Formal Specification and Analysis of Zeroconf Using Uppaal*

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

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

Abstract. We report on a case study in which the model checker Uppaal is used to formally model parts of Zeroconf, a protocol for dynamic configuration of IPv4 link-local addresses that has been defined in RFC 3927 of the IETF. Our goal has been to construct a model that (a) is easy to understand by engineers, (b) comes as close as possible to the informal text (for each transition in the model there should be a corresponding piece of text in the RFC), and (c) may serve as a basis for formal verification. Our 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 either by hand or by using Uppaal-Tiga.

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.

* 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), http://ametist.cs.utwente.nl, the DFG/NWO bilateral cooperation project Validation of Stochastic Systems (VOSS2), and NWO/GBE project 612.000.103 Fault-tolerant Real-time Algorithms Analyzed Incrementally (FRAAI). A preliminary version of this paper appeared as [21]. All the Uppaal models described in this paper are available on-line at http://www.cs.ru.nl/ita/publications/papers/fvaan/zeroconf/.

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 implementors 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. [18, 11, 19, 28, 34, 23, 17, 35]). 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 et al., 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 [33].

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.[3] Bruns and Staskauskas [11] used (a well-defined subset of) C to describe the SONET Automatic Protection Switching (APS) protocol and report that developers found their C description easy to understand and superior to that which appeared in the APS standard. 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 [10],

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

> "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 the UPPAAL model checker 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 [5, 4] is an integrated tool environment for specification, validation and verification of real time systems modeled as networks of timed automata [2]. The tool is available for free for non-profit applications at `www.uppaal.com`. The language for the new version UPPAAL 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.

Recently, efficient on-line algorithms for solving reachability and safety games based on timed game automata have been put forward [12] and made available within the tool UPPAAL-TIGA. In collaboration with Thomas Chatain, Alexandre David and Kim Larsen, we have used UPPAAL-TIGA to automatically check the correctness of one of our abstractions [13].

Zeroconf [16] is a protocol for dynamic configuration of IPv4 link-local addresses that has been defined by the IETF Network Working Group in RFC 3927 [15]. 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[4], but several other implementations are available.[5]

Contribution The contribution of this paper is, first of all, a formal model of (a critical part of) Zeroconf — a protocol with clear practical relevance — that is easy to understand, faithful to the RFC, and with an extensive discussion of the relationship between the model and the RFC. Our modeling efforts revealed several errors (or at least ambiguities) in the RFC that no one else spotted before. We 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 either by hand or by using UPPAAL-TIGA.

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 [9, 26]. The primary goal of [9] was to investigate the trade off between reliability and effectiveness of the protocol using a stochastic cost model. The model of [9], 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 [26] using the probabilistic model checker PRISM [27]. The model checking results reported in [26] are quite interesting, but the precise relationship between the model and the RFC is unclear (for instance, in the model of [26] address defense only occurs *before* a host is using an IP address). Our motivation for using UPPAAL instead of PRISM was that the input language of PRISM is too primitive for our purposes (no GUI, just a few datatypes, no support of C-like syntax,..). A toolset that combines the functionality of UPPAAL and PRISM would be ideal for dealing with the Zeroconf protocol. The compositional step simulation relations

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

between timed transition systems that we use to establish the correctness of our abstractions are inspired by the timed ready simulations from [25], and use the framework decribed in [6].

Paper 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 our model and RFC 3927 [15], 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 friends who are writing a joint paper and want to share a file. The goal of Zeroconf is to enable networking in the absence of configuration and administration services. The core of RFC 3927 [15] concerns 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, seperated by some delay, asking whether someone is already using the address. If one of the other hosts indicates that it is using the other address, the host starts all over again. Otherwise, it 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 different hosts are using the same IP address. The underlying algorithm used in Zeroconf is similar to Fisher's mutual exclusion algorithm [1, 30] and makes essential use of timing. However, whereas Fischer's algorithm uses a shared variable for communication between processes, Zeroconf uses broadcast communication. Within Zeroconf, hosts do not aim at acquiring access to a specific critical section (IP address); it is enough to obtain access to one of the 65024 available critical sections (IP addresses).

2.1 Fixing the Topology

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 `k` hosts. It may take arbitrary long before a host becomes active in the protocol and one may argue that in this way creation of new hosts is being captured. 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.[6]

The behavior of a 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 process identifier (PID) of the host they belong to. We assume a finite set of k PIDs, represented in UPPAAL by a scalar type:

```
typedef scalar[k] PIDtype;
```

Within UPPAAL, `scalar[k]` denotes the set $\{0, \ldots, k-1\}$, but on scalar types only a limited number of operations is permitted: assignment and identity testing. As a consequence, a scalar type is unordered and fully symmetric: the behavior of a model is invariant under arbitrary permutations of the elements of a scalar type [24, 22]. By using a scalar type rather than a subrange type, we specify that within our model all the PIDs (and therefore all the hosts) play a fully symmetric role. This makes it possible to exploit this symmetry during exploration of the state space.

2.2 The Underlying Network

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

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

The Address Resolution Protocol (ARP) [32] 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.

We assume a finite set of hardware addresses, represented abstractly by a scalar type:

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

Here `l` denotes the number of hardware addresses. As explained above, the advantage of using a scalar type is that we may benefit from symmetry reduction during verification. However, since we cannot refer explicitly to elements from a scalar type, we need a

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

trick to denote the all zeroes hardware address, which is referred to by the protocol: we introduce a state variable `nil` of type `HAtype`, whose value remains unchanged during any run of the system, and let the value of `nil` represent the all zeroes hardware address. We assume `l` > `k`, so that we can assign to each host a unique nonzero hardware address.

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 abstractly by a scalar type:

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

Again, we need a trick to refer to a specific element of this scalar type within our model: state variable `zero` of type `IPtype`, whose value remains unchanged during any run of the system, denotes the all zeroes IP address.

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

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

Here we use the convention that the `request` field is `true` for ARP requests and `false` for ARP replies.

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 another 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 underlying network as a set of `n` identical `Network` automata. Each of these automata takes care of handling a single ARP request at a time, 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. Fig. 1 schematically illustrates the operation of a `Network` automaton. After a request from a host comes in (`send_req`), this

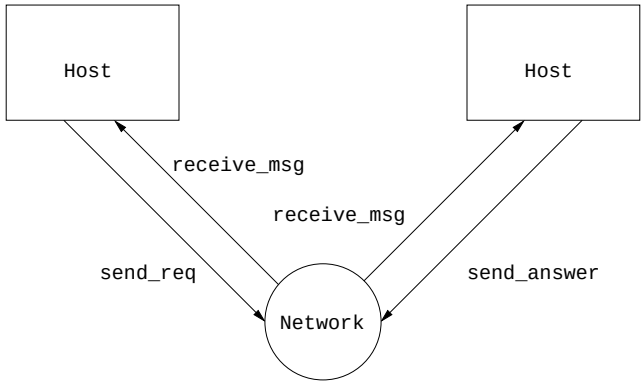

Fig. 1. Interaction between `Network` automaton and hosts.

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 `receive_msg` action, and broadcast to all other hosts via subsequent `receive_msg` actions. All these interactions take place within 1 second. After completing its 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, i.e., we adopt the synchrony hypothesis that is well-known from synchronous programming [8]. A desktop computer can realistically answer an ARP in 100μs. A device like a SitePlayer could take up to 10ms. Neither have a significant impact on achieving a round-trip delay under 1s. By taking the conceptual view that the 1s which `Network` may use to do its work *includes* the time needed by a host to generate a reply, we avoid cumbersome modeling of input buffers at each host.

Before explaining our 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

Each host in the system maintains a hardware address and an IP address:

```
HAtype HA[PIDtype];
IPtype IP[PIDtype];
```

A special initialization process assigns to each host a unique, nonzero hardware address (which remains fixed for the rest of the execution) and the initial all zeroes IP address (which will change as a result of address configuration).

Fig. 2 displays the automaton `Config[pid]`, which specifies how host `pid` configures a new IP address. The host starts in location `INIT`, where it stays

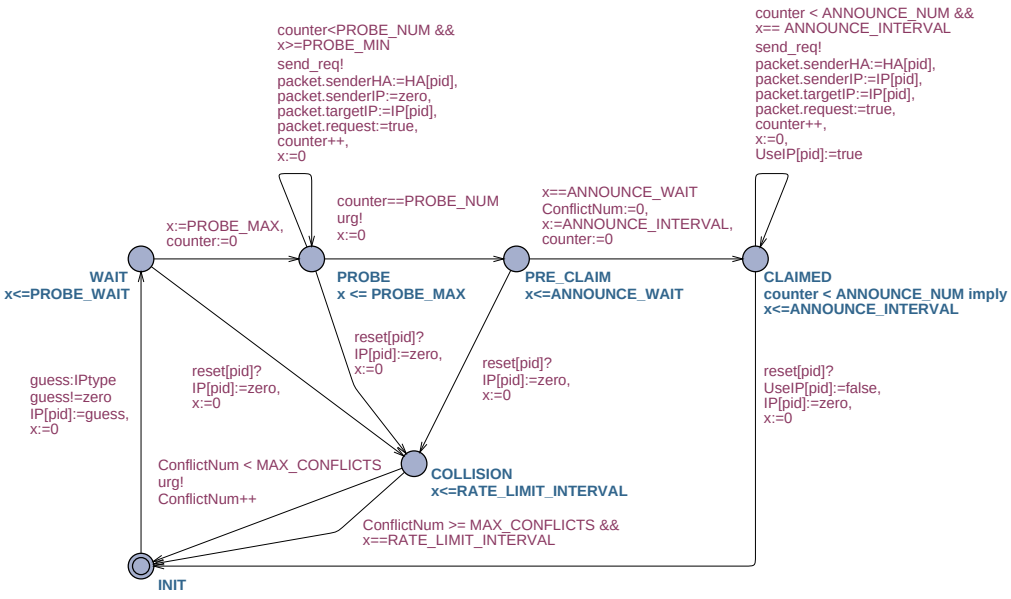

Fig. 2. Automaton `Config[pid]`.

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 to mark that an address has been selected. Via the UPPAAL 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[pid]`.

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 seconds, and should then send PROBE_NUM probe packets, each of these

> probe packets spaced randomly, PROBE_MIN to PROBE_MAX seconds apart."

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

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

Sending ARP Probes is modeled via actions `send_req!` that synchronize with the network. The actual packet is communicated via a global shared variable `packet` of type `ARP_packet`. In UPPAAL the assignments in an output (!) transition are executed before the assignments in a synchronizing input (?) transition, and this allows us to assign a value to `packet` in a `send_req!` transition, which is then picked up by a corresponding `send_req?` transition by a `Network` automaton. The lower and upper bounds of the probe interval are expressed in our model with a guard `x >= PROBE_MIN` on the sending transition and an invariant `x <= PROBE_MAX` on location `PROBE`, respectively. By setting `x` to `PROBE_MAX` in the transition from `WAIT` to `PROBE`, we express that the first probe is sent immediately. A local variable `counter` is used to record the number of probes that have been sent. After the probing phase is 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. 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 `CLAIM` indicates that the host has successfully claimed an address.

In our model, automaton `InputHandler[pid]` (which will be explained in Section 2.4) takes care of handling incoming messages. If `InputHandler[pid]` decides that, due to some conflict, a new address must be configured, it performs an

action `reset[pid]!`. This triggers a `reset[pid]?` transition to location `COLLISION` in `Config[pid]`. As part of this transition, `IP`[`pid`] 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."

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

In location `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 notion of an ARP Announcement is specified in the RFC as follows:

> "In this document, the term "ARP Announcement" is used to refer to an ARP Request packet, broadcast on the local link, identical to the ARP Probe described above, except that both the sender and target IP address fields contain the IP address being announced."

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` [14]. 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 [14]. 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 [14]. 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 [29], one will have to modify our model at this point.

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

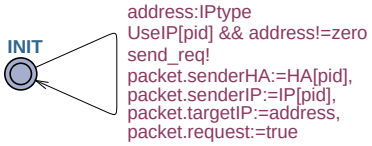

Fig. 3. Automation `Regular[pid]`.

2.4 The Input Handler

Automaton `InputHandler[pid]` receives incoming ARP packets and decides what to do with them. Input handling is described at various places in RFC 3937, which makes it nontrivial to determine the reaction to an arbitrary ARP packet, also because Zeroconf runs on top of the ARP protocol, which it sometimes follows but sometimes overrules. Automaton `InputHandler[pid]` is displayed in Fig. 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[j][pid]?` transition occurs, the automaton calls a function `ihandler`, which does the real work. The definition of `ihandler` is listed in Fig. 5. Function `ihandler` has a parameter `defend` which may be either `false` or `true`. This parameter indicates that a host will defend its IP address in case of a conflicting ARP request, and may be `true` only if there has been no other conflict during the last `DEFEND_INTERVAL` time units. Clock `y` is used to measure the time since the last conflict. Altogether, the input handler needs to distinguish 9 scenarios:

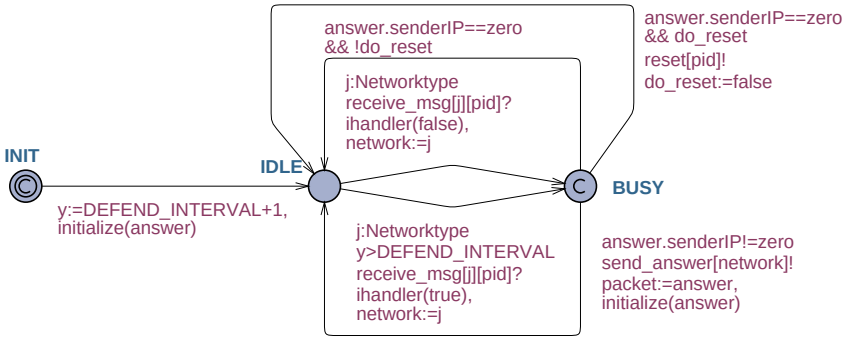

Fig. 4. Automaton `InputHandler[pid]`.

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

Scenario B. 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[pid]`. This occurs in Scenario C, which is described on [page 11, section 2.2.1]:

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

Scenarios D and E. In the previous scenario, `UseIP[pid]==false`. The case with `UseIP[pid]==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.

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

Fig. 5. Function `ihandler`.

A host MUST respond to a conflicting ARP packet as described in either (a) or (b) below:
(a) Upon receiving a conflicting ARP packet, a host MAY elect to immediately configure a new IPv4 Link-Local address as described above, or
(b) If a host currently has active TCP connections or other reasons to prefer to keep the same IPv4 address, and it has not seen any other conflicting ARP packets within the last DEFEND_INTERVAL seconds, then it MAY elect to attempt to defend its address by recording the time that the conflicting ARP packet was received, and then broadcasting one single ARP Announcement, giving its own IP and hardware addresses as the sender addresses of the ARP. Having done this, the host can then continue to use the address normally without any further special action. However, if this is not the first conflicting ARP packet the host has seen,

> and the time recorded for the previous conflicting ARP packet is recent, within DEFEND_INTERVAL seconds, then the host MUST immediately cease using this address and configure a new IPv4 Link-Local address as described above. This is necessary to ensure that two hosts do not get stuck in an endless loop with both hosts trying to defend the same address.
> A host MUST respond to conflicting ARP packets as described in either (a) or (b) above. A host MUST NOT ignore conflicting ARP packets."

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

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

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) [32] specifies that if a host receives an ARP request packet, it should return an ARP reply packet if it uses an IP address that equals the target protocol address of this request. In the reply packet the hardware and protocol field should be swapped, putting the local hardware and protocol addresses in the sender fields. Zeroconf

(RFC 3927) is not explicit about conformance to RFC 826, but in our model we take the view that once a host is using an IP address, it answers regular ARP requests in agreement with RFC 826 except when (a) the request has been broadcast by the host itself, or (b) there is a conflict. This is scenario H in our model. The final Scenario I occurs when the incoming packet is not conflicting with the IP address that the host is using, and no reply packet needs to be sent.

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 `j`, the automaton `Network[j]` is shown in Fig. 6. Initially the automaton is in its `IDLE` location. As soon as it receives a

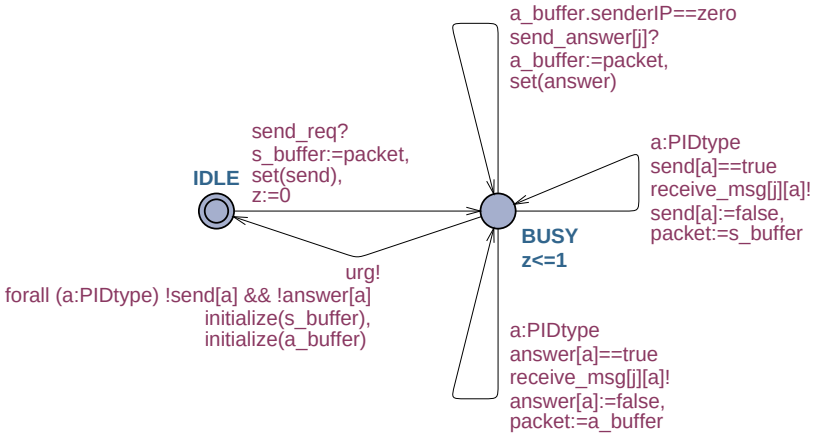

Fig. 6. Automaton `Network[j]`.

packet from a host via `send_req?`, it jumps to location `BUSY`. A local clock `z` is set to zero and an invariant `z` $\leq$ `1` ensures that within 1 second the network broadcasts the packet (and the answer if there is one) to all hosts. There is no lower bound on message delivery time. We assume that there is at most one host that wants to answer any given request, and that an answer does not induce subsequent answers. It is possible to handle multiple and successive answers, but this requires additional state variables and more complicated data structures. Our `Network` automaton maintains two local variables for storing packets: `s_buffer` stores the packet that was sent by the host and `a_buffer` stores an answer 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 Booleans of a Boolean array to `true`. Using the UPPAAL select statement on the `receive_msg[j][a]!` transitions, the automaton

nondeterministically selects in which order packets are delivered to the different hosts. The upper transition labeled with `send_answer[j]?` 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[j][a]!` is enabled as soon as there is an answer packet in `answer` buffer. The network returns to its `IDLE` location and resets its buffers to some initial values, as soon as all messages have been delivered.

2.6 Dimensioning the Complete Model

The RFC [page 25, section 9] specifies the following values for the different timing constants. These definitions are copied almost verbatim in the 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 [15]. These values are too big for automatic verification and with 3 hosts 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 the initial location via a reset in zero time. In fact, this behavior may be repeated `MAX_CONFLICTS` times in a row in zero time. Once a host is using an IP address, the number of messages in transit may increase even further (in fact unboundedly) since there is no lower bound on the time between successive ARP requests. UPPAAL forces us to bound the number of `Network` automata to some small number `n`.

3 Manual Verification

The model described in Section 2 is very close to the RFC definition of the protocol. However, the model is too big for UPPAAL to do a complete state space exploration for nontrivial instances without some drastic abstractions.

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:[7]

[7] 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.

1. Mutual exclusion, i.e., two hosts may not use the same IP address. This can be specified in UPPAAL as follows:

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

2. Absence of deadlock, i.e, in each reachable state a transition is possible. In UPPAAL syntax:

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

Using the latest version of UPPAAL (4.0), we only managed to establish `ME` and `DL` for the instance with 2 hosts, 1 link-local IP address and 2 network automata. Nevertheless, it is rather obvious that Zeroconf satisfies the mutual exclusion and absence of deadlock properties. 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 1. *For each instance of our Zeroconf model (i.e., any number of hosts, hardware addresses, IP addresses and network automata), the mutual exclusion property* `ME` *holds.*

Proof. (Sketch) Suppose `i` and `j` are hosts with `i != j`, and suppose that in some reachable state s, `UseIP[i]`, `UseIP[j]` and `IP[i]=IP[j]`. We derive a contradiction. Consider an execution α leading up to state s, i.e., a finite sequence of delay and action transitions in the timed transition system semantics of the model leading from the start state to s. Without loss of generality, we assume that host `j` enters the critical section before (but possibly at the same time as) `i`. Observe that before a host enters the "critical section" (where it may use its IP address) it resides at least 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 2 time units before entering the critical section, a host sends a (in fact, the last) probe packet.

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

(in)equalities:

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

We consider two cases:

1. See Figure 7. The probe arrives at host `j` before it enters the critical section.

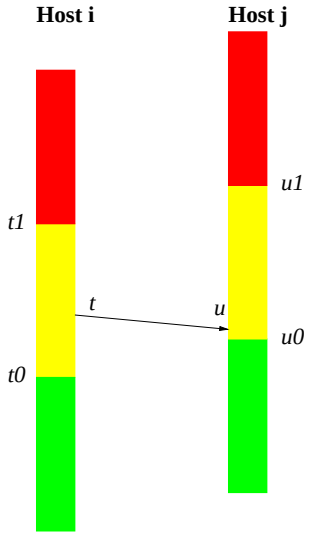

Fig. 7. Probe arrives at `j` before it enters critical section.

   Then `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 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 it enters the critical section. But this means that host `j`'s input handler, upon receipt of the probe, will return a reply message (Scenario H). Since we assume a roundtrip delay of

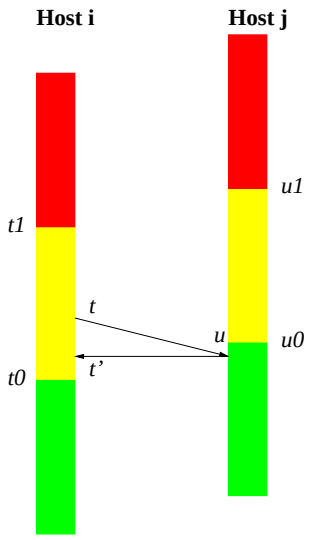

Fig. 8. Probe arrives at `j` after it enters critical section.

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 region. Contradication. QED

Formalization/mechanization of the proof of Theorem 1, for instance in PVS using the basic setup of [35], should be a routine exercise.

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 `DE` in a few seconds on a standard PC.

4 Verification by Model Checking and Abstraction

Although we have a manual (operational) proof of Theorem 1, 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 can only verify a single instance of the protocol, whereas we would like to establish correctness for all (infinitely many) instantiations of its parameters. Abstraction is a remedy to both problems.

The idea is to use an abstraction relation for UPPAAL that is *sound* for the property to be verified, meaning that when the properties hold in the abstract model, then it must hold in the concrete models. We need to make abstractions in a compositional way, which means that in a parallel composition of a set of timed automata, a subset can be replaced by its abstraction, thereby obtaining a total model that in turn is an abstraction of the original total model.

Section 4.1 describes the compositional abstraction framework. Section 4.2 introduces some small extra assumptions we need. Section 4.3 gives a finite abstract model, and Section 4.4 gives a feasible version of this model. Finally Section 4.5 gives the verification results.

4.1 Compositional Abstraction

The operational semantics of a UPPAAL model is defined on the model as a whole, see the UPPAAL help menu or [5]. Instead we need timed transition systems (TTSs) as semantical model for a *single* timed automaton, parallel composition on TTSs, and an abstraction relation for TTSs that is sound for invariant properties. Details and proofs of this compositional semantics and abstraction are described in [6].

Jensen, Larsen and Skou [25] present a compositional transition system semantics for UPPAAL which handles urgency and (a restricted form of) multi-writer/multi-reader semantics. However their model does not incorporate committed locations and their parallel composition operator is not associative [7], two aspects which are essential for our case study.

Basically, TTSs are labeled transition systems equipped with a bit of additional structure to support shared variables and committed transitions: states are defined as valuations of variables, and transitions may be committed, which gives them a certain priority in a parallel composition. TTSs can be placed in parallel and may communicate by means of shared variables and synchronization of actions. Like in CCS [31], two transitions may synchronize when their actions are complementary, leading to an internal transition in the composition.

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 $c?$ are called *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 are just the nonnegative real numbers in $\mathbb{R}_{\geq 0}$. 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. 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 *left-merge*, the combined function where f overrides g for all elements in the intersection of their domains. Formally, we define $f \rhd g$ to be 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 *right-merge* 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 their *merge* by $f \| g \triangleq f \rhd g$. 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. A new element in our definition of a TTS is that transitions are classified as either *committed* or *uncommitted*. Committed transitions have priority over time-passage transitions and over internal transitions that are not committed. Interestingly, whereas in UPPAAL committedness is an attribute of locations, it must be treated as an attribute of transitions in order to obtain a compositional semantics.

Definition 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 some other 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. 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 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 Fig. 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

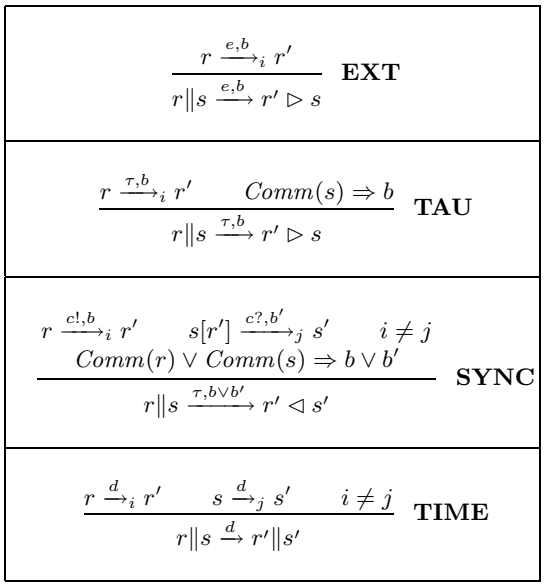

Fig. 9. Rules for parallel composition of TTSs

step d of the composition may occur when both components perform a time step d.

We refer to [6] for a proof that the composition of two TTSs is indeed a TTS. An important sanity check for our definitions is that parallel composition is both commutative and associative. Commmutativity immediately follows from the definitions. For a proof of associativity we refer to [6].

UPPAAL models can be mapped to TTSs in a straightforward manner [6]. 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 special variable $A.\mathsf{loc}$ to record the current location of this automaton. 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 element of the external variables 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 [6]:

- A location invariant does not depend on the 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.

In the Zeroconf model it is easy to see that all of the rules hold. The urgent action `urg!` can be viewed as an urgent internal action. Because `urg?` does not exist 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 (cf. the *restriction* operator in CCS [31]):

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

We claim that, modulo the "committed" Booleans, the resulting TTS is equal to the semantics for UPPAAL models as defined in [5]. For a proof we refer to [6].

Abstractions on TTSs are defined by *timed step simulation*, which is a relation on the states of the TTSs. Timed step simulation requires that both TTSs have the same external variables. The initial states are related. Related states have the same values for external variables, and if these values are changed by the environment two states are obtained that again are related. If an abstract state is committed, than so is every related concrete state. Each transition in the concrete TTS is mimicked by a transition between related states in the abstract TTS, except τ, which may be simulated by "doing nothing".

Definition 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. $\forall u \in \mathit{Val}(E_1) : s[u] \;\mathrm{R}\; r[u]$,
3. *if $\mathit{Comm}(r)$ then $\mathit{Comm}(s)$,*
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 verification. Theorem 2 states that invariants for an abstract system are also invariants for a related concrete system, Theorem 3 establishes that timed step simulations are compositional.

Theorem 2. *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be comparable timed transition systems 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 3. *Let $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$ be timed transition systems 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 Model without Network Addresses

Before we can make a finite abstraction of our model, we need some other abstractions and some minor changes to the original model.

A new assumption that we need is that after `reset[j]` in automaton `Config` at least 1 second is spent in location `INIT`. We model this by adding the conjunct $x \geq 1$ on the guard of the edge from `INIT` to `WAIT`. To keep the same behavior for the initial state of the model, we add an initial committed location to `Config` that has an edge to `INIT` which updates x to 1. Note that our new initial state differs in clock x from the original one, but this has no influence on behavior: an arbitrary amount of time may elapse in `INIT`, the transition to `WAIT` is enabled on the same conditions, and x is reset anyway. Figure 10 shows the changed part of `Config`.

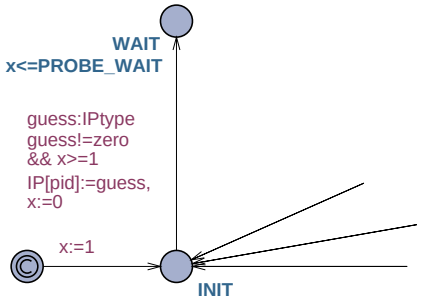

Fig. 10. Changed part of `Config`.

Using our new assumption we will show that we only have to prove a variant of the model where every host is associated with one specific `Network` automaton. The following argumentation can be formalized straightforward in terms of strong bisimularity of the underlying transition systems. The model contains n `Network` automata that together model the complete network. Everytime a host wants to send a message, one automaton `Network` is used. For reasons of symmetry, it does not matter which of the n automata is chosen. Suppose some host has sent a message using `Network` automaton A. Our new assumption makes sure that after resetting the host, a new message will only be sent after more than 1 second of delay has elapsed. At that point, A will surely be finished, and it is back in its `IDLE` location. Therefore automaton A may again be used for the next message that is sent.

We conclude that we can associate one `Network` automaton to each host that only serves that specific host. The new model is obtained from the old definition by replacing `Networktype` everywhere by `PIDtype`. To make sure `Network` only serves a designated host, we parametrize channel `send_req` with `PIDtype`. Given some PID $h \in$ `PIDtype` automata `Config`(h) and `Network`(h) will synchronize on `send_req`$[h]$.

The model for a single host is described by:

$$H(h) = \texttt{Config}(h) \| \texttt{InputHandler}(h) \| \texttt{Regular}(h) \| \texttt{Network}(h)$$
$$\text{, where } h \in \texttt{PIDtype}.$$

The total model becomes:

$$H(h_1) \| \cdots \| H(h_k) \| \texttt{Initializer}$$
$$\text{, where we have } k \text{ PIDs } h_1, \ldots, h_k \in \texttt{PIDtype}.$$

We will leave out `Initializer` as it is only used to give a number of global scalarsets their initial values, as UPPAAL does not support initialization of scalarsets.

4.3 A Finite Model Checking Problem

Here we construct a finite model checking problem, using a so called chaos automaton. In theory this model is decidable, however in the next sections we will introduce some more abstractions to make model checking tractable.

Because PIDs are modeled by the scalarset `PIDtype`, our mutual exclusion property holds if and only if the property below holds. Thus, we need to prove mutual exclusion only for two hosts:

$$\texttt{A[] (UseIP[}h_1\texttt{] \&\& UseIP[}h_2\texttt{]) imply IP[}h_1\texttt{]!=IP[}h_2\texttt{]}$$

We will build a so called chaos automaton `Chaos1` that is an abstraction for hosts $H(h_3), \ldots, H(h_k)$ in parallel, i.e., it is able to simulate all behavior of these hosts in parallel as observable by the environment.

From the automata we see that in synchronization between $H(h_1), \ldots, H(h_k)$, only channels `receive_msg` and `send_answer` play a role. Channels `reset` and `send_req` are only used within a single host.

Now `Chaos1` is the automaton able to do the following actions in arbitrary order and with arbitrary timing, where s, r will denote sender, receiver side of a synchronzation respectively. Note that variable `packet` is only used for value passing during synchronization, and therefore only needs to be set on the corresponding actions.

- `receive_msg`$[s][r]$? with arbitrary $s \in$ `PIDtype` and $r \in \{h_3 \ldots, h_k\}$,
- `send_answer`$[r]$! for $r \in \{h_1, h_2\}$ setting `packet` to an arbitrary value,
- `receive_msg`$[s][r]$! for $s \in \{h_3, \ldots, h_k\}$, arbitrary $r \in$ `PIDtype`, and setting `packet` to an arbitrary value,
- `send_answer`$[r]$? for $r \in \{h_3, \ldots, h_k\}$,
- we do not need an internal (non-synchronizing) action setting `packet` to an arbitrary value, because `packet` is only used to pass a packet on synchronization.

`Chaos1` is shown in figure 11. The array of Booleans `isConcretePID` is used to decide whether a PID is h_1 or h_2, in which case it denotes a concrete host. PIDs $h_3, \ldots, h_k$ denote a host abstracted by `Chaos1`. Array `isConcretePID` is initialized by the `Initializer` automaton. `PIDtype` is used as parameter of hosts H. UPPAAL generates an automaton for every value of `PIDtype`. Therefore, to make sure we only have hosts for h_1 and h_2, we strengthen the guard on the first edges of `Config`, `InputHandler`, and `Regular` to check whether they are initialized with h_1 or h_2. The strengthened guards make sure that in the cases $h_3, \ldots, h_k$, the automata cannot take any edges, and therefore do not play a role in the total system.

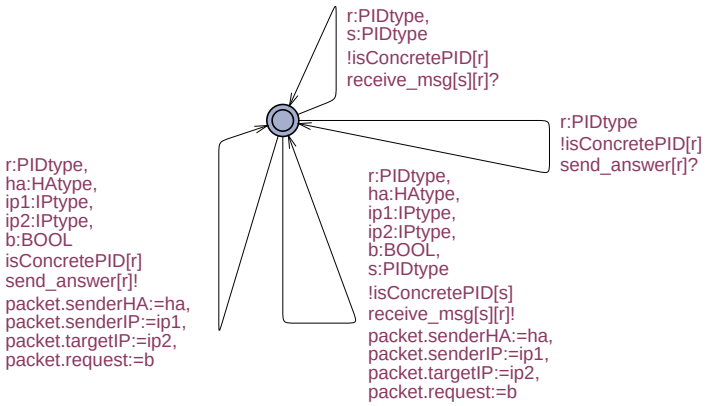

Fig. 11. `Chaos1`

It is clear from the model that messages to $h_3, \ldots, h_k$ will be caught by `Chaos1`, i.e., actions `receive_msg`$[s][r]!$, where $r \in \{h_3, \ldots, h_k\}$ are only possible in `Chaos1`. Moreover `Chaos1` doesn't place any constraints on synchronization, and the synchronization does not affect the behavior of `Chaos1`. We conclude that the actions can be left out. A very easy way to obtain this is by altering the `set` function in `Network`. We let it only set the elements to `true` that correspond to the concrete PIDs h_1 and h_2. In this way elements of `send` and `answer` that correspond to $h_3, \ldots, h_k$ are simply never changed.

Theorem 4 (Correctness of the Abstraction). *The mutual exclusion property for the original model holds, when it holds in:*

$$H(h_1) \| H(h_2) \| \texttt{Chaos1}$$

Proof. We have:

$$\texttt{Chaos1} \succeq H(h_3) \| \cdots \| H(h_k)$$

By applying Theorem 3 we get:

$$H(h_1) \| H(h_2) \| \texttt{Chaos1} \succeq H(h_1) \| \cdots \| H(h_k)$$

The mutual exclusion property is an invariant statement in terms of external variables. From Theorem 2 we conclude that we can model check mutual exclusion on $H(h_1)\|H(h_2)\|$`Chaos1`, and when mutual exclusion holds it will also hold in the original model. QED

4.4 Making Model Checking feasible

Here we will introduce some more abstractions to make the model checking problem tracktable.

Abstraction for Network Action `send_answer`$[j]$! with update `packet` $:= p$, where $j \in$ `Networktype` and $p \in$ `ARP_packet`, is possible in `InputHandler` and `Chaos1`. We want to be able to distinguish whether the answer was generated by one of the two concrete hosts or `Chaos1`. Therefore we add a parameter to the channel such that it becomes `send_answer`$[j][h]$!, where $h \in$ `PIDtype`.

Action `send_answer`$[\cdot]$? was only used in `Network`(h_1), `Network`(h_2), and `Chaos1` on a single transition. The transition needs to be replaced by multiple transitions, where each transition does exactly the same as the original transition, but synchronizes on a different PID. This is easily accommodated by using the UPPAAL select statement h:`PIDtype`. For `Chaos1` these changes lead to `Chaos2`, as shown in Fig. 12.

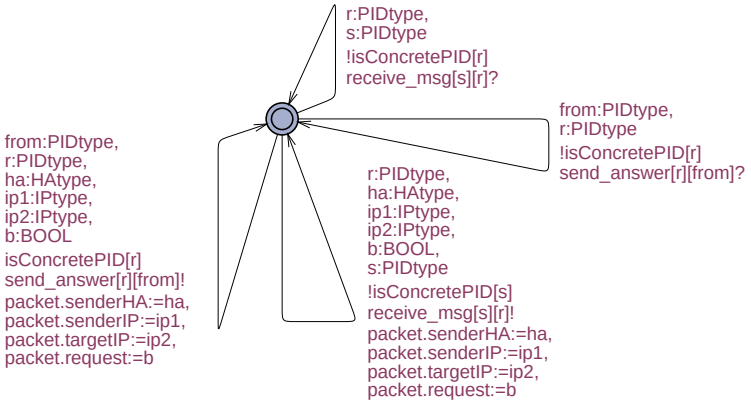

Fig. 12. `Chaos2`

We continue by strengthening the guard on receiving answers in `Network` such that it will only receive answers from concrete hosts $H(h_1)$ and $H(h_2)$. This is easily accomodated by `isConcretePID`. The result is given by `NetConcrAns`, as shown in Fig. 13.

Automaton `AbsAns` will handle the answers from chaos automaton `Chaos2`. `AbsAns` itself is also a chaos automaton, as it can do actions in any order and on

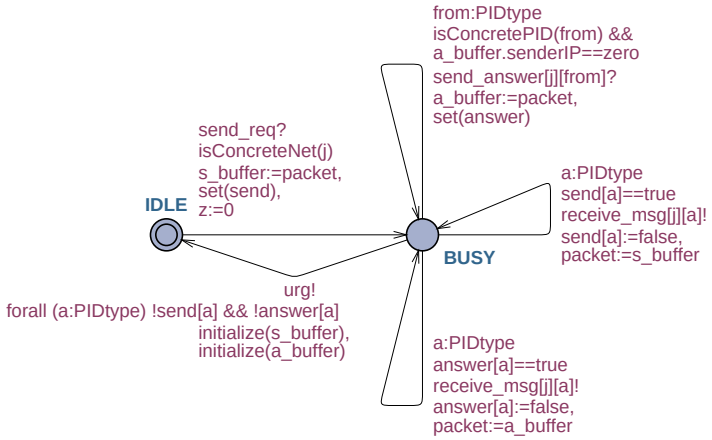

Fig. 13. Timed automaton $\texttt{NetConcrAns}(j)$

any time. Figure 14 shows the Uppaal timed automaton we can use for $\texttt{AbsAns}(j)$.

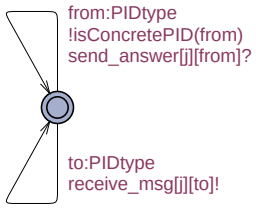

Fig. 14. Timed automaton $\texttt{AbsAns}(j)$

The next lemma formally states our new abstraction. Most recently, efficient on-line algorithms for solving reachability and safety games based on timed game automata have been put forward [12] and made available within UPPAAL-TIGA. In collaboration with Thomas Chatain, Alexandre David and Kim Larsen, we have been able to use UPPAAL-TIGA to automatically check the timed step simulation.

Lemma 1. *For arbitrary* $j \in \texttt{Networktype}$*:*

$$\texttt{NetConcrAns}(j) \| \texttt{AbsAns}(j) \succeq \texttt{Network}(j).$$

In `Chaos2` we have actions of the form: $\texttt{send\_answer}[j][h]!$, such that $h \in \{h_3, \ldots, h_k\}$. $\texttt{AbsAns}(h_1)$ and $\texttt{AbsAns}(h_2)$ are the only automata that can synchronize on these actions. As these synchronizations do not change the system

state we may as well delete all participating edges, both sending and receiving actions. For Fig. 14 this means the upper arc is removed. For `Chaos2` in Fig. 12 this means the arc on the left is removed. The resulting automaton `Chaos3` is the final one used for verification and is shown in Fig. 15. Action `send_answer`$[j][h]?$, which is only present in `NetConcrAns`, will only be parametrized with $h = h_1$ or $h = h_2$. Moreover, if `send_answer`$[j][h_1]?$ is possible, `send_answer`$[j][h_2]?$ is also possible and vice versa. We may as well return to our original situation where `send_answer` only has the first parameter, thus we may as well use again `Network` instead of `NetConcrAns`. We can easily see that:

$$\texttt{Chaos3} \succeq \texttt{AbsAns}(h_1) \| \texttt{AbsAns}(h_2).$$

Therefore we can as well leave out `AbsAns`(h_1) and `AbsAns`(h_2) in our abstract system. Recapitulating, the total model becomes:

$$H(h_1) \| H(h_2) \| \texttt{Chaos3},$$

where the only thing that has changed with respect to the previous model is `Chaos3`, which has no actions `send_answer`$[\cdot][\cdot]!$.

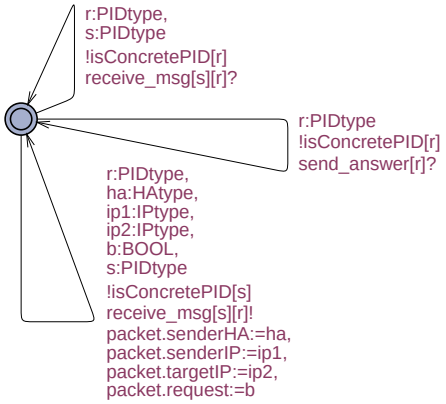

Fig. 15. `Chaos3`

Overapproximation By weakening guards, weakening invariants, or by making an urgent channel non-urgent, we add behavior to an automaton. The old behavior with the same values for the variables is still present. Adding more behavior to some automaton A using these methods will give an automaton B which simulates A, thus $B \succeq A$, according to Definition 3 (timed step simulation.) Using Theorem 2, if we manage to prove an invariant for the larger ("overapproximated") automaton B it will certainly hold for the smaller, original automaton A. As timed step simulation is compositional (Th. 3), weakening

one automaton of the model will give a total model that is an overapproximation of the original.

If, as a result of weakening, a variable is tested in none of the transitions and it is also not read by the context, it can be safely omitted from the model, which can again be characterized by timed step simulation. A variable is read by the context if it occurs in the property we are trying to prove, or is used by another automaton. In the case of Zeroconf, overapproximation and subsequent variable elimination can be applied in the following two situations:

1. We may weaken the guards of the two transitions from `COLLISION` to `INIT` in `Config(j)` to `true`, and remove the transition label `urg!`. In the resulting model local variable `ConflictNum` is no longer used and so we can eliminate it.
2. We may weaken the guard of the left `receive_msg[j]?` transition in automaton `InputHandler(j)` to `true`. In the resulting model local clock `y` is no longer used and it can eliminate it.

The basic idea behind abstractions (1) and (2) is that Zeroconf ensures mutual exclusion even when a host is allowed to always immediately select a new IP address after a reset, and may always defend the IP address that it is using. We continue to apply yet another technique. The final automata for `Config` and `InputHandler` are shown in Figures. 16 and 17.

Dead Variable Reduction Dead variable reduction is a well known static analysis technique, that has for instance been studied in the PhD thesis of Yorav [36]. In Yorav's terminology, a variable v is *used* in a transition if it appears in the guard or in the right hand side of an assignment. 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, i.e., they are strongly bisimilar, which in turn implies they simulate each other via timed step simulation. Setting variables as soon as they become dead to some default value will in general reduce the state space, because states that only differ in their dead variables will now become exactly the same.

In our Zeroconf model, variable `counter` of `Config(j)` is dead in locations `COLLISSION` and `INIT`. Hence, setting `counter:=0` upon occurrence of a reset transition will not affect whether the ME property holds or not. Another example is the variable `network`, which is dead in location `IDLE` of `InputHandler(j)`, and can be reset to a standard value. To have a standard value available we have defined global constant `Networktype net0`.

Only one InputHandler automaton Consider automaton `InputHandler`. The transition from `IDLE` to `BUSY` is never performed from a system state that is

committed. Therefore at most one automaton will be in location `BUSY`. Thus we may as well use only one automaton in our model. But `InputHandler` needs to know the PID of the host it is working for. This PID can be derived in the first transition and stored in a local variable.

The final abstract model We conclude this section by the final abstract model for the Zeroconf Protocol, and the verification results. With respect to the concrete model, timed automata `Regular` and `Initializer` have not changed. Figures 16 and 17 show the new versions of UPPAAL timed automata.

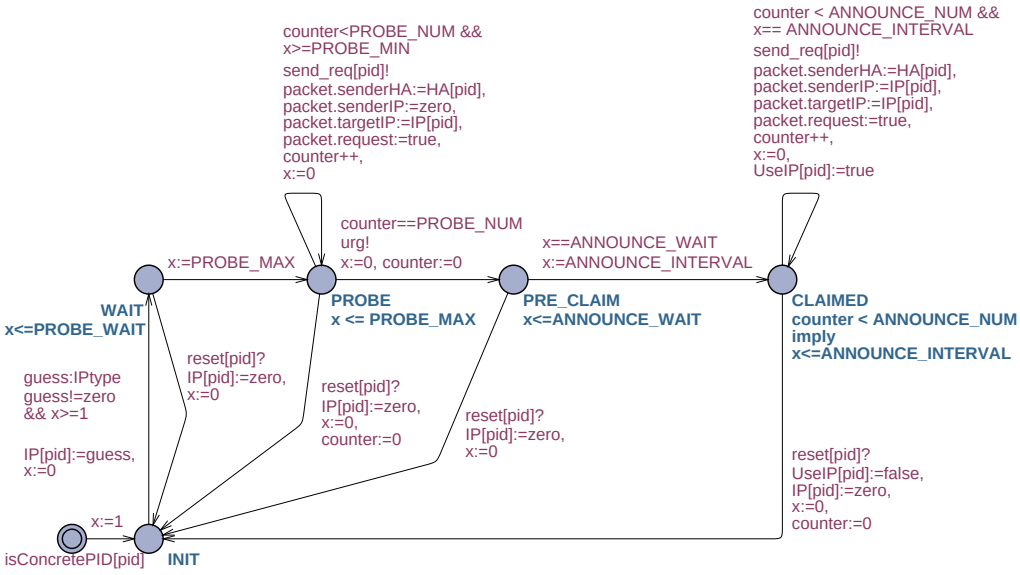

Fig. 16. Final abstract timed automaton `Config`(h)

4.5 Verification Results

We have been able to establish mutual exclusion for a system with an *arbitrary number of hosts, hardware addresses, and IP addresses.* We managed to check ME for for an arbitrary number of hosts, due to the assumption that hosts are idle for at least 1 second after a reset, and due to our abstraction of hosts by one chaos automaton. We managed to check ME for an arbitrary number of PIDs, hardware addresses, and IP addresses, due to a theorem of Ip and Dill [24] on *data saturation*. This theorem (which was proved in the setting of Murphi but can easily be shown to carry over to UPPAAL) states that for certain ("data") scalarsets, the state graph does not grow any further once the size of some

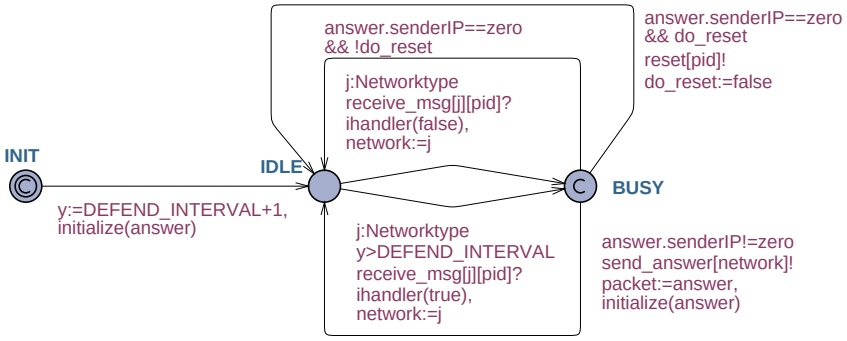

Fig. 17. Final abstract timed automaton `InputHandler`(h)

scalarset grows beyond the number of places in the system where that scalarset is used. This makes model checking with scalarsets of arbitrary size possible.

In our case, at the global level, we need 2 PIDs because we need to have two hosts for our abstract model to work. Furthermore, a select statement has the ability to choose a PID different from these two, which makes a difference for exectution, and therfore we need a total of at most 3 PIDs. PIDs are also used as index of the arrays `sent` and `replied` but as discussed only the elements in `sent` and `replied` that correspond to the concrete hosts are changed. Having more than two PIDs will not have an impact on the possible values for `sent` and `replied`.

For hardware addresses, at the global level there are:

- 2 hardware addresses in use in array `HA` (one per host).
- 1 hardware address is in use by `nil`.
- 1 hardware address is in use by `InputHandler`
- 2 by `Network` since packets use one hardware address.

As there are 2 `Network` automata, we get a total of 8 scalarset locations for hardware addresses. Furthermore, a select statement has the ability to choose a hardware address different from the others, which makes a difference for exectution, and therefore we need a total of at most 9 hardware addresses.

For IP addresses, at the global level there are:

- 2 IP addresses are in use in array `IP`.
- 1 hardware address is in use as `zero`.
- 2 IP addresses are in use by `InputHandler` in packet `answer`.
- 4 IP addresses are in use by `Network`, namely 2 in each of the two packets.

As there are 2 `Network` automata, the total model has 13 scalarset locations for IP addresses. Furthermore, a select statement has the ability to choose an IP address different from the others, which makes a difference for exectution, and therefore we need a total of at most 14 IP addresses.

Summarizing, all instances of the model consist of all possible combinations of $k \in \{1, \ldots, 3\}$ PIDs, $l \in \{1, \ldots, 9\}$ hardware addresses, and $m \in \{1, \ldots, 14\}$ IP addresses. Model checking all instances took approximately 75 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 ($k = 3, l = 9, m = 14$) uses 692739 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, for instance permitting us to write 0.0.0.0 for the all zero IP address instead of 0.
2. The ability to initialize clock and structure variables, allowing us to eliminate the initial transition in the `InputHandler[pid]` 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 in UPPAAL, as advocated in [20]. 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 [10], 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 [27], 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 five places where RFC 3927 [15] is incomplete/unclear:

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

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 and using UPPAAL-TIGA. 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 only modeled/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 [3]. An obvious challenge is to mechanize all these abstractions using either (an extension of) UPPAAL-TIGA 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 [9, 26] 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. We are grateful to Thomas Chatain, Alexandre David and Kim Larsen, for checking the correctness of one of our abstractions using UPPAAL-TIGA.

References

1. M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.
2. R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
3. J. Bauer. *Analysis of Communication Topologies by Partner Abstraction.* PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2006.
4. G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Third International Conference on the Quantitative Evaluation of SysTems (QEST 2006),* 11-14 September 2006, Riverside, CA, USA, pages 125–126. IEEE Computer Society, 2006.
5. G. Behrmann, A. David, and K.G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.
6. J. Berendsen and F.W. Vaandrager. Compositional abstraction in real-time model checking. Technical Report ICIS–R07027, Institute for Computing and Information Sciences, Radboud University Nijmegen, 2007. Available on-line at `http://www.ita.cs.ru.nl/publications/papers/fvaan/BV07.html`.
7. J. Berendsen and F.W. Vaandrager. Parallel composition in a paper of Jensen, Larsen and Skou is not associative. Technical note available at `http://www.ita.cs.ru.nl/publications/papers/fvaan/BV07.html`, September 2007.

8. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.
9. H. Bohnenkamp, P. van der Stok, H. Hermanss, and F.W. Vaandrager. Cost-optimisation of the IPv4 zeroconf protocol. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2003)*, pages 531–540, Los Alamitos, California, 2003. IEEE Computer Society.
10. E. Brinksma and A. Mader. On verification modelling of embedded systems. Technical Report TR-CTIT-04-03, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, January 2004.
11. G. Bruns and M.G. Staskauskas. Applying formal methods to a protocol standard and its implementations. In *Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 1998),* 20-21 April, 1998, Kyoto, Japan, pages 198–205. IEEE Computer Society, 1998.
12. F. Cassez, A. David, E. Fleury, K.G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, volume 3653 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2005.
13. Th. Chatain, A. David, and K.G. Larsen. Playing games with timed automata, 2007. Submitted.
14. S. Cheshire. Personal communication, February 2006.
15. S. Cheshire, B. Aboba, and E. Guttman. Dynamic configuration of IPv4 link-local addresses (RFC 3927), May 2005. `http://www.ietf.org/rfc/rfc3927.txt`.
16. S. Cheshire and D.H. Steinberg. *Zero Configuration Networking: The Definite Guide*. O'Reilly Media, Inc., 2005.
17. D. Chkliaev, J. Hooman, and E. de Vink. Verification and improvement of the sliding window protocol. In *Proceedings TACAS'03*, pages 113–127. Lecture Notes in Computer Science 2619, Springer-Verlag, 2003.
18. E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proc. CHDL*, pages 15–30, 1993.
19. M.C.A. Devillers, W.O.D. Griffioen, J.M.T Romijn, and F.W. Vaandrager. Verification of a leader election protocol: Formal methods applied to IEEE 1394. *Formal Methods in System Design*, 16(3):307–320, June 2000.
20. B. Gebremichael and F.W. Vaandrager. Specifying urgency in timed I/O automata. In *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005),* Koblenz, Germany, September 5-9, 2005, pages 64–73. IEEE Computer Society, 2005.
21. B. Gebremichael, F.W. Vaandrager, and M. Zhang. 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, pages 242–251. ACM Press, 2006.
22. M. Hendriks, G. Behrmann, K.G. Larsen, P. Niebert, and F.W. Vaandrager. 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, volume 2791 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
23. G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2004.

24. C.N. Ip and D.L. Dill. Better verification through symmetry. In David Agnew, Luc J. M. Claesen, and Raul Camposano, editors, *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, volume A-32 of *IFIP Transactions*, pages 97–111. North-Holland, 1993.
25. H.E. Jensen, K.G. Larsen, and A. Skou. Scaling up Uppaal: Automatic verification of real-time systems using compositionality and abstraction. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium, FTRTFT 2000,* Pune, India, September 20-22*, Proceedings*, volume 1926 of *Lecture Notes in Computer Science*, pages 19–30. Springer, 2000.
26. M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. Performance analysis of probabilistic timed automata using digital clocks. In K. Larsen and P. Niebert, editors, *Proc. Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, volume 2791 of *LNCS*, pages 105–120. Springer-Verlag, 2003.
27. M.Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 2.0: A tool for probabilistic model checking. In *Proceedings of the 1st International Conference on Quantitative Evaluation of Systems (QEST04)*, pages 322–323. IEEE Computer Society, 2004.
28. I. van Langevelde, J.M.T. Romijn, and N. Goga. 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, April 2003.
29. K.G. Larsen, M. Mikucionis, and B. Nielsen. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *the 5th ACM International Conference on Embedded Software*, pages 299 – 306. ACM Press New York, NY, USA, September 18–22 2005.
30. N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Fransisco, California, 1996.
31. R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.
32. D.C. Plummer. An Ethernet address resolution protocol (RFC 826), November 1982. `http://www.ietf.org/rfc/rfc826.txt`.
33. J.M.T. Romijn. Improving the quality of protocol standards: Correcting IEEE 1394.1 FireWire net update. *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*, 8:23–30, 2004. Available at `http://www.win.tue.nl/oas/index.html?iqps/`.
34. M. Stoelinga. Fun with FireWire: A comparative study of formal verification methods applied to the IEEE 1394 root contention protocol. *Formal Aspects of Computing Journal*, 14(3):328–337, 2003.
35. F.W. Vaandrager and A.L. de Groot. Analysis of a biphase mark protocol with Uppaal and PVS. *Formal Aspects of Computing Journal*, 18(4):433–458, December 2006.
36. K. Yorav. *Exploiting Syntactic Structure for Automatic Verification*. PhD thesis, The Technion, Israel Insitute of Technology, June 2000.