Formal Specification and Analysis of Zeroconf Using Uppaal

JASPER BERENDSEN, BINIAM GEBREMICHAEL and FRITS W. VAANDRAGER
Radboud University Nijmegen, The Netherlands
and
MIAOMIAO ZHANG
Tongji University, China

The model checker Uppaal is used to formally model and analyze 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 modeling efforts revealed several errors (or at least ambiguities) in the RFC that no one else spotted before. We present two proofs of the mutual exclusion property for Zeroconf (for an arbitrary number of hosts and IP addresses): a manual, operational proof, and a proof that combines model checking with the application of a new abstraction relation that is compositional with respect to committed locations. The model checking problem has been solved using Uppaal and the abstractions have been checked by hand.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution and Maintenance—*Documentation*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol verification*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; *Model checking*; *Validation*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs*; *Mechanical verification*; *Specification techniques*; D.2.1 [**Software Engineering**]: Requirements/Specifications—; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata (e.g., finite, push-down, resource-bounded)*; *Relations between models*

General Terms: Algorithms, Standardization, Verification

Additional Key Words and Phrases: Compositional reasoning, Compositional abstraction, modelling, Industrial case study, Simulation relation, Timed automata, Uppaal, Zeroconf protocol

Research 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), the DFG/NWO bilateral cooperation project Validation of Stochastic Systems (VOSS2), NWO/GBE project 612.000.103 Fault-tolerant Real-time Algorithms Analyzed Incrementally (FRAAI), and the European Community's Seventh Framework Programme under grant agreement no 214755 (QUASIMODO). A preliminary version of this paper appeared as Gebremichael et al. [2006]. All the Uppaal models described in this paper are available on-line at `http://www.cs.ru.nl/ita/publications/papers/fvaan/zeroconf/full.html`.
Corresponding author's address: F. Vaandrager, Institute for Computing and Information Sciences, Radboud University Nijmegen, Heijendaalseweg 135, 6500 GL Nijmegen, The Netherlands.

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), FireWire/iLink (IEEE 1394), HAVi, WAP, CAN 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 implementers from many different companies, with 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. Clarke et al. [1993], Bruns and Staskauskas [1998], Devillers et al. [2000], Langevelde et al. [2003], Stoelinga [2003], Holzmann [2004], Chkliaev et al. [2003], and Vaandrager and Groot [2006]). In most cases, these studies were carried out after completion of the standard, and involved guessing to fill in holes and resolve ambiguities. An exception is the work by Romijn [2004], 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.

In order to avoid holes and ambiguities in standards, the obvious way to go is to describe critical parts using 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. However — to the best of our knowledge — these languages have thus far not been used in the authoritative part of protocol standards. Some protocol standard have extended finite state machines (EFSMs) inside, but these are mostly illustrative, not completely formal, and sometimes contain mistakes.[1] Bruns and Staskauskas [1998] used (a well-defined subset of) C to describe the SONET Automatic Protection Switching (APS) protocol and report

[1]See, for instance, `ttp://www.inrialpes.fr/vasy/Press/firewire.tml`.

that developers found their C description easy to understand and superior to that which appeared in the APS standard. However, the lack of abstraction mechanisms is an obvious drawback of C.

The relationships between an (abstract) formal model of a protocol and the corresponding informal standard are typically obscure. As pointed out in Brinksma and Mader [2004],

> "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."

As a step towards the development of a systematic method, we report in this paper on the systematic construction of a verification model of a recent protocol standard. More specifically, we describe the use of Uppaal to model and analyze critical parts of Zeroconf, a protocol for dynamic configuration of IPv4 link-local addresses. 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 is a corresponding piece of text in the standard), and (c) may serve as a basis for formal verification.

Uppaal [Behrmann et al. 2004; Behrmann et al. 2006] is an integrated tool environment for specification, validation and verification of real time systems modeled as networks of timed automata [Alur and Dill 1994]. The tool is available for free for non-profit applications at `www.uppaal.com`. The language for the new version Uppaal 4.0 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. Due to these extensions, Uppaal is able to support modeling and analysis of critical parts of protocol specifications:

(1) The graphical syntax for EFSMs and 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.

Zeroconf [Cheshire and Steinberg 2005] is a protocol for dynamic configuration of IPv4 link-local addresses that has been defined by the IETF Network Working Group in RFC 3927 [Cheshire et al. 2005]. 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 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[2], but several other implementations are available.[3]

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 present two proofs of the mutual exclusion property for Zeroconf for an arbitrary number of hosts and IP addresses: a manual, operational proof, and a proof that combines model checking with the application of a new abstraction relation that is compositional with respect to committed locations. The model checking problem has been solved using Uppaal and the abstractions have been checked by hand.

Related Work. Zeroconf 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 Bohnenkamp et al. [2003] and Kwiatkowska et al. [2003]. The primary goal of Bohnenkamp et al. [2003] was to investigate the trade off between reliability and effectiveness of the protocol using a stochastic cost model. The model of Bohnenkamp et al. [2003], 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 than the RFC. Based on an earlier version of the present paper, a more detailed model has been presented in Kwiatkowska et al. [2003] using the probabilistic model checker PRISM [Kwiatkowska et al. 2004]. The model checking results reported in Kwiatkowska et al. [2003] are quite interesting, but the precise relationship between the model and the RFC is unclear (for instance, in the model of Kwiatkowska et al. [2003] 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 (just a few datatypes, no support of C-like syntax,..). A toolset that combines the functionality of Uppaal and PRISM would be ideal for dealing with the Zeroconf protocol. The compositional step simulation relations between timed transition systems that we use to establish the correctness of our abstractions are inspired by the timed ready simulations from Jensen et al. [2000], and use the framework described by Berendsen

[2]See http://developer.apple.com/networking/bonjour/.
[3]See http://en.wikipedia.org/wiki/Zeroconf.

and Vaandrager [2008].

Outline. The organization of the paper is as follows. In Section 2, we explain the protocol and our Uppaal model. Section 3 presents a manual correctness proof of the protocol. Section 4 shows how arbitrary instances of our model can be analyzed fully automatically after applying a series of abstractions. Finally, Section 5 presents some conclusions and directions for future research.

2. MODELING THE ZEROCONF PROTOCOL

In this section, we describe our Uppaal model of the Zeroconf protocol, and the relationship between this model and RFC 3927 [Cheshire et al. 2005], the official protocol standard.

A Zeroconf network is composed of a set of hosts on the same link. Hosts in the network can be devices that are present at home, office, embedded systems "plugged together" as in an automobile, or the laptops of some colleagues 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 Zeroconf is the dynamic configuration of IPv4 link-local addresses, and this is the part on which we 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, separated 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 will 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 two hosts use the same IP address. The question whether (or under which circumstances) this goal is actually achieved calls for verification and cannot be resolved by direct inspection and easy arguments. The underlying algorithm used in Zeroconf is similar to Fisher's mutual exclusion algorithm [Abadi and Lamport 1994; Lynch 1996] 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.

2.1 Fixing the Set of Hosts

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 positive number of 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. A phenomenon that may occur in practice, but which we have not modeled here, is that distinct Zeroconf networks are joined. We also do not model host failure or termination (although it would be easy to add this).

The behavior of a single host is modeled by three timed automata that run concurrently: `Config`, `InputHandler` and `Regular`. Automaton `Config` describes the configuration of a new IP address, `InputHandler` takes care of the incoming messages, and `Regular` abstractly models the activity of all the other processes running on the host. The three automata are parametrized by the unique hardware address (HA) of the host they belong to. We introduce a *scalar type* to represent the set of all `l` hardware addresses of hosts in the system:

```
typedef scalar[l] HAtype;
```

In Uppaal, the type `scalar[l]` denotes the set $\{0, \ldots, l-1\}$. On scalar types only a few operations are permitted: assignment of the value of one variable to another, and identity testing. As a consequence, scalar types are unordered and fully symmetric: the behavior of a model is invariant under arbitrary permutations of the elements of a scalar type [Ip and Dill 1993; Hendriks et al. 2004]. By using a scalar type rather than a subrange type, we specify that within our model all the HAs (and therefore all the hosts) play a fully symmetric role. This enables the use of symmetry reduction during exploration of the state space.

2.2 The Underlying Network

We assume the presence of an underlying network via which nodes may communicate. RFC 3927 states the following assumption about this 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) [Plummer 1982] is a widely used method for converting protocol addresses (e.g., IP addresses) to local network ("hardware") addresses (e.g., Ethernet addresses). It takes care of dynamic distribution of the information needed to build tables to translate protocol addresses to hardware addresses. Within Zeroconf, all messages are ARP packets.

The goal of Zeroconf is to configure a *link-local* IP address. Altogether there are $2^{16} - 2 \times 256 = 65024$ link-local addresses:

> "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."

The total number of link-local addresses occurs as a parameter `m` in our model. The only IP addresses used by Zeroconf are link-local addresses and the all zeroes IP address 0.0.0.0, which serves as a special 'unknown' or 'undefined' value in the protocol. We represent the set of used IP addresses by a scalar type:

```
typedef scalar[m+1] IPtype;
```

Actually, because the IP address 0.0.0.0 plays a special rôle, the set of IP addresses is not fully symmetric. We use a trick to denote the all zeroes IP address: we introduce a special state variable `zero` of type `IPtype`, whose value is never changed, and define this value to be the all zeroes IP address. In this way, we do not refer directly to an element of the scalar type. But since variable `zero` is never changed, it acts as a constant and thus, effectively, it refers to a fixed element of the scalar type.

For our model, the relevant[4] 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 packet type, which can be either "request" or "reply". Hence, an ARP packet can be defined in Uppaal 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.

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."

A host that is looking for the hardware address of a host with IP address `x`, broadcasts an ARP request packet with the target IP address set to `x`. A host with IP address `x` will then return an ARP reply packet with the sender hardware address set to its local network address.

We model the network as a set of `n` identical `Network` automata. Each of these automata takes care of handling a single ARP request at a time, and is parametrized by an element of the scalar type:

```
typedef scalar[n] Networktype;
```

The main reason for having `n` automata rather than just one, is that this allows us to model round-trip latencies in Uppaal: each network automaton has its own clock to keep track of timing. Figure 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 an answer (this may be a reply or a request packet), this is transferred from the host to the network automaton using a `send_answer` action, and broadcast to all the hosts via subsequent `receive_msg` actions. All these interactions take place within 1 second. After completing this task, a `Network` automaton returns to its initial location, ready to handle a new request.

To simplify our model, we assume that hosts handle incoming ARP requests in zero time, that is, we adopt the synchrony hypothesis that is well-known from synchronous programming [Berry and Gonthier 1992]. A desktop computer can

[4]ARP packets also contain a target hardware address, but this can be ignored in our model since Zeroconf uses broadcast for all messages.

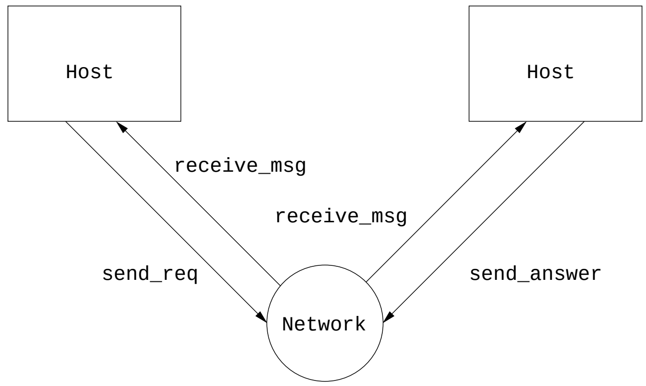

Fig. 1. Interaction between `Network` automaton and hosts.

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 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.

2.3 Address Configuration

For each host, we introduce a state variable `IP` to store the IP address of that host:

```
IPtype IP[HAtype];
```

Figure 2 displays the automaton `Config(h)`, which specifies how host `h` configures a new IP address. The host starts in location `INIT`, where it stays 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."

A transition from location `INIT` to location `WAIT` takes place when an address has been selected. Via a so-called select statement `guess:IPtype`, we nondeterministically bind identifier `guess` to a value of type `IPtype`. This means that there is an instance of the transition for each element of the type. The transition is enabled if a value different from `zero` has been selected, that is, a link-local address. In this way we express that a link-local IP address is chosen nondeterministically. The selected address is stored in state variable `IP[h]`. To mark the time at which the address has been selected, we reset a local clock `x`.

The RFC continues [page 11, section 2.2.1]:

> "When ready to begin probing, the host should then wait for a random time interval selected uniformly in the range zero to PROBE_WAIT

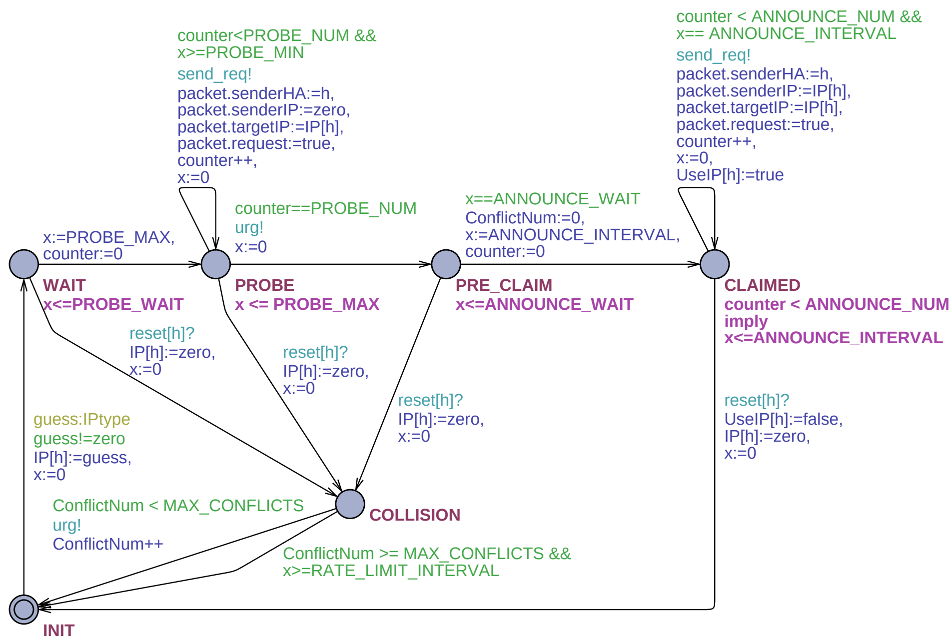

Fig. 2. Automaton `Config(h)`.

> seconds, and should then send PROBE_NUM probe packets, each of these probe packets spaced randomly, PROBE_MIN to PROBE_MAX seconds apart."

The use of the word "should" in the above sentence is somewhat ambiguous. In our model, we assume that it has the same meaning as the keyword "MUST" as defined in RFC 2119, that is, the definition is an absolute requirement of the specification. In Section 3, we will discuss the alternative interpretation in which "should" has the same meaning as "SHOULD" in the sense of RFC 2119. This keyword means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course. We will see that the protocol may fail in case no probes are sent at all.

The initial waiting period is modeled by by bounding the time that the host may stay in `WAIT` via an invariant `x <= PROBE_WAIT`. At any point the host may move to location `PROBE`, where it starts sending probes. Probes are defined 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 an action `send_req!` that synchronizes with a matching action `send_req?` of the network. The packet is communicated via a global shared variable `packet` of type `ARP_packet`. In Uppaal, assignments in an output (!) transition are executed before assignments in a synchronizing input (?) transition, and this allows us to assign a value to `packet` in a `send_req!` transition, which is then picked up by a matching `send_req?` transition of the network. Lower and upper bounds on timing are expressed 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 when location `PROBE` is entered. A local variable `counter` is used to record the number of probes that have been sent. After the probing phase is completed, the automaton immediately jumps to location `PRE_CLAIM`. The urgent broadcast channel `urg` ensures that this transition is taken as soon as it is enabled, that is, immediately after sending the last probe. As the reader can check, the translation from the RFC description of the probing phase to our model 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 `CLAIMED` indicates that the host has successfully claimed an address.

In our model, automaton `InputHandler(h)` (which will be discussed in Section 2.4) takes care of handling incoming messages. If `InputHandler(h)` decides that, due to some conflict, a new address must be configured, it performs an action `reset[h]!`. This triggers a `reset[h]?` transition in `Config(h)`. As part of this transition, `IP[h]` is set to `zero` and clock `x` is reset. 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."

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 model is straightforward.

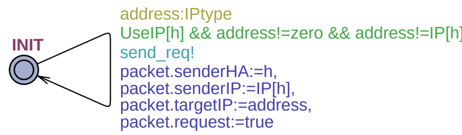

Fig. 3. Automation `Regular(h)`.

In location `CLAIMED` 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` [Cheshire 2006]. 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 [Cheshire 2006]. 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 [Cheshire 2006]. We decided to follow the RFC and not specify accuracy requirements, but in order to use our model for automatic generation of tests, for instance using the Uppaal-Tron toolset [Larsen et al. 2005], one would have to modify our model at this point.

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

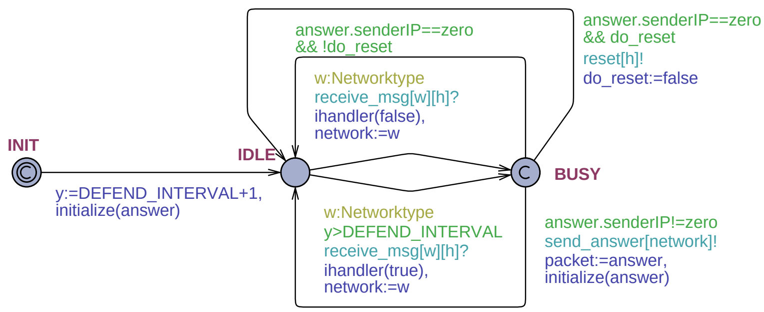

Fig. 4. Automaton `InputHandler(h)`.

2.4 The Input Handler

For each host `h`, automaton `InputHandler(h)` 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. Conceptually, we think it is natural to describe input handling in terms of a single component or function. Implementations of the protocol will typically also do this.

Automaton `InputHandler(h)` is displayed in Figure 4. The automaton starts with a transition to initialize its local variables: clock `y` is set to a large value, and packet variable `answer` is set to the undefined value. When a new packet arrives, that is, when a `receive_msg[w][h]?` transition occurs, the automaton calls a function `ihandler`, which does the real work. The definition of `ihandler` is listed in Figure 5. The Boolean parameter `defend` indicates whether the host will defend its IP address in case of a conflicting ARP request. The host may only defend its address if there has been no other conflict during the last `DEFEND_INTERVAL` time units. Clock `y` measures the time since the last conflict. The input handler must distinguish between 9 scenarios:

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

Scenario B. Incoming packets sent by the host 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[h]`. This scenario is described in the RFC as follows [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

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

Fig. 5. Function `ihandler`.

> 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[h]==false`. The case with `UseIP[h]==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 ongoing 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 im-

> mediately 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 may occur when the topmost `receive_msg` transition in the automaton has been taken, which sets `defend` to `false`, Case (b) corresponds to scenario D. This scenario may occur when the lower `receive_msg` transition in the automaton has been 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 above quotation 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* address. Thus one could also argue that in this situation a host may defend its address. To model this interpretation, one has 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."

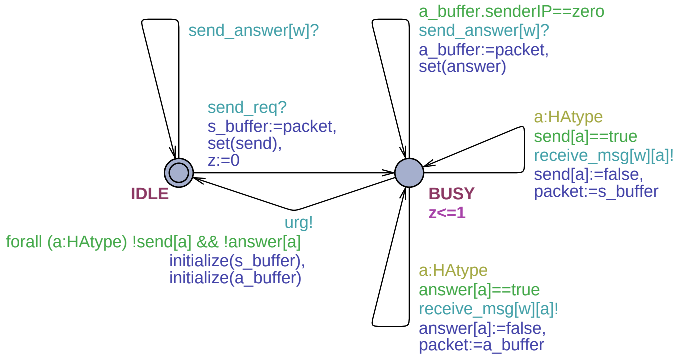

Fig. 6. Automaton `Network(w)`.

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) [Plummer 1982] 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 sender fields contain the local hardware address and local IP address, and the target field contains the value of the sender field of the received packet. Zeroconf (RFC 3927) is not explicit about conformance to RFC 826 (it assumes a link-layer technology that "supports ARP"), 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.

2.5 The Network Automaton

As explained in Section 2.2, we model the underlying network as a set of `n` identical `Network` automata. For index `w`, the automaton `Network(w)` is shown in Figure 6. Initially the automaton is in its `IDLE` location. As soon as it receives a packet via a `send_req?` transition, it jumps to location `BUSY`. A local clock `z` is set to zero and an invariant `z <= 1` ensures that within 1 second the network broadcasts the packet (as well as the answer if there is one) to all hosts. We assume no lower bound on message delivery time, but we do assume that there is at most one host that answers any given request, and that an answer does not induce subsequent answers. It is possible to model multiple and successive anwers, but this will require additional state variables and more complicated data structures.

Our `Network` automaton maintains two local variables for storing packets: `s_buffer` holds the packet that was sent by the host and `a_buffer` holds an answer to a request when it arrives. In addition, `Network` maintains Boolean arrays `send` and `answer` to record to which hosts packets still need to be delivered. The function `set` is used to set all entries of a Boolean array to `true`. Via a select statement on the `receive_msg[w][a]!` transitions, the automaton nondeterministically selects in which order packets are delivered to the different hosts. The upper transition labeled with `send_answer[w]?` occurs when a host returns an answer upon receipt of a request, as explained in Subsection 2.4. The lower transition labeled with `receive_msg[w][a]!` is enabled as soon as there is an answer packet in `answer` buffer. The network returns to its `IDLE` location and resets the buffers to their initial value, as soon as all messages have been delivered.

2.6 Dimensioning the Model

The RFC [page 25, section 9] specifies the following values for the different timing constants. These definitions are copied verbatim in the declaration section of our Uppaal 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 [Cheshire et al. 2005]. These values are too big for automatic verification: with 3 hardware addresses and 65024 IP addresses even the 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 packets 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 its 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 small number `n`.

3. MANUAL VERIFICATION

The RFC does not specify what properties the protocol must satisfy. However, it is clear that at least the following two correctness properties are desirable:[5]

(1) Mutual exclusion, that is, two hosts may not use the 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
```

[5]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 mutual exclusion will be restored within a specified amount of time, provided no further joins occur.

(2) Absence of deadlock, that is, in each reachable state a transition is possible. In Uppaal syntax:

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

The model described in the previous section is very close to the RFC definition of the protocol, but too big for Uppaal to do a complete state space exploration for nontrivial instances without some drastic abstractions. Using the latest version of Uppaal (4.0), we only managed to establish `ME` and `DL` for the instance with 2 hardware addresses, 1 link-local IP address and 2 network automata. Nevertheless, it is not too hard to see that Zeroconf satisfies mutual exclusion and absence of deadlock. In the remainder of this section, we sketch a manual proof of mutual exclusion. We claim that our model has no deadlocks but do not present the (long and tedious) proof here.

Theorem 3.1. *For each instance of the Zeroconf model (i.e., any number of hardware addresses, IP addresses and network automata), the mutual exclusion property* `ME` *holds.*

Proof. (Sketch) Suppose that `i` and `j` are distinct hardware addresses and suppose that in some reachable state s, $\texttt{UseIP}[\texttt{i}] \wedge \texttt{UseIP}[\texttt{j}] \wedge (\texttt{IP}[\texttt{i}] = \texttt{IP}[\texttt{j}])$. We derive a contradiction. Consider an execution α leading up to state s, that is, a finite sequence of delay and action transitions in the timed transition system semantics of the model leading from the start state to s. Observe that before a host enters the "critical section" (where it may use its IP address) it resides at least

```
PROBE_MIN + PROBE_MIN + ANNOUNCE_WAIT = 1 + 1 + 2 = 4
```

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

```
Config(i).WAIT || Config(i).PROBE || Config(i).PRE_CLAIM ||
    (Config(i).CLAIMED && !UseIP[i])
```

and the critical section is defined by

```
UseIP[i]
```

Moreover, exactly `ANNOUNCE_WAIT=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`. Without loss of generality, assume that host `j` enters the critical section before (but possibly at the same time as) `i`. Then

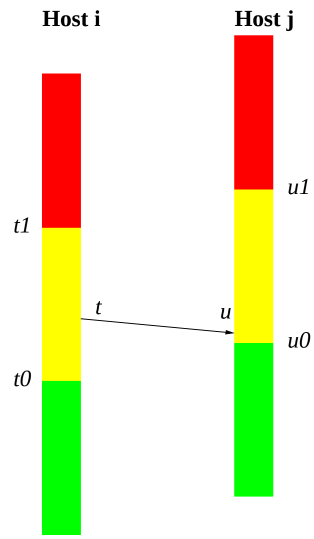

Fig. 7. Probe arrives at j before it enters critical section.

we have the following (in)equalities:

$$\begin{aligned} t0 &\geq u0 \\ t0 - t1 &\geq 4 \\ u0 - u1 &\geq 4 \\ t &= t0 - 2 \\ u &\geq t \\ u &\leq t + 1. \end{aligned}$$

We consider two cases:

(1) See Figure 7. The probe arrives at host `j` before `j` enters the critical section. At this moment, `j` must be in its trying region since:

$$u \geq t = t0 - 2 \geq u0 - 2 > u0 - 4 \geq u1.$$

But this means that host `j`'s input handler, upon receipt of the conflicting probe, will generate a reset (Scenario F) and immediately drive `Config(j)` back to its initial state, i.e, out of the trying region. Contradiction.

(2) See Figure 8. The probe arrives at host `j` after `j` has entered 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

$$t0 = t + 2 > t + 1 \geq t' \geq t = t0 - 2 > t0 - 4 \geq t1.$$

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

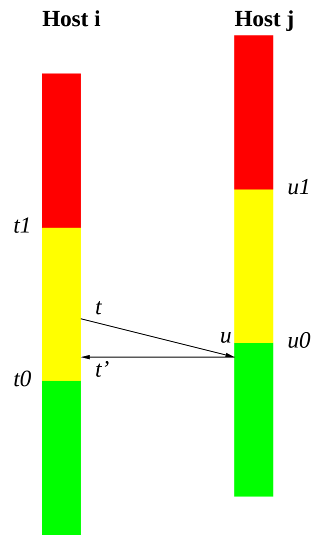

Fig. 8. Probe arrives at `j` after it enters critical section.

region. Contradiction. QED

□

We expect that formalization/mechanization of the proof of Theorem 3.1, for instance in PVS using the basic setup of Vaandrager and Groot [2006], will be routine although it will involve a significant amount of work.

Inspection of the proof indicates that Zeroconf is extremely 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 `DL` in a few seconds on a standard PC.

4. VERIFICATION BY MODEL CHECKING AND ABSTRACTION

Next to the operational proof of mutual exclusion (Theorem 3.1) described in the previous section, we also would like to have a proof that is obtained in a more automatic and structured way. Model checking is of course such an automatic way, but it suffers from state space explosion. Moreover, model checking usually can only verify a single instance of a protocol, whereas one would like to establish correctness for all (possibly infinitely many) instantiations of its parameters (`l`, `m`). We will show that abstractions are a remedy to both problems.

We use an abstraction relation that is *sound* for the property to be verified, meaning that when the property holds in the (simple) abstract model, then it also holds in the (more complex) concrete model. We will apply abstractions in a compositional way, which means that in a parallel composition of a set of automata, a subset is replaced by an abstraction, thereby obtaining a new full model that in turn is an abstraction of the original full model.

Section 4.1 introduces our compositional abstraction framework. Section 4.2 establishes soundness of an abstraction that only uses two hosts. Section 4.3 derives an even more abstract system that can effectively be verified by Uppaal. Section 4.4, finally, presents our model checking results.

4.1 Compositional Abstraction

The standard operational semantics of a Uppaal model is defined on the model as a whole, see Behrmann et al. [2004] or the Uppaal help menu. For compositional verification we use the approach described in Berendsen and Vaandrager [2008], in which a timed transition systems (TTSs) is associated to each individual timed automaton. TTSs can be composed in parallel and a compositional abstraction relation is defined that is sound for invariant properties.

Basically, TTSs are labeled transition systems equipped with some additional structure to support shared variables and committed transitions: states are defined as valuations of variables, and transitions may be committed, which gives them priority in a parallel composition. TTSs can be composed in parallel and may communicate by means of shared variables and synchronization of actions. Like in CCS [Milner 1989], two transitions may synchronize when their actions are complementary, leading to an internal transition in the composition.

Below we write $\mathbb{R}_{\geq 0}$ for the set of nonnegative real numbers, $\mathbb{N}$ for the set of natural numbers, and $\mathbb{B} = \{1, 0\}$ for the set of Booleans. We let d range over $\mathbb{R}_{\geq 0}$, i, j, k, n over $\mathbb{N}$, and $b, b', \ldots$ over $\mathbb{B}$.

We consider three different types of state transitions, corresponding to three different types of actions. We assume a set $\mathcal{C}$ of *channels* and let c range over $\mathcal{C}$. The set of *external actions* is defined as $\mathcal{E} \triangleq \{c!, c? \mid c \in \mathcal{C}\}$. Actions of the form $c!$ are called *output actions* and actions of the form $c?$ *input actions*. We assume the existence of a special *internal action* τ, and write $\mathcal{E}_\tau$ for $\mathcal{E} \cup \{\tau\}$, the set of *discrete actions*. Finally, we assume a set of *durations* or *time-passage actions*, which in this paper is just $\mathbb{R}_{\geq 0}$. We write Act for $\mathcal{E}_\tau \cup \mathbb{R}_{\geq 0}$, the set of *actions*.

TTSs are capable of communication over a universal set $\mathcal{V}$ of typed *variables*, with a subset $\mathcal{X} \subseteq \mathcal{V}$ of *clock variables* or *clocks*. Clocks have domain $\mathbb{R}_{\geq 0}$. A *valuation* for a set $V \subseteq \mathcal{V}$ is a function that maps each variable in V to an element in its domain. We let $u, v, w, \ldots$ range over valuations, and write $Val(V)$ for the set of valuations for V. For valuation $v \in Val(V)$ and duration $d \in \mathbb{R}_{\geq 0}$, we define $v \oplus d$ to be the valuation for V that increments clock variables by d, and leaves the other variables untouched, that is, for all $y \in V$,

$$(v \oplus d)(y) \triangleq \begin{cases} v(y) + d & \text{if } y \in \mathcal{X} \\ v(y) & \text{otherwise.} \end{cases}$$

We write $dom(f)$ to denote the domain of a function f (in our case a valuation). For functions f and g, we let $f \rhd g$ denote the combined function where f overrides g for all elements in the intersection of their domains. Formally, $f \rhd g$ is the function with $dom(f \rhd g) = dom(f) \cup dom(g)$ satisfying, for all $z \in dom(f \rhd g)$,

$$(f \rhd g)(z) \triangleq \begin{cases} f(z) & \text{if } z \in dom(f) \\ g(z) & \text{if } z \in dom(g) - dom(f). \end{cases}$$

We define the dual operator by $f \lhd g \triangleq g \rhd f$. Two functions f and g are *compatible*, notation $f \heartsuit g$, if they agree on the intersection of their domains, that is, $f(z) = g(z)$ for all $z \in dom(f) \cap dom(g)$. For compatible functions f and g, we define $f \| g \triangleq f \rhd g$. Whenever we write $f \| g$, we implicitly assume $f \heartsuit g$. We write $f[g]$ for the *update* of function f according to g, that is $\forall z \in dom(f) : f[g](z) = (f \lhd g)(z)$.

The state variables of a TTS are partitioned into external and internal variables. Internal variables may only be updated by the TTS itself and not by its environment. This in contrast to external variables, which may be updated by both the TTS and its environment. Transitions are classified as either *committed* or *uncommitted*. Committed transitions have priority over time-passage transitions and over internal transitions that are not committed.

Definition 4.1 *TTS.* A *timed transition system (TTS)* is a tuple

$$\mathcal{T} = \langle E, H, S, s^0, \longrightarrow^1, \longrightarrow^0 \rangle,$$

where $E, H \subseteq \mathcal{V}$ are disjoint sets of external and internal variables, respectively, $V = E \cup H$, $S \subseteq Val(V)$ is the set of states, $s^0 \in S$ is the initial state, and the transition relations $\longrightarrow^1$ and $\longrightarrow^0$ are subsets of $S \times Act \times S$.

We write $r \xrightarrow{a,b} s$ if $(r, a, s) \in \longrightarrow^b$. The value b determines whether or not a transition is committed. We often omit b when it equals 0. A state s is called committed, notation $Comm(s)$, iff it enables an outgoing committed transition, that is, $s \xrightarrow{a,1}$ for some a. We require the following axioms to hold, for all $s, t \in S$, $a, a' \in Act$, $b \in \mathbb{B}$, $d \in \mathbb{R}_{\geq 0}$ and $u \in Val(E)$,

$$s \xrightarrow{a,1} \wedge s \xrightarrow{a',b} \quad \Rightarrow \quad a' \in \mathcal{E} \vee (a' = \tau \wedge b) \qquad \text{(Axiom I)}$$

$$s[u] \in S \qquad \text{(Axiom II)}$$

$$s \xrightarrow{c?,b} \quad \Rightarrow \quad s[u] \xrightarrow{c?,b} \qquad \text{(Axiom III)}$$

$$s \xrightarrow{d} t \quad \Rightarrow \quad t = s \oplus d. \qquad \text{(Axiom IV)}$$

Axiom I states that in a committed state neither time-passage steps nor uncommitted τ's may occur. The axiom implies that committed transitions always have a label in $\mathcal{E}_\tau$. Note that a committed state may have outgoing uncommitted transitions with a label in $\mathcal{E}$. The reason is that, for instance, an uncommitted $c?$-transition may synchronize with a committed $c!$-transition from another component, and thereby turn into a committed τ-transition. Axiom II states that if the external variables of a state are changed, the result is again a state. Axiom III states that enabledness of input transitions is not affected by changing the external variables. This is a key property that we need in order to obtain compositionality.

$$\frac{r \xrightarrow{e,b}_i r'}{r\|s \xrightarrow{e,b} r' \rhd s}\ \textbf{EXT}$$

$$\frac{r \xrightarrow{\tau,b}_i r' \qquad Comm(s) \Rightarrow b}{r\|s \xrightarrow{\tau,b} r' \rhd s}\ \textbf{TAU}$$

$$\frac{\begin{array}{c} r \xrightarrow{c!,b}_i r' \qquad s[r'] \xrightarrow{c?,b'}_j s' \qquad i \neq j \\ Comm(r) \vee Comm(s) \Rightarrow b \vee b' \end{array}}{r\|s \xrightarrow{\tau,b\vee b'} r' \lhd s'}\ \textbf{SYNC}$$

$$\frac{r \xrightarrow{d}_i r' \qquad s \xrightarrow{d}_j s' \qquad i \neq j}{r\|s \xrightarrow{d} r'\|s'}\ \textbf{TIME}$$

Fig. 9. Rules for parallel composition of TTSs

Axiom IV, finally, asserts that if time advances with an amount d, all clocks also advance with an amount d, and the other variables remain unchanged.

In our setting, parallel composition is a partial operation that is only defined when TTSs are *compatible*: the initial states must be compatible functions and the internal variables of one TTS may not intersect with the variables of the other.

Definition 4.2 *Parallel composition.* Two TTSs $\mathcal{T}_1$ and $\mathcal{T}_2$ are *compatible* if $H_1 \cap V_2 = H_2 \cap V_1 = \emptyset$ and $s_1^0 \heartsuit s_2^0$. In this case, their *parallel composition* $\mathcal{T}_1\|\mathcal{T}_2$ is the tuple $\mathcal{T} = \langle E, H, S, s^0, \longrightarrow^1, \longrightarrow^0\rangle$, where $E = E_1 \cup E_2$, $H = H_1 \cup H_2$, $S = \{r\|s \mid r \in S_1 \wedge s \in S_2 \wedge r \heartsuit s\}$, $s^0 = s_1^0\|s_2^0$, and $\longrightarrow^1$ and $\longrightarrow^0$ are the least relations that satisfy the rules in Figure 9. Here i, j range over $\{1, 2\}$, r, r' range over S_i, s, s' range over S_j, b, b' range over $\mathbb{B}$, e ranges over $\mathcal{E}$ and c over $\mathcal{C}$.

The external and internal variables of the composition are simply obtained by taking the union of the external and internal variables of the components, respectively. The states (and start state) of a composed TTS are obtained by merging the states (resp. start state) of the components. The interesting part of the definition consists of the rules in Figure 9. Rule **EXT** states that an external transition of a component induces a corresponding transition of the composition. The component that takes the transition may override some of the shared variables. Similarly, rule **TAU** states that an internal transition of a component induces a corresponding transition of the composition, except that an uncommitted transition may only occur if the other component is in an uncommitted state. Rule **SYNC** describes the synchronization of components. If $\mathcal{T}_i$ has an output transition from r to r', and if $\mathcal{T}_j$ has a corresponding input transition from s, updated by r', to s', the composition has a τ transition to $r' \lhd s'$. The synchronization is committed iff one of the participating transitions is committed. However, an uncommitted synchronization may only occur if both components are in an uncommitted state. Rule **TIME**, finally, states that a time step d of the composition may occur when both components perform a d-step. We refer to Berendsen and Vaandrager [2008] for proofs that the composition of two TTSs is indeed a TTS, and that parallel composition is both commutative and associative modulo structure isomorphism.

Uppaal models can be mapped to TTSs in a straightforward manner [Berendsen and Vaandrager 2008]. Each variable in a Uppaal model corresponds to a variable in a TTS. We treat each element in a Uppaal array as a distinct variable. For each timed automaton A we introduce a fresh variable $A.\mathsf{loc}$ to record the location. The location and local variables of an automaton A are always classified as internal. If v is a local variable of automaton A then $A.v$ becomes an internal variable of the TTS associated to A. Each global variable in a Uppaal model becomes an external variable of *all* automata. A discrete transition is committed if and only if it starts from a state with a committed location.

For the axioms of a TTS to hold we need timed automata to comply with the following rules as defined in Berendsen and Vaandrager [2008]:

—Location invariants do not depend on external variables.

—Satisfaction of guards on input transitions does not depend on the external variables.

—In a committed location always at least one edge is enabled.

—Urgent edges do not synchronize and their guards do not depend on the values of clocks.

It is easy to see that all these rules hold for the Zeroconf model. The urgent action `urg!` can be viewed as an urgent internal action. Because `urg?` is used nowhere, this broadcast synchronization will only involve a single automaton.

Given a timed automaton A, we write $\mathsf{TTS}(A)$ to denote its TTS semantics. The semantics of a complete Uppaal model $A_1, \ldots, A_n$ is obtained by associating a TTS to each individual automaton in the model, taking the composition of all these TTSs, and then removing all synchronization transitions from the resulting TTS using the *restriction* operator $\backslash\mathcal{E}$ from CCS [Milner 1989]:

$$(\mathsf{TTS}(A_1) \| \cdots \| \mathsf{TTS}(A_n))\backslash\mathcal{E}.$$

We claim that, modulo the "committed" Booleans that label transitions, the resulting TTS is equal to the semantics for Uppaal models as defined in Behrmann et al. [2004]. For a proof we refer to Berendsen and Vaandrager [2008].

Abstractions on TTSs can be defined by *timed step simulations*, which are relations on the states of TTSs. Timed step simulation requires that (a) both TTSs have the same external variables, (b) the initial states are related, (c) related states have the same values for external variables, and (d) if these values are changed by the environment the resulting states are again related, (e) if an abstract state is committed then so is every related concrete state, and (f) each transition in the concrete TTS can be mimicked by a transition between related states in the abstract TTS, except τ, which may be simulated by "doing nothing".

Definition 4.3 *Timed step simulation.* Two TTSs $\mathcal{T}_1$ and $\mathcal{T}_2$ are *comparable* if they have the same external variables, that is $E_1 = E_2$. Given comparable TTSs

$\mathcal{T}_1$ and $\mathcal{T}_2$, we say that a relation $\mathrm{R} \subseteq S_1 \times S_2$ is a *timed step simulation* from $\mathcal{T}_1$ to $\mathcal{T}_2$, provided that $s_1^0 \,\mathrm{R}\, s_2^0$ and if $s \,\mathrm{R}\, r$ then

(1) $\forall y \in E_1 : s(y) = r(y)$,
(2) $\forall u \in Val(E_1) : s[u] \;\mathrm{R}\; r[u]$,
(3) if $Comm(r)$ then $Comm(s)$,
(4) if $s \xrightarrow{a,b} s'$ then either there exists an r' such that $r \xrightarrow{a,b} r'$ and $s' \,\mathrm{R}\, r'$, or $a = \tau$ and $s' \,\mathrm{R}\, r$.

We write $\mathcal{T}_1 \preceq \mathcal{T}_2$ when there exists a timed step simulation from $\mathcal{T}_1$ to $\mathcal{T}_2$.

The following two theorems play a key role in our approach. Theorem 4.4 states that invariants for an abstract system are also invariants for a related concrete system, Theorem 4.5 establishes that timed step simulations are compositional.

THEOREM 4.4. *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be comparable TTSs such that $\mathcal{T}_1 \preceq \mathcal{T}_2$. Let ϕ be an invariant over the external variables of $\mathcal{T}_1$ (and $\mathcal{T}_2$), then*

$$\phi \text{ holds in } \mathcal{T}_2 \Rightarrow \phi \text{ holds in } \mathcal{T}_1.$$

THEOREM 4.5. *Let $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$ be TTSs such that $\mathcal{T}_1$ and $\mathcal{T}_2$ are comparable, and both $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible with $\mathcal{T}_3$. If $\mathcal{T}_1 \preceq \mathcal{T}_2$ then $\mathcal{T}_1 \| \mathcal{T}_3 \preceq \mathcal{T}_2 \| \mathcal{T}_3$.*

4.2 An Abstraction with Two Hosts

In this section we will establish that, for the purpose of proving mutual exclusion, a model with just two hosts is a sound abstraction of the model with `l` hosts that we presented in Section 2. Figure 10 presents an overview of all the abstraction steps that we are going to make as part of our verification effort. The figure will be explained in the next paragraphs.

Step 1: Reorder. The first row of Figure 10 shows the situation of the original, unabstracted model. A box `Hostx` denotes the host with hardware address `x`. Altogether there are `l` hosts with addresses $\{\mathtt{0}, \ldots, \mathtt{l} - \mathtt{1}\}$. Each host consists of three automata `Config`, `InputHandler` and `Regular`, as illustrated for `Host0`. In addition there are the network automata. We write

$$\mathtt{Net} \;=\; \|_{i \in \mathtt{Networktype}} \;\; \mathtt{Network}(i).$$

Whenever `Config` or `Regular` sends a message, an automaton $\mathtt{Network}(i)$ is activated, for some i. To simplify reasoning, we associate network automata to specific hosts in the system. More specifically, we introduce one copy of `Net` for each $\mathtt{Config}(j)$ and each $\mathtt{Regular}(j)$ automaton in the system. Let $\mathtt{Networktype}'$ denote the set of indices of `Network` automata in the modified system. Then we have an injective mapping

$$\langle \cdot \rangle : \{C, R\} \times \mathtt{Networktype} \times \mathtt{HAtype} \;\longrightarrow\; \mathtt{Networktype}',$$

where $\langle C, i, j \rangle$ refers to the copy of $\mathtt{Network}(i)$ for use by $\mathtt{Config}(j)$, and $\langle R, i, j \rangle$ refers to the copy of $\mathtt{Network}(i)$ for use by $\mathtt{Regular}(j)$. We also have projections

$$\mathtt{HA} : \mathtt{Networktype}' \;\longrightarrow\; \mathtt{HAtype}$$
$$\mathtt{Comp} : \mathtt{Networktype}' \;\longrightarrow\; \{C, R\},$$

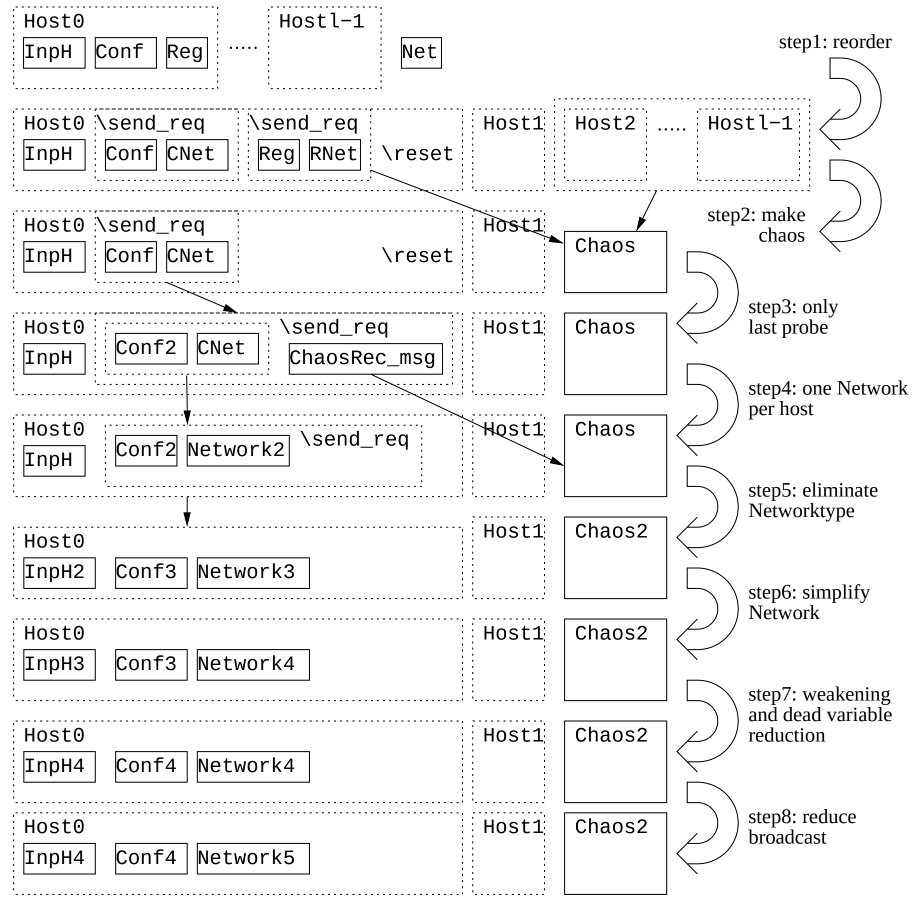

Fig. 10. Overview of abstractions

that assign to each Network automaton in the new system the corresponding hardware address and component, that is, for all c, i and j, $\mathtt{HA}(\langle c,i,j\rangle) = j$ and $\mathtt{Comp}(\langle c,i,j\rangle) = c$. We define, for $j \in \mathtt{HAtype}$,

$$\begin{aligned} \mathtt{CNet}(j) &= \|_{i \in \mathtt{Networktype}} \ \mathtt{Network}[\langle C,i,j\rangle] \\ \mathtt{RNet}(j) &= \|_{i \in \mathtt{Networktype}} \ \mathtt{Network}[\langle R,i,j\rangle]. \end{aligned}$$

The modified system is now obtained by first removing automaton Net, replacing each automaton $\mathtt{Config}(j)$ by

$$(\mathtt{Config}(j) \| \mathtt{CNet}(j)) \backslash \mathtt{send\_req},$$

and similarly replace each automaton $\mathtt{Regular}(j)$ by

$$(\mathtt{Regular}(j) \| \mathtt{RNet}(j)) \backslash \mathtt{send\_req}.$$

The proof that this is a proper abstraction is simple though not compositional. Observe that when an automaton $\mathtt{Network}(i)$ in the original system is in location

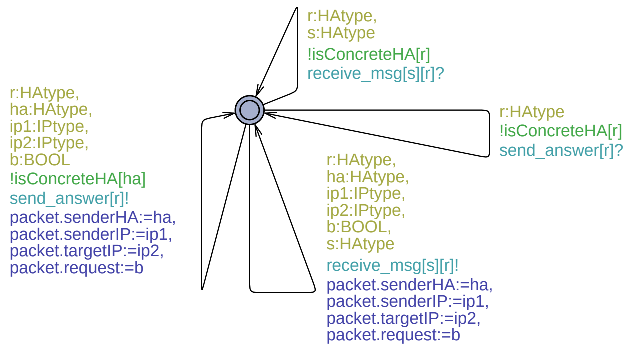

Fig. 11. `Chaos`

`BUSY`, we may infer from the value of variable `s_buffer.senderHA` which host has sent the message that is currently being broadcast. Moreover, if `s_buffer` is a probe or announcement then we know that it has been sent by `Config`; otherwise it has been sent by `Regular`. This allows us to map each state of the old model to a state of the new model: if $\texttt{Network}(i)$ is busy transmitting a message from $\texttt{Config}(j)$, we map the substate of $\texttt{Network}(i)$ to the identical substate of $\texttt{Network}[\langle C, i, j\rangle]$, and if $\texttt{Network}(i)$ is busy transmitting a message from $\texttt{Regular}(j)$, we map the substate of $\texttt{Network}(i)$ to the identical substate of $\texttt{Network}[\langle R, i, j\rangle]$. The substates of the `Config` and `Regular` automata remain unchanged. The substates of `InputHandler` remains unchanged, except that local variable `network` is mapped to the index of the network automaton to which `Network[network]` is mapped. It is straightforward to prove that this map in fact determines a timed step simulation. Note that the modified model contains more `Network` automata (and therefore has more behaviors) than the original model. Hence our first abstraction adds to the number of states rather than reducing it. Using the standard CCS distributive laws for the restriction operator [Milner 1989], we may push the restriction `\reset` inside. Thus we obtain subexpressions

$$(\texttt{InputHandler}(j) \| (\texttt{Config}(j) \| \texttt{CNet}(j)) \backslash \texttt{send\_req}) \backslash \texttt{reset}.$$

The second row of Figure 10 illustrates the new situation.

Step 2: Introducing chaos. The mutual exclusion property `ME` that we want to prove states that two hosts may not use the same IP address. Since all hosts in the network are fully symmetrical, this is equivalent to proving that two specific hosts, say those with HAs 0 and 1, may not use the same IP address:

```
A[] (UseIP[0] && UseIP[1]) imply IP[0]!=IP[1].
```

It turns out that in order to prove the mutual exclusion property for two specific hosts, the behavior of all the other hosts is completely irrelevant. Thus we may over-approximate the behavior of these hosts with a drastic abstraction `Chaos`, an automaton that is able to do all externally visible actions of the abstracted automata in any order and with any timing (cf. the `CHAOS` process in CSP [Hoare 1985]). Automaton `Chaos`, depicted in Figure 11, can do both input and output actions on channels `receive_msg` and `send_answer` in arbitrary order and with arbitrary timing.

The shared variable `packet`, which is used for value passing, is set arbitrarily whenever an output action is performed. For $h \in$ `HAtype`, predicate `isConreteHA`(h) holds if and only if $h \in \{0,1\}$. For $r \in$ `Networktype`$'$, predicate `isUsedByConcreteConfig`$[r]$ holds if and only if `HA`$(r) \in \{0,1\}$ and `Comp`$(r) = C$.

It is straightforward to prove that `Chaos` is an abstraction of all hosts with a HA different from 0 or 1. Moreover, `Chaos` is an abstraction of all the `Regular` automata. Finally, `Chaos` is an abstraction of the composition of two `Chaos` automata. Formally, the following abstraction relations can be shown to hold:

$$\begin{aligned}(\texttt{InputHandler}(j) \| (\texttt{Config}(j) \| \texttt{CNet}(j)) \backslash \texttt{send\_req}) \backslash \texttt{reset} &\preceq \texttt{Chaos} \text{ if } j \notin \{0,1\} \\ (\texttt{Regular}(j) \| \texttt{RNet}(j)) \backslash \texttt{send\_req} &\preceq \texttt{Chaos} \\ \texttt{Chaos} \| \texttt{Chaos} &\preceq \texttt{Chaos}.\end{aligned}$$

Since abstractions are compositional (Theorem 4.5), we can replace hosts 2 to $l-1$ as well as `Regular`(0) and `Regular`(1) and their associated network automata with a single `Chaos` automaton, and obtain a new system as depicted on the third row of Figure 10, where the arrows denote existence of timed step simulations.

Step 3: Only send last probe. The proof of Theorem 3.1 only considers the last probe sent by a host. Here we take a similar approach by over-approximating all the other probes with the chaos automaton. As illustrated in Figure 10, new automata `Config2` and `ChaosRec_msg` are constructed in such a way that:

$$(\texttt{Config}(h) \| \texttt{CNet}(h)) \backslash \texttt{send\_req} \preceq (\texttt{Config2}(h) \| \texttt{Cnet}(h)) \backslash \texttt{send\_req} \| \texttt{ChaosRec\_msg}(h).$$

`Config2` is obtained from `Config` by replacing all `send_req` transitions, except for the last probe, by an internal transition. The upper four locations of Figure 13 illustrate the changes. Automaton `ChaosRec_msg`(h) is able to do all possible actions `receive_msg`$[s][r]!$ just like `Chaos`. Therefore `ChaosRec_msg`$(h) \preceq$ `Chaos` as indicated by the arrow in Figure 10, and the `ChaosRec_msg` automata can be abstracted away in the next step.

In order to prove the correctness of this abstraction, we establish a timed step simulation from the LHS network to the RHS network. Each state of the original model is related to a state of the modified model iff (1) the substate of `Config` is exactly matched by that of `Config2`, and (2) for each automaton `Network`(w), either the substates in LHS and RHS match, or the RHS automaton is in location `IDLE`. Note that also the values of external variables `IP` and `UseIP` are matched by the simulation. If in the LHS model there is a synchronization on `send_req` to send the last probe, the RHS model can do exactly the same, and also the substates of the involved `Network` in the original and modified model match. If in the original model there is a synchronization on `send_req` different from the last probe, `Config2` can perform a τ-transition to preserve the simulation relation. If some `Network`(w) of the LHS model is in location `BUSY` and this is not the case in the RHS model, the actions `receive_msg`$[s][r]!$ can be mimicked by `ChaosRec_msg`. The actions `send_answer`$[r]?$ are mimicked by `Network`(w) in the modified model, since these are also possible from the location `IDLE`.

Step 4: Only one network automaton per host. Let automaton `Network2` be equal to `Network`, except that its parameter j has become a local variable j. In addition, Network2 has a self-loop in location IDLE that non-deterministically updates j to any value from Networktype$'$ that is in use by the host. We claim that[6]

$$(\texttt{Config2(h)}\|\texttt{CNet(h)})\backslash\texttt{send\_req} \preceq (\texttt{Config2(h)}\|\texttt{Network2})\backslash\texttt{send\_req}.$$

Suppose Config2 sends a message (which can only be a last probe) to the Network automaton A, by synchronizing on send_req. It is easy to see that a next message can only be sent after resetting the host, that is, after more at least 2 seconds of delay. At that point, A will surely be back in its IDLE location. Thus, from all Network automata in CNet at most one is in location BUSY. Hence the new automaton Network2 is able to simulate the behavior of CNet in the given context.

Step 5: Eliminate Networktype. Since only one Network automaton per host is needed, we can get rid of Networktype$'$ and use HAtype instead. To make sure that a network automaton only serves one designated host, we parametrize channel send_req with HAtype. Given some hardware address h, automata Config(h) and Network(h) will synchronize on send_req[h]. We adapt our model in the obvious way to use channel types

$$\begin{aligned}&\texttt{send\_req}[\texttt{HAtype}],\\&\texttt{receive\_msg}[\texttt{HAtype}][\texttt{HAtype}],\\&\texttt{send\_answer}[\texttt{HAtype}].\end{aligned}$$

Let InputHandler2, Config3, Network3 and Chaos2 denote the new automata in the model. In Chaos2, the predicate isUsedByConcreteConfig has been replaced by the predicate isConcreteHA, since the Network automata that are used by concrete hosts now have a concrete hardware address. We also move the restriction operators for send_req to the outside again. Correctness of this transformation step can be established via a routine simulation proof.

4.3 Further Reduction of the State Space

With the reductions carried out thus far, in theory model checking arbitrary instances of the model is possible. However, it turns out that the state space of our model is too big to be fully explored by Uppaal. Therefore we need some further abstractions to make the model checking problem tractable.

Step 6: Simplifying the Network automata. The Chaos2 automaton may generate almost arbitrary send_answer[r] messages at any time. In particular, it may generate send_answer[0] and send_answer[1] messages that are picked up by the network automata of hosts 0 and 1. The corresponding packets are stored by these network automata, thus contributing to the total number of reachable states of the system. To reduce the state space, we replace the automaton Network3 by an automaton Network4 that is identical, except that incoming send_answer messages from non-concrete hosts (that is, from Chaos2) are ignored. The template Network4 is displayed in Figure 12. Here function handle_answer is defined by

```
void handle_answer() {
    if (isConcreteHA[packet.senderHA])
```

[6] In fact, there exists a bisimulation between the two networks.

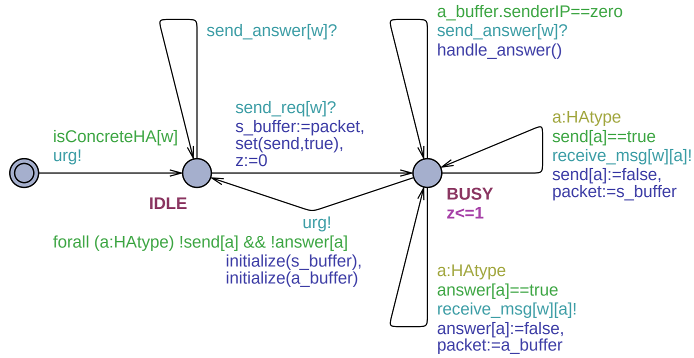

Fig. 12. Timed automaton `Network4(w)`.

```
        {a_buffer:=packet; set(answer,true); }
}
```

Then clearly, for arbitrary $j \in$ `HAtype`,

$$\texttt{Network3}(j) \preceq \texttt{Network4}(j) \| \texttt{Chaos2}.$$

Thus, we may replace the `Network3` by `Network4`.

Step 7: Weakening, dead variable reduction and state merging. By weakening guards, weakening invariants, or by making an urgent channel non-urgent, we add behavior to a timed automaton. The old behavior with the same values for the variables is still present. Adding more behavior to an automaton A using these methods will give an automaton B which simulates A, that is $A \preceq B$, in the sense of Definition 4.3 (timed step simulation).

If, as a result of weakening, a variable is tested in none of the transitions and it is also not read by the environment, it can be safely omitted from the model, an abstraction which can again be justified as a timed step simulation. 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(h)` to `true`, and remove the transition label `urg!`. In the resulting model local variable `ConflictNum` is no longer used and so we can eliminate it. But now, since `COLLISION` only has outgoing transitions to `INIT` with no guards and no effect on the state, we may just as well merge these two locations.

(2) We may weaken the guard of the lower `receive_msg[w]?` transition in automaton `InputHandler(h)` to `true`. In the resulting model local clock `y` is no longer used and it can be eliminated.

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 to always defend the IP address that it is using.

Dead variable reduction is a well known static analysis technique that has, for instance, been studied in the PhD thesis of Yorav [2000]. In Yorav's terminology, a variable v is *used* in a transition if it appears in the guard or in the right hand

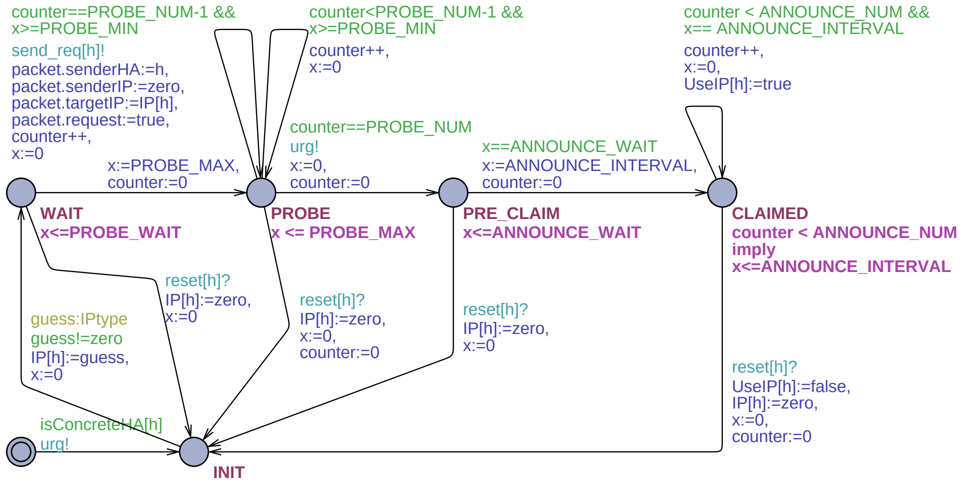

Fig. 13. Final abstract timed automaton `Config4(h)`.

side of an assignment. A variable is used in a location if it appears in the invariant of that location. 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, automata that only differ in the values of dead variables are equivalent in a very strong sense, that is, they are strongly bisimilar, which in turn implies they simulate each other via timed step simulation. Setting variables to a default value as soon as they become dead will reduce the state space, since states that only differ in their dead variables will now become identical.

In our Zeroconf model, variable `counter` of `Config(h)` is dead in locations `PRE_CLAIM` and `INIT`. Hence, setting `counter:=0` upon entering these locations will not affect whether the ME property holds or not. Another example is the variable `network`, which is dead in location `IDLE` of `InputHandler(h)`, and can be reset to a standard value. To make a standard value available we introduce a global constant `HAtype ha0`. The final abstractions of the configuration and input handler automata are displayed in Figure 13 and Figure 14, respectively.

Step 8: Reduced broadcast. There is no real need for `Network4(w)` to do `receive_msg`$[w][a]!$ actions for a's that are not concrete: the only effect of these actions is that they update elements of the `send` and `answer` arrays. Let `Network5(w)` be obtained from `Network4(w)` by slightly altering the `set` function: it only sets the elements to `true` that correspond to concrete HAs. Via a trivial simulation relation we can show that `Network4(0)` and `Network4(1)` can be replaced by `Network5(0)` and `Network5(1)`, respectively.

4.4 Verification Results

We have been able to establish mutual exclusion for a system with an *arbitrary* number of hosts, an *arbitrary* number of HA addresses, and an *arbitrary* number of

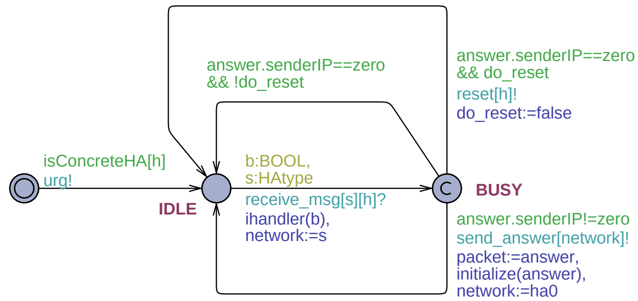

Fig. 14. Final abstract timed automaton `InputHandler4(h)`.

IP addresses. We can handle an arbitrary number of hosts since, as we have shown above, all but 2 automata can be abstracted away by a chaos automaton.

We can handle an arbitrary number of HAs since any instance of Zeroconf with more than 3 HAs can be simulated by the same instance but then with 3 HAs: we just abstract all HAs to 2, except the concrete HAs 0 and 1. This is a sound abstraction since, apart from a number of transitions which test whether a HA is concrete, the only places where HAs are tested are in the input handlers: the input handler of Host0 may test whether the hardware address of an incoming message is equal to 0, and the input handler of Host1 may test whether the hardware address of an incoming message is equal to 1. The outcomes of these tests (and hence the behavior of the protocol) is not affected by an abstraction that identifies all HAs ≥ 2.

We can handle an arbitrary number of IP addresses, due to a result of Ip and Dill [1993] on *data saturation*. This result (which was proven in the setting of Murphi but carries over to Uppaal) states that for certain ("data") scalar types, the state graph does not grow any further once the number of possible values in some scalar type grows beyond the number of places in the system where that scalar type is used. Places consist of variables, but also each select statement on a single edge offers a 'place' for a new value to be chosen. This makes model checking with scalar types of arbitrary size possible.

For IP addresses, at the global level 1 is in use as `zero`, and per (concrete) host:

—1 is in use in global array `IP`

—2 are in use by `InputHandler` in packet `answer`.

—4 IP addresses are in use by `Network`, namely 2 in each of the 2 packets.

In `InputHandler`, the 2 IPs of `answer` are only assigned when entering the committed location. Some committed locations are used for initialization, but after the total system is initialized, only the 2 committed locations of the concrete `InputHandler` automata play a role. It is easy to see these are never visited at the same time, and therefore there will be no interleaving until `InputHandler` has left its committed location. The IP values are passed to other automata, and are after that not used anymore. Moreover they are never tested for their contents. Therefore we do not have to count these variables as extra places where an IP is used.

IPs selected by an select statement are never tested to be different from all other used IPs. Therefore we do not need an extra IP for the select statements, leading to a total of 11 IPs.

Summarizing, all instances of the model are all possible combinations of $l \in \{1, \ldots, 3\}$ HAs, and $m \in \{1, \ldots, 11\}$ IP addresses. Model checking all instances took approximately 10 hours on the following hardware: 2 x Dual-Core Opteron 280 2.4 GHz, 8 GB RAM. Note that we used 4 processing cores to work parallel on different instances. The biggest instance ($l = 3, m = 11$) takes the full time of 10 hours, using 140 MB of memory, exploring $1.754 \cdot 10^6$ symbolic states.

5. 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.

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

(1) A richer syntax for datatypes and functions, in particular a notion of enumerated types.

(2) The ability to initialize clock and structure variables, allowing us to eliminate the initial transition in the `InputHandler(h)` automaton.

(3) 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.

(4) The introduction of urgent transitions (or deadlines) in Uppaal, as advocated in Gebremichael and Vaandrager [2005], Sifakis [1999] and Sifakis and Yovine [1996]. 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 the Zeroconf standard. These models will help to clarify the RFC and to prevent incorrect interpretations due to ambiguity in the text. The Uppaal simulator is also very useful to obtain insight in the operation of the protocol.

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 Brinksma and Mader [2004], 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 two 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 [Kwiatkowska et al. 2004], 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. Support for a more object-oriented specification style appears to be desirable.

Verification. Our modeling efforts revealed six places where RFC 3927 [Cheshire et al. 2005] is incomplete/unclear:

(1) It is not clear whether a host "MUST" or "SHOULD" send ARP Probes before using a new address.
(2) 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.
(3) It does not specify whether a host may immediately start using a newly claimed address or whether it must first send out all ARP Announcements.
(4) It does not specify the tolerance that is permitted on the timing of ARP Announcements.
(5) 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.
(6) It is not exactly clear in which situations a host may defend its address.

The 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 of Section 2. In order to verify the general system automatically, we had to apply some drastic abstractions. The soundness of these abstractions has been proven manually. In our view, it is highly desirable to further 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 study, we have modeled and analyzed a fragment of Zeroconf in a restrictive setting without faulty nodes, merging of subnetworks, etc. In order to deal with dynamically changing network topologies, a more sophisticated use of abstractions will be required, for instance along the lines of Bauer [2006]. An obvious challenge is to mechanize all these abstractions using either (an extension of) Uppaal-Tiga [Cassez et al. 2005] or a general purpose theorem prover. The timing behavior of Zeroconf becomes really interesting when studied within a setting in which also the probabilistic behavior is modeled. The performance analysis of Zeroconf reported in Bohnenkamp et al. [2003] and Kwiatkowska et al. [2003] 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 Cobelens (Free University, Amsterdam) for answering all our questions about Zeroconf. Martijn Hendriks, 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. Finally, we thank the anonymous reviewers for some insightful comments.

REFERENCES

Abadi, M. and Lamport, L. 1994. An old-fashioned recipe for real time. *ACM Trans. Program. Lang. Syst. 16,* 5 (Sept.), 1543–1571.

Alur, R. and Dill, D. 1994. A theory of timed automata. *Theoretical Computer Science 126*, 183–235.

Bauer, J. 2006. Analysis of communication topologies by partner abstraction. Ph.D. thesis, Universität des Saarlandes, Saarbrücken, Germany.

Behrmann, G., David, A., and Larsen, K. 2004. A tutorial on Uppaal. In *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, 2004, Revised Lectures*, M. Bernardo and F. Corradini, Eds. Lecture Notes in Computer Science, vol. 3185. Springer, 200–236.

Behrmann, G., David, A., Larsen, K. G., Håkansson, J., Pettersson, P., Yi, W., and Hendriks, M. 2006. Uppaal 4.0. In *Third International Conference on the Quantitative Evaluation of SysTems (QEST 2006),* 11-14 September 2006, Riverside, CA, USA. IEEE Computer Society, 125–126.

Berendsen, J. and Vaandrager, F. 2008. Compositional abstraction in real-time model checking. In *Proceedings Sixth International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2008),* September 15-17, 2008, Saint-Malo, France. Lecture Notes in Computer Science, vol. 5215. Springer Berlin / Heidelberg, 233–249.

Berry, G. and Gonthier, G. 1992. The Esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program. 19,* 2 (Nov.), 87–152.

Bohnenkamp, H., Stok, P. v. d., Hermanss, H., and Vaandrager, F. 2003. Cost-optimisation of the IPv4 zeroconf protocol. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2003).* IEEE Computer Society, Los Alamitos, California, 531–540.

Brinksma, E. and Mader, A. 2004. On verification modelling of embedded systems. Tech. Rep. TR-CTIT-04-03, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands. January.

Bruns, G. and Staskauskas, M. 1998. 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. IEEE Computer Society, 198–205.

Cassez, F., David, A., Fleury, E., Larsen, K., and Lime, D. 2005. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, M. Abadi and L. de Alfaro, Eds. Lecture Notes in Computer Science, vol. 3653. Springer, 66–80.

Cheshire, S. 2006. Personal communication.

Cheshire, S., Aboba, B., and Guttman, E. 2005. Dynamic configuration of IPv4 link-local addresses (RFC 3927). `http://www.ietf.org/rfc/rfc3927.txt`.

Cheshire, S. and Steinberg, D. 2005. *Zero Configuration Networking: The Definite Guide*. O'Reilly Media, Inc.

Chkliaev, D., Hooman, J., and de Vink, E. 2003. Verification and improvement of the sliding window protocol. In *Proceedings TACAS'03*. Lecture Notes in Computer Science 2619, Springer-Verlag, 113–127.

Clarke, E. M., Grumberg, O., Hiraishi, H., Jha, S., Long, D. E., McMillan, K. L., and Ness, L. A. 1993. Verification of the Futurebus+ cache coherence protocol. In *Proc. CHDL*. 15–30.

Devillers, M., Griffioen, W., Romijn, J., and Vaandrager, F. 2000. Verification of a leader election protocol: Formal methods applied to IEEE 1394. *Formal Methods in System Design 16,* 3 (June), 307–320.

Gebremichael, B. and Vaandrager, F. 2005. 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. IEEE Computer Society, 64–73.

Gebremichael, B., Vaandrager, F., and Zhang, M. 2006. Analysis of the Zeroconf protocol using Uppaal. In *Proceedings 6th Annual ACM & IEEE Conference on Embedded Software (EMSOFT 2006),* Seoul, South Korea, October 22-25, 2006. ACM Press, 242–251.

Hendriks, M., Behrmann, G., Larsen, K., Niebert, P., and Vaandrager, F. 2004. Adding symmetry reduction to Uppaal. In *Proceedings First International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS 2003),* September 6-7 2003, Marseille, France. Lecture Notes in Computer Science, vol. 2791. Springer-Verlag.

Hoare, C. 1985. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs.

Holzmann, G. 2004. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley.

Ip, C. and Dill, D. 1993. Better verification through symmetry. In *Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications - CHDL '93,* Ottawa, Ontario, Canada, 26-28 April, 1993, D. Agnew, L. J. M. Claesen, and R. Camposano, Eds. IFIP Transactions, vol. A-32. North-Holland, 97–111.

Jensen, H., Larsen, K., and Skou, A. 2000. Scaling up Uppaal: Automatic verification of real-time systems using compositionality and abstraction. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium, FTRTFT 2000,* Pune, India, September 20-22*, Proceedings*, M. Joseph, Ed. Lecture Notes in Computer Science, vol. 1926. Springer, 19–30.

Kwiatkowska, M., Norman, G., and Parker, D. 2004. PRISM 2.0: A tool for probabilistic model checking. In *Proceedings of the 1st International Conference on Quantitative Evaluation of Systems (QEST04)*. IEEE Computer Society, 322–323.

Kwiatkowska, M., Norman, G., Parker, D., and Sproston, J. 2003. Performance analysis of probabilistic timed automata using digital clocks. In *Proc. Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, K. Larsen and P. Niebert, Eds. LNCS, vol. 2791. Springer-Verlag, 105–120.

Langevelde, I. v., Romijn, J., and Goga, N. 2003. Founding FireWire bridges through Promela prototyping. In 8^{th} *International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA)*. IEEE Computer Society Press.

Larsen, K., Mikucionis, M., and Nielsen, B. 2005. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *the 5th ACM International Conference on Embedded Software*. ACM Press New York, NY, USA, 299 – 306.

Lynch, N. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Fransisco, California.

Milner, R. 1989. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs.

Plummer, D. 1982. An Ethernet address resolution protocol (RFC 826). http://www.ietf.org/rfc/rfc826.txt.

Romijn, J. 2004. Improving the quality of protocol standards: Correcting IEEE 1394.1 FireWire net update. *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica 8*, 23–30. Available at http://www.win.tue.nl/oas/index.html?iqps/.

Sifakis, J. 1999. The compositional specification of timed systems - a tutorial. In *Proceedings of the 11th International Conference on Computer Aided Verification,* Trento, Italy, N. Halbwachs and D. Peled, Eds. Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, 2–7.

Sifakis, J. and Yovine, S. 1996. Compositional specification of timed systems (extended abstract). In *STACS*, C. Puech and R. Reischuk, Eds. Lecture Notes in Computer Science, vol. 1046. Springer, 347–359.

Stoelinga, M. 2003. 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.

Vaandrager, F. and Groot, A. d. 2006. Analysis of a biphase mark protocol with Uppaal and PVS. *Formal Aspects of Computing Journal 18,* 4 (December), 433–458.

Yorav, K. 2000. Exploiting syntactic structure for automatic verification. Ph.D. thesis, The Technion, Israel Insitute of Technology.