Analysis of a Protocol for Dynamic Configuration of IPv4 Link Local Addresses Using UPPAAL*

Biniam Gebremichael[1], Frits Vaandrager[1], and Miaomiao Zhang[2]

[1] Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
{B.Gebremichael,F.Vaandrager}@cs.ru.nl
[2] Tongji University, China
miaomiao@mail.tongji.edu.cn

Abstract. Formal methods have been applied frequently to analyze (critical parts of) standards for communication protocols and it has been demonstrated that their application may help to improve the quality of these standards. Nevertheless, despite several decades of formal methods research, formal methods notations have rarely been included in the authoritative part of protocol standards. Also, the relationships between (abstract) formal models and informal protocol standards are typically obscure. It is our ambition to improve this situation. To establish the current state-of-the-art, we report in this paper on a case study in which UPPAAL is used to formally model parts of Zeroconf, a protocol for dynamic configuration of IPv4 link-local addresses that has been defined in RFC 3927 of the IETF. Our goal has been to construct a model that (a) is easy to understand by engineers, (b) comes as close as possible to the informal text (for each transition in the model there should be a corresponding piece of text in the RFC), and (c) may serve as a basis for formal verification. Our conclusion is that UPPAAL, which combines extended finite state machines, C-like syntax and concepts from timed automata theory, is able to model Zeroconf in a faithful and intuitive manner, using notations that are familiar to protocol engineers. Our modeling efforts revealed several errors (or at least ambiguities) in the RFC that no one else spotted before. We also identify a number of points where UPPAAL still can be improved. After applying a number of abstractions, UPPAAL is able to fully explore the state space of an instance of our model with three hosts, and to establish some correctness properties.

* This work was supported by PROGRESS project TES4199, Verification of Hard and Softly Timed Systems (HaaST), the European Community Project IST-2001-35304 Advanced Methods for Timed Systems (AMETIST), http://ametist.cs.utwente.nl, and the DFG/NWO bilateral cooperation project Validation of Stochastic Systems (VOSS2). Author names are listed alphabetically; all three authors made an equally significant contribution to this paper.

1 Introduction

Our society increasingly depends on the correct functioning of modern communication technology. Most prominent are (mobile) phones and Internet, but there are also networks in modern cars, trains, and airplanes, and the new generation of consumer electronics allows all sorts of devices to communicate with each other. The most important and most often used protocols that describe the operation of these networks are standardized. Examples of this are the Internet protocol (TCP/IP) and all its derivatives, FireWire/iLink (IEEE 1394), HAVi, WAP and BlueTooth. Due to a combination of factors, the complexity of these protocol standards is often very high: rapid changes in the capabilities of the underlying hardware, the fact that often many (industrial) parties are involved in standardization, each with its own interests, and market demands to extend the functionality of the protocol. Since these standards serve as a guide to implementors from many different companies, with very different backgrounds, it is vital that standards only allow for one clear interpretation, are complete and ensure the required functionality for each implementation. For most protocol standards this is clearly not the case. In fact, it is surprising that protocols that are of such immense importance to our society are typically written in informal language, with frequent ambiguities, omissions and inconsistencies. They also fail to state what properties are expected of a network running the protocol, and what it means for an implementation to conform to a standard.

By now there is ample evidence that formal (mathematical) techniques and tools may help to improve the quality of protocol standards. Numerous publications describe the formal modeling and analysis of critical parts of protocols, and via these case studies many previously undetected bugs have been detected (see e.g. [12, 7, 15, 22, 27, 18, 11, 28]). In most cases, these studies were carried out after the standard had been completed and involved guessing to fill in holes and resolve ambiguities in the standard. An exception is the work by Romijn and her colleagues at the Eindhoven University of Technology, who aim at applying formal methods already during the standard development process. Their efforts have resulted, for instance, in the discovery and correction of many errors, omissions and inconsistencies, as well as the addition of correctness properties, in the IEEE 1394.1 FireWire Net Update standard [26].

In order to avoid holes and ambiguities in standards the obvious way to go is to describe critical parts using programming and/or formal specification languages, similar to the way in which diagrams are used to specify the electrical circuits and mechanical parts. There have been joint attempts of academia and industry to arrive at formal description languages for protocols. The most notable attempts at this have been the LOTOS and SDL standardization efforts. Interestingly — to the best of our knowledge — these languages have never been used in the authoritative part of protocol standards. Some ISO standards written in the 90's had informative annexes written in a formal language such as LOTOS, but in case of ambiguity, the informal (natural language) part had precedence. Apparently, standardization bodies either did not trust/understand the formal specifications themselves or were afraid implementors would misinter-

pret them. Some protocol standard have extended finite state machines (EFSMs) inside, but these are mostly illustrative, not completely formal, and sometimes contain mistakes.[3] Bruns and Staskauskas [7] used C to describe the SONET Automatic Protection Switching (APS) protocol and report that developers found their C description easy to understand and superior to that which appeared in the APS standard. The lack of abstraction mechanisms is an obvious drawback of C. Bruns and Staskauskas suggest that abstraction can be introduced through an implementation relation.

The relationships between an (abstract) formal model of a protocol and the corresponding informal standard is typically obscure. As pointed out by [6], "current research seems to take the construction of verification models more or less for granted, although their development typically requires a coordinated integration of the experience, intuition and creativity of verification and domain experts. There is a great need for systematic methods for the construction of verification models to move on, and leave the current stage that can be characterized as that of model hacking. The ad-hoc construction of verification models obscures the relationship between models and the systems that they represent, and undermines the reliability and relevance of the verification results that are obtained." It is our ambition to improve this situation. To establish the current state-of-the-art of formal methods and tools, we report in this paper on a case study where we use Uppaal to formally model parts of Zeroconf, a protocol for dynamic configuration of IPv4 link-local addresses that is defined in RFC 3927 of the IETF. Our goal has been to construct a model of Zeroconf that (a) is easy to understand by engineers, (b) comes as close as possible to the informal text (for each transition in the model there should be a corresponding piece of text in the RFC), and (c) may serve as a basis for formal verification.

Uppaal [3] is an integrated tool environment for specification, validation and verification of real time systems modeled as networks of timed automata [2]. The tool is available for free for non-profit applications at `www.uppaal.com`. The language for the new version Uppaal 3.6 features a subset of the C programming language, a graphical user interface for specifying networks of EFSMs, and timed automata syntax for specifying timing constraints between events. Due to these extensions, the Uppaal syntax appears to be sufficiently expressive for the description of critical parts of protocol specifications:

1. The graphical syntax for EFSMs in combination with the C-like syntax are easy to understand for protocol designers and implementers, and very close to notations they use anyway.
2. Uppaal allows one to specify timing constraints between events, which is quite important in many protocol specifications.
3. The Uppaal language does have formal semantics and the transitions provide a simple abstraction mechanism for the C-like syntax: the semantics of a program is defined in terms of its effect on the observable state variables.
4. The Uppaal toolset supports simulation and model checking.

[3] See, for instance, `http://www.inrialpes.fr/vasy/Press/firewire.html`.

Zeroconf In this paper, we describe and analyze (critical parts of) *Zeroconf* [10], a protocol for dynamic configuration of IPv4 link-local addresses that has been defined by the IETF Network Working Group in RFC 3927 [9]. There are many situations in which one would like to use the Internet Protocol for local communication, for instance in the setting of in home digital networks or to establish communication between laptops. For these type of applications it is desirable to have a plug-and-play network in which new hosts automatically configure an IPv4 address, without using external configuration servers, like DHCP and DNS, or requiring users to set up each computer by hand. The Zeroconf protocol has been proposed by the IETF to achieve exactly this. It describes how a host may automatically configure an interface with an IPv4 address within the 169.254/16 prefix that is valid for communication with other devices connected to the same physical (or logical) link. The most widely adopted Zeroconf implementation is Bonjour from Apple Computer[4], but several other implementations are available.[5]

Contribution The contribution of this paper is, first of all, a formal model of a critical part of Zeroconf — a protocol with clear practical relevance — that is easy to understand, faithful to the RFC, and with an extensive discussion of the relationship between the model and the RFC. Our modeling efforts revealed several errors (or at least ambiguities) in the RFC that no one else spotted before. We also identify several directions where Uppaal still can be improved. Finally, after applying several nontrivial abstractions we manage, using Uppaal, to fully explore the state space of an instance of our model with three hosts, and establish some interesting correctness properties.

Related Work The Zeroconf protocol involves a number of probabilistic aspects that are not incorporated in our Uppaal model: hosts select IP-addresses randomly using a pseudo-random number generator, and at some point during the protocol they wait for a random amount of time selected uniformly from an interval. The probabilistic behavior of Zeroconf has been studied in [5, 20]. The primary goal of [5] was to investigate the trade off between reliability and effectiveness of the protocol using a stochastic cost model. The model of [5], which only involves a single host, is quite appropriate in capturing the probabilistic behavior of IP address configuration and conflict handling, but the analysis takes place at a level that is much more abstract then the RFC. Based on an earlier version of the present paper, a more detailed model has been presented in [20] using the probabilistic model checker PRISM [21]. The model checking results reported in [20] are very interesting, but the precise relationship between the model and the RFC is unclear (for instance, in the model of [20] address defense only occurs *before* a host is using an IP address). Our motivation for using Uppaal instead of PRISM was that the input language of PRISM is too primitive for our purposes (no GUI, just a few datatypes, no support of C-like syntax,..).

[4] See `http://developer.apple.com/networking/bonjour/`.
[5] See `http://en.wikipedia.org/wiki/Zeroconf`.

A toolset that combines the functionality of Uppaal and PRISM would be ideal for dealing with the Zeroconf protocol.

Paper outline The organization of the paper is as follows. In Section 2 we explain the protocol and our Uppaal model of it. In Section 3, a manual proof of correctness of the protocol is presented. We also define an abstracted model, which can be automatically explored in the case of 3 hosts. Finally, in Section 4 we present our conclusions and several directions for future research.

The Uppaal models described in this paper are available on-line at

`http://www.cs.ru.nl/ita/publications/papers/fvaan/zeroconf/`.

2 The Protocol

In this section, we describe the Zeroconf protocol, our Uppaal model of it, and the relationship between our model and RFC 3927 [9], the official protocol standard.

A Zeroconf network is composed of a set of hosts on the same link. Hosts in the Zeroconf network can be devices that are present at home, office, embedded systems “plugged together” as in an automobile, or the laptops of three friends who are writing a joint paper and want to share a file. The goal of Zeroconf is to enable networking in the absence of configuration and administration services. The core of RFC 3927 [9] concerns the dynamic configuration of IPv4 link-local addresses, and this is the part on which we will focus in this paper.

The basic idea of Zeroconf is trivial and easy to explain. A host that wants to configure a new IP link-local address randomly selects an address from a specified range and then broadcasts a few identical messages to the other hosts, seperated by some delay, asking whether someone is already using the address. If one of the other hosts indicates that it is using the other address, the host starts all over again. Otherwise, it may start using the address after waiting a certain amount of time.

One may view Zeroconf as a distributed mutual exclusion algorithm in which the resources are IP addresses. A goal of Zeroconf is to prevent that at any point two different hosts are using the same IP address. The underlying algorithm used in Zeroconf is similar to Fisher’s mutual exclusion algorithm [1, 24] and makes essential use of timing. However, whereas Fischer’s algorithm uses a shared variable for communication between processes, Zeroconf uses broadcast communication. Within Zeroconf, hosts do not aim at acquiring access to a specific critical section (IP address); it is enough to obtain access to one of the 65024 available critical sections (IP addresses).

2.1 Basic Modelling Assumptions

RFC 3927 assumes a *set* of hosts. This set is not fixed and host may join and leave while the protocol is running. Since Uppaal does not support dynamic

process creation, we assume a fixed number of `k` hosts. It may take arbitrary long before a host becomes active in the protocol and one may argue that in this way creation of new hosts is being captured. We do not model host failure or termination but it would be easy to add this.[6] In our model, a host that has configured an IP address may stop sending messages. From an observational point of view this is the same as a (stopping) failure. A phenomenon that may occur in practice, and which we have also not modeled here, is that previously separate Zeroconf networks are joined.

The behavior of each host is modeled by three timed automata that are composed in parallel: `Config`, `InputHandler` and `Regular`. Automaton `Config` models the configuration of a new IP address, `InputHandler` takes care of the incoming messages, and `Regular` is an abstract model of the activity of all the other processes running on the host. All three automata are parametrized by the hardware address of the host they belong to. For convenience, in our model a hardware address is a natural number in the range $\mathtt{0}$ to $\mathtt{k}-\mathtt{1}$. Within UPPAAL, the scalarset type `scalar`[k] denotes the set $\{\mathtt{0},\ldots,\mathtt{k}-\mathtt{1}\}$.

```
typedef scalar[k] HAType;
```

On scalarsets only restricted operations are permitted. As a consequence, a scalarset is a fully symmetric type and the behavior of a model is invariant under arbitrary permutations of the elements of a scalarset [19, 17]. By defining a scalarset type rather than a subrange, we tell UPPAAL that within our model all the hardware addresses (and therefore also the hosts) play a fully symmetric role, which makes it possible to exploit this symmetry during exploration of the state space.

As already discussed in the introduction, our model abstracts from probabilistic issues, and contains non deterministic choice whenever the RFC specifies probabilistic choice.

2.2 The Network

RFC 3927 states the following assumption about the underlying network [page 4, section 1.3]:

> "This specification applies to all IEEE 802 Local Area Networks (LANs) [802], including Ethernet [802.3], Token-Ring [802.5] and IEEE 802.11 wireless LANs [802.11], as well as to other link-layer technologies that operate at data rates of at least 1 Mbps, have a round-trip latency of at most one second, and support ARP [RFC826]."

The Address Resolution Protocol (ARP, [25]) is a widely used method for converting protocol addresses (e.g., IP addresses) to local network ("hardware") addresses (e.g., Ethernet addresses). It allows dynamic distribution of the information needed to build tables to translate protocol addresses to hardware

[6] Notationally it would be somewhat cumbersome as UPPAAL still lacks a notion of hierarchical state.

addresses. Within Zeroconf all messages are ARP packets. For our model, the relevant information in an ARP packet consists of (1) a sender hardware address, (2) a sender IP address, (3) a target IP address, and (4) the type of the packet, which can be either "request" or "reply". Hence, an ARP packet can be defined as a UPPAAL C data type as follows:

```
typedef struct{
        HAType   senderHA;  // sender hardware address
        IPType   senderIP;  // sender IP address
        IPType   targetIP;  // target IP address
        bool     request;   // is the packet a Request or a Reply
}ARP_packet;
```

Here we use the convention that the `request` field is `true` for ARP requests and `false` for ARP replies. A host that is looking for the local network address of another host with IP address x, broadcasts an ARP request packet with the field `targetIP` set to x. A host with IP address x will then return an ARP reply packet with the field `senderHA` set to its local network address.

In Zeroconf, all ARP packets are broadcast [page 13, section 2.5]:

> "All ARP packets (*replies* as well as requests) that contain a Link- Local 'sender IP address' MUST be sent using link-layer broadcast instead of link-layer unicast. This aids timely detection of duplicate addresses."

We model the underlying network as a set of `n` identical `Network` automata. Each of these automata takes care of handling a single ARP request at a time. To express that all the automata are symmetric, we define a type

```
typedef scalar[k] NetworkType;
```

and parametrize each automaton by an element `j` from this type.

The main reason for having `n` automata is that this allows us to model round-trip latencies in UPPAAL. Fig. 1 schematically illustrates the operation of a `Network` automaton. After a request from a host comes in (`send_req`), this is broadcast to all hosts (`receive_msg`). In case there is a corresponding answer (this may be a reply or a request packet) this is accepted (`answer`) and also broadcast to all hosts (`receive_msg`). All these interactions take place within 1 second. After completing its task the `Network` automaton returns to its initial location, ready to take care of a new request.

To simplify our model, we assume that a host handles an incoming ARP request in zero time, i.e., we adopt the synchrony hypothesis that is well-known from synchronous programming [4]. A desktop computer can realistically answer an ARP in 100μs. A device like a SitePlayer could take up to 10ms. Neither have a significant impact on achieving a round-trip delay under 1s. By taking the conceptual view that the 1s which `Network` may use to do its work *includes* the time needed by a host to generate a reply, we avoid cumbersome modeling of input buffers at each host.

Before explaining our UPPAAL model of the `Network` automaton in detail (in Section 2.5), we now turn our attention to the core part of RFC 3927, which concerns address configuration.

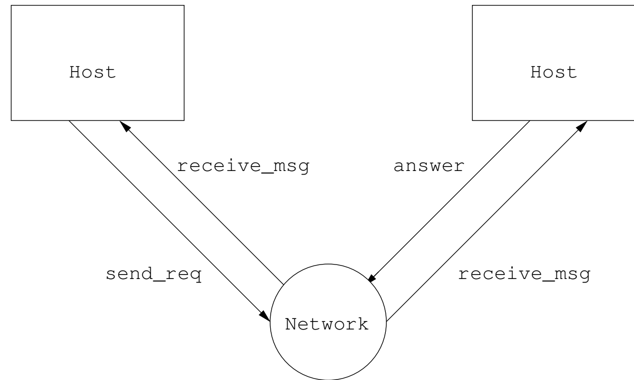

Fig. 1. Interaction between `Network` automaton and hosts.

2.3 Address Configuration

Fig. 2 displays the automaton `Config[j]`, which specifies how host `j` configures a new IP address. Each host starts in location `INIT`, where it resides until it has selected an IP address. According to the RFC [page 9, section 2.1]:

> "When a host wishes to configure an IPv4 Link-Local address, it selects an address using a pseudo-random number generator with a uniform distribution in the range from 169.254.1.0 to 169.254.254.255 inclusive. The IPv4 prefix 169.254/16 is registered with the IANA for this purpose. The first 256 and last 256 addresses in the 169.254/16 prefix are reserved for future use and MUST NOT be selected by a host using this dynamic configuration mechanism."

Just to keep the code simple, we abstract sligthly from the naming of IP addresses. An IP address simply is a number in the range `0` to `m`, where `m` denotes the number of available link-local addresses:

```
typedef int[0,m] IPType;
```

The address `0` corresponds to the all zeroes IP address 0.0.0.0, which is used as a special 'unknown' or 'undefined' value in the protocol, and the addresses `1` to `m` correspond to the addresses registered with the IANA, listed in increasing order. Due to the special role of the address `0`, we cannot declare `IPType` as a (fully symmetric) scalarset, and thus we declare it as a subrange instead. A transition from location `INIT` to location `WAIT` takes place when an address has been selected. Via the UPPAAL select statement `address:int[1,m]` we nondeterministically bind identifier `address` to a value in the interval `[1,m]`. This means that there is an instance of the transition for each number in this interval. In this way, we express that an IP address is chosen nondeterministically. The selected address is stored in state variable `IP`[`j`].

The RFC continues [page 11, section 2.2.1]:

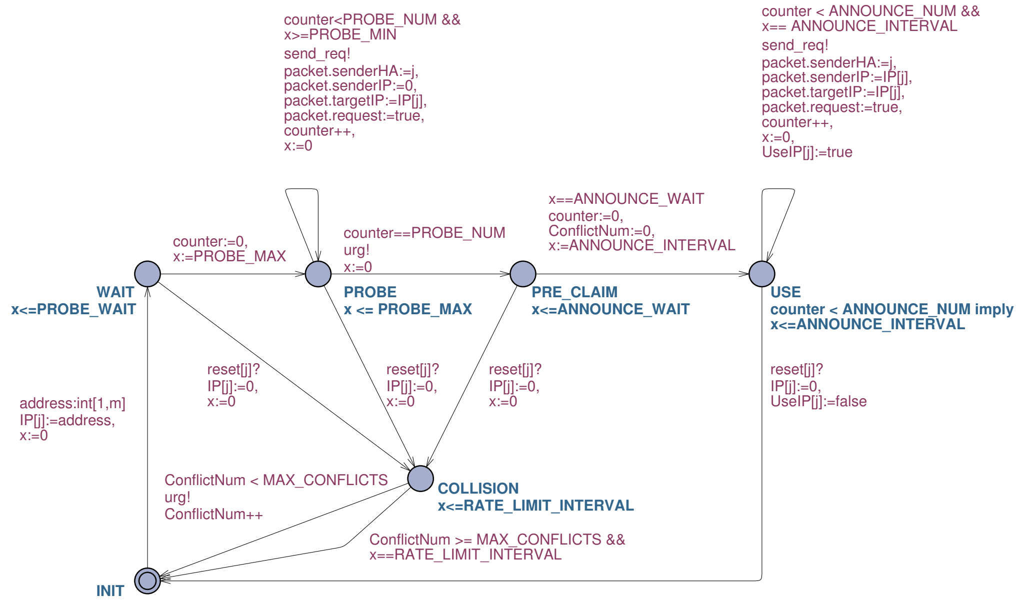

Fig. 2. Automaton `Config`.

> "When ready to begin probing, the host should then wait for a random time interval selected uniformly in the range zero to PROBE_WAIT seconds, and should then send PROBE_NUM probe packets, each of these probe packets spaced randomly, PROBE_MIN to PROBE_MAX seconds apart."

The waiting period is modeled by resetting a local clock `x` upon entering location `WAIT` and by bounding the time the host may stay in `WAIT` with an invariant `x <= PROBE_WAIT`. At any point the host may move to location `PROBE`, where it starts sending "probes". The notion of an ARP Probe is specified in the RFC as follows:

> "A host probes to see if an address is already in use by broadcasting an ARP Request for the desired address. The client MUST fill in the 'sender hardware address' field of the ARP Request with the hardware address of the interface through which it is sending the packet. The 'sender IP address' field MUST be set to all zeroes, to avoid polluting ARP caches in other hosts on the same link in the case where the address turns out to be already in use by another host. The 'target hardware address' field is ignored and SHOULD be set to all zeroes. The 'target IP address' field MUST be set to the address being probed. An ARP Request constructed this way with an all-zero 'sender IP address' is referred to as an "ARP Probe"."

Sending ARP Probes is modeled via actions `send_req[j]!` that synchronize with the network. The actual packet is communicated via a global shared variable `packet` of type `ARP_packet`: in UPPAAL the assignments in an output (!) transition are executed before the assignments in a synchronizing input (?) transition, and this allows us to assign a value to `packet` in a `send_req[j]!` transition, which is then picked up by a corresponding `send_req[j]?` transition by a `Network` automaton. The lower and upper bounds of the probe interval are expressed in our model with a guard `x >= PROBE_MIN` on the sending transition and an invariant `x <= PROBE_MAX` on location `PROBE`, respectively. By setting `x` to `PROBE_MAX` in the transition from `WAIT` to `PROBE`, we express that the first probe is sent immediately. A local variable `counter` is used to record the number of probes that have been sent. After the probing phase is successfully completed, the automaton jumps to location `PRE_CLAIM`. The urgent broadcast channel `urg` ensures that this transition is taken as soon as it is enabled. As the reader can check, the translation from the RFC description of the probing phase to UPPAAL is straightforward.

According to the RFC:

> "If, by ANNOUNCE_WAIT seconds after the transmission of the last ARP Probe no conflicting ARP Reply or ARP Probe has been received, then the host has successfully claimed the desired IPv4 Link-Local address."

Clock `x` is used to ensure that exactly `ANNOUNCE_WAIT` time units are spent in location `PRE_CLAIM`. A transition from location `PRE_CLAIM` to location `USE` is taken to indicate that the host has successfully claimed an address.

In our model, automaton `InputHandler[j]` (which will be explained in Section 2.4) takes care of handling incoming messages. If `InputHandler[j]` decides that, due to some conflict, a new address must be configured, it sends a `reset[j]` signal to automaton `Config[j]`. Upon receiving this signal, `Config[j]` sets `IP[j]` to 0 and jumps to location `COLLISION`. According to the RFC:

> "A host should maintain a counter of the number of address conflicts it has experienced in the process of trying to acquire an address, and if the number of conflicts exceeds MAX_CONFLICTS then the host MUST limit the rate at which it probes for new addresses to no more than one new address per RATE_LIMIT_INTERVAL. This is to prevent catastrophic ARP storms in pathological failure cases, such as a rogue host that answers all ARP Probes, causing legitimate hosts to go into an infinite loop attempting to select a usable address."

A counter `ConflictNum` is used in our model to record the number of conflicts that have occurred during the process of acquiring an IP address. Depending on the value of `ConflictNum`, the automaton returns to location `INIT` immediately or first waits for `RATE_LIMIT_INTERVAL` time units. Again, the correspondence between the RFC text and our UPPAAL model is straightforward.

In location `USE` the host announces the new address that it has just claimed [page 12, section 2.4]:

> "Having probed to determine a unique address to use, the host MUST then announce its claimed address by broadcasting ANNOUNCE_NUM ARP announcements, spaced ANNOUNCE_INTERVAL seconds apart. An ARP announcement is identical to the ARP Probe described above, except that now the sender and target IP addresses are both set to the host's newly selected IPv4 address. The purpose of these ARP announcements is to make sure that other hosts on the link do not have stale ARP cache entries left over from some other host that may previously have been using the same address."

The RFC does not specify upper and lower bounds on the time that may elapse between sending the last ARP Probe and sending the first ARP Announcement. However, according to the protocol designers upper and lower bound both equal `ANNOUNCE_WAIT` [8]. Also, the RFC does not specify whether a host may immediately start using a newly claimed address (in parallel with sending the ARP Announcements), or whether it should first send out all announcements. According to the designers, a host should send the first ARP Announcement, and then it can immediately start using the address [8]. So the second announcement goes out `ANNOUNCE_INTERVAL` seconds later, but other traffic does not need to be held up waiting for that. Finally, the RFC does not specify the tolerance that is permitted on the timing of ARP Announcements. Since no physical device can consistently send messages spaced *exactly* `ANNOUNCE_INTERVAL` seconds apart, strictly speaking it is impossible for an implementation to conform to the RFC. According to the designers, the RFC does not specify accuracy requirements, partly because the protocol is robust to a wide range of variations, so it does not matter [8]. We decided to follow the RFC and not specify accuracy requirements, but if someone wants to use our model for automatic generation of tests, for instance using the UPPAAL-TRON toolset [23], he or she will have to modify our model at this point.

With this additional information, the modeling of the announcement phase in UPPAAL is straightforward and analogous to that of the probing phase. After sending the first announcement, Boolean variable `UseIP[j]` is set to `true`. This enables automaton `Regular[j]`, displayed in Fig. 3, to start sending out regular ARP requests packets with the `senderIP` field set to `IP[j]` and the `targetIP` field set to an arbitrary link-local address. However, even when a host is using an IP address still at any moment a conflict may arise. When this happens automaton `Config[j]` returns to its initial location and `UseIP[j]` is set to `false` again.

2.4 Input Handler

Automaton `InputHandler[j]` receives incoming ARP packets and decides what to do with them. Input handling is described at various places in RFC 3937, which makes it nontrivial to determine the reaction to an arbitrary ARP packet, also because Zeroconf runs on top of the ARP protocol, which it sometimes follows but sometimes overrules. Automaton `InputHandler` is displayed in Fig. 4. When a new packet arrives, that is, when a `receive_msg[j]?` transition occurs, the automaton calls a function `ihandler` to find out what to do. This function computes

Fig. 3. Automation `Regular[j]`.

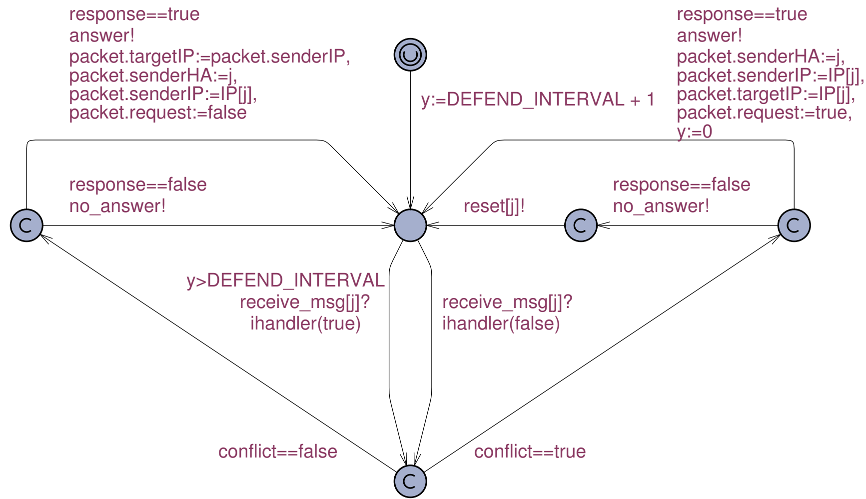

Fig. 4. Automaton `InputHandler[j]`.

two bits, `conflict` and `response`: if `conflict==true` then some other host is using or trying to use the IP address the host has selected and if `response==true` then a packet will be send in response. Thus the value of the two bits determines the reaction of the input handler to the incoming packet:

1. If `conflict==true` and `response==false`, a `reset[j]` signal is sent.
2. If `conflict==true` and `response==true`, an ARP Announcement is broadcast.
3. If `conflict==false` and `response==true`, an ARP Reply is broadcast.
4. If `conflict==false` and `response==false`, the packet is ignored.

The definition of `ihandler` is listed in Fig. 5. Function `ihandler` has a parameter `defend` which may be either `false` or `true`. This parameter, which indicates that a host will defend its IP address in case of a conflicting ARP request, may be `true` only if there has been no other conflict during the last `DEFEND_INTERVAL` time units. Clock `y` is used to measure the time since the last conflict. Altogether, the input handler has to distinguish 9 scenarios.

```
void ihandler(bool defend)
{
  if (IP[j]==0) // Scenario A: I have not selected an IP address
        {response:=false; conflict:=false;}
  else if (packet.senderHA==j) // Scenario B: I have sent the packet myself
        {response:=false; conflict:=false;}
  else if (packet.senderIP==IP[j]) //There is a conflict: somebody else is using my IP address!
        {
        conflict:=true;
        if (not UseIP[j]) // Scenario C: select a new address
              response:=false;
        else if (defend) // Scenario D: I am going to defend my address
              response:=true;
        else  // Scenario E: I will not defend my address
              response:=false;
        }
  else if (not UseIP[j])
        {
        response:=false;
        if (packet.targetIP==IP[j] && packet.request && packet.senderIP==0) // Scenario F: conflicting probe
             conflict:=true;
        else  //Scenario G: Packet is not conflicting with IP address that I want to use
             conflict:=false;
        }
  else // Incoming packet is not conflicting with IP address that I am using
        {
        conflict:=false;
        if (packet.targetIP==IP[j] && packet.request) // Scenario H: answer regular ARP request
             response:=true;
        else // Scenario I: no reply message required
             response:=false;
        }
}
```

Fig. 5. Function `ihandler`.

Scenario A. Clearly, if a packet comes in when a host has not yet selected an IP address it should be ignored. This scenario is not listed explicitly in the RFC but should be obvious.

Scenario B. Packets that a host has sent itself can be ignored. Also this scenario is implicit in the RFC.

Scenario C. A conflict may arise when another host sends a packet with the `senderIP` field set to `IP[j]`. This occurs in Scenario C, which is described on [page 11, section 2.2.1]:

> "If during this period, from the beginning of the probing process until ANNOUNCE_WAIT seconds after the last probe packet is sent, the host receives any ARP packet (Request *or* Reply) on the interface where the probe is being performed where the packet's 'sender IP address' is the address being probed for, then the host MUST treat this address as being in use by some other host, and MUST select a new pseudo-random address and repeat the process."

Scenarios D and E. In the previous scenario, `UseIP[j]==false`. The case with `UseIP[j]==true` is also described in the RFC [page 12, section 2.5]:

> "Address conflict detection is not limited to the address selection phase, when a host is sending ARP Probes. Address conflict detection is an on-going process that is in effect for as long as a host is using an IPv4 Link-Local address. At any time, if a host receives an ARP packet (request *or* reply) on an interface where the 'sender IP address' is the IP address the host has configured for that interface, but the 'sender hardware address' does not match the hardware address of that interface, then this is a conflicting ARP packet, indicating an address conflict.
> A host MUST respond to a conflicting ARP packet as described in either (a) or (b) below:
> (a) Upon receiving a conflicting ARP packet, a host MAY elect to immediately configure a new IPv4 Link-Local address as described above, or
> (b) If a host currently has active TCP connections or other reasons to prefer to keep the same IPv4 address, and it has not seen any other conflicting ARP packets within the last DEFEND_INTERVAL seconds, then it MAY elect to attempt to defend its address by recording the time that the conflicting ARP packet was received, and then broadcasting one single ARP Announcement, giving its own IP and hardware addresses as the sender addresses of the ARP. Having done this, the host can then continue to use the address normally without any further special action. However, if this is not the first conflicting ARP packet the host has seen, and the time recorded for the previous conflicting ARP packet is recent, within DEFEND_INTERVAL seconds, then the host MUST immediately cease using this address and configure a new IPv4 Link-Local address as described above. This is necessary to ensure that two hosts do not get stuck in an endless loop with both hosts trying to defend the same address.
> A host MUST respond to conflicting ARP packets as described in either (a) or (b) above. A host MUST NOT ignore conflicting ARP packets."

Case (a) corresponds to our scenario E. This scenario occurs when the right `receive_msg?` transition in the automaton is taken, which sets `defend` to `false`, Case (b) corresponds to scenario D. This scenario occurs when the left `receive_msg?` transition is taken, which sets `defend` to `true`.

The interpretation of the sentence "and it has not seen any other conflicting ARP packets within the last DEFEND_INTERVAL seconds" in the previous quotation from the RFC is not entirely clear. Is a host allowed to defend its address if there has been a recent conflict concerning a *different* address (but no previous conflict concerning the current address)? Strictly speaking, the host has seen a conflicting packet and it may not defend. However, the conflict concerned a different address, and the motivation for recording the time since the last conflict has been to rule out a scenario in which two hosts get stuck in an endless loop trying to defend the *same* addess. Thus one could also argue that in this situation

a host may defend its address. To model this interpretation, one would have to add an assignment `y := DEFEND_INTERVAL+1` to the reset transition of the input handler.

Scenarios F and G. The RFC specifies one more conflict scenario [page 11, section 2.2.1]:

> "In addition, if during this period [from the beginning of the probing process until ANNOUNCE_WAIT seconds after the last probe packet is sent] the host receives any ARP Probe where the packet's 'target IP address' is the address being probed for, and the packet's 'sender hardware address' is not the hardware address of the interface the host is attempting to configure, then the host MUST similarly treat this as an address conflict and select a new address as above. This can occur if two (or more) hosts attempt to configure the same IPv4 Link-Local address at the same time."

In the `ihandler` code, this corresponds to scenario F. Scenario G, which is implicit in the RFC, occurs when the incoming packet is not conflicting and the host is not yet using an IP address. In this case the incoming packet is ignored.

Scenario H and I. The Address Resolution Protocol (RFC 826) [25] specifies that if a host receives an ARP request packet, it should return an ARP reply packet if it uses an IP address that equals the target protocol address of this request. In the reply packet the hardware and protocol field should be swapped, putting the local hardware and protocol addresses in the sender fields. Zeroconf (RFC 3927) is not explicit about conformance to RFC 826, but in our model we take the view that once a host is using an IP address, it answers regular ARP requests in agreement with RFC 826 except when (a) the request has been broadcast by the host itself, or (b) there is a conflict. This is scenario H in our model. The final Scenario I occurs when the incoming packet is not conflicting with the IP address that the host is using, and no reply packet needs to be sent.

Note that in automaton `InputHandler[j]` some of the locations are committed (C). In UPPAAL, when a system reaches a committed location, the next transition has to be an outgoing transition from that location.[7] The use of committed locations here is a modeling trick. When a network automaton delivers a packet to an input handler via a `receive_msg` synchronization, the input handler has to return an answer (if there is one) instantaneously (by the synchrony hypothesis). But since in general there are many network automata active, we need to ensure that the answer is picked up by the right automaton. Introducing separate channel names for each network automaton or pi-calculus like private channels would create too much overhead. Our trick is that a network automaton may only synchronize on an `answer` action right after performing a `receive_msg` action. By making the locations of the input handler following a `receive_msg`

[7] Or from a committed location from another component, but such a situation does not occur in our model.

transition committed, we ensure that the reply is picked up by the right network automaton. Essentially, the `receive_msg` and `answer` synchronizations take place in a single atomic transaction. In case the input handler does not generate an answer, it uses a `no_answer` action to inform the network automaton about this. This synchronization is an artifact of our model since in reality no signal is sent in this case.

2.5 The `Network` Automaton

The `Network` automaton is shown in Fig. 6. Initially the automaton is in its `IDLE`

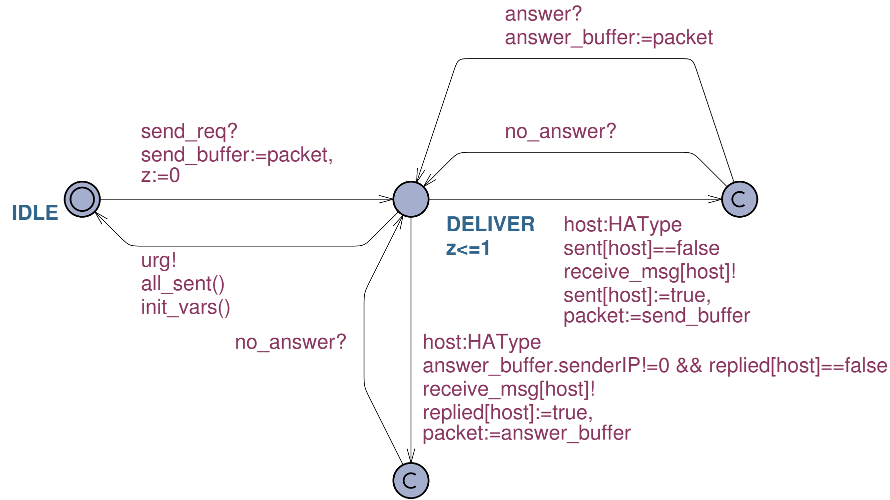

Fig. 6. The `Network` automaton.

location. As soon as it receives a packet from a host via `send_req`, it jumps to the `DELIVER` location. Since there is no lower bound on message delivery time, message delivery may start immediately. A local clock `z` is reset to zero and an invariant $z \leq 1$ ensures that within 1 second the network broadcasts the packet (and the answer if there is one) to all hosts. In our model we assume that there is at most one host that wants to answer any given request, and that an answer does not induce subsequent answers. It is possible to modify the `Network` automaton so that it can handle multiple and successive answers, but this requires additional state variables and more complicated data structures. Our `Network` automaton has two local buffers: `send_buffer` stores the packet that was sent by the host and `answer_buffer` stores an answer when it arrives. In addition, `Network` maintains Boolean arrays `sent` and `replied` to records to which hosts the packets have already been delivered. Using the Uppaal select

statement, the automaton non deterministically selects in which order a packet is delivered to the different hosts. A host may return an answer upon receipt of a request, as explained in Subsection 2.4. The lower transition labeled with `receive_msg` is enabled as soon as there is an answer packet in `answer_buffer`. The network returns to its `IDLE` location and resets its buffers, as soon as all messages have been sent. This is checked by the Boolean function `all_sent`:

```
bool all_sent(){
// the request packet has been sent to all
// and if there is an answer it has been sent to all
return ((forall (i : HAType) sent[i]) and
        ((answer_buffer.senderIP!=0) imply
         (forall (i : HAType) replied[i]))); }
```

Upon return to the `IDLE` location all variables are re-initialized.

2.6 Dimensioning the Complete Model

The RFC [page 25, section 9] specifies the following values for the different timing constants. These definitions are copied almost verbatim in the Uppaal declaration section of our model.

```
"PROBE_WAIT          1 second   (initial random delay)
 PROBE_NUM           3          (number of probe packets)
 PROBE_MIN           1 second   (minimum delay till repeated probe)
 PROBE_MAX           2 seconds  (maximum delay till repeated probe)
 ANNOUNCE_WAIT       2 seconds  (delay before announcing)
 ANNOUNCE_NUM        2          (number of announcement packets)
 ANNOUNCE_INTERVAL   2 seconds  (time between announcement packets)
 MAX_CONFLICTS      10          (max conflicts before rate limiting)
 RATE_LIMIT_INTERVAL 60 seconds (delay between successive attempts)
 DEFEND_INTERVAL    10 seconds  (minimum interval between defensive ARPs)."
```

In general, a Zeroconf network has 65024 IP addresses available and it is suitable for up to 1300 hosts [9]. These values are too big for automatic verification and with 3 hosts and 65024 IP addresses also the Uppaal simulator runs out of memory.

A next issue regarding the dimensioning of the model is the number `n` of `Network` automata, i.e., the maximal number of ARP requests that may be in transit at any given point. In our model, a host may select an IP address, send a probe, and return to the initial location via a reset in zero time. In fact, this behavior may be repeated `MAX_CONFLICTS` times in a row in zero time. Once a host is using an IP address, the number of messages in transit may increase even further (in fact unboundedly) since there is no lower bound on the time between successive ARP requests. Uppaal forces us to bound the number of `Network` automata to some number `n`.

3 Verification

The model described in Section 2 is very close to the RFC definition of the protocol. However, as a result the model is too big for Uppaal to do a complete

state space exploration for nontrivial instances, even when we use symmetry reduction.

The RFC specification of the protocol does not specify what properties the protocol must satisfy. However, it is clear that at least the following two correctness properties are desirable:[8]

1. Mutual exclusion, i.e., no two hosts may use same IP address. This can be specified in UPPAAL as follows:

```
ME = A[] forall (i: HAType) forall (j: HAType)
        (UseIP[i] && UseIP[j] && IP[i]==IP[j]) imply i==j
```

2. The network has no deadlock, i.e, in each reachable state a transition is possible. Or in UPPAAL syntax:

```
DL = A[] not deadlock
```

Using the latest version of UPPAAL (3.6 beta), we only managed to establish `ME` and `DL` for the instance with 2 hosts, 1 IP address and 2 network automata. Nevertheless, it is rather obvious that Zeroconf satisfies the mutual exclusion property and is free of deadlocks. In the remainder of this section, we first present a sketch of a manual proof of mutual exclusion and then describe an abstracted version of our model that can be fully explored by UPPAAL in the case of 3 hosts and used to prove mutual exclusion automatically for this instance. We claim that the full model has no deadlocks but do not present the (long and tedious) proof here. Since the abstract model overapproximates the full model, absence of deadlock in the first does not imply absence of deadlock in the second.

3.1 Manual Proof of Mutual Exclusion

Theorem 1. *For each instance of our Zeroconf model (i.e., any number of IP addresses, any number of hosts, and any number of network automata), the mutual exclusion property* `ME` *holds.*

Proof. (Sketch) Suppose `i` and `j` are hardware addresses with `i != j`, and suppose that in some reachable state s, `UseIP[i]`, `UseIP[j]` and `IP[i]=IP[j]`. We derive a contradiction. Consider an execution α leading up to state s, i.e., a finite sequence of delay and action transitions in the semantics of the model leading from the start state to s. Without loss of generality, we may assume that host `j` enters the critical section before (but possibly at the same time as) `i`. Observe that before a host enters the "critical section" (where it uses an IP address) it resides at least 6 time units in the "trying region" (where it has selected an IP address but is not yet using it). Formally, the trying region of host `i` is characterized by the predicate

[8] Mutual exclusion will not hold in an extension of our model in which Zeroconf networks can be merged. In such an extension the specification should be weakened: mutual exclusion may be violated after a join, but as soon as the violation is detected (due to an ARP packet) mutual exclusion will be restored within a specified amount of time (provided meanwhile no further joins occur).

```
Config(i).WAIT || Config(i).PROBE || Config(i).PRE_CLAIM ||
      (Config(i).USE && Config(i).counter==0)
```

and the critical section by

```
Config(i).USE && Config(i).counter>0
```

Moreover, exactly 2 time units before entering the critical section, a host sends a (in fact, the last) probe packet.

Assume that host `i` is in its critical section from time $t0$ onwards, and is in its trying region from time $t1$ to $t0$. Similarly, host `j` is in its critical section from time $u0$ onwards, and is in its trying region from time $u1$ to $u0$. Let t be the time at which host `i` sends its last probe and let u be the time at which this probe is received by the input handler of host `j`. Then we have the following (in)equalities:

$$\begin{aligned} t0 &\geq u0 \\ t0 &\geq t1 + 6 \\ u0 &\geq u1 + 6 \\ t &= t0 - 2 \\ u &\geq t \\ t &\geq u - 1 \end{aligned}$$

We consider two cases:

1. See Figure 7. The last probe arrives at host `j` before it enters the critical section. Then `j` must be in its trying region since:

   $$u \geq t = t0 - 2 \geq u0 - 2 > u0 - 6 \geq u1.$$

   But this means that host `j`'s input handler, upon receipt of the conflicting probe, will generate a reset (Scenario F) and drive `Config(j)` back to its initial state, i.e, out of the trying region. Contradication.
2. See Figure 8. The last probe arrives at host `j` after it enters the critical section. But this means that host `j`'s input handler, upon receipt of the probe, will return a reply message (Scenario H). Since we assume a roundtrip delay of at most 1 time unit, this reply message will arrive at `i` at some time t' with $t' \leq t + 1$. At time t' host `i` is still in its trying region since

   $$t0 = t + 2 > t + 1 \geq t' \geq t = t0 - 2 > t0 - 6 \geq t1.$$

   Hence, the input handler will generate a reset upon receipt of this reply message (Scenario C) and drive `Config(i)` back to its initial state, i.e, out of its trying region. Contradication. QED

Formalization/mechanization of the proof of Theorem 1, for instance in PVS using the basic setup of [28], should be a routine exercise.

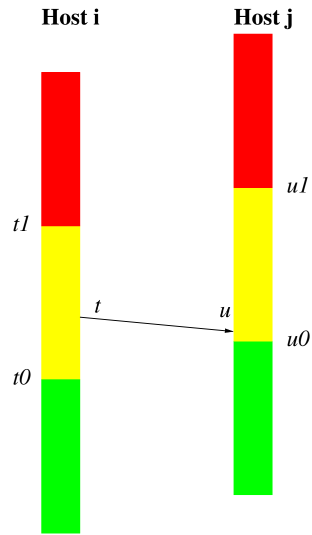

Fig. 7. Last probe arrives at `j` before it enters critical section.

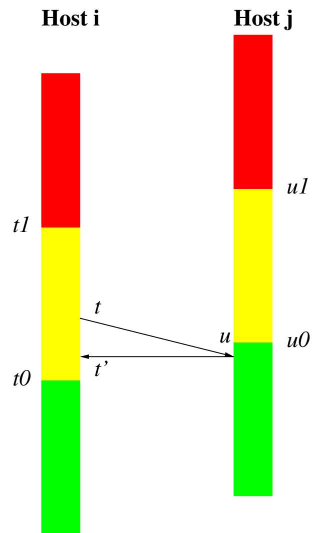

Fig. 8. Last probe arrives at `j` after it enters critical section.

Inspection of the proof indicates that Zeroconf is extremly robust: the protocol has been designed to handle all kinds of error scenarios (loss of messages, failure of hosts, merge of networks) which do not occur within our idealized model. Without these errors, it suffices (for mutual exclusion) to send out a single probe (`PROBE_NUM=1`), there is no need for sending announcements (`ANNOUNCE_NUM=0`), and a host may start using an address after waiting any time longer than the maximal communication delay. For a model of this simplified protocol with 3 hosts Uppaal can verify `ME` and `DE` in a few seconds on a standard PC.

3.2 Abstractions

To make automatic verification of mutual exclusion possible for the full protocol in the case with 3 hosts, we applied a combination of several abstractions (on top of the abstractions that are already applied by Uppaal). Also, we had to make the additional assumption that at any time for each host there is at most one outgoing message in transit. This allows us to associate a single network automaton to each host, which only accepts packets from this host when empty.

Dead Variable Reduction Dead variable reduction is a well known static analysis technique, that has for instance been studied in the PhD thesis of Yorav [29]. In Yorav's terminology, a variable v is *used* in a transition if it appears in the guard or in the right hand side of an assignment. Variable v is *defined* in a transition if it is in the left hand side of an assignment. Notice that in an assignment $v := v + 1$, v is first used, and then it is defined. A variable v is said to be *dead* at a location l if on every execution path from l, v is defined before it is used, or is never used at all. Clearly, systems that only differ in the values of dead variables are equivalent in a very strong sense (bisimilar). In our Zeroconf model, variable `counter` of `Config(j)` is dead in locations `COLLISSION` and `INIT`. Hence, setting `counter:=0` upon occurrence of a reset transition will not affect whether the ME property holds or not. Another example are the variable `conflict` and `response`, which are dead in the non-urgent locations of `InputHandler(j)`, and can be reset to `false` upon entering these locations. Finally, variable `packet` is only used within synchronization transitions and it can be set to a default value following each transition.

Overapproximation By weakening guards or by making an urgent channel non-urgent, we add behavior to an automaton. If we manage to prove an invariant for the larger ("overapproximated") automaton it will certainly hold for the smaller, original automaton. If, as a result of weakening, a variable is tested in none of the transitions and it also does not occur in the invariant that we are trying to prove, it can be safely omitted from the model. In the case of Zeroconf, overapproximation and subsequent variable elimination can be applied in the following two situations:

1. We may weaken the guards of the two transitions from `COLLISION` to `INIT` in `Config(j)` to `true`, and remove the transition label `urg!`. In the resulting

model local variable `ConflictNum` is no longer used and so we can abstract it away.
2. We may weaken the guard of the left `receive_msg[j]?` transition in automaton `InputHandler(j)` to `true`. In the resulting model local clock `y` is no longer used and it can be abstracted away.
3. Once a host starts to use the claimed IP address (after the first announcement), the remaining announcements can be seen as ARP packets sent by automaton `Regular`.

The basic idea behind abstractions (1) and (2) is that Zeroconf ensures mutual exclusion even when a host is allowed to always immediately select a new IP address after a reset, and may always defend the IP address that it is using.

Verification Results Using the combination of the above abstractions, we were able to prove mutual exclusion for instances of Zeroconf with 2 hosts and up to 5 IP addresses, and an instance with 3 hosts and 1 IP address.

We also did some experiments with the use of symmetry reduction for IP addresses. Since in Zeroconf the IP address 0 (i.e., 0.0.0.0) plays a special role, and UPPAAL can only handle fully symmetric data types, this required some rewriting of the model. Using symmetry reduction for IP addresses, we were able to establish mutual exclusion for a system with 2 hosts and an *arbitrary* number of IP addresses. Essentially, this is due to a theorem of Ip and Dill [19] on *data saturation*. This theorem (which was proved in the setting of Murphi but can easily be shown to carry over to UPPAAL) states that for certain ("data") scalarsets, the state graph does not grow any further once the size of the scalarsets grows beyond the number of scalarset locations in the system. In the case of 2 hosts, the number of scalarset locations for IP addresses in the model equals 12 (1 for each `Config[j]` automaton, 4 for each `Network` automaton, and 2 for the `packet` variable). In fact, data saturation already happens starting from scalarsets of size 5.

Actually, we conjecture that there exists a bisimulation between a model with n IP addresses, for any n, and the model with just one (nonzero) IP address, via which a proof of ME for the general model can be reduced to a proof of ME for the model with just one address.

4 Conclusions

Our goal has been to construct a model of Zeroconf that (a) is easy to understand by engineers, (b) comes as close as possible to RFC 3927, and (c) may serve as a basis for formal verification. Did we succeed?

Understandability Of course, it is not to us to judge whether our model is understandable for others. The present paper aims to place the cards on the table as a basis for a discussion. The UPPAAL syntax, which combines extended finite state machines, C-like syntax and concepts from timed automata, will certainly

be familiar to protocol engineers, except maybe for the use of clock variables. However, our experience is that timed automata notation is easy to explain, also to people without expertise in theoretical computer science. Clocks provide a simple and intuitive means to specify the various timing constraints in Zeroconf. The automata `Config` and `InputHandler` would be the obvious candidates for inclusion in a standard. The only elements in these automata which may be considered less intuitive are the use of committed locations in the `InputHandler` and the sending of a `no_reply` signal in situations where no reply packet is sent (this is an artifact of the model since in reality there is no such signal). However, we can easily remove these elements from the `InputHandler` automaton at the price of making the `Network` automata (considerably) more complicated.

There are a number of extensions of the Uppaal syntax that would help us to further improve the readability of our model:

– A richer syntax for datatypes, for instance permitting us to write 0.0.0.0 for the all zero IP address instead of 0.
– The ability to initialize clock variables, allowing us to eliminate the initial transition in the `InputHandler[j]` automaton.
– The ability to test the value of clocks within the body of functions, allowing us to move the test `y>DEFEND_interval` into the definition of `ihandler`, where it belongs conceptually.
– The introduction of urgent transitions in Uppaal, as advocated in [16]. This would allow us to eliminate the urgent channel `urg`, which is a modeling trick that is hard to explain to non-specialists. Also, it would allow us to replace the invariant `counter < ANNOUNCE_NUM imply x <= ANNOUNCE_INTERVAL` in automaton `Config` by an urgency predicate `x <= ANNOUNCE_INTERVAL`. In our opinion urgency predicates are more intuitive than location invariants.

Once these extensions have been implemented, a good case can be made for inclusion of the `Config` and `InputHandler` automata (with the `ihandler` code) in a Zeroconf standard. These models definitely help to clarify the RFC and to prevent incorrect interpretations due to ambiguity in the textual part. The Uppaal simulator is also very useful to obtain insight in the operation of the protocol.

Our modeling efforts revealed five places where RFC 3927 [9] is incomplete/unclear:

1. It does not specify upper and lower bounds on the time that may elapse between sending the last ARP Probe and sending the first ARP Announcement.
2. It does not specify whether a host may immediately start using a newly claimed address or whether it should first send out all ARP Announcements.
3. It does not specify the tolerance that is permitted on the timing of ARP Announcements.
4. Although it states that Zeroconf requires an underlying network that supports ARP (RFC 826), we identified some cases where Zeroconf does not conform to RFC 826.
5. It is not exactly clear in which situations a host may defend its address.

Faithfulness and Traceability We have shown that UPPAAL is able to model Zeroconf faithfully. Basically, for each transition in the model we can point towards a corresponding piece of text in the RFC. The relationships between our model and the RFC have been described in great detail in this paper, including the design choices and abstractions that we made. Following [6], our aim has been to make the model construction *transparent*, so that our model may be more easily understood and checked by others, making its quality measurable in (at least) an informal sense.

We see at least three ways in which UPPAAL can be improved to allow for even more faithful/realistic modeling of Zeroconf and better traceability:

- Zeroconf involves a number of probabilistic aspects that are not incorporated in our UPPAAL model. An extension with probabilities, along the lines of PRISM [21], is clearly desirable.
- UPPAAL supports modeling of systems that are described as networks of a *fixed* number of automata with a *fixed* communication structure. This modeling approach, although very convenient as a starting point for verification, does not fit very well with the highly dynamic structure of Zeroconf networks where hosts may join and leave, subnetworks may be joined, etc.
- To support traceability it would help to add a feature to UPPAAL by which comments are displayed when a user clicks on (or points at) a transition.

The first two items require a major research effort, whereas the last item should be easy to implement.

Complexity and Tractability The formal model of Zeroconf that we presented in Section 2 cannot be analyzed by UPPAAL for interesting instances with 3 or more hosts. We presented a simple manual proof of mutual exclusion for the model that we considered in this paper (no message loss, host failure and merging of networks). In order to verify a system with 3 hosts, we had to apply some drastic abstractions. We have argued informally that these abstractions are sound.

A challenging question for us is to come up with (automatically generated) additional abstractions that allow for the automated analysis of larger instances of the protocol. One possibility here would be to try to apply the technique of counterexample guided abstraction refinement [14, 13]. A basic idea in the design of Zeroconf is that it does not harm to send additional ARP messages; they have only been added because they may help to ensure (or restore) mutual exclusion in the case of faults. Thus far, we have not been able to come up with abstractions that capture this idea.

In our view, it is highly desirable to extend UPPAAL with (semi-)automatic support for proving correctness of abstractions. Only abstractions can bridge the gap between realistic and tractable models.

Future Work In this paper, we have only modelled/analyzed a few simple instances of a part of Zeroconf in a restrictive setting without faulty nodes, merging of subnetworks, etc. So clearly, there are many directions in which our modeling

effort can be extended. The timing behavior of Zeroconf becomes really interesting when studied within a setting in which also the probabilistic behavior is modelled. The performance analysis of Zeroconf reported in [5, 20] has been carried out for an abstract probabilistic model of Zeroconf. A challenging question is whether these results also hold for a (probabilistic extension) of our more realistic model.

Acknowledgments We thank Peter van der Stok (Philips Research) for suggesting the problem to us, Stuart Cheshire (Apple Computer, Inc.) and Boris Cobbelens (Free University, Amsterdam) for answering all our questions about Zeroconf. Martijn Hendriks, Jasper Berendsen, Jozef Hooman and the students of the Analysis of Embedded Systems course in Nijmegen commented on earlier versions and came with modeling suggestions. Martijn also helped with Uppaal and noted the occurrence of data saturation. Guy Leduc, Hubert Garavel, Judi Romijn and Ken Turner commented on the use of formal description languages within protocol standards.

References

1. M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.
2. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
3. G. Behrmann, A. David, and K.G. Larsen. A tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM-RT 2004),* Bertinoro, Italy, September 13-18*, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.
4. G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.
5. H. Bohnenkamp, P. van der Stok, H. Hermanns, and F.W. Vaandrager. Cost-optimisation of the IPv4 zeroconf protocol. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2003)*, pages 531–540, Los Alamitos, California, 2003. IEEE Computer Society.
6. E. Brinksma and A. Mader. On verification modelling of embedded systems. Technical Report TR-CTIT-04-03, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, January 2004.
7. G. Bruns and M.G. Staskauskas. Applying formal methods to a protocol standard and its implementations. In *Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 1998),* 20-21 April, 1998, Kyoto, Japan, pages 198–205. IEEE Computer Society, 1998.
8. S. Cheshire. Personal communication, February 2006.
9. S. Cheshire, B. Aboba, and E. Guttman. Dynamic configuration of IPv4 link-local addresses (RFC 3927), May 2005. `http://www.ietf.org/rfc/rfc3927.txt`.
10. S. Cheshire and D.H. Steinberg. *Zero Configuration Networking: The Definite Guide*. O'Reilly Media, Inc., 2005.

11. D. Chkliaev, J. Hooman, and E. de Vink. Verification and improvement of the sliding window protocol. In *Proceedings TACAS'03*, pages 113–127. Lecture Notes in Computer Science 2619, Springer-Verlag, 2003.
12. E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proc. CHDL*, pages 15–30, 1993.
13. E.M. Clarke, A. Fehnker, Z. Han, B.H. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *Int. J. Found. Comput. Sci.*, 14(4):583–604, 2003.
14. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.
15. M.C.A. Devillers, W.O.D. Griffioen, J.M.T Romijn, and F.W. Vaandrager. Verification of a leader election protocol: Formal methods applied to IEEE 1394. *Formal Methods in System Design*, 16(3):307–320, June 2000.
16. B. Gebremichael and F.W. Vaandrager. Specifying urgency in timed I/O automata. In *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005),* Koblenz, Germany, September 5-9, 2005, pages 64–73. IEEE Computer Society, 2005.
17. M. Hendriks, G. Behrmann, K.G. Larsen, P. Niebert, and F.W. Vaandrager. Adding symmetry reduction to uppaal. In Kim Guldstrand Larsen and Peter Niebert, editors, *FORMATS*, volume 2791 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2003.
18. G.J. Holzmann. *The Spin model checker: primer and reference manual*. Addison-Wesley, 2003.
19. C.N. Ip and D.L. Dill. Better verification through symmetry. In David Agnew, Luc J. M. Claesen, and Raul Camposano, editors, *Computer Hardware Description Languages and their Applications, Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications - CHDL '93, sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC, Ottawa, Ontario, Canada, 26-28 April, 1993*, volume A-32 of *IFIP Transactions*, pages 97–111. North-Holland, 1993.
20. M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. Performance analysis of probabilistic timed automata using digital clocks. In K. Larsen and P. Niebert, editors, *Proc. Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, volume 2791 of *LNCS*, pages 105–120. Springer-Verlag, 2003.
21. M.Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 2.0: A tool for probabilistic model checking. In *Proceedings of the 1st International Conference on Quantitative Evaluation of Systems (QEST04)*, pages 322–323. IEEE Computer Society, 2004.
22. I. van Langevelde, J.M.T. Romijn, and N. Goga. Founding FireWire bridges through Promela prototyping. In 8th *International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA)*. IEEE Computer Society Press, April 2003.
23. K.G. Larsen, M. Mikucionis, and B. Nielsen. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *the 5th ACM International Conference on Embedded Software*, pages 299 – 306. ACM Press New York, NY, USA, September 18–22 2005.
24. N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Fransisco, California, 1996.

25. D.C. Plummer. An Ethernet address resolution protocol (RFC 826), November 1982. `http://www.ietf.org/rfc/rfc826.txt`.
26. J.M.T. Romijn. Improving the quality of protocol standards: Correcting IEEE 1394.1 FireWire net update. *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*, 8:23–30, 2004. Available at `http://www.win.tue.nl/oas/index.html?iqps/`.
27. M. Stoelinga. Fun with FireWire: A comparative study of formal verification methods applied to the IEEE 1394 root contention protocol. *Formal Aspects of Computing Journal*, 14(3):328–337, 2003.
28. F.W. Vaandrager and A.L. de Groot. Analysis of a biphase mark protocol with Uppaal and PVS. *Formal Aspects of Computing Journal*, 2006. To appear. Also available as Technical Report NIII-R0445, NIII, Radboud University Nijmegen, 2004.
29. K. Yorav. *Exploiting Syntactic Structure for Automatic Verification*. PhD thesis, The Technion, Israel Insitute of Technology, June 2000.