Modeling and Validating Distributed Autoconfiguration of Mobile Ad hoc Networks using Coloured Petri Nets

by

Brian Pedersen

Master's Thesis

University of Aarhus
Faculty of Science
Department of Computer Science
Denmark

Advisor: Lars M. Kristensen

Århus, 2007

Preface

Abstract

A mobile ad hoc network (MANET) is a wireless, multi-hop network designed to operate without existing communication infrastructure, and its topology may change rapidly due to high mobility of its members. MANETs are *self-configuring* and *self-healing* networks, whose members discover other devices automatic. Users of a traditional cabled or wireless network enjoy the benefits of automatic configuration via, e.g., IPv6 Stateless Address Autoconfiguration. This process known as *autoconfiguration* must be an integrated part of mobile ad hoc networks. This thesis presents a formal verification of a proposal for distributed autoconfiguration in Mobile IPv6 ad hoc networks. A model has been constructed using Coloured Petri Nets that captures the main aspects of the specification. Coloured Petri Nets (CPN) is a graphical modeling language designed for validation of distributed and concurrent systems. Application areas for CPN include network protocols, distributed algorithms and embedded systems. Prior to engaging in a formal verification, simulations were conducted on the constructed CPN model, and documented as message sequence charts. Standard state space analysis was applied to the constructed model, and formed the base for a formal verification. Customized query functions investigated properties specific to the constructed model. The simulation results and state space analysis helped identifying problematic scenarios.

Resumé

Et mobile ad hoc netværk (MANET) er et trådløst, multi-hop netværk designet til at operere uden eksisterende kommunikationsinfrastruktur, hvis netværkstopologi kan ændres hurtigt grundet høj mobilitet af dets medlemmer. Mobile ad hoc netværk er selv-konfigurerende og selv-helende netværk, hvis medlemmer opdager andre brugere automatisk. Brugere af traditionelle netværk, trådløst eller ej, kan drage fordel af automatisk konfiguration via f.eks. IPv6 Stateless Address Autoconfiguration. Denne proces, kendt som *autokonfigurering*, skal ogsåvære en del af mobile ad hoc netværk. Dette speciale præsenterer en formel verifikation af distribueret autokonfiguration af mobile IPv6 ad hoc netværk. En model er konstrueret vha. Farvede Petri Net (CPN) som indfanger hovedtrækkene i specifikationen. Farvede Petri Net er et grafisk modelleringssprog designet til validering af distribuerede og samtidige systemer. Anvendelsesområder for CPN inkluderer netværksprotokoller, distribuerede algoritmer og indkapslede systemer (embedded systems). Simuleringer af den konstruerede model blev foretaget inden den formelle verifikation indledtes, og dokumenteret som *message sequence charts*. Hjørnestenen til den formelle verifikation blev lagt via grundlæggende state space analyse af den konstruerede model. Specialfremstillede funktioner undersøgte de modelspecifikke egenskaber. Simuleringerne og state space analysen gav anledning til identifikation af problematiske scenarier.

Acknowledgements

I would like to express my sincere gratitude towards my thesis advisor, Lars Michael Kristensen, for his continuous guidance during the writing of this thesis. Throughout the grueling process of completing the thesis, he provided constructive criticism to help me steer it in the right direction. Also for lending me the processing power to complete the analysis in a resonantly amount of time.

To my study group throughout the majority of my studies, Gabriel Siegel, Martin Jørgensen and Bjarke Laustsen, with whom I share a lot of memories. Without their help and support I would not have come this far. Thank you.

To my friends, on and off campus, and family for taking my mind of the thesis work every now and then. Finally, I would like to thank my parents for their support, not just through the process of writing this thesis, but my entire education, and life in general.

Brian Pedersen,
Århus, June 21, 2007

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Description	Introduced
AODV	Ad hoc On-demand Distance Vector routing	4
BRITNeY	Basic Real-time Interactive Tool for Net-based animation	89
CPN	Coloured Petri Net	5
DAD	Duplicate Address Detection	10
DHCP	Dynamic Host Configuration Protocol	9
EUI-64	Extended Universal Identifier (64 bits)	20
IEEE	Institute of Electrical and Electronics Engineers	3
IPv4	Internet Protocol version 4	2
IPv6	Internet Protocol version 6	2
IPv6 SAA	IPv6 Stateless Address Autoconfiguration	9
MAC	Media Access Control	31
MANET	Mobile Ad hoc Network	2
MSC	Message Sequence Chart	88
NA	Neighbor Advertisement	21
NDP	Neighbor Discovery Protocol	21
NS	Neighbor Solicitation	21
RARP	Reverse Address Resolution Protocol	9
UUID	Universally Unique ID	15

Chapter 1

Introduction

With the increasing number of portable devices capable of connecting wirelessly to the Internet and/or other devices, a greater demand for wireless accessibility needs to be met. Traditional computer networks, wireless or cabled, requires existing infrastructure such as routers and switches in order to operate. Being able to communicate with other devices in areas without existing network infrastructure, or even power, could be vital. Rescue missions in a disaster area could be one such scenario. Communication between paramedics, police officers, firefighters, and hospitals is vital to rescue as many victims as possible. If the disaster has wiped out the entire communication infrastructure, this could become problematic. Using handheld devices (e.g. laptops and PDAs) could provide the necessary communication if they were used to form a network.

A mobile ad hoc network[2] could be the answer to this problem. Mobile ad hoc networks are characterized by being autonomous systems of mobile devices connected through wireless interfaces. They are designed to work in unpredictable environments with high mobility of its users, and without existing communication infrastructure. The networks are *self-configuring* and *self-healing*, whose members discover other devices automatic. Mobile ad hoc networks are capable of connecting to external networks, e.g., the Internet. Devices called *gateways* provide the link between the Internet and the rest of the mobile ad hoc network. These gateways need not to be specialized hardware, but can be any member of the network sufficiently close to an Internet connection. Before a device can take advantage of an Internet connection, it must know how to contact the gateway(s). The process of obtaining information about the gateway is called *gateway discovery*.

1.1 Mobile Ad hoc Networks

A mobile ad hoc network (MANET) has four distinct characteristics[3]:

1. MANETs are autonomous systems with high mobility of the individual *nodes* (e.g. computers, PDAs, and mobile phones) that constitute the network. Nodes are free to move arbitrarily hence their connection point may change rapidly. This means that there is no fixed topology of the network.

2. Even though the transmission speed of wireless network equipment grows rapidly, the typical wireless connection has a lower bandwidth than their hardwired counterpart. Noise and other kinds of interference are more common in wireless communication as opposed to using cabled network connections.

3. When using any kind of handheld device, the battery level is of great concern. An important criterion for designing MANET mechanisms is to optimize for battery conservation.

4. Due to the increasing number of devices capable of connecting to other devices and/or the Internet, any new technology should be able to cope with scalability.

With regard to the scalability, most new protocols and other mechanisms developed today, exist in an IPv4 [4] and IPv6 [5] version, and this thesis will focus on the latter. The amount of available addresses in IPv6 provides scalability regarding the possible number of concurrent devices in a MANET.

A mobile ad hoc network is also characterized by the nodes willingness to forward data packets on behalf of others. Figure 1.1 depicts a situation in which node A wants to communicate with node B, but is not within direct wireless range, as indicated by the circles representing the transmission range. In this scenario a node, say C, might be within reach of both A and B, and will forward packets between A and B.

If a node is within communication range of two otherwise separated MANETs, these two networks become connected (or *merged*) by means of this intervening node. On the other hand, if this node moves outside the range of its two closest neighbors, these two networks become disconnected (or *partitioned*). Figure 1.2 illustrates a scenario where the green node is sufficiently close to nodes in Network1 and Network2, in order to merge them into a single network.

A MANET can be connected to other external networks using a *gateway*. Any node positioned close enough to an Internet connection, can act as a gateway. Gateways do not have to be a computer in the traditional sense, but can be small

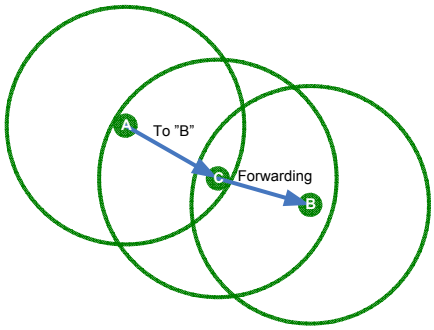

Figure 1.1: Node C forward packets to B on behalf of A.

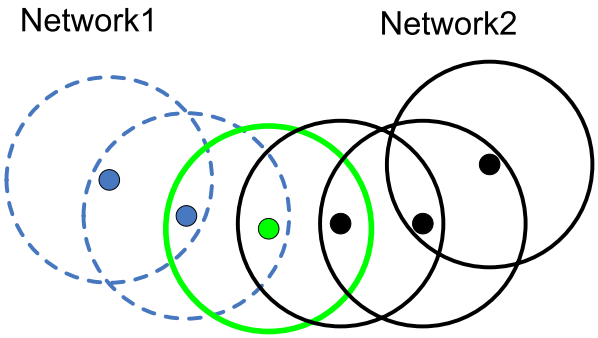

Figure 1.2: Merging two separated MANETs.

devices such as mobile phones or PDAs with very limited computational power. MANETs that are connected to an external network are referred to as *hybrid*, while MANETs without external connections are called *isolated*.

An isolated MANET is a network that is setup among various wireless devices all located in the same geographical area, thus an obvious use is *home networking*. Due to the high number of Internet connections available at home, this home network could be connected to the Internet via a gateway, thereby characterized as hybrid. The example mentioned in the introduction where paramedics, firefighters, and police officers are forming an emergency network at an accident site, is another application of MANETs. Other examples include military battlefield communication, conferencing away from the office infrastructure, electronic surveillance connecting video cameras, alarms, touch sensors and guards in an ad hoc fashion.
The main part of the above-mentioned usage of MANETs can be applied in a hybrid fashion. Connecting a MANET to an external network such as a cellular Wide Area Networks (e.g. GSM or GRPS) or a Wireless Local Area Network (e.g. IEEE 802.11) can be achieved via gateways. By using strategically placed

gateways, wireless Internet connections can be extended into sparsely populated areas with an otherwise poor or non-existing communication infrastructure.

1.2 Autoconfiguration

Connecting to any type of network requires *configuration*, before being able to communicate. The possibility of a rapidly changing topology in case of a mobile ad hoc network increases the probability of switching network connection during node movement. Whenever a user moves from one network to another, a reconfiguration is needed. Users of a traditional network enjoy the benefits of automatic configuration (using the Dynamic Host Configuration Protocol[6] or IPv6 Stateless Address Autoconfiguration[7]), and this process, denoted *autoconfiguration*, must be an integrated part of MANETs.

There are various ways of performing autoconfiguration, depending on the type of MANET (isolated or hybrid). The overall strategy of autoconfiguration is to generate an IP address, and assign it to a network interface provided that others do not occupy the address. These addresses are often only routable within the network itself, and extra configuration is required to obtain a globally reachable IP address.

Global connectivity in MANETs is provided by nodes acting as gateways. Before non-gateway nodes can benefit from the Internet connection, they must establish a route to the gateway. The process of obtaining information about the gateway is called *gateway discovery*. A node discovers the gateway by receiving advertisements containing the necessary information. There are two overall methods for gateway discovery; the gateway periodically distributes the information in advertisements or nodes requests information when needed. The latter is often used when the MANET implements an on-demand routing protocol such as the Ad hoc On-Demand distance vector protocol (AODV[8]). Rather than inventing a completely new protocol for gateway discovery, an existing protocol can be extended. The Neighbor Discovery Protocol[9] can be extended with a new message type providing the necessary information. A proposal for autoconfiguration using an extended neighbor discovery protocol is treated in section 2.2.2.

1.3 Coloured Petri Nets

Implementing an autoconfiguration mechanism is a massive task, and requires a vast amount of resources. Therefore, ensuring the correctness of this *before* the implementation phase begins, is vital. Constructing a model that reflects the specification is one way of analyzing a system. Using this model to verify

correctness of the mechanism, could potentially save time and money, should any defects be discovered.

Coloured Petri Nets (CPN[10] or CP-nets) is a graphical modeling language designed for validation of distributed and concurrent systems. Examples of application areas for CPN include network protocols, distributed algorithms and embedded systems. CPN combines Petri Nets[11] with a high-level programming language. The high-level programming language behind CPN is CPN ML, which is based on Standard ML[12].

CPN Tools[13] is a tool for modeling and validating CP-nets, and is used for constructing the model for this thesis. Without such tools for creation and verification of CPN models, analysis would have to be conducted by hand. Conducting this analysis by hand is very tedious, time consuming, and error prone. After completing the verification of the system, other aspects of the mechanism might be of interest. This includes experiments investigating the performance of the mechanism e.g. how fast can we configure a new device? What is the average setup time? Does the mechanism scale? By using a tool only supporting the logical aspect of model verification such as Alloy[14], a separate model supporting timing must be constructed. CPN Tools has one huge advantage that tools only for logical verification do not have, namely the possibility to extend the model with timing. This time concept enables us to conduct performance analysis by investigating properties such as delay, queue length, and throughput. With a little effort, compared to create a new model from scratch, a CPN model is easily extended to support timed analysis.

A CPN model describes the *states* of a system, and the events (*transitions*) that cause the system to change state. Complicated CPN models are often constructed with a number of modules interconnected in a hierarchy. Building this hierarchy improves the readability of the model because a large portion of the model can be abstracted away in e.g. a *send packets*- or *handle packets*-module. Each module may contain sub modules, which gives a hierarchical structure. By conducting simulations of the constructed model, it is possible to investigate the behavior of the system. Simulation can be conducted in one of two ways; interactively or automatically, where the former is equivalent to single step debugging known from conventional programming. The automatic simulation keeps selecting events until no more events are enabled.

Even though CPN models are graphical representations of a system, they have mathematical definitions of syntax and semantics. These definitions can be used as a basis to verify various system properties, i.e., proving or disproving certain

properties.
Verification of system properties uses the idea of a *state space* graph. The idea behind state spaces is to compute all reachable states and state changes of the CPN model. Using this state space graph, we can answer a number of verification questions such as:

- Are there any deadlocks?
- Can we reach state X?
- If we terminate, did we terminate in the desired state?

By calculating a state space, we can obtain counter examples if certain properties are violated, providing the path of execution for reaching these undesirable states.

1.4 Thesis aim

This thesis aims at proving or disproving the correctness of distributed IPv6 autoconfiguration of Mobile Ad hoc Network nodes, as described by Lee et al.[15] using Coloured Petri Nets. The constructed model is analyzed using standard facilities in CPN Tools such as state spaces and strongly connected component graphs, and will form the basis of the verification. Customized query functions will aid the exploration of model specific properties. Communicating through a wireless network requires various network related mechanisms such as routing protocols and handling security issues. This thesis, however, focuses on configuration of nodes, and modeling these auxiliary mechanisms is out of scope. Section 4.1 describes the assumptions and simplifications made in greater detail.

1.5 Structure of this thesis

This thesis is organized into seven chapters:

Chapter 1: Introduction gives an introduction to this thesis and a brief introduction to the fundamental background material.

Chapter 2: Autoconfiguration in mobile ad hoc networks describes autoconfiguration for mobile ad hoc networks in more detail. It presents an overview of five proposals to autoconfiguration all having different properties and working domains.

Chapter 3: Distributed IPv6 Addressing Technique for MANETs is a detailed description of the autoconfiguration proposal[15] subject for validation

in this thesis.

Chapter 4: Modeling distributed autoconfiguration presents basic terminology regarding CPN modeling, and provides a detailed description of the constructed CPN model.

Chapter 5: Validation by means of simulation introduces six scenarios in which the model has been tested. These simulations form a starting point for the formal verification presented in chapter 6.

Chapter 6: State space analysis presents the formal verification of the constructed model. It introduces the basic concepts of state space analysis.

Chapter 7: Conclusion summarizes the results of the thesis work. It draws conclusions about the analysis, and discusses future work.

1.6 Reader requirements and guide

This thesis deals with a specialized area of IPv6 networking, but detailed knowledge about autoconfiguration and Mobile Ad hoc Networks in general is not required. However, the reader is assumed to be familiar with the basic concepts of networking, including fundamental knowledge of IPv6. The basic concepts of Coloured Petri Nets will be explained throughout chapter 4, and no prior knowledge of CPN and analysis hereof is required.

For readers with no or limited knowledge of mobile ad hoc networks, it is recommended to read chapter 1, which introduces basic concepts. Chapter 2 introduces several suggestions on autoconfiguration, and can be skipped by readers who are not interested in different views on autoconfiguration. If the reader does not have solid knowledge of the autoconfiguration scheme presented by Lee et al.[15], it is recommended to read chapter 3. Readers with knowledge of autoconfiguration of MANETs and CPN concepts can read the chapters in any order.

Chapter 2

Autoconfiguration in mobile ad hoc networks

Before a device in any type of network can communicate with other devices, it must be configured properly. The most basic part of a configuration for computers attached to a TCP/IP network is an *IP address*. Additional information needed includes router address, subnet mask, and name server address. The process of configuring a device without user interaction is known as *autoconfiguration*. Traditional IPv4 computer networking has a number of different ways for configuring a device. The *Reverse Address Resolution Protocol* (RARP[16]) operates by broadcasting a request to a server, and waits until it receives a response. Another server-based autoconfiguration scheme is the Dynamic Host Configuration Protocol (DHCPv4[17]), which allows for more information to be received in a single message, than its predecessor. IPv6 defines *Stateless Address Autoconfiguration* (SAA[7]), which is used instead of RARP to obtain the necessary information. A host sends a link-local request via multicast to receive its configuration parameters. If stateless autoconfiguration is not supported, a host can utilize the IPv6 version of DHCP (DHCPv6[6]). Due to the properties of a MANET, stateless IPv6 autoconfiguration cannot be adopted in its original form. The multi-hop nature of a MANET prevents the SAA protocol from working mainly because there is no well-defined notation of a common link. There are, however, two interpretations of "being on-link" in a MANET setting[18]. The first interpretation only allows a link-local address to be valid in a *one-hop range*. Another way of interpreting the scope of a link-local address is *MANET wide*, allowing nodes to communicate through the link-local address if they belong to the same MANET. This thesis adopts the first interpretation of link-local address scope. Choosing the other interpretation could break the interoperability between a node in the MANET and an exterior node.

Many suggestions for autoconfiguration have been made, all with different advantages, disadvantages, and properties. All proposals share one common element, namely the management of a pool of addresses or *the address space*. Managing the address space consists of assigning addresses to nodes, and handling complications following network partitioning and merging. Pre-assigning addresses for MANET-use before a device leaves the factory is not feasible. This is due to the mobility of each individual node causing frequent address changes.

Using a centralized server implementing DHCPv6 for address space management is not very common. There are, however, autoconfiguration mechanisms based on DHCP[19]. Regardless of autoconfiguration proposition, MANET nodes should carry a common prefix used within a particular network to ensure scalability. If a common prefix is not used throughout the MANET, individual node must store a potentially large data structure, informing the node whether or not a given destination resides within the MANET.

Many papers on autoconfiguration for MANETs borrow their idea from stateless address autoconfiguration for IPv6. Some make minor or major modifications to SAA, while others just use the SAA-skeleton. SAA divide the task into the following three steps or phases:

1. Construction of a link-local address.

2. Check if the address is in use (Duplicate Address Detection).

3. Create a global and site-local address.

There are various ways of constructing a link-local address. Some suggestions combine a prefix with an (unique) identifier, while others use a probability function to construct an IP address. Determining uniqueness of the generated address is usually the task of a Duplicate Address Detection (DAD) algorithm. The main idea behind DAD is to send a message to the generated address, and listen for any replies. If a reply is received, the address is in use, and another must be generated. A more detailed description of the DAD procedure used in Lee et al.[15] can be found in section 3.3. The global and site-local addresses are typically based on the unique link-local address.

Autoconfiguration for MANETs can be divided into two main categories. Propositions using fixed "servers" for managing the address space are called *stateful*, while those not relying on a centralized entity are called *stateless*. Section 2.1 and 2.2 provides an overview of various autoconfiguration propositions for isolated and hybrid MANETs.

2.1 Autoconfiguration in isolated MANETs

The requirements for configuring nodes in isolated MANETs are less strict than the hybrid counterpart. Configuring a globally routable IP address is not necessary in isolated MANETs, as they have no connection to external networks. The assigned IP addresses needs, however, to be unique within the MANET, even after merging multiple independent networks. Address conflicts may occur after merging independently configured networks, and the autoconfiguration scheme must cope with these.

Three approaches for autoconfiguration in isolated MANETs are discussed below, each having their own properties.

2.1.1 MANETconf: Configuration of Hosts in a MANET

MANETconf[20] is a proposal for configuring nodes in isolated MANETs supporting partitioning and merging.

Overview

When a node n_r, called the *requester*, enters an established MANET, it chooses a reachable node n_i (the *initiator*) that performs address configuration on behalf of n_r. n_i chooses an unallocated address X as a tentative address, and attempts to acquire permission from all other nodes in the MANET to assign X to n_r. This message is flooded throughout the network to ensure all nodes receive the request. Each receiving node either grants or denies permission to use the tentative address, and replies to n_i. Only if all replies are "positive", the initiator can assign X on a more permanent basis to n_r.
Leaving the MANET in a graceful manner is also supported by this proposal. When a node wishes to leave a network, it broadcasts an *address cleanup* message, stating that it wants to leave. Each node receiving this message updates its knowledge about addresses in use i.e. the address is released.

The question is why does the requester n_r need a configured node to perform address assignment on its behalf? One of the reasons is that the newly arrived node n_r does not yet have an IP address. This means that if it moves around, and thereby change its connectivity, the other nodes does not know how to forward the address reply to n_r. Another reason is that if n_r were to broadcast its choice of IP on the link, any replies must be broadcast or multicast on that same link. However, MANETs are multi-hop networks, and not all nodes can communicate with n_r through link-layer broadcast, thus not all replies would be received. Furthermore, there are no entries in the routing tables to help them forward responses to n_r.

Joining the MANET

To get a better understanding of how this proposal works, a small example is presented below containing three configures nodes marked n_i, m_1 and m_2. Each node is assumed to have an IP address denoted by the node name, e.g., node m_1 has IP address m_1 and so on.
Figure 2.1 shows a new node n_r joining an existing MANET, and broadcasts a *neighbor query* (NQ). If multiple neighbor replies are received, n_r chooses a node n_i as initiator and discards all subsequent replies.

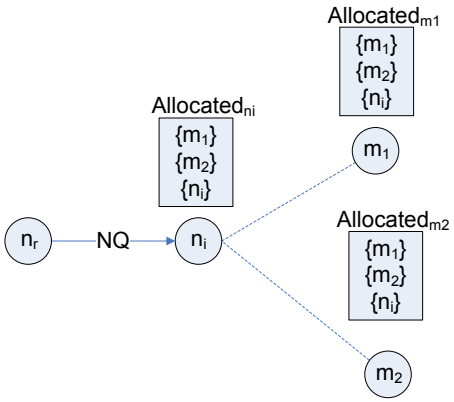

Figure 2.1: Node n_r broadcasts a neighbor query (NQ), which is received by initiator n_i.

Each configured node maintains the following data structures:

- $Allocated_{n_j}$: the set of known allocated addresses as seen from n_j.
- $AllocatedPending_{n_j}$: the set of IP addresses, which has been reserved, but not yet allocated. Again this is seen from node n_j's perspective.

The entries in $AllocatedPending_{n_j}$ have the form $\{address, initiator\}$, and has an associated validity period (not shown on figure 2.1). Adding the initiators IP address to *AllocatedPending* can help sorting priority issues in case of concurrent attempts to assign the same address. If two different initiators attempt to use an address, the initiator with the *lowest* IP address gets highest priority. Assume that recipient n_k has entry (x, l) in $AllocatedPending_{n_k}$, and it receives an address request (x, j). If $l < j$ then node n_l has highest priority, and a negative reply is sent to n_j. Upon reception of a neighbor query, the initiator n_i selects an address m_3 not present in $Allocated_{n_i}$ nor $AllocatedPending_{n_i}$. The initiator adds the entry $\{m_3, n_i\}$ to $AllocatedPending_{n_i}$, and broadcasts an *initiator request* to all other nodes (figure 2.2).

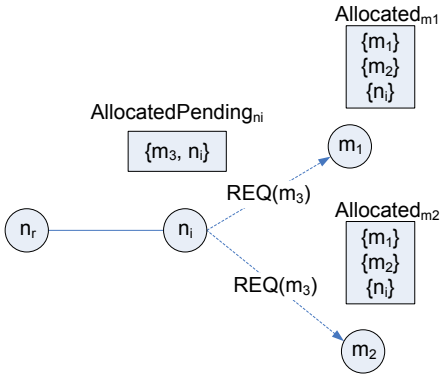

Figure 2.2: n_i broadcasts an initiator request to nodes m_1 and m_2, requesting permission to use m_3.

The receiving nodes m_1 and m_2 checks its *Allocated* and *AllocatedPending* tables in order to grant or deny permission for use of address m_3. Nodes granting permission adds $\{m_3, n_i\}$ to its *AllocatedPending* and sends an affirmative response to the initiator (figure 2.3).

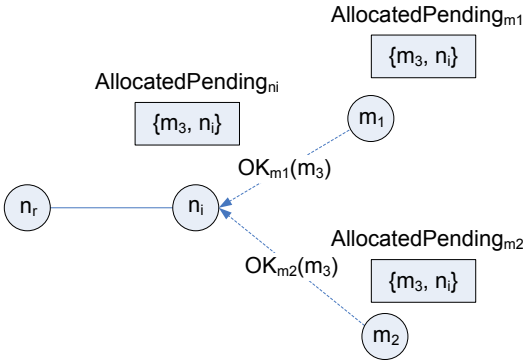

Figure 2.3: Nodes m_1 and m_2 grants permission for use of m_3.

Nodes denying permission will send a negative response. When all replies are received, the initiator must do one of two tasks. Either it must choose another address m_4 in case of at least one negative reply and repeat the process, or it can assign the address to the requester. Assigning the address is a three step procedure where the initiator:

1. assigns the address m_3 to n_r
2. adds m_3 to $Allocated_{n_i}$
3. floods the information to the MANET nodes m_k, in order for them to move $\{m_3, n_i\}$ from $AllocationPending_{m_k}$ to $Allocated_{m_k}$.

The three steps of assigning the address to n_r is shown in figure 2.4 and 2.5.

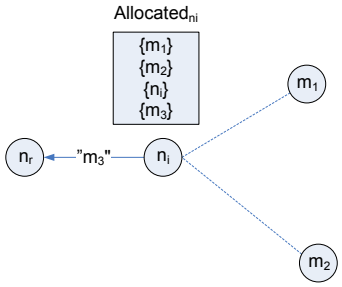

Figure 2.4: n_i sends address m_3 to n_r and updates its table.

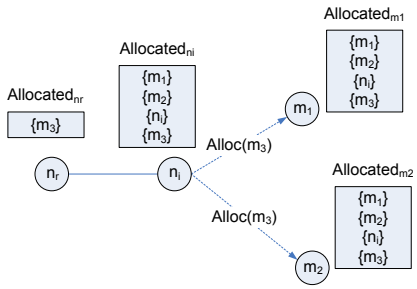

Figure 2.5: Nodes m_1 and m_2 updates its table.

A series of improvements can make the proposal more robust. The use of *address allocation timers* will ensure that if an initiator node crashes before the address is assigned, the requester selects another initiator. Using timers to detect abruptly departed nodes is also a part of the suggestion. Message loss due to network failure is compensated by retransmission a fixed number of times. If a node n_c does not receive a flooded message from initiator n_i, it cannot update its tables accordingly. An address x is not added to the $AllocatedPending_{n_c}$, and is absent from $Allocated_{n_c}$ if permission is granted. This is, however, not a problem regarding the existence of duplicate addresses. If n_c is approached by a requester, in principle it could request address x since it is not present in its $AllocatedPending_{n_c}$ or $Allocated_{n_c}$. But at least one other node has received the request message, and has the correct tables. Assuming they are mutually reachable, this node will send a negative reply to n_c, and it must choose another address.

Preventing address leaks

A networking issue also applying to MANETs, is the possibility of loosing messages. If an address cleanup message, indicating that address x is no longer a part of the MANET, is lost, the tables become outdated. A node n_j that did not receive a cleanup message will have address x as part of its $Allocated_{n_j}$ table. An initiator handling a request could use address x as tentative address, and ask for permission to use it for its requester, but n_j denies permission based on outdated information. If we are not dealing with this problem, known as an *address leak*, address x becomes unavailable for allocation. A new data structure, $SeqNum_{n_j}$, is introduced to prevent address leaks. Each IP address is associated with a sequence number to resolve the problem. This data structure contains the sequence number associated with all IP addresses in the range(s) of allowed addresses within the MANET. $SeqNum_{n_j}$ is initialized to all zeros indicating that

no address is allocated. When node n_j knows that an address has been allocated *or* released, the corresponding sequence number is incremented by one. The entry $SeqNum_{n_j}[x]$ represents the sequence number for address x at n_j. An odd value for $SeqNum_{n_j}[x]$ indicates that n_j considers address x as occupied, whereas an even value indicates that x is unallocated. n_j must update its sequence number for address x to the *highest* known value when a message regarding x is received. Each message sent by n_j regarding address x, also contains the sequence number for that particular address. When node n_j receives an initiator request about x, it consults $SeqNum_{n_j}[x]$ to check if the received sequence number is an *even* number greater than its own. If this is true, node n_j's information is outdated and it sends a *positive* reply to the initiator.

Network partitioning and merging

Handling partitioning of a network is relatively simple. Each node in one partition can conclude that all nodes in the other partition(s) have departed abruptly. Due to this abruptly departure, the nodes did not send address cleanup messages, thus they are still a part of the *Allocated*-tables in all partitions. The initiator sending a request to all nodes in its allocated table, associates a timer with this request. If it for some node m does not receive a reply before the timer expires, this node is flagged. The set of flagged nodes will receive a fixed number of retries to ensure that timer expiration is not due to long delays. When all retransmission are completed, and the initiator still has not received a reply, the nodes are considered to be out of reach, and are removed from the *Allocated*-tables.

If two nodes of separate MANETs M_1 and M_2 are sufficiently close to each other, the networks will merge. Merging MANETs raises an interesting question. How can a node n_i in M_1 distinguish if a node n_j has become a direct neighbor within M_1, or if n_j belongs to M_2, leading to a merge? It cannot, unless nodes maintain and exchange additional information.
Each MANET has an ID, which is a tuple $\{lowest\ IP\ address, UUID\}$, where *lowest IP address* is the lowest IP address in use, and *UUID* is a Universally Unique ID . This UUID is proposed by the node n_{low} with the lowest IP in the MANET. Splitting a MANET into two partitions, P_1 and P_2, means that one partition, say P_1, will contain n_{low}, and thereby keep its ID, while the other partition is without an ID. Both partitions perform cleanup, deleting non-reachable nodes from their allocated tables as described above. Partition P_2 determines the node n'_{low} with the lowest IP address, and it selects a new UUID, which is propagated throughout P_2.

Detecting whether or not partitions of n_i and n_j must be merged, is based on the partition ID. If the partition IDs differ, they merge into a single MANET, otherwise n_i and n_j has become neighbors due to node movement. When merg-

ing partitions, n_i and n_j exchange their *Allocated* table, which is propagated throughout the counterparts partition. All nodes update $Allocated_{n_k}$ to be the union of their current table, and the newly arrived addition. IP addresses that occur in the original table, and the addition are part of a conflict. One of the two *conflicting nodes* must acquire a new IP address using the initiator-requester approach described earlier.

2.1.2 PACMAN: Passive Autoconfiguration for MANETs

PACMAN is an approach for distributed autoconfiguration in mobile ad hoc networks[1].

Overview

While the other proposals discussed in this chapter are categorized as stateless *or* stateful, PACMAN is considered a hybrid approach. Information about the addresses assigned is collected (as in a *stateful* approach), but the nodes are responsible for assigning addresses them selves (as in a *stateless* approach). A low overhead is achieved by detecting address conflicts in passive manner using information derived from the routing protocol traffic.

Architecture

PACMAN utilizes a modular architecture with a *PACMAN manager*, which handles these six components:

- Routing protocol packet parser
- Address en-/decoding
- Address assignment
- Address change management
- Conflict resolution
- Passive duplicate address detection

Figure 2.6 outline the modular architecture of PACMAN, and the connections between components and manager. A brief description of each module is given in this section, and a more detailed description in the following subsections.

The module *Routing protocol packet parser* is a part of the *network layer* in the OSI protocol stack[21]. This module extracts information from the packets and passes it to the manager. The protocol packet parser is not restricted to use a particular routing protocol, but can use any of the available protocols (e.g.

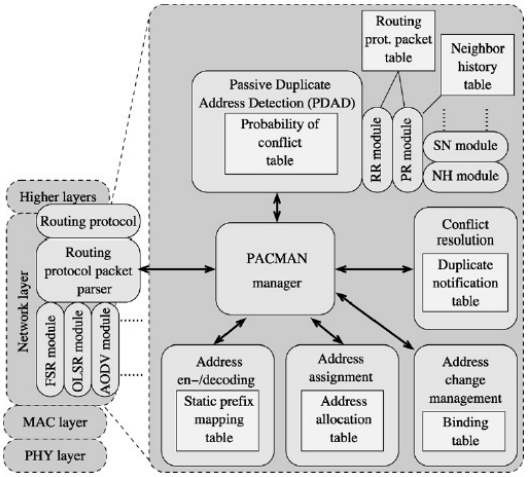

Figure 2.6: The modular architecture of PACMAN[1].

AODV[8] and OLSR[22]). The *address assignment* module selects an address via a probabilistic algorithm (an example is presented later), and is responsible for maintaining a table with addresses in use. Merging two networks could potentially cause an address conflict, meaning that two or more nodes possess the same address. Checking for these address conflicts is the task of the *Passive Duplicate Address Detection (PDAD)* component. Should any conflicts be discovered, the *conflict resolution* component notifies the parties involved. Preventing communication in the transport layer from being broken, the *address change management* module can notify the communication partners about this address change. PACMAN supports an optional reduction in packet size by compressing to variable-sized addresses. This is handled by the *address encoding* component.

Address encoding

Only a fixed number of nodes can be uniquely identified in a network. This is determined by the size of the address space. Using an oversized address space for small MANETs can reduce the throughput on the network. A reduction in address space implies a smaller protocol packet, and thereby a slightly higher throughput (in theory). A variable-sized address space prevents bandwidth wasting by only increasing the size when necessary. To preserve the compatibility with the standard IP addressing architecture, PACMAN encodes the addresses in outgoing packets *below* the network layer. Upon reception, the addresses of incoming packets are decoded to the common IP address format. For communication within a MANET we can use an IP address with the MANET-prefix `FEC0:0:0:FFFF::/64` for IPv6 and `169.254/16` for IPv4. These MANET-addresses are not globally routable, hence uniqueness is only necessary within this particular MANET. If we were using all 64-bits available for address space,

we could potentially create a network with $2^{64} \approx 1.8 \cdot 10^{19}$ individual nodes. This is clearly an overkill. Therefore a variable-sized *virtual address space* is used. Addresses in the virtual address space have a maximum length of 64-bits (128 bits for IPv6 addresses minus the prefix), and the address assignment component chooses from this virtual address space.

Encoding the address by using only a part of the IP address as identifier reduces the packet size. This insinuates that the remaining bits are 0, and can be compressed. One type of address encoding is *run-length coding (RLC)*[23], and can be applied to an address in the following manner. An IPv4 address is used for simplification, but the procedure is the same. Suppose we want to compress the address `169.254.0.13`. The binary representation of this address is

```
10101001 11111110 00000000 00001101
```

where the 16 leftmost bits are the MANET IPv4 prefix, and the last four bits in the rightmost block is the node ID. Each RLC-block has a size of **4-bits**. Applying the RLC returns the address

```
11111101
```

The leftmost bit indicates that the MANET-prefix has been removed, while the next bit informs the decoder-module that RLC has been applied. The next two 1's represent the number of blocks of 0's that has been compressed by the encoding. In this case, the 11 (3 in decimals) indicate that the maximum of three 4-bit blocks of 0's have been compressed. The last four bits is direct copy of the original node ID. The example shows how we can represent a 32-bit address using only 8 bits. In the above example we save 24 bits, and 120 bits in case of the corresponding IPv6 address with each packet sent through the MANET. But compared to the maximum allowed data frame size for packets sent from application layer to the MAC layer in e.g. IEEE 802.11 (max 2304 bytes[24]), this reduction in packets size is minuscule.

Address assignment

Nodes using PACMAN are responsible for assigning their own address, using a probabilistic algorithm to generate this address. The algorithm depends on a predefined conflict probability, an estimation of the network size, and an allocation table, which contains addresses occupied in the network. Based on these values, the algorithm randomly generates an address from the virtual address space. It utilizes the allocation table to ensure uniqueness of this generated address. Assigning the address is carried out immediately, resulting in a fast configuration

time (in the range of milliseconds). There is, however, a probability of an address conflict, but due to the allocation table, this is usually close to zero. A problematic scenario emerge with multiple simultaneously joins, e.g., when merging MANETs, causing the allocation table to be outdated. Any address conflicts that may arise are resolved by the PDAD component. The choice of address compression is a trade off between compression rate and conflict probability. A small virtual address space can quickly result in a high probability of address conflicts. For IPv4 MANETs, the probability is approximately 50% when using 16-bit address space size in a network with 300 nodes.

Address change management

As mentioned in the architectural overview, PACMAN can prevent communication in the transport layer from being broken whenever a conflict is detected. If two nodes are in conflict, at least one has to change its address. On address change, communication in the transport layer can be broken, which is similar to network-layer mobility in Mobile IPv6[25]. To prevent this from happening, PACMAN uses the idea of route optimization from Mobile IPv6. The node that changes its IP address sends a notification to the other conflicting node in form of a *binding update* message. The old address is now used as *home address*. A mobile IPv6 home address is an address, which is assigned to a node upon attachment to the home link. The node is reachable through this address regardless of its location on the network. The conflicting node can then use the routing header to send packets to the new address. These packets are then de-capsulated, and arrive at the transport layer with the old IP destination address.

Passive duplicate address detection

PACMAN uses passive duplicate address detection (PDAD) to resolve any address conflicts. A node analyzes incoming traffic to derive hints about conflicts. The idea behind PDAD is utilizing events in the implemented routing protocol that

1. never occurs if the address is unique, but always occurs when duplicate addresses are present or;

2. rarely occurs when routing unique address, but often when dealing with a duplicate address.

Each node maintains a table containing the probability of an address conflict. Whenever the PDAD algorithm detects a hint about a possible conflict, it adjusts this value. Should the value reach a predetermined threshold, the module notifies the conflict resolution component.

An example of the probability algorithm is called *PDAD-SN* (SN being an abbreviation for Sequence Number). The sequence number is an integrated part of routing protocol packets. It is assumed that each node only uses a sequence number once in the time interval t_d. t_d is an estimated upper bound on the packet distribution time using a given routing protocol. This sequence number is incremented for each message sent by a node. The basic idea of PDAD-SN is as follows. If node B receives a message from node A having the same address $Addr_{AB}$, but with a lower sequence number, node B cannot detect the duplication. Node B cannot distinguish if this message is previously sent by itself, or if it came from a node with the same address. On the other hand, if node A receives a packet with same address, but higher sequence number, this packet must have originated from a different node with address $Addr_{AB}$. Node A must notify the conflict resolution module to handle this situation.

Conflict resolution

PACMAN issues an *Address Conflict Notification (ACN)* when a conflict is detected, and is unicast to the duplicate address. The ACN message is sent in the direction from which the packet was received. This way, the node that has most recently joined the MANET tends to change its address, and thereby no additional information must be distributed.

2.1.3 IPv6 Autoconfiguration in Large Scale MANETs

To cope with the increasing number of devices capable of wireless communication, MANETs must be able handle a large number of simultaneously connected nodes. *IPv6 Autoconfiguration in Large Scale MANETs*[26] presents a proposal for autoconfiguration in a large scale MANET.

Overview

The overall strategy to achieve a scalable solution is to define a hierarchy, where special nodes are responsible for parts of the address configuration. IPv6 already defines stateless address autoconfiguration, but recall that it cannot be adopted for MANET-use directly. IPv6 SAA begins by constructing a link-local address based on the well-known link-local prefix `FE80::/10` and a unique interface identifier. The address constructed by these values is considered a *tentative address*. The Institute of Electrical and Electronics Engineers (IEEE[27]) defines a 64-bit Extended Universal Identifier (EUI-64), which is used as the interface identifier. The EUI-64 is intended to be globally unique, but network interface manufactures may use unregistered identifiers, thus it cannot be guaranteed.
After constructing this tentative link-local address, its uniqueness must be ensured. The Duplicate Address Detection phase issues a Neighbor Solicitation

(NS) message with the well-known *unspecified address*, `::/128`, as source, and the *solicited-node multicast address* as destination. This address is constructed from the prefix `FF02::1:FF00:0/104`, and the last 24-bits of the IPv6 address that is being resolved. If this address is already occupied, the corresponding node issues a *Neighbor Advertisement (NA)* with the *all-node multicast address* as its destination. Receiving an advertisement alerts the sender that the address in question duplicate.

Hierarchical approach to stateless address autoconfiguration

Algorithms for MANET-routing all requires address uniqueness for every node. To fulfill this requirement, DAD must *flood* the network to ensure no such duplication exists, but flooding an entire network does not scale. To prevent flooding, each node defines its broadcast link (called *scope*) as a group of nodes that are located r_s hops or less away. Special *leader nodes*, elected using an *election algorithm* (described later), establish a hierarchy.
In order to perform DAD in a hierarchically fashion, a node issues a modified NS message containing the tentative address, and a hop limit of r_s (as opposed to 255 in a standard NS message). Limiting the hop count to r_s reduces the scope to which the message is sent.

Detecting duplicate addresses in a MANET is not trivial. A node receives echoes of NA and NS messages multiple times depending on the number of adjacent nodes. Furthermore, a message sent by a node cannot be distinguished from a message sent by another node performing DAD (with the same tentative address). The messages are completely identical. The source is the unspecified address, and the destination is solicited-node multicast address, and it is targeting the tentative address. Defining a new option to the Neighbor Discovery Protocol (NDP[9]) message is one solution to the above-mentioned scenario. The *MANET-option* contains a *Random Source ID (RS-ID)* field with a new random value for each new message. Upon reception of neighbor discovery messages, nodes cache the RS-IDs and only forward messages whose ID are not present in the cache.

Leader election algorithm

As described earlier, special leader nodes must be elected in order to create and maintain a hierarchical structure. These leader nodes are chosen via an election algorithm. There are two requirements to achieve an efficient election algorithm:

1. The number of leaders must be small compared to the total number of nodes present in the network.

2. Leaders should not change too frequently in order to keep the network topology stable.

Choosing the node with the highest number of neighbors ensure that the number of leader nodes is kept to a minimum. To keep track of which node is elected leader, three states are introduced. First is a leader state L, secondly a candidate state C, and finally a host state H. The leader node is in the leader state L. Potential leader candidates are non-leader nodes with the highest number of neighbors in the scope, and are placed in candidate state C. All remaining nodes are in the host state H.
Based on the number of received neighbor solicitations, the number of neighbors can be calculated. If a leader node receives a solicitation, the number of neighbors is stored in n_l. Otherwise it is compared to the number of neighbors n_c of the current candidate. n_m denotes the number of neighbors a node m have, and if this is greater than n_c, m is promoted to candidate. Otherwise the node stays in the host state.
Leaders are elected for a certain period, which is determined by a DAD timer. When this timer expires, the number of neighbors of the corresponding node is compared to the neighbors in the leader and candidate node. To prevent oscillation of leader nodes, a threshold n_{thres} is introduced. If node m is in state L, and has fewer neighbors than the current candidate, when n_{thres} is taken into consideration, the leader steps down. In other words, if $n_m < n_c - n_{thres}$, m withdraws its leadership. If a node m' satisfy the inequality $n_{m'} > n_l + n_{thres}$, this nodes has significantly more neighbors than the leader. All nodes satisfying this inequality are considered leaders, and the one with the highest RS-ID is elected.

Forming a site-local address

The newly elected leader node has a number of tasks to perform. It subscribes to the *all-router multicast group*, it chooses a random subnet ID, constructs a site-local address based on this subnet ID, and its link-local address. It issues a router advertisement (RA) containing the subnet ID to all nodes within its scope. All nodes receiving this RA can generate their own site-local address. One problematic aspect of generating a site-local address based on a node's link-local address is, that link-local addresses are not guaranteed to be unique throughout the entire MANET. Ensuring a unique site-local address can be achieved by choosing a unique subnet ID. Leader nodes must therefore communicate via the all-routers multicast group to prevent duplicate subnet ID's.

2.2 Autoconfiguration in hybrid MANETs

Connecting to the outside world requires a link between the MANET and the Internet. Providing this link is a task for *gateways*. The following sections describe autoconfiguration in MANETs with one or more gateways present.

2.2.1 Autoconfiguration using prefix continuity

A proposal for autoconfiguration using neighborhood discovery, while maintaining a logical structure within the network using *prefix continuity*[28] is presented below.

Prefix continuity

Prefix continuity is the concept of ensuring existence of a path between a node N and its gateway G, such that all intervening nodes carry the same network prefix. The main goal of keeping prefix continuity is to form a logical structure within the hybrid network, in the sense that the network is partitioned into subnets. Figure 2.7 shows a MANET divided into two subnets where nodes A through D are associated with gateway $GW1$, while E and F are associated with $GW2$. Nodes having $GW1$ as its gateway all use the prefix bound to $GW1$ thereby forming a subnet within the MANET.

The autoconfiguration mechanism relies on two major components; neighborhood discovery and selecting the upstream neighbor, which is the direct neighbor on the path towards the gateway. Both of these issues will be discussed below.

Neighborhood discovery

Each gateway has the responsibility to periodically advertise the prefix. The 64-bits global prefix is combined with the EUI-64 identifier of a node to construct a globally routable IPv6 address. This prefix is sent encapsulated inside a *GW_INFO (GateWay INFOrmation)* message, and all nodes in the MANET will receive these from its upstream neighbor. If a non-upstream neighbor receives a GW_INFO message, the receiving node must silently discard it. Figure 2.7 shows the propagation of GW_INFO messages in a MANET using prefix continuity. The number on the arc in figure 2.7 between two nodes indicates the hop distance to the associated gateway. GW_INFO messages includes the following fields:

- the global address of the gateway, and the prefix length (usually 64 bits).
- the hop-distance to the gateway.
- a sequence number.

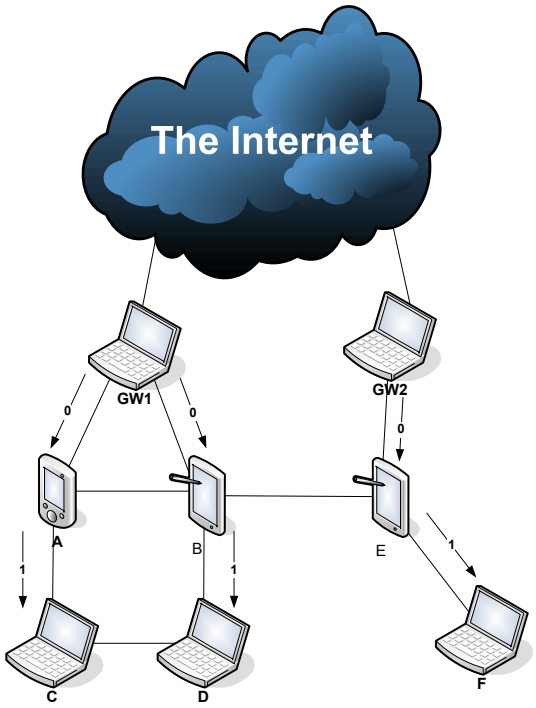

Figure 2.7: Propagation of GW_INFO messages in a prefix continuity enabled MANET.

The initial GW_INFO message sent by the gateway has a hop-distance of zero. This value is stored and incremented by each neighbor prior to forwarding it downstream. Using this stored sequence number, a node can detect loss of messages by looking for gaps in the sequence numbers arriving.

Selecting the upstream neighbor

There are two different ways in which a node can select its upstream neighbor. The first option is to select the neighbor that provides the shortest path to the gateway. The other option is to select a node that provides maximum stability, i.e., selecting the neighbor allowing a node to keep its global address as long as possible. In the following, the former will be referred to as *distance option*, and the latter as *stability option*.

Selecting an upstream neighbor using the distance option is rather simple. The neighbor is selected based on the gateway advertisement with the lowest hop count. In case of multiple gateways, the distance option tends to form well-balanced networks in the sense that the sub networks are of equal size (statistically speaking). A drawback of the distance option is the possibility of frequently address changes, due to selecting the shortest path to the gateway.
As mentioned earlier, the stability option prevents these frequently changing ad-

dresses. Having specified the stability option, a node simply ignores GW_INFO messages sent by nodes attached to a different subnet. Although selecting the upstream neighbor does not guarantee the globally shortest gateway path, it is guaranteed to be the shortest within the subnet.

2.2.2 Global connectivity for IPv6 MANETs

Global connectivity obtained via a gateway, either by using an extended Neighbor Discovery Protocol, or the implemented routing protocol is described in *Global connectivity for IPv6 MANETs*[29].

Overview

MANET nodes learn about gateways by receiving advertisements. These advertisements can be distributed periodically, but is typically only sent by request when implementing a reactive routing protocol, e.g., AODV[8]. Nodes can solicit gateway advertisements when needing global connectivity. In order to facilitate these solicitation and advertisements, modifications to the Neighbor Discovery Protocol (NDP) and each MANET routing protocol are required. Advertisement contains prefix information, and processing these includes resolving a route to the gateway. Each MANET node can use this prefix to construct a globally routable IPv6 address.

Conceptual data structures

In order to realize the autoconfiguration scheme, all MANET nodes must support a collection of data structures. Most of these structures are similar to those found in NDP, but often extended with additional information.

- **Internet Gateway List**. A list of available gateways to which packets can be sent. A timer is associated with each entry in the list, and is used to remove entries when no longer advertised. Information in these entries includes Internet gateway global address, lifetime and an optional MANET-local address (only for internal communication).
- **Internet Gateway Prefix List**. A list of prefixes that are advertised by gateways. A node uses gateway advertisements to create the entries in the prefix list. Each of these entries is also associated with a timer. Information in prefix list entries includes gateway prefix address and prefix address length.
- **Associated MANET node list**. Internet gateways manage a list of associated MANET nodes to which it offers global connectivity. Only the global address of these nodes is stored.

Gateway advertisements or gateway solicitation messages are implemented in one of two ways, extending NDP or extending the routing protocol.

Extending NDP

Three new messages are introduced in addition to standard NDP, namely solicitation message *IGWSOL-N*, advertisement *IGWADV-N* and conformation message *IGWCON-N*. The *N* indicates that extension is made in NDP. Adding these extensions to the solicitation message is a matter of introducing a new type to insert in the message header (figure 2.8).

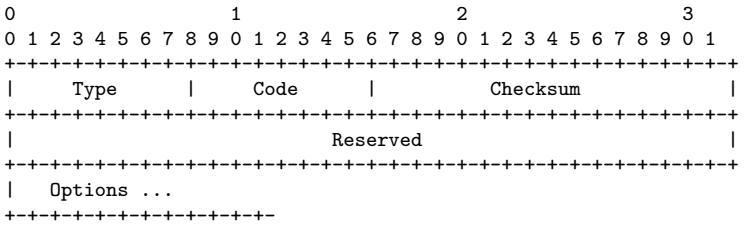

Figure 2.8: Standard NDP gateway solicitation message.

Advertisements *IGWADV-N* are similar in message structure to router advertisements in NDP. However, these messages are destined for the MANET only, and must not be forwarded to Internet nodes (nodes connected to the Internet from outside the MANET). Figure 2.9 shows the *IGWADV-N* header.

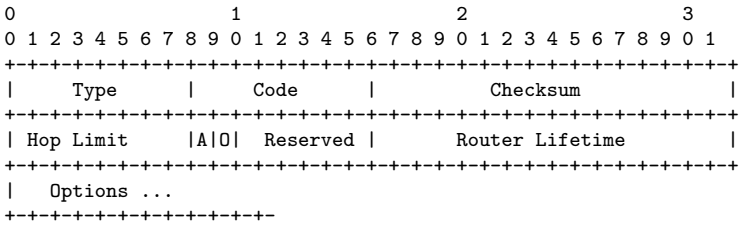

Figure 2.9: Advertisement header for extended NPD.

The extended version of NPD must also include a prefix information option with the globally routable prefix. This option has not been modified for MANET use. The last message *IGWCON-N* is only used if a gateway requests an acknowledgment (by setting the A-flag in IGWADV-N figure 2.9). These acknowledgment requests are used to construct and maintain the associated MANET node list.

Extending the routing protocol

There are various ways of extending an existing routing protocol, depending on which protocol is used. For version 2 of OLSR (OLSRv2[30]) a new message

is defined. This message is for global connectivity information is illustrated in figure 2.10.

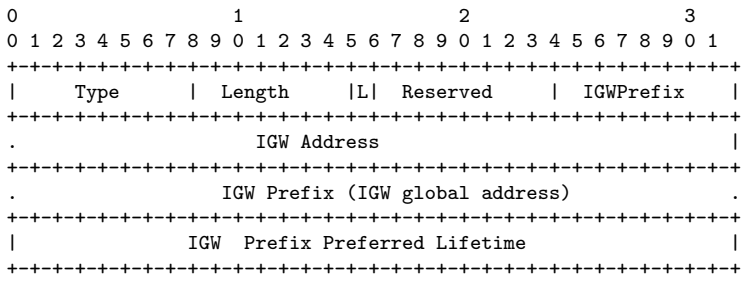

Figure 2.10: The extension message to the OLSRv2 protocol.

The Dynamic MANET On-demand Routing protocol (DYMO[31]) already specifies the concept of a gateway. A new block for global connectivity is specified and is carried out by RREP (Route REPly) messages. To facilitate gateway confirmation, a new C-flag is added in the "reserved field", and if set, the message is a confirmation message. Figure 2.11 illustrates the extended DYMO message.

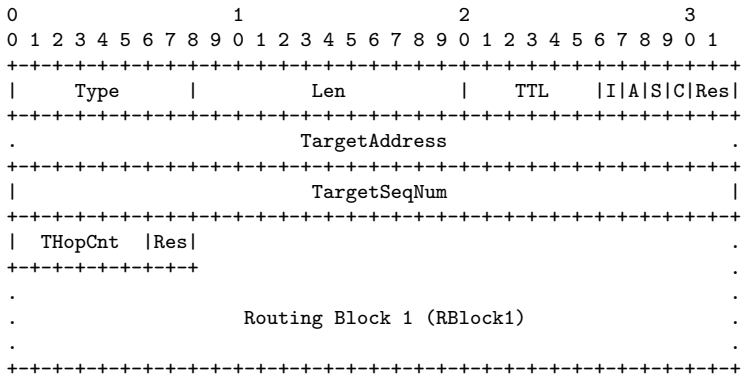

Figure 2.11: Extended DYMO message.

2.3 Summary

This chapter summarized three approaches to autoconfiguration in isolated mobile ad hoc networks, and two for hybrid MANETs. Each proposal had distinct properties, and different target areas. Section 2.1.1 introduced an autoconfiguration scheme, where configured nodes acquired unused IP addresses on behalf of newly joined nodes. PACMAN in section 2.1.2 was a mixture of stateless and stateful autoconfiguration, allowing nodes to configure their own addresses. It used passive duplicate address detection to detect any address conflicts. Auto-

configuration in large-scale mobile ad hoc networks was introduced in section 2.1.3. Achieving this was done by creating and maintaining a hierarchy, limiting the scope of flooded messages. Special leader nodes were elected to maintain this hierarchically structure.
Autoconfiguration in hybrid MANETs using prefix continuity was the topic of section 2.2.1. Prefix continuity divided the MANET into smaller subnet, each associated with a gateway. The prefix was periodically propagated throughout the subnets. The nodes have responsibility of preserving these subnets by selecting an upstream neighbor, who forwards the correct prefix to downstream nodes.
Section 2.2.2 presented a scheme for obtaining gateway information using either the routing protocol or an extended version of the Neighbor Discovery Protocol. In both case new messages was introduced facilitating the additional information required to propagate gateway information.

Chapter 3

Distributed IPv6 Addressing Technique for MANETs

This chapter serves as a detailed description of the autoconfiguration suggestion[15] that is subject for validation in this thesis. The chapter is divided into subsections, each describing a part of the autoconfiguration process. Section 3.1 provides an overview of the autoconfiguration process. Generating a link-local address for MANET use is the subject of section 3.2. Ensuring uniqueness of the generated link-local address is taken care of via the Duplicate Address Detection described in section 3.3. Section 3.4 introduces the IP address format used in this proposal. Obtaining a globally routable IP address, and ensuring that no two nodes is assigned the same address, is described in section 3.5. Section 3.6 describes the operation required by special proxy nodes in order to ease the workload of the gateway(s). Addresses assigned to proxy nodes must be managed by the gateway to prevent address leaks, and this is the subject of section 3.7. Finally, section 3.8 summarizes.

3.1 Overview

When a new node N enters a mobile ad hoc network, it creates an IPv6 link-local address. To ensure uniqueness of this address, it performs duplicate address detection as ordinary IPv6 stateless autoconfiguration[7]. This link-local address is only valid within a one-hop distance, and the node must acquire a global address from the gateway G. Recall that MANETs are multi-hop networks, and not all nodes are mutually reachable throughout communication using link-local addresses.

So if N is not within direct wireless range of G, it cannot use the available ad

hoc routing protocols to send a request for reasons discussed in section 2.1.1 on page 11. Node N chooses a fully configured, directly reachable neighbor I called the *initiator*, which forwards the request on behalf of N. Broadcast of a *neighbor request* is carried out to reach at least one fully configured neighbor. N chooses a single initiator from the set of *neighbor replies* received. To allocate a global IPv6 address, N sends an *address request* to the selected initiator, which in turn forwards it to G. Node I receives an address from G, which is sent back to the requester N. N can finally configure its interface with this address, and is now globally reachable.

Due to the multi-hop nature of a MANET, a node far away (in hops) from the gateway can experience a long delay between the address request and the address reply. To reduce setup time for these nodes, special *proxy nodes* are introduced. Proxy nodes act as a middleman, assigning addresses on behalf of the gateway(s). If a proxy node P is present on a path from the initiator I to gateway G, P intercepts the address request. P has been granted a pool of IP addresses, which it can hand out to requesting nodes. A node is promoted to proxy if it receives its address reply from a gateway.

To illustrate the autoconfiguration procedure, a continuous example is presented throughout this chapter as Message Sequence Charts. Message Sequence Charts are formally introduced in section 5.1. Unless stated otherwise, the autoconfiguration of the two requesters is assumed to be conflict-free, i.e., both requesters generate unique link-local addresses. Figure 3.1 shows an overview of the network topology considered in this chapter. It consists of two requesters, a proxy node, a non-proxy node, and a gateway providing the Internet access.

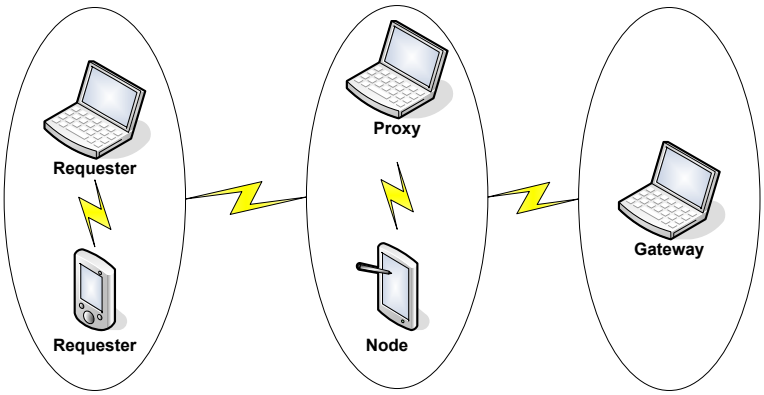

Figure 3.1: The network topology considered in this example.

The lightning bolts between nodes in figure 3.1, illustrates how the nodes are interconnected. If there exist a lightning bolt between a pair of nodes inside a

circle, it means that they are in direct wireless range of each other. The lightning bolt between the circles surrounding the nodes illustrates the reachability between pairs of nodes. The *proxy* and the non-proxy *node* are the only nodes capable of reaching all nodes.

3.2 Generating a link-local address

Creating a link-local address is a relatively simple procedure. A node needs to generate a link-local address for an interface if

- the interface is attached to the link (MANET) for the first time; or
- the interface becomes enabled after having been disabled administratively by the user.

Appending a unique identifier to the well-known link-local prefix`FE80::0` creates a link-local address. An interface identifier of length n replaces the n rightmost bits in the prefix to form the address. If the identifier is more than 118 bits for some reason, the configuration has to be done manually. The prefix `FE80` occupies the 10 most significant bits ($FE80_{hex}$ = **1111 1110 10**00 0000), which means that a total of 118 bits are available for the identifier. The IEEE has defined a format for a typical interface identifier, consisting of 64 bits (EUI-64). This EUI is similar to the 48-bit MAC (Media Access Control) address as most interfaces uses today. Like the MAC address, EUI is split into two blocks; the upper 24 bits constitute the *organizationally unique identifier (OUI)*, and the remaining 40 is a host identifier. A variation of this EUI-64 is adopted for IPv6 interfaces, called the *modified EUI-64*. To modify the standard EUI-64 identifier, the 7th bit from the left, the *universal/local or U/L bit*, is changed from 0 to 1. A 48-bit MAC can be converted into a modified EUI-64. Copy the OUI portion of the MAC address to the upper 24 bits of the EUI-64, and place the remaining 24 bits as the rightmost bits. The 16-bit gap is filled with `FFFE`, and the universal/local bit is change to obtain the modified identifier. To illustrate this process, assume we want to configure a node which has an interface with MAC address `0050:56C0:0001`. Figure 3.2 shows the process of transforming a 48-bit MAC address into a modified EUI-64 identifier.
The link-local address generated for the node is `FE80::0250:56C0:0001`.

Figure 3.3 shows the message sequence chart after generating the link-local address. Based on the MAC addresses, the two requesters generate a link-local address as described above.

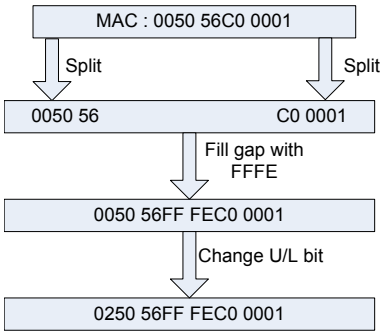

Figure 3.2: The process of transforming a 48-bit MAC address into a modified EUI-64 identifier.

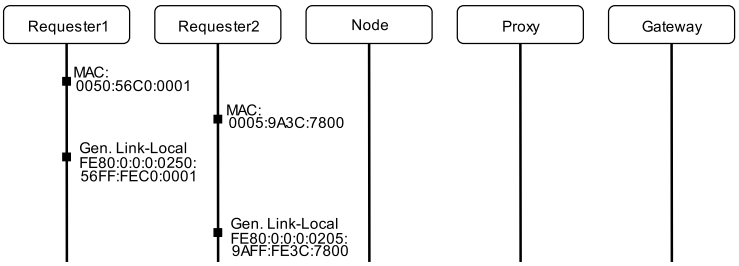

Figure 3.3: Message sequence chart showing the generated link-local address.

3.3 Duplicate address detection

Detecting the existence of duplicate addresses in a network is relatively straightforward. The specification[15] suggest using Stateless Address Autoconfiguration for IPv6[7]:

> *In IPv6, stateless address auto-configuration can be used in MANET. This mechanism allows a node to pick a tentative address randomly and then use a Duplicate Address Detection(DAD) procedure to detect duplicate addresses.*

Duplicate address detection for IPv6 stateless autoconfiguration uses neighbor solicitation and neighbor advertisement messages to detect duplications. Should the procedure detect any address collisions, the address is not suitable as a link-local address. If this address was derived from the interface identifier and the link-local prefix, a new identifier must be assigned. Another option is to manually configure the IP address for that particular interface.
The generated address is considered tentative, and must not be assigned until the procedure has successfully been completed. The interface with this tentative address must only accept neighbor solicitation and/or advertisements, until the procedure successfully terminates.

Sending neighbor solicitation messages

An interface must join the all-nodes multicast address, `FF02::1`, and the solicited-node multicast address for the tentative address, before sending any neighbor solicitations. Joining the all-nodes multicast address ensures that the node receives neighbor advertisements from another node already occupying that address. Unsolicited or solicited advertisements with a source being the unspecified address are sent to this multicast address.
Detecting the presence of another node attempting to use the same address is the purpose of joining the solicited-node multicast address. This address is constructed by taking the lower 24 bits of the tentative address and replace the $x's$ in the address `FF02:0:0:0:0:1:FFxx:xxxx`. Continuing the example from section 3.2, the solicited-node multicast address for this node is `FF02:0:0:0:0:1:FFC0:0001`. Checking if an address is occupied, a node sends *DADTransmits* neighbor solicitation messages (default value of *DADTransmits* is 1) with an interval of *RetransmitTimer* milliseconds in between. Figure 3.4 shows the message format of a neighbor solicitation message (not including the base IPv6 header).

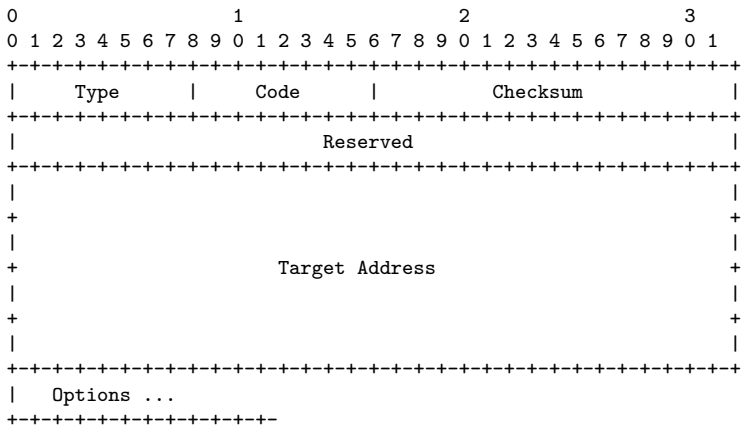

Figure 3.4: Neighbor solicitation message format.

The *target address* of a NS message is the tentative address being checked, and the source is the unspecified address. The destination is set to the solicited-node multicast address of the target address. The target and destination are part of the IPv6 base header (the IPv6 base header is shown in the IPv6 specification[5]).

If the first message sent by a node is a neighbor solicitation, it should delay sending it by a random delay between 0 milliseconds and a predetermined maximum value. This prevents congestion when multiple nodes join a common link at the same time, and helps to avoid race conditions. These race conditions can occur when more than one node tries to solicit for the same address.

Receiving neighbor solicitation messages

When a node receives a neighbor solicitation its actions depends on whether or not the target address is tentative. Determining if an address is tentative or not can be achieved in several ways. One solution is to add a new option[32] for NDP. This option replaces the *Source Link-Local Address* option when performing Optimistic DAD[33]. Optimistic DAD is an optimization to the existing Neighbor Discovery[9] and Stateless Address Autoconfiguration[7], intending to minimize configuration delays in successful cases. Neighbor solicitations include this new option called *Tentative Source Link-Local Address Option* (TSLLAO), together with the unspecified address as source. If the receiver is unable to interpret the TSLLAO option, it is assumed that DAD is performed when the source is the unspecified address.

If it is not a tentative address (i.e. it is assigned to the receiving interface), the solicitation message is processed as specified in the neighbor discovery protocol specification. Receiving a solicitation message with the source address *not* being the unspecified address, the solicitation includes a source link-layer address option. The node should create/update the neighbor cache entry for the IP address indicated by the messages source address.

Receiving a solicitation with a tentative target address, the type of source address dictates which actions must be taken. If the source address is a unicast address, the sender is performing address resolution on the target, and the message must be silently discarded. An unspecified address as source informs the receiver that the solicitation originated from a node performing DAD. The address is a duplicate address if the solicitation came from another node, and should not be used by any of the involved nodes. Finally, the message could originate from the node itself due to loop back of multicast packets, and the solicitation does not indicate that duplicate addresses are present.

Detecting whether or not an address is unique can be accomplished in several ways:

- Two nodes performing DAD simultaneously, with the same tentative address X, will result in one node receiving a solicitation prior to having sent one (due to the random delay selected). This alerts the node that another node is interested in acquiring X.
- The number of solicitations received can be used to detect duplications. If this number exceeds the expected number, the tentative address is duplicate.

Continuing our example, the message sequence chart in figure 3.5 shows the ex-

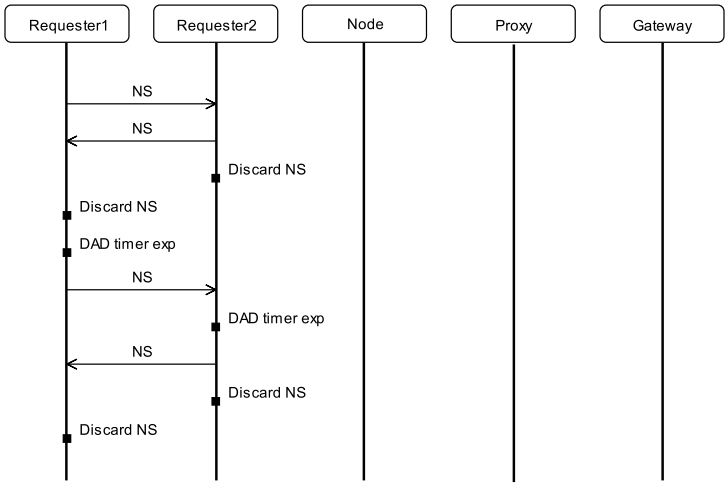

Figure 3.5: Requesters exchanging and discarding neighbor solicitations.

change of neighbor solicitations, and the discarding hereof. *Requester1* starts sending a neighbor solicitation as indicated by the arrow with caption *NS*. *Requester2* sends a solicitation and correctly discards the incoming NS. The DAD-timer expire at both requesters and they resend the solicitation, and discard the incoming packet.

Sending neighbor advertisements

Neighbor advertisements (NA) are sent in reply to solicited neighbor messages. The structure of the advertisement is similar to the solicitations, where only three bits in the "reserved" field distinguish them. These three bits are denoted R, S and O and is a router, solicitation, and override flag, respectively. The R-bit indicate that the source is a router, and the S-bit is set if the advertisement came in response to a solicitation. The override flag indicate that the advertisement should override an existing cache entry, and update the cached link-layer address. *Source address* of a neighbor advertisement is the interface from which the advertisement is sent. *Destination address* is the source address of the invoking neighbor solicitation, unless it is the unspecified address. If the source is the unspecified address, the destination is the all-nodes multicast address. The *target address* is the same as the target address of the solicitation message.

Receiving a neighbor advertisement

Similar to receiving a solicitation message, handling a neighbor advertisement depends on the target address. If the target address is assigned to the receiver (i.e. *not* tentative) the node must proceed as follows. Receiving a neighbor advertisement triggers a search in the neighbor cache for the targets entry. If the

search is unsuccessful, the neighbor advertisement should be silently discarded. Creation of an entry is not necessary since the receiver has not initiated any communication with the target. When an entry exists, the action taken by the receiver is among other dependent on the state of the cache entry and the flags in the incoming message. For a detailed description on how to update the cache entries, see section 7.2.5 of the neighbor discovery specification[9].

If the target address *is* tentative, this address is not unique, and the configuration process fails.

Figure 3.5 shows the sending of an advertisement due to clash in link-local address. Both requesters send a neighbor solicitation as part of the duplicate address detection phase. *Requester1* detects the address clash, resets its configuration, and sends an advertisement to *Requester2* as indicated by the arrow with caption *Send NA*. *Requester2* receives this advertisement and resets its autoconfiguration procedure.

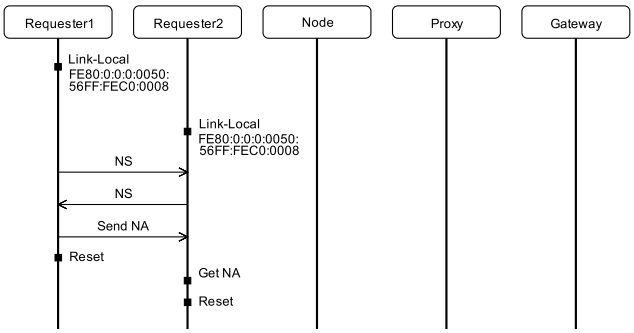

Figure 3.6: A clash in link-local address leads to the sending of a neighbor advertisement.

A successful completion of duplicate address detection is, however, not a guarantee that the link-local address continues to be unique. A merge of otherwise independent MANETs could lead to clashes in link-local addresses. The technique discussed above can only detect conflicts a configuration time, and not subsequent conflicts due to MANET merging or connectivity changes. Detecting address conflicts post configuration can be achieved by a passive duplicate address detection mechanism as discussed in section 2.1.2.

3.4 Address allocation

IPv6 addresses used in this autoconfiguration scheme are divided into three parts, *Global Network Prefix*, *Ad hoc Prefix* and *Host ID*. Figure 3.7 shows the address division.

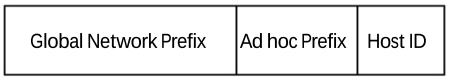

Figure 3.7: An IPv6 address split into global network prefix, ad hoc prefix and host ID.

The global network prefix is identical for all addresses allocated, but the ad hoc prefix and host ID vary. The ad hoc prefix is allocated by the gateway, and is used to identify the proxy. Each proxy can allocate addresses by selecting a unique host ID.
If we assume that the gateway prefix is 64 bits, and the ad hoc prefix is 48, then 16 bits are available for the host ID. The gateway allocating a new address selects an unused ad hoc prefix and initializes the host ID to all 0's. Assuming that the gateway is allocating the following address:

`3FFE:2E01:2B:1111:2222:2222:2222:0000`

the first 64 bits, `3FFE:2E01:2B:1111`, is the global network prefix. The middle portion, `2222:2222:2222`, is the ad hoc prefix and the 16-bit block of 0's is the initial host ID. The new node receiving this address is automatically promoted to proxy-node, enabling it to allocate addresses upon reception of an address request. The range from which the new proxy can allocate addresses is:

`3FFE:2E01:2B:1111:2222:2222:2222:0001`

to

`3FFE:2E01:2B:1111:2222:2222:2222:FFFF`

The specification recommends the following address allocation for the proxy node

> *As usual, the proxy node uses the first free address as its own node address. And the address, which all bits of Host ID are set to 0, must be used as a proxy multicast address.*

Using this three part partitioning of IPv6 addresses enables the gateway to allocate addresses for a maximum of $2^{48} \approx 2.8 \cdot 10^{14}$ proxy nodes, each capable of assigning $2^{16} - 2$ or 65,534 addresses. This clearly provides the scalability regarding the maximum number of concurrently connected nodes.

3.5 Initiator selection

After having successfully completed duplicate address detection, the requester R can initiate a request for a global reachable address. R broadcasts a *neighbor*

request to nodes within a one-hop distance. If multiple replies are received, R selects one of the responders as *initiator*, and silently discards subsequent *neighbor replies*. R sends an address request to the initiator, which acquires an address on behalf of R. If the chosen initiator is aware of any proxy node within a one-hop distance, it forwards the address request to this proxy. The initiator knows about proxies from the periodically *proxy advertisements* being sent by the proxy nodes. When no proxy node is within reach, the initiator sends the request to the gateway. Should any proxy exist on the path towards the gateway, it intercepts the message, and performs address allocation as described in section 3.4. If no proxy exists on the path, the gateway sends a new "address range", and the requesting node is upgraded to proxy. How can a newly connected node distinguish if the address came from the gateway (and thereby being promoted to a proxy), or it came from an existing proxy node? Recall that addresses allocated by a gateway has `0000` as host ID, thus the node can examine this ID to determine whether or not it originated from a gateway. The address sent by a proxy has a host ID greater than or equal to `0002`, assuming that the implementation follows the recommend address allocation.
The node examines this host ID to determine whether or not it must perform proxy duties. Figure 3.8 shows the initiator selection at *Requester1* and *Requester2*, together with the subsequent address request.

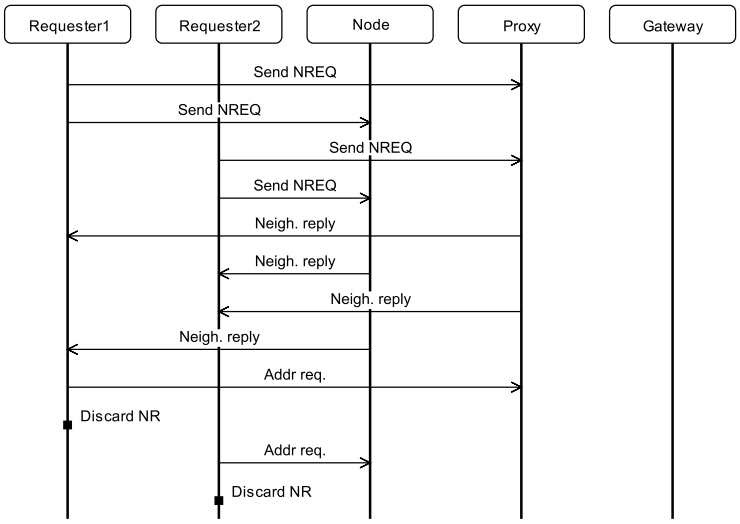

Figure 3.8: R broadcast neighbor requests, and initiates a request for a globally routable IP address.

Requester1 and *Requester2* start by broadcasting neighbor requests to all nodes within a one-hop distance. Both requesters receive neighbor replies from *Proxy* and *Node*. *Requester1* selects *Proxy* as initiator, while *Requester2* selects *Node*. The remaining neighbor reply is correctly discarded.

3.6 Proxy node operation

A node promoted to proxy examines the destination of all incoming packets. Since all nodes in mobile ad hoc networks perform routing, this is not an additional overhead. Because of its proxy status, if the destination of a packet is the gateway, it examines the type. When the type is an *address request*, it handles the address assignment by selecting an available address from its local address space. The selected address is sent in an *address reply*.

Due to the mobility of MANET node, all proxies could potentially huddle up in one corner of the network. This results in unavailable proxy services for new nodes joining the network in particular areas. Therefore, a regular node may be upgraded to proxy node, in order to balance out any skew proxy distribution. If a node does not receive any proxy advertisements for some period of time, it sends a *proxy address request* message to the gateway. Receiving this request, the gateway allocates a new address space for that node. Now the node can perform proxy duties for any new node in that area of the network.

3.7 Address management

When the gateway G allocates a new block of IP addresses to a proxy P, it records this address in an *addresses allocated table* with associated lifetime timers. This timer value is included in the address reply to inform the proxy about its lifetime. In order not to expire this timer, proxy nodes should periodically send *address refresh*s to extend its lifetime. If G does not receive an address refresh before the timer associated with P expires, it multicast a refresh request using the multicast address of P as destination. Assuming all nodes having the ad hoc prefix of P is reachable from G, they must receive this message, and respond to G. No response to the request alerts G that no nodes are occupying addresses with that ad hoc prefix, and it can be removed from its table. This address space is now available for reallocation upon reception of an address request.

The prefix management illustrated as message sequence chart is shown in figure 3.9.
This example assumes that only the proxy node has an address with ad hoc prefix `1111:1111:1111`. The initial event disconnects the proxy from the network, and thereby no address refresh packets are received. The gateway G proceeds by multicasting refresh requests, but none of the connected nodes occupies this prefix, and silently discards the request. The lack of refresh replies informs the gateway that the prefix is unoccupied, and can be returned to the pool of available prefixes.

The scenario in which G receives at least one refresh reply is shown in figure

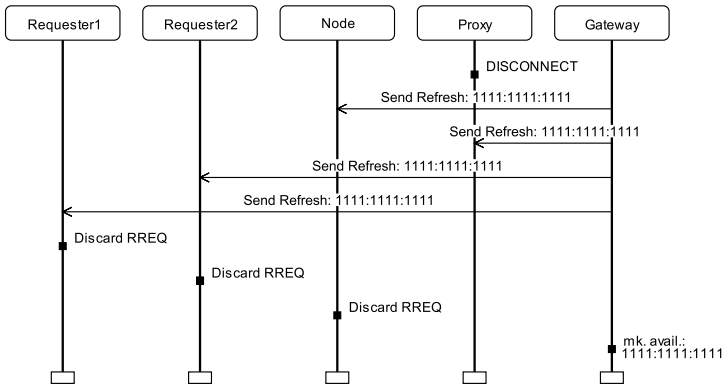

Figure 3.9: The gateway multicasts refresh requests, but no replies are sent.

3.10. It is assumed that all non-gateway nodes occupies prefix P_1 of the proxy node. After G stops receiving address refresh packets from P it multicasts refresh requests and all receivers respond as indicated by the arrows labeled *Send RREP* (send refresh reply). Receiving at least one reply informs the gateway that there is no proxy, but that the prefix is occupied. In this case, G should *not* remove the entry from its allocation table.

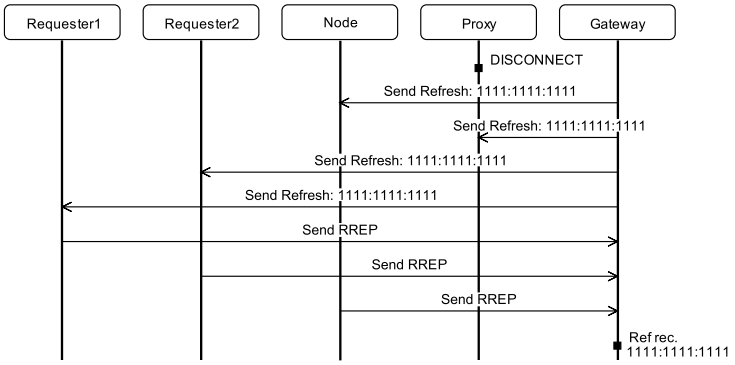

Figure 3.10: The gateway multicasts refresh requests, and the MANET nodes respond.

Achieving reliable multicasting is difficult given the dynamic nature of MANETs. A study has shown that the packet delivery ratio can be as low as 25% when multicasting in a highly mobile MANET[34]. In other words, only one packet is successfully received by the multicast group members for every four packets sent. Using an on-demand multicast routing protocol (ODMRP[35]) can, according to experiments[36], increase the packet delivery ratio to between 97.3% - 99.3% for a highly mobile MANET with between 10-50 receivers and one sender. Incorporating ODMRP into the autoconfiguration protocol will ensure a reliable construction of routes from gateway to receivers with a low channel and storage

overhead. Properties of ODMRP include:

- Low channel and storage overhead.
- Reliable construction of routes and forwarding group.
- Exploitation of the broadcast nature of wireless environments.
- Scalability using efficient flooding.

To reduce the waiting time for joining nodes, the gateway may select a tentative proxy T among the responders, and send a refresh reply to T. T can, however, not allocate addresses to other nodes, but it must send address refresh messages periodically in order to prevent G from sending additional refresh requests. A refresh reply with an extended lifetime instructs the proxy (tentative or not) with ad hoc prefix A to multicast this information. This informs the nodes using prefix A that the lifetime has been extended.

3.8 Summary

The chapter served as a detailed description of the autoconfiguration proposal subject for evaluation. A tentative link-local address was created by appending an extended universal identifier to the well-known link local prefix `FE80::0`. Duplicate address detection was applied to investigate the uniqueness of this generated link-local address. The process used neighbor solicitation and advertisements to resolve any address clashes. After obtaining a unique link-local address, a globally routable address must be obtained. Gateways connecting the MANET to the Internet were responsible for allocating these addresses. To ease the workload of these gateways, host nodes can be promoted to proxy nodes, allowing them to handout addresses on behalf of the gateways. Tables with the current allocated addresses helped gateways and proxies to manage the address space. If the timer associated with each address was not refreshed, the address was returned to the address pool.

Chapter 4

Modeling distributed autoconfiguration

This chapter presents a detailed description of the CPN model constructed for this thesis. For the reader with limited knowledge of Coloured Petri Nets, key concepts and terms are introduced throughout the chapter using examples from the constructed model. Section 4.1 discusses the assumptions and simplifications made in order to construct the model. Section 4.2 describes how the specification[15] is turned into a model, introducing various concepts such as *scope* and *partition*. A brief overview of the model, and its hierarchically structure, is presented in section 4.3. Definitions of the basic building blocks such as the network packets are the subject of section 4.4. Detailed descriptions of each module constituting the model are found in section 4.5. Finally, section 4.6 summarizes.

4.1 Assumptions and simplifications

A number of assumptions and simplifications have been made in order to model the autoconfiguration proposition presented by Lee et al. [15].

- **Neighbor discovery**. Prior to initiating the configuration process, a requester must first detect the presence of a MANET. This model assumes that the requester has already detected the presence of the MANET, and is ready to initiate the autoconfiguration process as described in chapter 3.
- **Routing**. Sending packets, e.g., from requester to initiator requires routing of packets. This thesis, however, focuses on the configuration aspect, and modeling a routing protocol such as AODV[8] or OLSR[22] is out of scope.

- **Address conflicts**. Address conflicts occur when multiple nodes have identical MAC addresses. In the model constructed for this thesis, possible MAC address clashes only occur between *requesting nodes*. Regardless of its status (fully configured or in the process of acquiring an address), if an address clash is detected, both parties is assumed to conduct a full reset of its configuration parameters.
- **Node type**. A MANET may consist of a broad range of diverse nodes, e.g., PDAs, laptops and cellular phones, all with their own characteristics. In the model, it is not taken into account that devices may have different wireless transmission range and quality.
- **Network reliability**. Real world networks are not 100% reliable, and loss, overtaking and corruption of packets are part of network communication. In this model it is assumed that the network is 100% reliable, rendering retransmission unnecessary.
- **No proxy advertisements**. Due to the properties investigated in this thesis, it is not necessary to periodically transmit proxy advertisements.
- **Link-local address range**. This thesis assumes that link-local addresses are limited to operate within a one-hop distance[18]. If a requester changes connectivity *before* it has been properly configured, a partial reset must be conducted, and the link-local address must be rechecked as a consequence of this assumption. Within a one-hop distance of the new connection, another node may occupy the same link-local address as the requester. The model will reset to retransmitting neighbor solicitations if the connectivity changes before it has been fully configured.
- **Security issues**. A large number of security issues surround wireless networking, but dealing with these issues regarding MANET and wireless communication is out of scope.

Assuming that the requesting node already has detected the presence of a MANET will not affect the analysis of the model, because the procedure does not depend on events occurring prior to this detection. Omitting the implementation of a routing protocol is a safe assumption due to the flexibility of the proposal

> *Our proposal enables a node in MANET to auto-configure a unique IPv6 global address. The proposed approach is flexible enough to be integrated with many different routing protocols.*

Ignoring the security issues regarding wireless network transmission is also a reasonable assumption. The operations of the autoconfiguration scheme are independent of which security measures have been made to ensure safe communication. Disregarding proxy advertisements is also a fairly reasonable assumption.

Receiving proxy advertisements merely assists the initiator in redirecting address requests. As a routing algorithm is not implemented, and the model is not created with performance analysis in mind, sending periodically advertisements only complicates the model further. Assuming only requesting nodes are involved in address conflicts, can also be justified. If a conflict occurs between a requester and a fully configured, non-requesting node, they must be handed a new global IP. They select an initiator and proceeds as described in chapter 3 eventually ending up with a new globally routable IP address. Only if the node were a proxy node significant changes are made. The conflicting nodes would have to get new addresses, leaving a prefix P without an associated proxy. Thus the periodic address refresh messages are not sent to the gateway, forcing nodes with prefix P to send replies. But this operation is no different from handling a departed proxy node from the network. This would simply disable the opportunity of assigning new hosts with addresses using prefix P, which already is covered by the model.

Assuming 100% reliability of the underlying network, and uniform transmission range and quality, is not realistic to expect when dealing with wireless networks. Lots of routing protocols include retransmission of packets in order to ensure successful communication. Implementing these retransmission mechanisms would inevitable complicate the model by adding retransmission timers. Most routing protocols handles all complications due to network transmission failure, and since routing is not the main issue, it is assumed that the packet will eventually arrive at its destination.

4.2 Converting the specification into a CPN model

Turning the specification into a model is a non-trivial task. Various CPN constructs must be introduced to capture the specification at a certain level of detail. Due to the mobile nature of MANETs, the model must reflect topology changes. As CPN is not a network modeling language, but a general discrete-event based modeling language, there are no preexisting modules ready for simulating MANETs or wireless communication. Features found in network simulation tools such as NS2 [37] must manually be implemented to simulate, e.g., topology changes.

One of the constructs introduced in this model is the notion of a *partition*. Simulating a MANET partitioning implies splitting the its nodes into multiple disjoint sets. This model, however, only supports partitioning the MANET into two disjoint sets. This is achieved by assigning a new *partition*-value to a node, having the value of either *part1* or *part2*. The initial *partition*-value is *part1*, and all nodes in the MANET carry this value. Simulating topology changes are conducted by changing the partition value to *part2* for a set of nodes. The nodes

carrying this new partition value can *only* receive a packet if it originates from a node having this partition value. These partition values are part of the packet sent, and a more detailed description of the packets can be found in section 4.4. Packets originating from, e.g., *part1*, destined for a node now belonging to *part2* cannot be received, simulating that the node is out of reach.

The notion of *scope* is introduced to help simulating the multi-hop nature of a MANET. A *scope* in the model is an integer stating to which scope a node belongs. Each node is assigned an initial scope, and determines which nodes are directly reachable, and for which forwarding is required. All nodes within a particular scope are within directly wireless range of each other. Furthermore, a node can also reach nodes with a scope value *one higher* or *one lower* than its own scope. Figure 4.1 shows the organization of nodes into interconnected rings. The lines illustrate the link between nodes.

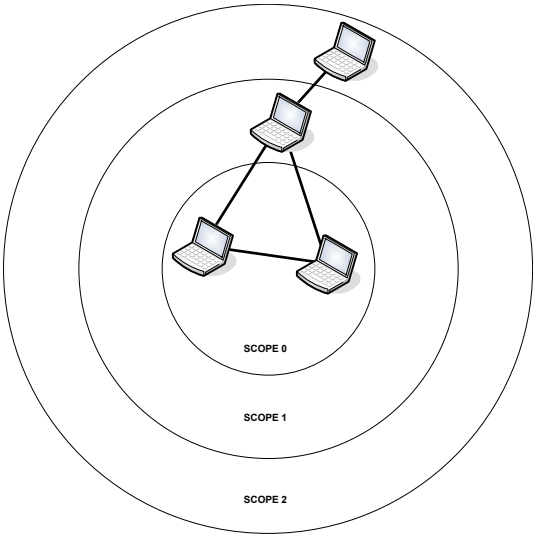

Figure 4.1: The organization of nodes into scopes.

Since scope 0 is the lowest scope number, nodes with this scope can reach nodes in scope 0 and scope 1. Nodes within scope 1, can reach nodes in scope 0, scope 1 and scope 2 and so on. Each packet contains a field, *CurrentScope*, stating at which scope it currently resides. Nodes may only receive a packet if *CurrentScope* is identical to the scope of the receiver. To contact a node in e.g. scope 2 from scope 0, the packet must pass through a node in scope 1, performing routing based on the implemented routing protocol. Since routing of packets is not the main issue of this thesis, routing merely consists of updating the scope at which the packets currently resides.

The advantage of organizing the nodes into a partial connected mesh network

as shown in figure 4.1 is, that no explicit topology information must be stored and updated. This topology, however, assumes a bidirectional connection between nodes within the scope, and nodes in the neighboring scope(s). This is not always a safe assumption, but adequate for this analysis. Using this approach, a node must only keep track of which scope it belongs to, and not, e.g., a list of pairs explicitly connecting it to the other reachable nodes.

4.3 Model Overview

The topology chosen for the CPN model was introduced in figure 3.1 in section 3.1. The topology is repeated in figure 4.2 for convenience.

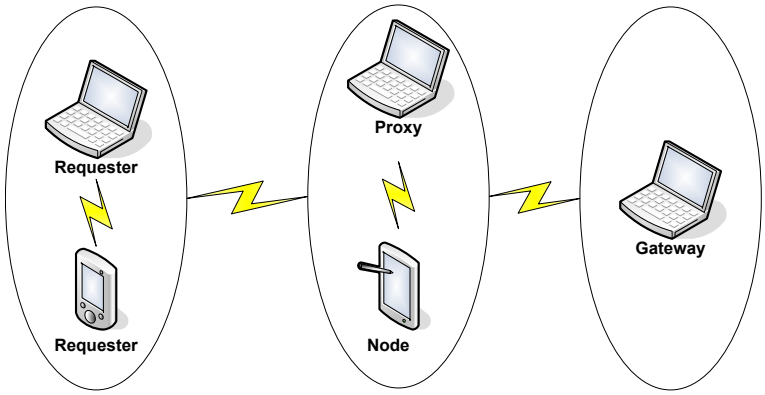

Figure 4.2: The network topology considered in this example.

Choosing this particular setup enables us to investigate various aspects of the model. Given that there are two non-configured requester-nodes, we can investigate the protocol properties when handling multiple simultaneously requests. At the same time it is possible to determine model behavior in case of duplicate addresses (both requesters have the same MAC address). Having a model with two configured nodes (beside the gateway) allow the requesters to choose either the proxy or non-proxy node as initiator.

There are no limitations in the model that prevent us from choosing a different configuration, but minor changes in the configuration parameters are, however, necessary. The hierarchical structure of the constructed CPN model is shown in figure 4.3.

Figure 4.3 reflects the modules that comprise the CPN model, and how they are interrelated. Nodes in figure 4.3 correspond to modules, while edges describe the sub-module relation between modules. A module at the arrow-tip is a *sub-module* of the module from which the arrow originates. As an example, *Select MAC* is a sub-module of *Requester*, which in turn is a sub-module of *MANET*.

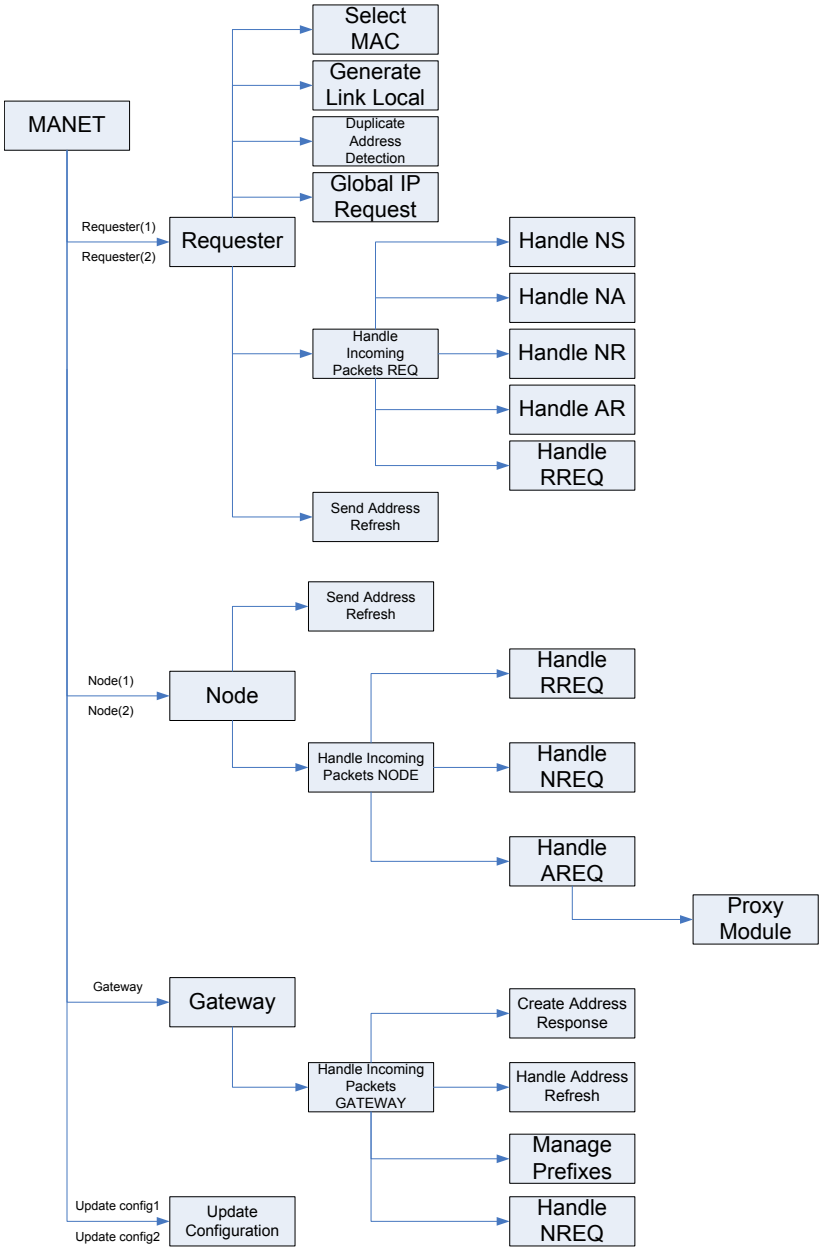

Figure 4.3: The module hierarchy.

A given module is implemented on a *page* in a CPN model.

At the top of the hierarchy, we find the highest modeling abstraction, which is the *MANET* module. This module consists of two *Requester*s, a general-purpose *Node*, a *Proxy* node, and a *Gateway*. There is no difference in the module representing a *Node* and a *Proxy* node. All *Node*s have a *Proxy Module* (section 4.5.17), but its configuration determines whether or not this module is active.

Each of the three modules (*Requester*, *Node/Proxy* and *Gateway*) is comprised by a number of sub-modules or *sub-pages*, which are described in detail in section 4.5. Figure 4.4 depicts the topmost page in the hierarchy, namely the *MANET* page. We explain figure 4.4 in more detail below.

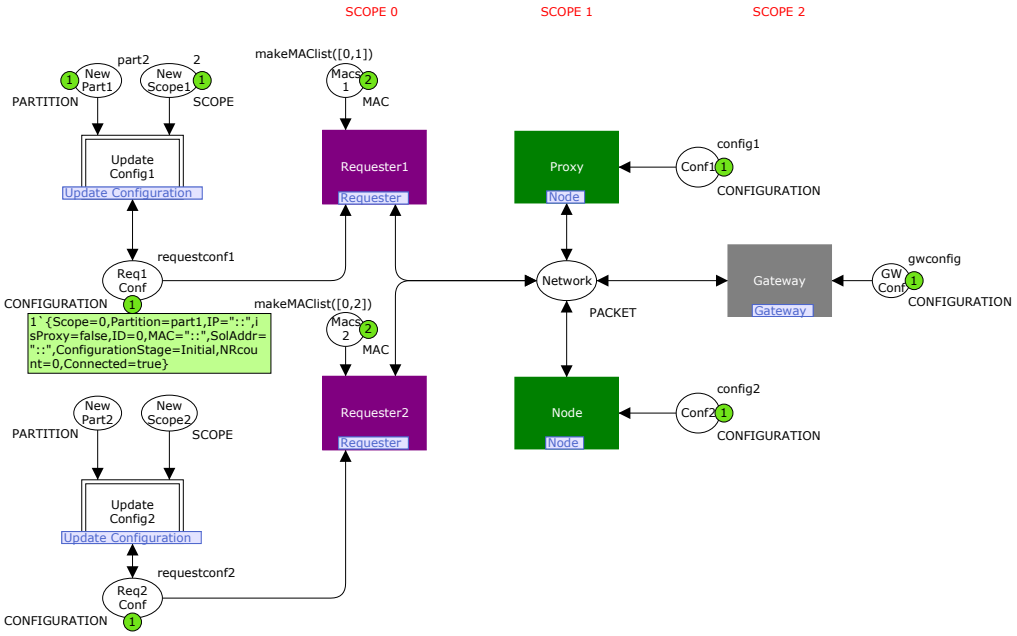

Figure 4.4: The topmost page of the constructed CPN model.

CPN models are constructed of two fundamental building blocks, namely *places* and *transitions*, represented as ellipsis and rectangles, respectively. A third vital component of a Coloured Petri Net is *inscriptions*, which are *expressions* written in CPN ML. These expressions are built from constants, variables, operators, and functions. The assembled expressions can only be evaluated when all expression variables are bound to values of the correct type.

Places represent the *state* of a given model, and may contain a number of *tokens*. Each token has an associated data value or *token colour*. An example of a place is *Req1Conf* representing the configuration parameters for *Requester1*. The name written inside a place/transition has no formal meaning, similar to variable names in conventional programming. Using well-chosen names, however, eases the readability of the model.

Combining the number of tokens, and the token colour on the individual places, represents the system state. This representation of system state is known in CPN terminology as a *marking*. The *current marking* of a place is graphically

represented by a small circle next to it, containing an integer stating how many tokens it currently holds. Next to the circle is a box listing the token colour of all tokens currently available on the place. The current marking of a place is not a set in the mathematically sense, but a *multi-set* e.g. a set with repetitions. In the remainder of this thesis, *set* and *multi-set* will be used interchangeably when referring to multi-sets of tokens.

An example of a *current marking* is shown in figure 4.4, where the place *Req1Conf* currently holds *one* token with the token colour listed below:

```
{Scope = 0, Partition = part1, IP = "::", isProxy = false, ID = 0,
 MAC = "::", SolAddr = "::", ConfigurationStage = Initial,
 NRcount = 0, Connected = true}
```

The set of token colours permitted on a place is called the *colour set*. Each place is restricted to hold only tokens of a particular colour set. This is equivalent to only have permission to assign, e.g., strings to string-variables in a programming language. A convention is to write the colour set to the lower right of a place, typically in all uppercase letters. An example of a colour set is *CONFIGURATION*, which is the colour set of places *Req1Conf* and *Req2Conf* in figure 4.4. Colour sets are defined using the CPN ML keyword **colset**, and the definition of the CONFIGURATION colour set is given in listing 4.1.

```
colset CONFIGURATION = record
                       (*The scope at which it resides*)
                       Scope              : SCOPE              *
                       (*The partition at which it resides*)
                       Partition          : PARTITION          *
                       (*The IP address which it is assigned*)
                       IP                 : IP                 *
                       (*Is the node a proxy node?*)
                       isProxy            : BOOL               *
                       (*An unique ID*)
                       ID                 : ID                 *
                       (*The MAC address selected*)
                       MAC                : MAC                *
                       (*Solicitation address created based on MAC*)
                       SolAddr            : IP                 *
                       (*Configuration stage at which the node is*)
                       ConfigurationStage : CONFIGURATIONSTAGE *
                       (*The number of neighbor replies received*)
                       NRcount            : INT                *
                       (*Is the node connected to the MANET?*)
                       Connected : BOOL;
```

Listing 4.1: The colour set definition of *CONFIGURATION*.

The CONFIGURATION colour set is a ML record consisting of ten fields, each having an associated type. The token(s) residing, e.g., on place *Req1Conf* must be a token of this colour set, i.e. a record with these named fields and types.

The types *scope* and *partition* was introduced in section 4.2. *IP* and *MAC* are mnemonic names for colour sets of Strings, while *ID* is the colour set of integers. The field *ConfigurationStage* identifies which stage the configuration process is currently at. There are six steps in the configuration process, and each step has been assigned a value in the *CONFIGURATIONSTAGE* colour set. The colour set definition of *CONFIGURATIONSTAGE* is shown in figure 4.5.

```
colset CONFIGURATIONSTAGE = with
                            Initial        |
                            SelectMAC      |
                            LinkLocal      |
                            DAD            |
                            Global         |
                            FullyConfigured;
```

Figure 4.5: The five configuration stages.

Starting the simulation, a requester resides in the *Initial* stage, which is changed to *SelectMAC* when a MAC address is selected. Generating the link-local address changes the configuration state to *LinkLocal*. Creating the first neighbor solicitation in the duplicate address detection step changes the stage from *LinkLocal* to *DAD*. Initiating the request for a globally routable IP address changes the configuration stage to *Global*. Finally, when the address reply is received, and the IP is extracted from the packet, the requester can update its configuration stage to *FullyConfigured*.

A set of tokens may be preassigned to a place, making them available upon model initialization. The number of tokens available at a place on simulation start, and their colours, is known as the *initial marking* of the place. An initial marking is given by a CPN ML expression, typically written to the upper right of a place, and evaluates to a set of tokens. Seven places in figure 4.4 have initial markings. An example is *Macs1*, with an initial marking determined by the CPN ML function *makeMAClist*. The function takes a list of integers, returning the entries corresponding to the integer values from a predetermined list of possible MAC values.

All transitions in figure 4.4 are *substitution transitions*. Unlike regular transitions (described later), substitution transitions are marked with a label, typically placed below or at the bottom, with the name of the module associated with it. Substitution transitions are vital ingredients in modeling a complex system because they allow for representing compound behavior in a single transition. These substitution transitions enable us to organize the model into smaller interconnected modules in a hierarchical fashion as depicted in figure 4.3. As in

conventional programming, we must define an interface through which the modules communicate. In CPN, we are only allowed to connect places to transitions or vice versa, rendering place-to-place and transition-to-transition connections illegal. A connection between a place and a transition (or vice versa) is represented by an arrow or *arc* in CPN terminology.

Providing this interface between modules are special types of places called *port/socket* places. Socket-places and port-places are connected in pairs consisting of a socket and a port place, providing the interface between substitution transition and a sub-module. An example of a socket is *Network* in figure 4.4 on page 49, modeling the communication channel, e.g., between a *Node* and the *Gateway*. In the sub-module is the port-place providing the entry point for the super-module. Port places typically carries the same name as its complementary socket, although this is not a requirement. There are three different types of port places. There are those who allow import and export of tokens, and those who only allow one of these. A port place annotated with a label *I/O*, which is an abbreviation for In/Out, allows tokens to be imported *and* exported through this place. All I/O-ports in this model are colored in solid black for easier distinction from regular places. Places annotated with *In* are only allowed to import tokens, and conversely for places with *Out*. Example of *In*- and *Out*-ports are presented in sections 4.5.1 and 4.5.3 respectively.

4.4 Modeling messages

Modeling the autoconfiguration protocol requires numerous colour set definitions, ranging from trivial definitions, e.g., *IP* being a mnemonic name for the set of strings, to more elaborate constructions. This section review the most important colour set such as the colour set definitions used to model packets.

Many types of messages are exchanged throughout the model simulation, e.g., Neighbor solicitations, Refresh requests, and Address replies. Each of these message types is represented by a colour set. In the real world, these messages are series of bits flowing through the network. Long series of bits are, however, hard for humans to comprehend, so another approach is used. Packets are translated into CPN ML records with named fields corresponding to the information stored. An example is *address replies*, which has a field, *GlobalIP*, containing the global address allocated for the requester.

Listing 4.2 shows the definitions of each message type used in the model. The comments provide a description of each field's contribution.

```
(*Address reply*)
colset ARep     = record
                  (*The scope from which it originates*)
                  SourceScope       : SCOPE        *
                  (*The scope at which it currently resides*)
                  CurrentScope      : SCOPE        *
                  (*The scope at which it is destined*)
                  DestinationScope  : SCOPE        *
                  (*The partition from which it is sent*)
                  Partition         : PARTITION    *
                  (*The destination of the packet*)
                  Destination       : IP           *
                  (*The IP that the requester is assigned*)
                  GlobalIP          : IP           *
                  (*A list of nodes through which it travels*)
                  Route             : IDlist       *
                  (*The validity period of the address*)
                  Lifetime          : INT;

(*Address request *)
colset AReq     = record
                  SourceScope       : SCOPE        *
                  CurrentScope      : SCOPE        *
                  DestinationScope  : SCOPE        *
                  (*The IP from which the request originates*)
                  Source            : IP           *
                  Destination       : IP           *
                  Route             : IDlist       *
                  Partition         : PARTITION;

(* Neighbor advertisements *)
colset NA       = record
                  SourceScope       : SCOPE        *
                  CurrentScope      : SCOPE        *
                  DestinationScope  : SCOPE        *
                  Partition         : PARTITION    *
                  Source            : IP           *
                  (*The target IP*)
                  Target            : IP           *
                  Route             : IDlist       *
                  Destination       : IP           *
                  (*The ID of the receiving node*)
                  ReceiverID        : ID;

(* Neighbor request *)
colset NR       = record
                  Source            : IP           *
                  SourceScope       : SCOPE        *
                  Partition         : PARTITION    *
                  Route             : IDlist       *
                  (*An unique ID identifying the node*)
```

```
                ReceiverID          : ID;

(* Neighbor reply *)
colset NRep     = record
                SourceScope         : SCOPE         *
                CurrentScope        : SCOPE         *
                DestinationScope    : SCOPE         *
                Partition           : PARTITION     *
                Source              : IP            *
                Route               : IDlist        *
                Destination         : IP            *

(*Neighbor solicitation*)
colset NS       = record
                SourceScope         : SCOPE         *
                CurrentScope        : SCOPE         *
                DestinationScope    : SCOPE         *
                Partition           : PARTITION     *
                Source              : IP            *
                Target              : IP            *
                Destination         : IP            *
                Route               : IDlist        *
                ReceiverID          : ID;

(*Refresh request*)
colset REFRESH = record
                Source              : IP            *
                (*The prefix that is being refreshed*)
                Prefix              : PREFIX        *
                Partition           : PARTITION     *
                ReceiverID          : ID;

(*Address Refresh*)
colset AR       = record
                Partition           : PARTITION     *
                CurrentScope        : SCOPE         *
                Prefix              : PREFIX        *
                (*Is the reply from a proxy node?*)
                FromProxy           : BOOL;
```

Listing 4.2: Colour set definitions of each message used.

In combination with the comments, most fields are self-explanatory, but one that may need a more detailed description is the *Route*-field found, e.g., in the *AREP* colour set. When a request is created, the requester adds its own ID to the *Route*-list. If the request needs to be forwarded, e.g., an address request forwarded to the gateway, the intervening nodes add their own ID to the front of the list. When the request reaches its intended target, it will create the response and pass it along the reverse path, i.e., to the head of the list. In forwarding a *reply*, the node must remove the first element of the list (its own ID), and forward

it to the next list entry. Sending the response back on the exact reverse path might not be the correct approach in a real routing protocol, but this aids the simulations described in chapter 5.

The colour set *PACKET* is a *union* of all packet types of listing 4.2. By creating this union of data types, a constructor expression must be provided, and the actual *PACKET* colour set provides these constructors. Listing 4.3 shows the definition of the *PACKET* colour set.

```
colset PACKET = union
                NS             : NS      + (*Neighbor solicitation*)
                NA             : NA      + (*Neighbor advertisement*)
                NR             : NR      + (*Neighbor requests*)
                NRep           : NRep    + (*Neighbor reply*)
                AReq           : AReq    + (*Address request*)
                ARep           : ARep    + (*Address reply*)
                RefreshRequest : REFRESH + (*Refresh request*)
                AddressRefresh : AR;       (*Address Refresh*)
```

Listing 4.3: The colour set definition of *PACKET*

Constructing, e.g., an *address refresh* message is a matter of applying the constructor to a record containing the required fields. As an example, an address refresh message is represented by the following ML value:

```
AddressRefresh({Partition = part1, CurrentScope = 2,
                Prefix = 1111:1111:1111, FromProxy = false})
```

4.5 Modules

This section provides a detailed description of the CPN model, with a named section for each of its sub-pages.

4.5.1 MANET

The topmost page, MANET, has already been briefly described in section 4.3, and shown in figure 4.4. To summarize, it consist of two *Requester* substitution transitions (*Requester1* and *Requester2*), a *Node*, a *Proxy* and a *Gateway*. Not previously described are the two substitution transitions *UpdateConfig1* and *UpdateConfig2*. Figure 4.6 shows the associated module.

The purpose of this module is to model topology changes as the simulation progresses. It consists of four places and two regular transitions. Like places, arcs can be annotated with general CPN ML inscriptions, evaluating to a set of tokens. The arc inscriptions in figure 4.6, such as *partition* on the arc from

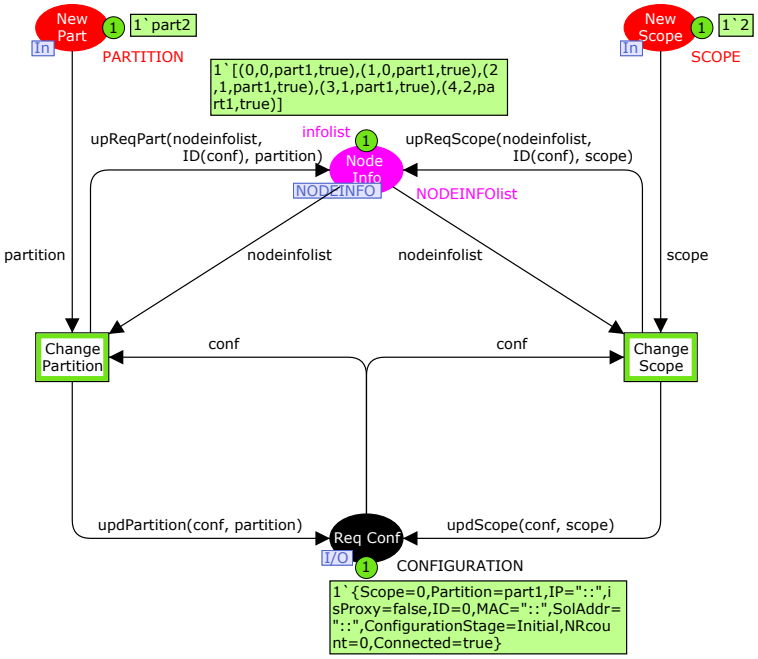

Figure 4.6: The *Update Configuration* module *before* firing any transitions.

NewPart to *ChangePartition*, are trivial variable bindings. A variable *partition* is defined having the type *PARTITION*, and the token(s) on *NewPart* can be bound to this variable. Only one token is available on these places, but if multiple tokens were present, the variable could be bound to either of these tokens. Variables are defined in a CPN model by the keyword **var**, followed by a name and type:

```
var partition : PARTITION;
```

We can consider a *binding* from figure 4.6:

$$
\begin{aligned}
&\langle scope = 2 \\
&\quad conf = \{Scope = 0, Partition = part1, IP = \text{"::"}, isProxy = false, \\
&\qquad\quad ID = 0, MAC = \text{"::"}, SolAddr = \text{"::"}, \\
&\qquad\quad ConfigurationStage = Initial, NRcount = 0, \\
&\qquad\quad Connected = true\} \\
&\rangle
\end{aligned}
$$

that bind the variable *scope* to *2* and *conf* to the CPN ML record. For this binding, the arc expression on the arc from *NewScope* evaluate to following token colour:

$$scope \rightarrow 2$$

where → should be read as "evaluates to".

A green border surrounding a transition in figure 4.6 indicates that the transition is *enabled*, i.e., it is able to occur with the current marking. The occurrence of a transition is known as *firing a transition*. Before a transition can become enabled, there must exist *bindings* for all variables on the surrounding arc expression. In the above example, only one enabled binding of *scope* exists, since only one token is available on the connected place. If multiple tokens were available on *NewScope*, the variable *scope* may be bound to either of the tokens. The arc expression of each incoming arc must evaluate to a multi-set of token colour on the place from which the arc originates. An arc from a place P to a transition T will *remove* a set of tokens from P, while an arc from T to P will *add* a set of tokens to P. A CPN construction can disable an otherwise enabled transition. This construct is called a *guard* and is introduced in section 4.5.5. When firing either *ChangePartition* or *ChangeScope*, the token on place *NodeInfo* is also updated. The colour set *NODEINFOlist* represents a list containing information about the nodes in the MANET. *NODEINFO* is a four tuple:

```
colset NODEINFO = product ID        *
                          SCOPE     *
                          PARTITION *
                          CONNECTED;
```

The individual entries of *NODEINFO* maps an ID to the scope and partition at which its requester belong, and whether or not the node is connected to the MANET. This information is used throughout the model, and its use is described later. Place *NodeInfo* is a special type of place known as a *fusion place*. All places tagged with the same label, e.g., *NODEINFO* in figure 4.6 share the same marking. For easier distinction, fusion sets are colored purple in this model. Fusion places enables us to make the *NODEINFO*-list available on other pages without using port/socket connections.

By firing *ChangePartition*, the token on *NewPart* and *ReqConf* is removed, and a new configuration token with updated information is added to *ReqConf*. Figure 4.7 depicts the *Update Configuration* module *after* the transitions have fired, showing the updated configuration token on *ReqConf* and *NodeInfo*. Firing transition *ChangePartition* simulates a partitioning of a node. Therequester is now only capable of receiving packets originating from nodes within its new partition. Changing the scope, i.e., firing *ChangeScope*, simulates a change in connectivity. The arc from transition *ChangePartition* to place *ReqConf* shows a non-trivial arc inscription. The inscription *updPartition(conf, partition)* is a CPN ML function, taking an existing configuration (bound to the variable *conf*), and returns a new configuration token with the newly updated partition.

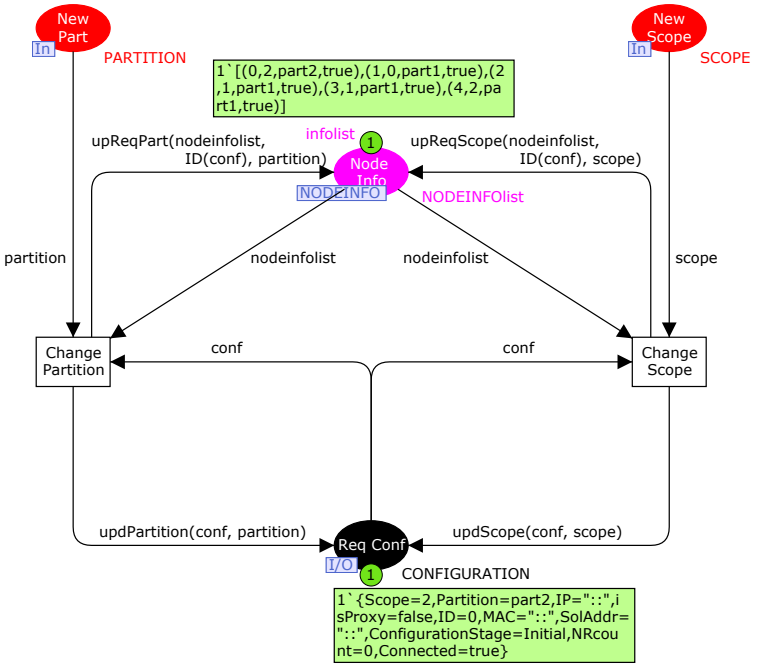

Figure 4.7: The *Update Configuration* module **after** firing *change scope* and *change partition*.

The place *ReqConf* in figure 4.6 is an example of an *I/O port*-place, allowing the module to import and export configuration tokens, while *NewPart* and *NewScope* are examples of *In*-ports. These places are only allowed to *import* tokens via their corresponding socket-place on the *MANET*-page. To separate the input ports from regular places, they are colored in a solid red in this model.

In figure 4.6 and 4.7 we saw how the *NODEINFO*-list and configuration was updated with the information stored at *NewPart* and *NewScope*. This corresponded to two steps in the CPN model, with each step represents an occurrence of a transition with enabled bindings. These steps can be written as a pair consisting of a transition, and occurring binding hereof. In CPN terminology, this pair is known as a *binding element*. Table 4.1 shows the binding elements for the transitions in figure 4.6, in which *ChangePartition* is fired first, and *ChangeScope* secondly. Two binding elements are said to be in *conflict* if both are simultaneously enabled, but only one of them can occur. Section 4.5.3 will show an example of conflicting binding elements.

Step	Binding element			
1	(*ChangePartition*, ⟨	*partition*	=	part2,
		conf	=	{Scope = 0, Partition = part1, SolAddr = "::",isProxy = false, IP = "::", ID = 0, MAC = "::", ConfigurationStage = Initial, NRcount = 0, Connected = true},
		nodeinfolist	=	[(0,0,part1,true), (1,0,part1,true), (2,1,part1,true), (3,1,part1,true), (4,2,part1,true)]
	⟩			
	)			
2	(*ChangeScope*, ⟨	*scope*	=	2,
		conf	=	{Scope = 0, Partition = part2, SolAddr = "::",isProxy = false, IP = "::", ID = 0, MAC = "::", ConfigurationStage = Initial, NRcount = 0, Connected = true},
		nodeinfolist	=	[(0,0,part2 ,true), (1,0,part1,true), (2,1,part1,true), (3,1,part1,true), (4,2,part1,true)]
	⟩			
	)			

Table 4.1: Binding elements for the *Update Configuration* module.

4.5.2 Requester

Figure 4.8 shows the *Requester* sub-module. It consist of six substitution transitions, one for each step in the autoconfiguration process, one for handling the incoming packets, and one for sending address refresh packets. The latter is only enabled if it has become a proxy node.

The configuration parameters are key components of a node. These are stored in a CPN ML record with named fields holding the information shown in listing 4.1 on page 50. This configuration token is needed by each of the six modules, and resides on place *ReqConf*. Connecting this place to each of the six modules are double-headed arrows. A *double arc*, as it is known, is an abbreviation for two separate arcs, pointing in opposite directions with the same arc inscription.

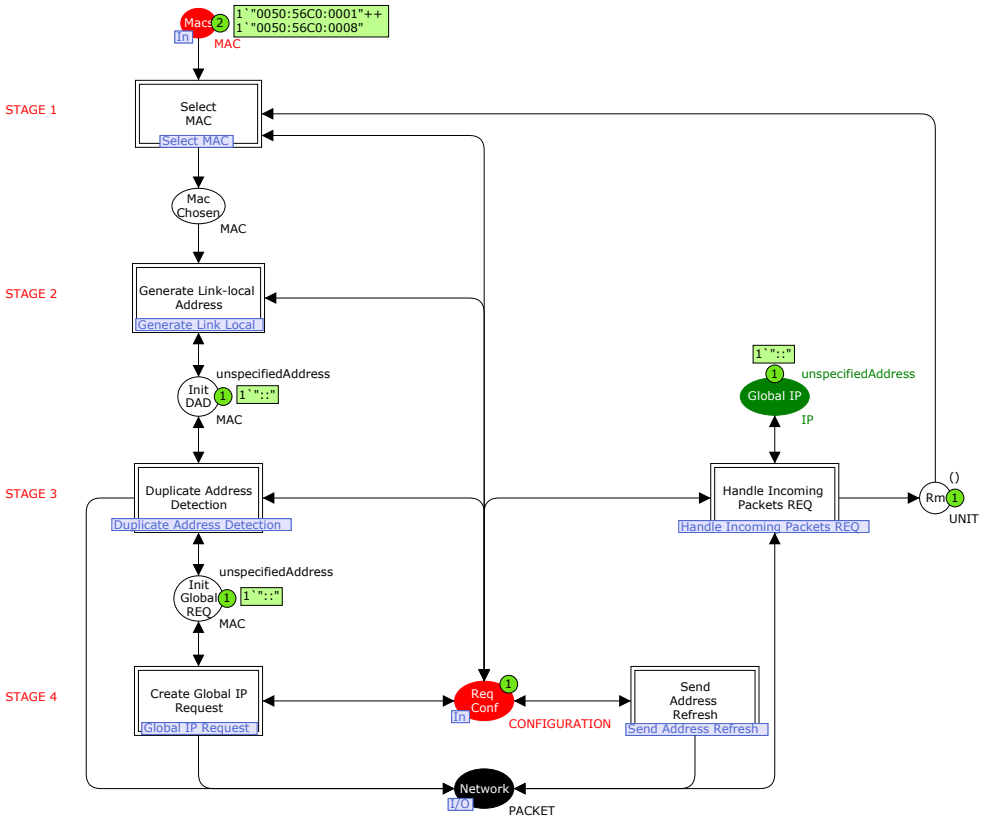

Figure 4.8: The requester page.

Firing a transition connected with double arcs will technically remove the token, and add a new token with an identical colour.

The initial step of configuring a node is handled by the *Select MAC*-module (section 4.5.3). This module has an additional input socket connected to it. Place *Macs* contains a multi-set of MAC addresses that the requester can choose from. The *Select MAC* module communicates with *Generate Link-local Address* via the place *MacChosen*. Generating the link-local address used is handled by the *Generate Link Local* module. Duplicate address detection must be completed on the generated address, and is handled in the dedicated module. The *Duplicate Address Detection* module is connected to *Network* in order to transmit neighbor solicitations. After successfully completing this step, the request for a globally routable IP address can be initiated. The *Duplicate Address Detection* module sends a token to place *InitGlobalREQ*, and the *Global IP Request* module creates and transmit this request. *Send Address Refresh* is only enabled if the node is promoted to proxy, and is responsible for sending periodically address refresh messages to the gateway. The final substitution transition on the page

is *Handle Incoming Packets REQ*. The associated module handles all incoming packets, and is responsible for creating the appropriate response (if any). Receiving a solicitation or advertisement for the generated link-local address, adds a token to the *Rm* place. A token on this place enables the *Select MAC* module to select a new MAC address, and the configuration process resets. The place *GlobalIP* will eventually contain the IP allocated for the requester, assuming the autoconfiguration process is successfully completed.

4.5.3 Select MAC

A user is not capable of selecting a MAC address for a specific interface under normal circumstances. This feature is added in order to simulate two interfaces having the same MAC address, leading to a failure in the duplicate address detection step. Figure 4.9 shows the *Select MAC* module.

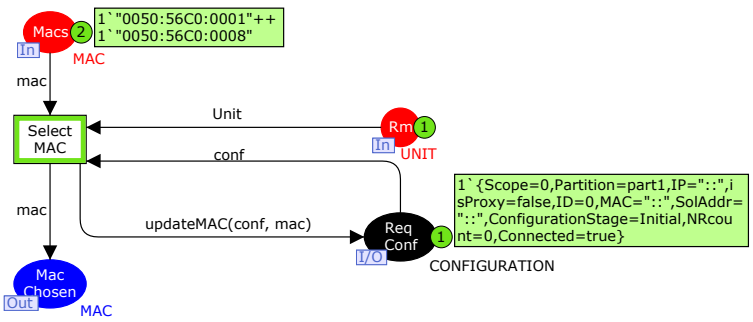

Figure 4.9: The *Select MAC* module.

The initial marking consists of two MAC addresses. This fairly limited selection of available MAC addresses is made with the formal validation in mind. The calculation time can increase manyfold by adding a third MAC address. This phenomenon is described in further detail in section 6.1. The module contains conflicting binding elements for transition *SelectMAC* in the marking shown in figure 4.9:

$$
\begin{aligned}
SM_1 = (SelectMAC, \langle Unit = (), mac = \text{"}0050:56C0:0001\text{"}, \\
conf = \{Scope = 0, Partition = part1, \\
IP = \text{" :: "}, isProxy = false, \\
ID = 0, MAC = \text{" :: "}, \\
SolAddr = \text{" :: "}, \\
ConfigurationStage = Initial, \\
NRcount = 0, Connected = true\} \\
\rangle \\
)
\end{aligned}
$$

$$
\begin{aligned}
SM_2 = (SelectMAC, \langle Unit = (), mac = \text{"}0050:56C0:0008\text{"} \\
conf = \{Scope = 0, Partition = part1, \\
IP = \text{" :: "}, isProxy = false, \\
ID = 0, MAC = \text{" :: "}, \\
SolAddr = \text{" :: "}, \\
ConfigurationStage = Initial, \\
NRcount = 0, Connected = true\} \\
\rangle \\
)
\end{aligned}
$$

Both binding elements are enabled initially, but only one can occur since they both need the token on *Rm*.

This page introduces three new constructs to the CPN arsenal. Firing the *SelectMAC* transition will select one of the MACs residing on place *MACs*. This token is added to place *MacChosen*, which is an example of an output-port. Output ports in this model are colored in a solid blue, and has a label *Out* positioned typically to the lower left. In this particular page-instance, *MACs* has an initial marking consisting of two tokens with colour `0050:56C0:0001` and `0500:56C0:0008`. Prefixing the tokens is a 1´, which indicate that a single token of that particular colour in the set of MAC addresses. The binary operator ´ takes a number, and a token, producing a multi-set of tokens. Between the two tokens is the binary operator ++. This operator takes two multi-sets and returns their union. The arc from transition *SelectMAC* to place *ReqConf* shows another example of a non-trivial arc inscription. *updateMAC(conf, mac)* is a CPN ML function taking an existing configuration, and a MAC address, updating the MAC field hereof. While updating the MAC entry in the configuration token, the *Con-*

figurationStage is changed to *SelectMAC*. On the right side of the page is a place, *Rm*, which is connected to the corresponding *Rm*-place on the *Requester* page. This input port will receive a token if duplicate addresses were detected. Figure 4.10 shows the *Select MAC* module after firing the *SelectMAC* transition in the binding SM_2.

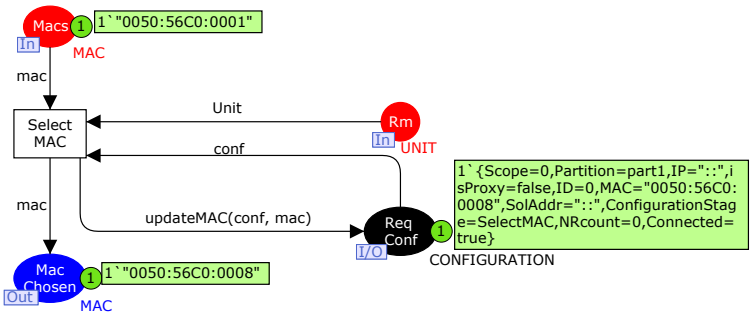

Figure 4.10: The *Select MAC* module *after* the transition *SelectMAC* has fired.

4.5.4 Generate Link Local

After having selected the MAC address, a link-local address must be generated. This is taken care of in the *Generate Link Local* module, depicted in figure 4.11.

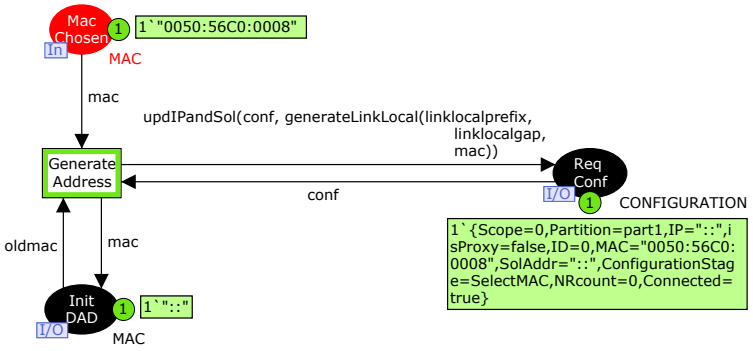

Figure 4.11: The *Generate Link Local* module *before* generating the link local address.

This is a relatively simple module consisting of only three places and a single transition. The function *updIPandSol* is responsible for generating the address and updating the requester's configuration token. A link-local address is generated based on the well-known link-local prefix together with the selected MAC address. These parameters are necessary components in order to construct the EUI-64 identifier (section 3.2) used for generating the link local address. Figure 4.12 shows the updated configuration token after *GenerateAddress* has occurred.

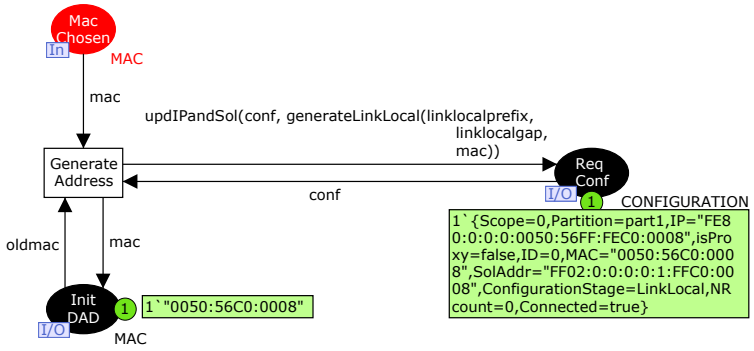

Figure 4.12: After generating the link local address.

The *generateLinkLocal* function is shown in listing 4.4.

```
fun generateLinkLocal(prefix : PREFIX, gap : STRING, mac : MAC) =
let
  val partOne = if String.size mac >= 8 then
                   String.substring(mac : STRING, 0, 7)
                else
                   ""
  val partTwo = if String.size mac >= 8 then
                   String.extract(mac : STRING, 7, NONE)
                else
                   ""
in
  if String.size mac >= 8 then
     prefix ^ partOne ^ gap ^ partTwo : IP
  else
     "ERROR"
end
```

Listing 4.4: The *generateLinkLocal* function.

This function is relatively straightforward. It splits the incoming MAC address in two parts, and wedges the "gap filler" `FFFE` in between. The final step is to append this to the link-local prefix. While updating its configuration, the MAC address token is sent to the *InitDAD* place, and the old MAC is removed. The removal of this old MAC address is necessary when the requester selects a new MAC due to duplicate link-local addresses.

4.5.5 Duplicate Address Detection

The constructed link-local address must pass the duplicate address detection phase. Transmitting neighbor solicitations as part of this step is taken care of in the module depicted in figure 4.13.

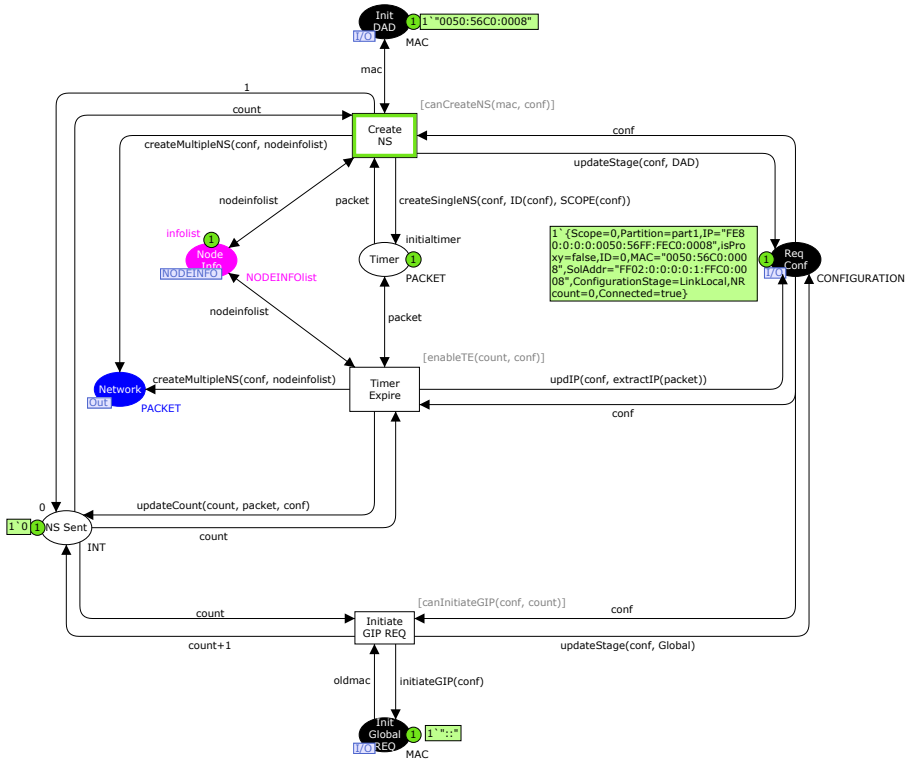

Figure 4.13: The *Duplicate Address Detection* module.

Checking for address conflicts is accomplished by sending a neighbor solicitation, listing for response. Firing *CreateNS* adds neighbor solicitations at two places while updating the current *ConfigurationStage* to *DAD*. As described in section 3.3, the requester must retransmit solicitations a number of times. Therefore, a copy of the transmitted neighbor solicitation is added to *Timer* for later retransmission. Connecting *CreateNS* and *Network* is an arc with the inscription *createMultipleNS(conf).Network* is connected to *Network* on the *Requester*-page, providing the outgoing buffer for message transmission. To simulate real network behavior, each node must intercept the packet, and decide whether or not to process it, or discard it. To achieve this behavior, *createMultipleNS* generate multiple neighbor solicitations. To simplify the model, neighbor solicitations is only generated for requesting nodes, as they are the only nodes involved in address clashed. To simulate loss of packets if the receiving node belongs to a different partition, *createMultipleNS* uses the *NODEINFOlist* stored at *NodeInfo*. If the node transmitter and receiver belongs to different partitions, *createMultipleNS* returns the empty set. Thus we simulate a that a solicitations never reaches its intended target(s). This pattern is repeated throughout the model whenever a request and/or response is transmitted. Upon reception, the node determines which action must be taken based on its own configuration. Each

node has a unique ID stored in its configuration token. Only if a node's ID and the ID stored in the solicitation agree, can it intercept the packet and act accordingly. Simulating timer expiration is the purpose of transition *TimerExpire*, which resends neighbor solicitations. In the top left corner of *TimerExpire* we find an expression, *enableTE(count, conf)*, between a set of square brackets. This expression is called a *guard*, and must be a *list* of Boolean expressions. Guards are used to enable or disable transitions, only allowing them to fire if the sufficient amount of tokens is available *and* all guard expressions evaluates to true. For easier distinction between guard expressions and regular inscriptions, guards are colored in a dark grey in this model. The guard associated with *TimerExpire* ensures that retransmission only occurs *DADlimit* times. When this limit is reaches and no advertisements have been received, the *InitiateGIPREQ* (initiate global IP request) is enabled, and hand control over to the *Create Global IP request* substitution transition. Finally, the *ConfigurationStage* is updated to *Global*. Figure 4.14 shows the module after sending the first solicitation.

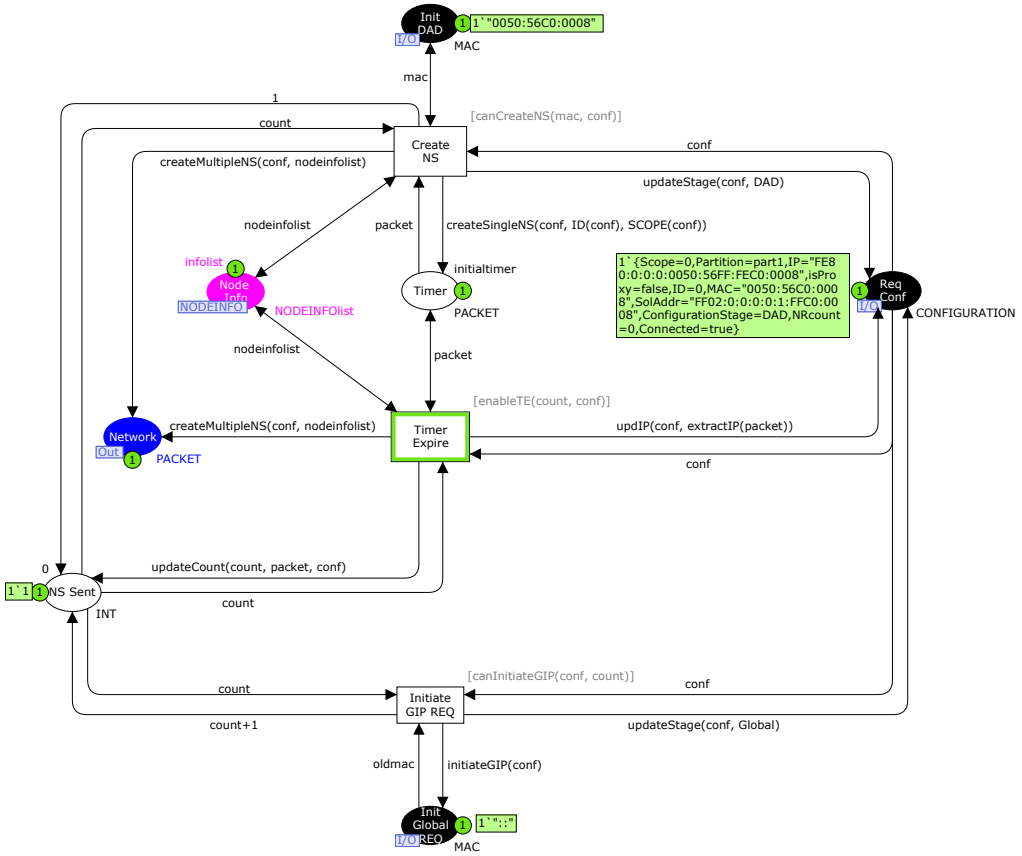

Figure 4.14: The *Duplicate Address Detection* module after sending the first neighbor solicitations.

4.5.6 Global IP Request

Figure 4.15 shows the *Global IP Request* module.

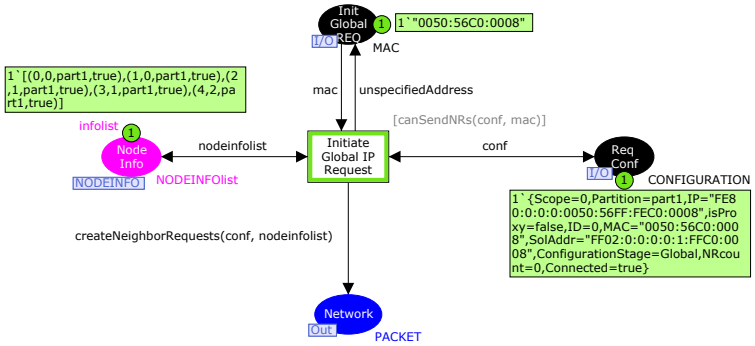

Figure 4.15: The *Global IP Request* module.

The purpose of this module is to initiate a request for a globally routable IP address by broadcasting neighbor requests, connecting the requester to an initiator. The information stored in the *NODEINFO* list determines to whom the requester must send these requests. Since the requester only has a link-local address, it can only communicate with nodes within a one-hop distance. The function *createNeighborRequests* utilizes the scope information stored in the list at *NodeInfo* to determine these reachable nodes. To be eligible for receiving neighbor requests, the node must be a fully configured node, while non-configured, non-connected nodes, and nodes belonging to a different partition are ignored. The code for creating neighbor requests is shown in listing 4.5.

```
fun createNeighborRequests(conf as {IP = ip, Partition = part,
                                    Scope = s,
                                    ConfigurationStage = cs,
                                    ID = sid,...} : CONFIGURATION,
                           list : NODEINFOlist) =
let

fun inList(nil,_) = false
|   inList( (i,s)::xs, id : ID) =
    if i = id then true else inList(xs, id)

fun inBroadcastList(nil, _) = false
|   inBroadcastList(i::xs, id) =
    if i = id then true else inBroadcastList(xs, id)

val withinscopelist =
List.filter(fn (id, scope, part, con) =>
          (id <> sid) andalso withinscope(scope, s) andalso
          not(inList(requesterscopelist, id)) andalso con andalso
          inBroadcastList(broadcastRec, id)) list

fun NRaux(nil) = empty
|   NRaux((i,src, p, con)::xs) =
    if notSamePartition(conf, list, i) then NRaux(xs) else
    1`NR({Route = [sid], Source = ip, Partition = part,
          SourceScope = s, ReceiverID = i})++NRaux(xs)

in
  NRaux(withinscopelist)
end
```

Listing 4.5: Code for generating neighbor requests.

The constructed neighbor requests are transmitted via *Network*, which in turn is connected to *Network* on the *Requester* page. The module is shown in figure 4.16 after initiating the request for a global IP.

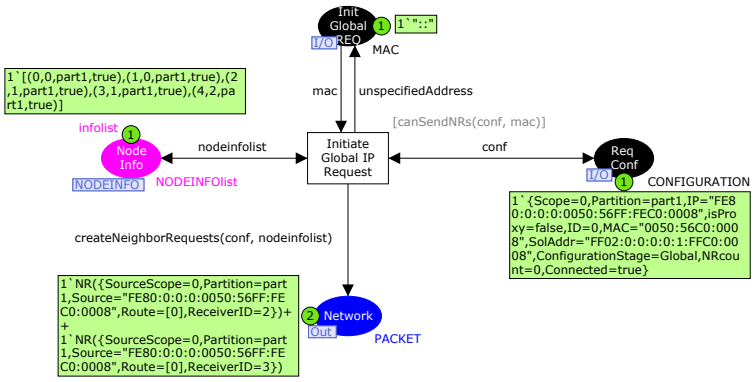

Figure 4.16: After initiating the request for a global IP.

4.5.7 Handle Incoming Packets REQ

The module *Handle Incoming Packets REQ*, shown in figure 4.17, act as a dispatcher by forwarding incoming packets to the appropriate module.

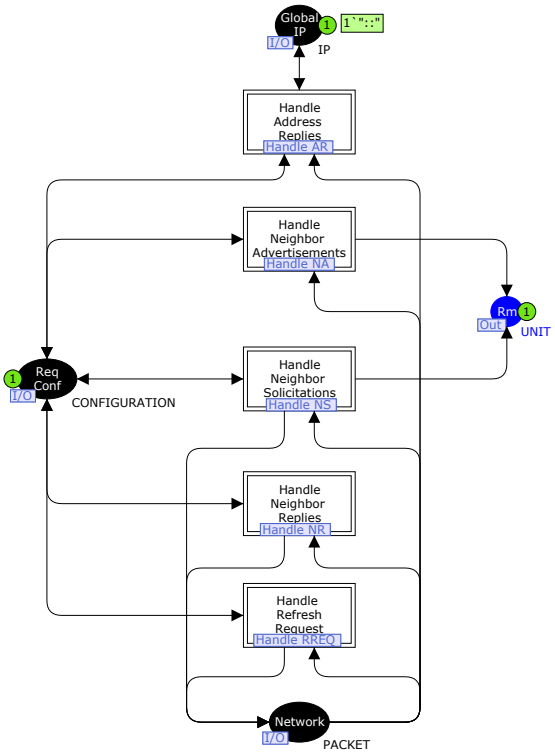

Figure 4.17: The *Handle Incoming Packets REQ* module.

Connected to substitution transitions *Handle Neighbor Solicitations* and *Handle Neighbor Advertisements* is the place *Rm* that will inform the *Select MAC* module to select a new MAC. The place is connected to *Rm* of the *Requester* page, whose role was described in section 4.5.2. *Handle Address Replies* is connected to places *GlobalIP*, which eventually will contain the IP assigned to the requester. This value is also updated in the configuration token.

The following five sections describe how the handling of each of the five response packets is modeled.

4.5.8 Handle Neighbor Solicitations (NS)

The module for handling incoming neighbor solicitations is shown in figure 4.18. Receiving a neighbor solicitation not destined for this node enables transition *DiscardNS*, and removes the solicitation from place *Network*. On the contrary, if the destination of the solicitation is the node in question, an advertisement is

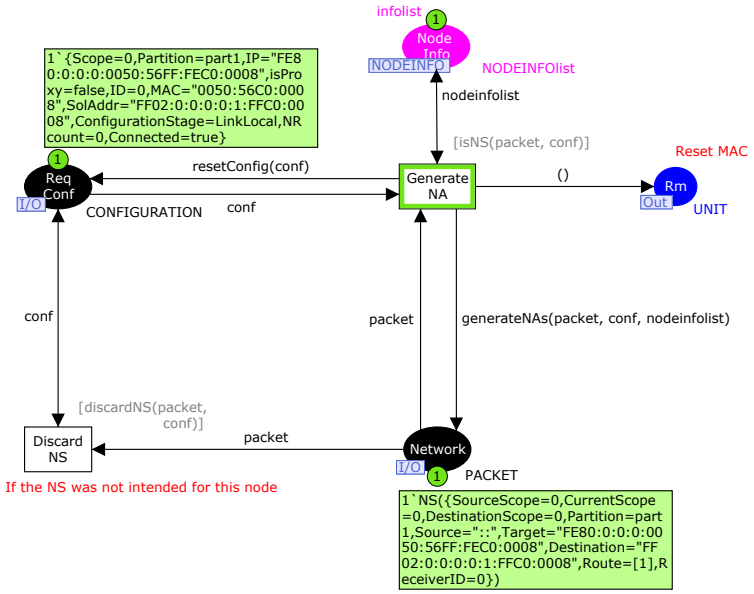

Figure 4.18: The *Handle NS* module for handling neighbor solicitations.

created, and sent by firing transition *GenerateNA*. The function responsible for generating the advertisement uses the node information stored at *NodeInfo*. If the transmitter and receiver belongs to different partitions, no advertisement is generated thus simulating a packet loss. Generating neighbor advertisements resets the configuration stage to *Initial*. A token is added to *Rm* which is made available to the *Select MAC* module. The *Handle NS* module after having generated an advertisement is shown in figure 4.19.

The neighbor solicitations are discarded if it is not intended for the receiving node. A solicitation is also discarded if the node has already received an advertisement for the generated address. Having received this advertisement, the node has already acknowledged the problem, and has initiated the selection of a new MAC address. If the node were to handle this solicitation as well, it would lead to selecting yet another MAC even though the process has been reset.

4.5.9 Handle Neighbor Advertisements (NA)

Handling of neighbor advertisements is equally simple. The module consists of two transitions and three places. The module is depicted in figure 4.20.

The transition *GetNA* removes the advertisement from *Network*, and notifies the *Select MAC* module by sending a token to *Rm*. *DiscardNA* is enabled if the node has already reacted to a neighbor solicitation by restarting the configuration process. Handling neighbor advertisements also include resetting the configuration stage to *Initial*.

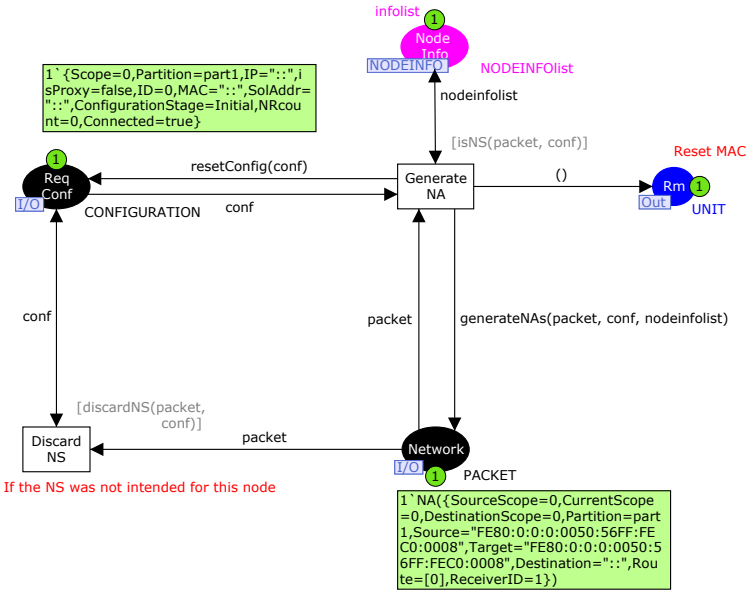

Figure 4.19: After having received a neighbor solicitation for an address which lead to duplicate link local addresses.

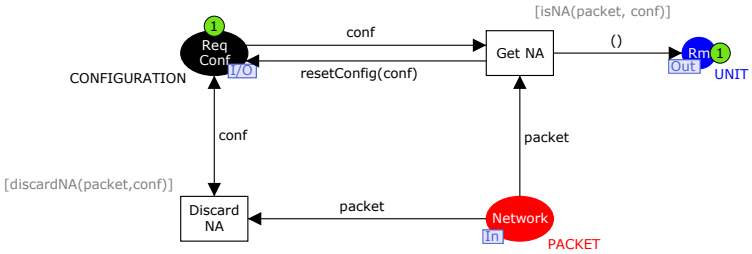

Figure 4.20: The *Handle NA* module for handling neighbor advertisements.

4.5.10 Handle Neighbor Replies (NR)

Handling incoming *neighbor replies* is the purpose of the *Handle NR* module depicted in figure 4.21.

Recall that a requester must broadcast a neighbor request and await response. Should multiple neighbors respond, the requester must select one of these replies. *GetNR* removes the incoming neighbor reply, and creates an *address request*. This request is sent back to the responder unless it has become disconnected from the MANET, e.g., partitioned from the network. The configuration tokens *NRcount*-field keeps track of how many neighbor requests have been received. Any subsequent neighbor replies are discarded by firing the newly enabled transition *DiscardNR*. Figure 4.22 shows the enabling of *DiscardNR* due to having received, and handled, one of the incoming neighbor replies.

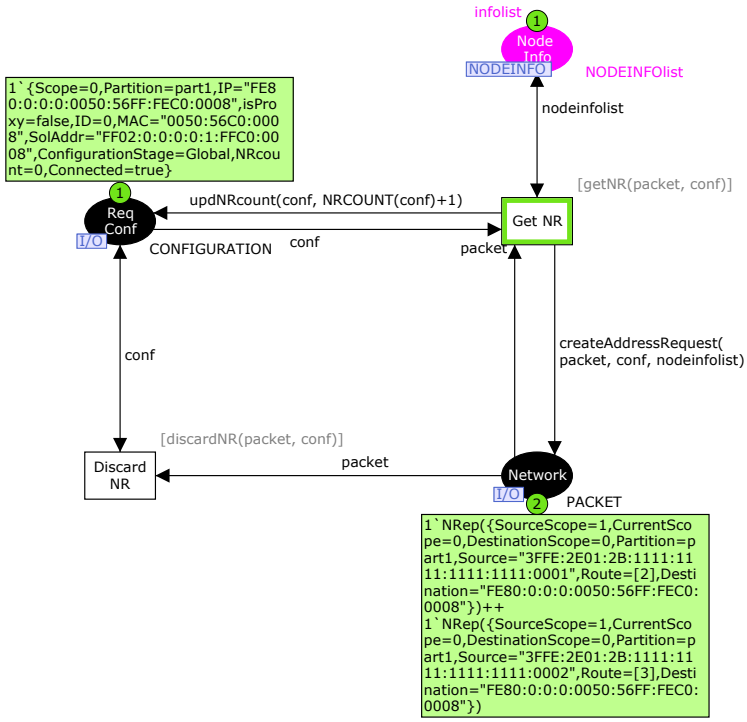

Figure 4.21: The *Handle NR* module for handling neighbor replies.

4.5.11 Handle Refresh Requests (RREQ)

Refresh requests are sent by gateways when no address refresh message is received for a given ad hoc prefix. The module responsible for handling refresh requests is shown in figure 4.23.

The refresh requests sent to a general-purpose node help the gateway to determine whether or not a prefix is unoccupied. The requester discards any incoming refresh request if it is not yet fully configured. This is achieved by firing transition *Discard*. *Discard* is also enabled if the maximum number of permitted refresh requests has been sent, determined by the token on place *RefreshCounter*. To simulate that a particular prefix has no proxy node attached, a node is degraded from proxy if the maximum number of refresh requests has been met. The inscription on the arc from *Discard* to *ReqConf* handles degrading the node to a general node. The arc adds the original configuration token to *ReqConf* as long as the limit has not reached.

If a node is fully configured and the refresh request is destined for the nodes prefix, *SendReply* is enabled (unless they belong to different partitions). The refresh request token is removed from *Network*, and firing the transition creates an address refresh packet.

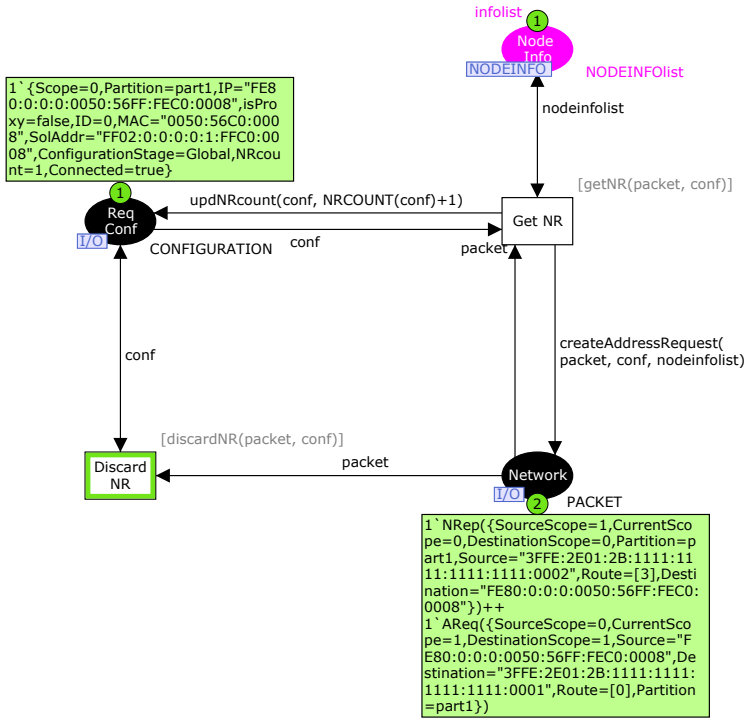

Figure 4.22: Enabling *DiscardNS* after having processed one neighbor reply.

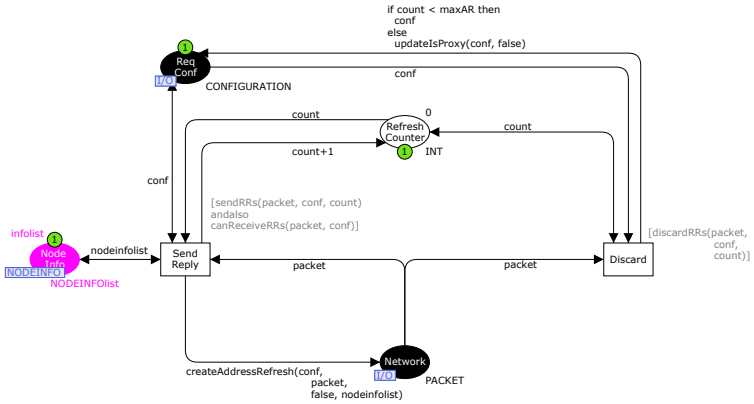

Figure 4.23: The *Handle RREQ* module for handling refresh requests.

4.5.12 Handle Address Replies (AR)

This module is responsible for handling address replies. The module is shown in figure 4.24.

The incoming reply is either received through *GetAddressReply* or *PromoteToProxy*, based on the host ID of the global address contained in the reply. Promoting a node to proxy occurs if the host ID is all 0's. Using function *create-*

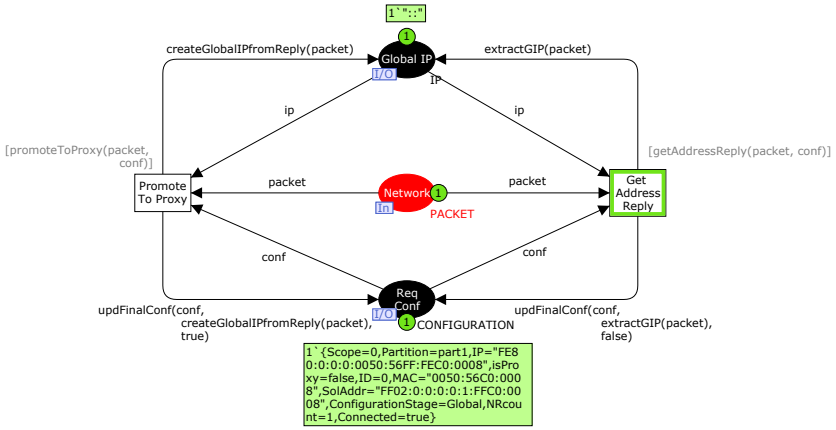

Figure 4.24: The *Handle AR* module.

GlobalIPfromReply, it allocates the first available address to itself, updates the configuration token, and adds the IP to place *GlobalIP*. If the received IP is a regular IP (a non-zero host ID), the node updates its configuration, and adds the IP token to *GlobalIP* using the function *extractGIP*. The requester is now fully configured and changes *ConfigurationStage* to *FullyConfigured*. Figure 4.25 shows the module after having processed the incoming address reply.

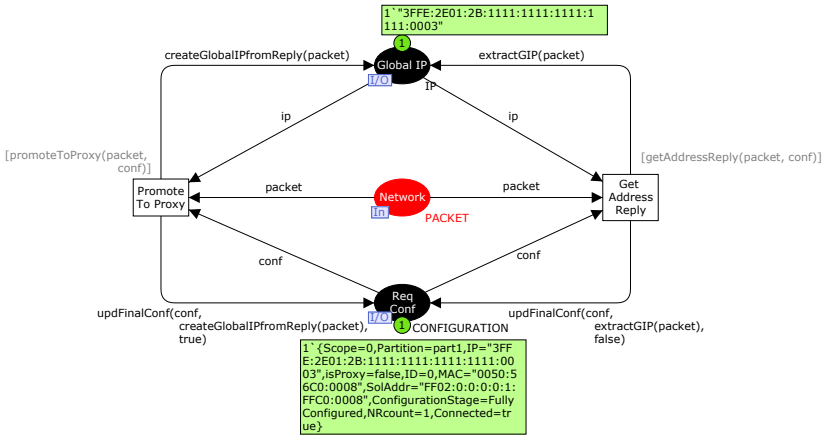

Figure 4.25: The *Handle AR* module after handling the incoming address reply.

4.5.13 Send Address Refresh

Figure 4.26 shows the module for sending address refresh packet.

The *Send Address Refresh* module consists of four places and a single transition. Recall that a proxy node must periodically send address refresh messages to refresh its prefix lifetime. When transition *SendAddressRefresh* is fired, an

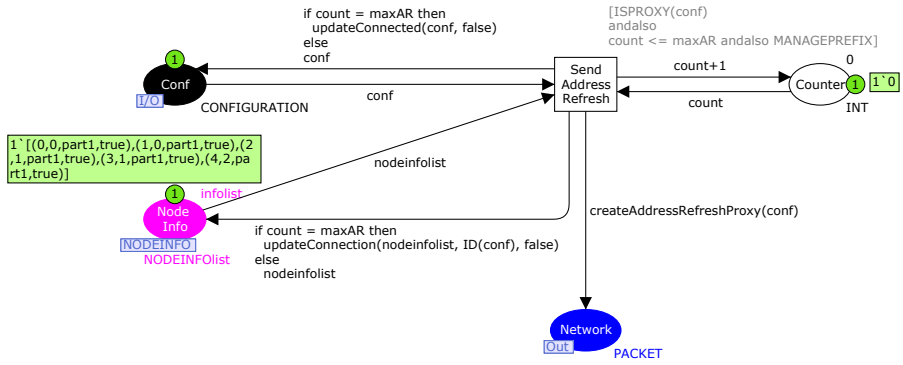

Figure 4.26: The *Send Address Refresh* module.

address refresh message is created, and sent via *Network*. The allowed number of address refresh packets is limited to a maximum number (*maxAR*). When this limit has reached, the proxy is no longer able to send these messages. In a real situation, a loss of network connectivity may cause the lack of address refresh messages. Updating the list of *NODEINFO*, by altering the proxy node's connected-entry to false, simulates this.

4.5.14 Node

Figure 4.27 shows the sub-module representing a fully configured, non-gateway node.

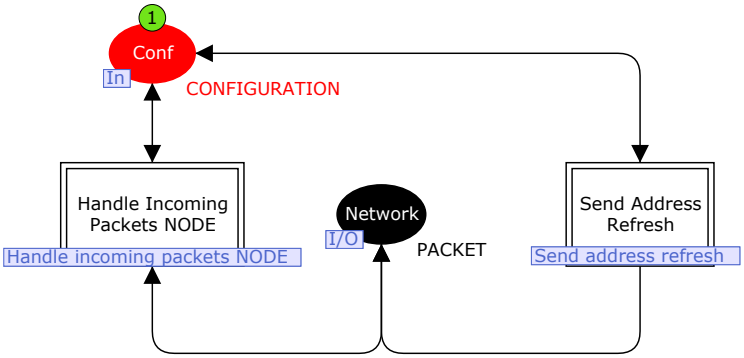

Figure 4.27: The page representing a *Node*.

The *Node* page has two substitution transitions, one for handling incoming packets, and one for sending address refresh messages. The latter was described in section 4.5.13. Similar to the *Requester* page, a central component of a *Node* is the configuration token on place *Conf*. The *Handle Incoming Packets NODE* module is otherwise similar to that of the requester, except the type of packets it must handled. The module is described in more detail in the following section.

4.5.15 Handle Incoming Packets NODE

The module for handling a *Node*s incoming packets is similar in structure to that of a requesting node. The *Handle Incoming Packets NODE* module is shown in figure 4.28.

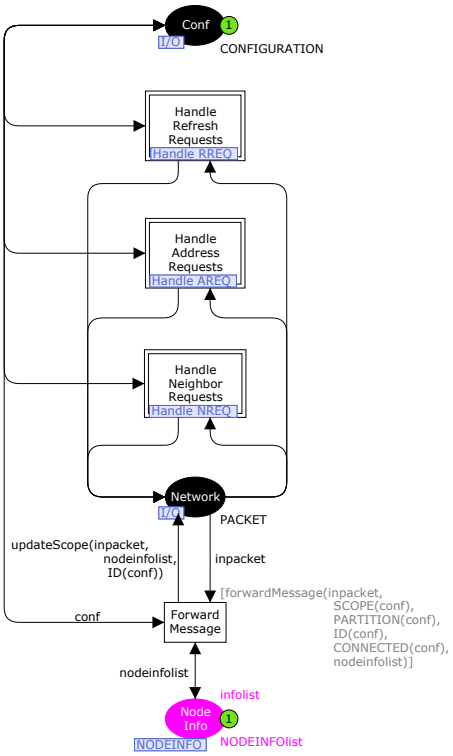

Figure 4.28: The *Handle Incoming Packets NODE* module.

A *Node* needs only to handle three types of packets, namely refresh request, address request, and neighbor requests. A substitution transition represents each of the three message types. Opposed to handling of packets at the *Requester*, this module has an additional transition *ForwardMessage*. Recall that every configured node in a MANET participates in the routing of packets, forwarding them as a specialized router in a traditional network. The *ForwardMessage* transition helps simulating the routing of messages. If a packet is destined for a scope different than the *CurrentScope*-field, a *Node* can intercept this packet and forward it. Listing 4.6 shows the guard function, and its auxiliary functions that decides whether a packet needs forwarding or not. The function may update packets of type *address reply*. The other types of packets cannot be forwarded since the requester only has a link-local address, which is limited to a one-hop range. In case of a non-proxy initiator, the address request is forwarded to the gateway, and is handled by the *Handle AREQ* module described in section 4.5.16.

```
fun forwardMessage(ARep{CurrentScope = cs, DestinationScope = ds,
                        Partition = part, Route = r, ...},
                   myscope : SCOPE, p : PARTITION,
                   id : ID, con : BOOL,
                   userlist : NODEINFOlist) =

forwardAux(cs, ds, myscope) andalso (part = p) andalso
not(userMoved(List.last r, userlist, ds)) andalso
hd r = id andalso con

|     forwardMessage(_) = false


(* Auxiliary functions *)
fun forwardAux(current : SCOPE, dest : SCOPE, myScope : SCOPE) =
(current <> dest) andalso (current = myScope)

fun userMoved(userid : ID, list : NODEINFOlist, ds : SCOPE) =
let
  val (id, scope,_,_) = List.nth(list, userid)
in
  scope <> ds
end
```

Listing 4.6: Guard function *forwardMessage* and its auxiliary functions.

A *Node* is allowed to forward a packet if its *CurrentScope* and *DestinationScope* differ, and it currently resides within its own scope. Forwarding basically consists of updating the current scope based on information about the *source scope* and *destination scope*. Based on this information the node can decide whether to increment or decrement the *CurrentScope*-field. The updating of scope is handled by function *updateScope*. *updateScope* uses an auxiliary function, *updateScopeAux*, for determining the new value of *CurrentScope*. This function is shown in listing 4.7.

```
fun updateScopeAux(sourcescope : SCOPE,
                   currentscope : SCOPE,
                   destinationscope : SCOPE ) =

if destinationscope < sourcescope then
  currentscope-1
else
  (if destinationscope = sourcescope then
      currentscope
   else
      currentscope+1)
```

Listing 4.7: The auxiliary function updating the current scope when forwarding

messages.

The module for handling refresh requests is exactly the same as that of the requester described in section 4.5.11. The remaining modules are described below.

4.5.16 Handle Address Requests (AREQ)

Handle AREQ is responsible for handling address requests, and is depicted in figure 4.29.

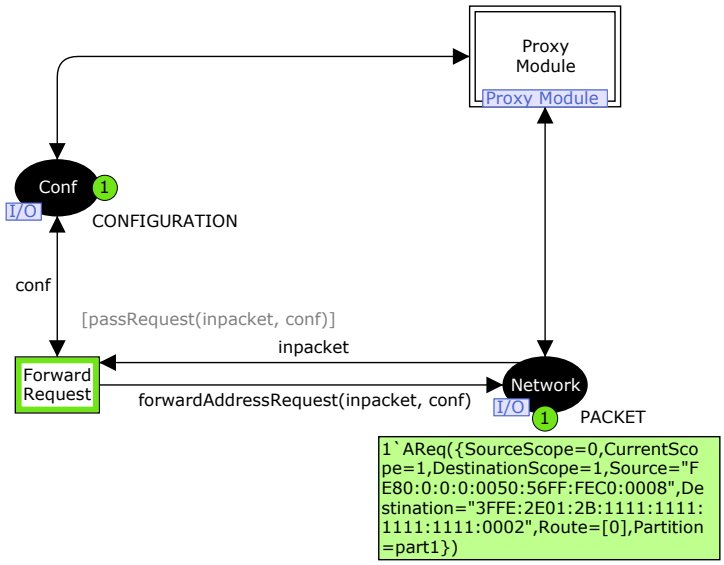

Figure 4.29: The *Handle AREQ* module for handling address requests.

The incoming address request can follow one of two paths. If the node is a *Proxy*-node, the incoming request is sent to the *Proxy Module* substitution transition. In case of a non-proxy node, it simply updates the packets destination using *forwardAddressRequest* when firing transition *ForwardRequest*. The new destination is the gateway, and the *CurrentScope* is updated according to the gateway's scope. Figure 4.30 depict the newly updated address request, which now is destined for the gateway.

4.5.17 Proxy module

The proxy module is a fairly simple module, consisting of three places and a single transition. The module is shown in figure 4.31.

Recall that the typical proxy node uses reserves first available host ID as part of its own IP address. When the address request enters the proxy module, *CreateReply* is fired, choosing the next available host ID. This packet is lost if the receiver belongs to another partition, but the *NewAddr* token is still incremented.

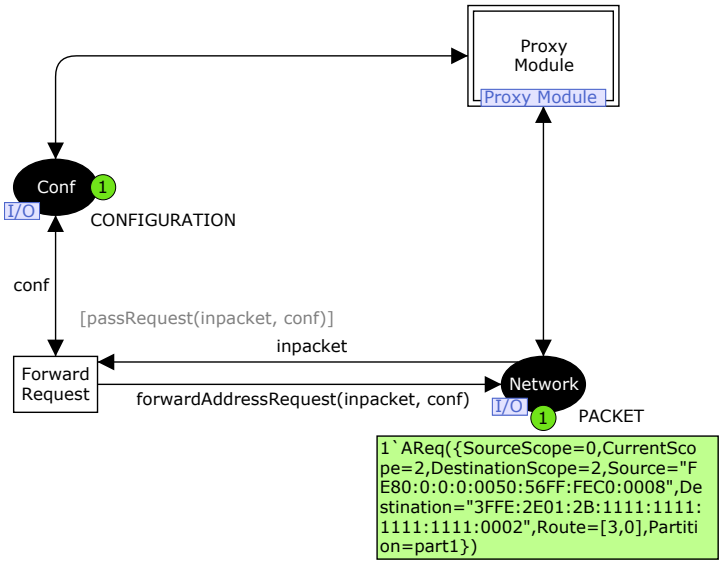

Figure 4.30: The *Handle Address Request*-module after forwarding the incoming address request.

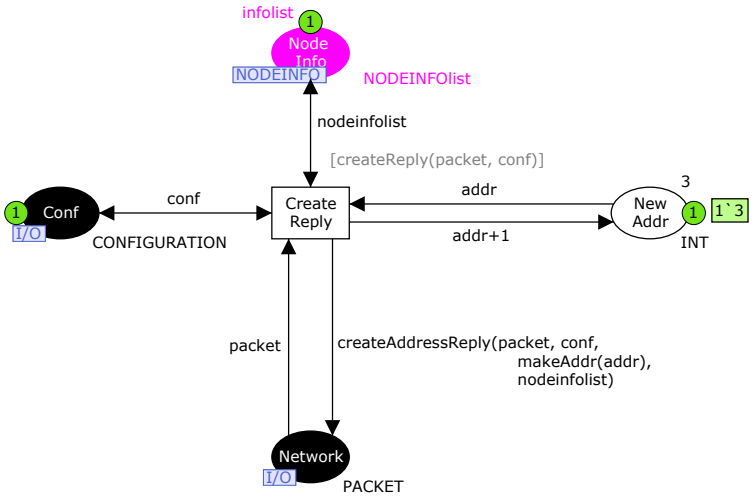

Figure 4.31: The *Proxy Module* for performing proxy duties.

The specification[15] does not state how this host ID is chosen, but only that 0000 is reserved and that the proxy typically uses the first available host ID as its own. The simplest scheme is an incremental host ID, starting at the first unoccupied value. This scheme is adopted in this model. The first available ID is 0003, since 0000 is reserved, and *Proxy* and *Node* occupy 0001 and 0002 respectively. It chooses the value residing on place *NewAddr*, and increments it. Function *createAddressReply* converts this number into a proper IP address, which is sent

in an address reply packet.

4.5.18 Handle Neighbor Requests (NREQ)

Handling incoming neighbor requests is the purpose of the *Handle NREQ* module depicted in figure 4.32.

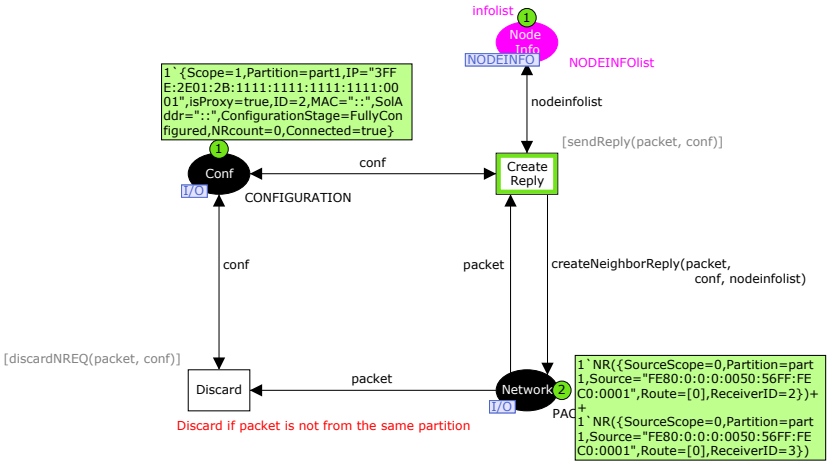

Figure 4.32: The *Handle NREQ* module for handling neighbor requests.

Handling neighbor request is likewise a relatively simple procedure. It takes the incoming request, creates a reply based on information in its configuration token and the packet. Only if the *Node*'s ID matches *ReceiverID* of the request can it be removed by the node. The *Discard* transition is enabled if the request originated from the other partition. This simulates the situation where a request never reaches its target because they belong to two seperate MANETs. Figure 4.33 shows *Handle NREQ* after having created a reply.

4.5.19 Gateway

The gateway module only consists of one substitution transition for handling incoming packets, and one transition for forwarding packets. The module is shown in figure 4.34.

The configuration token is also central to the gateway, and resides on place *GWConf*. The purpose of *ForwardMessage* was described in section 4.5.15.

4.5.20 Handle Incoming Packets GATEWAY

Handling incoming packets at the gateway is no different from handling packets at e.g. the *Requester*. The *Handle Incoming Packets GATEWAY* page (figure 4.35) act as a dispatcher, sending the packet to *Handle Address Refresh*, *Handle Neighbor Requests*, or *Create Address Response*.

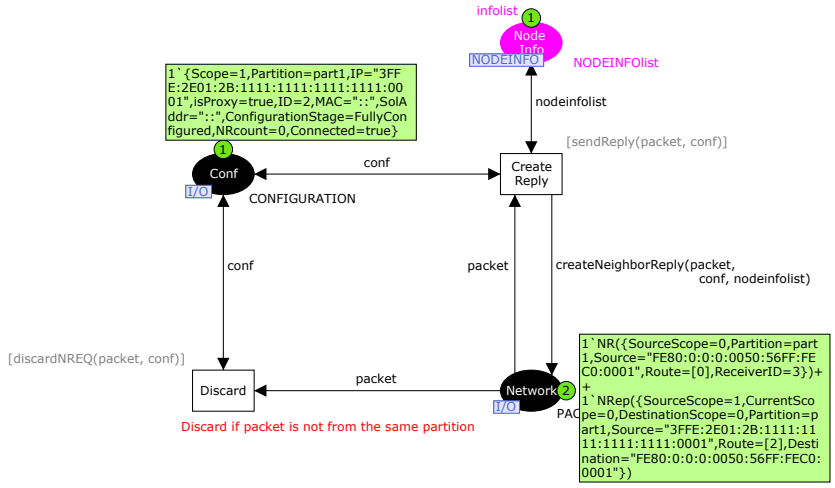

Figure 4.33: The *Handle NREQ* module after creating the neighbor reply.

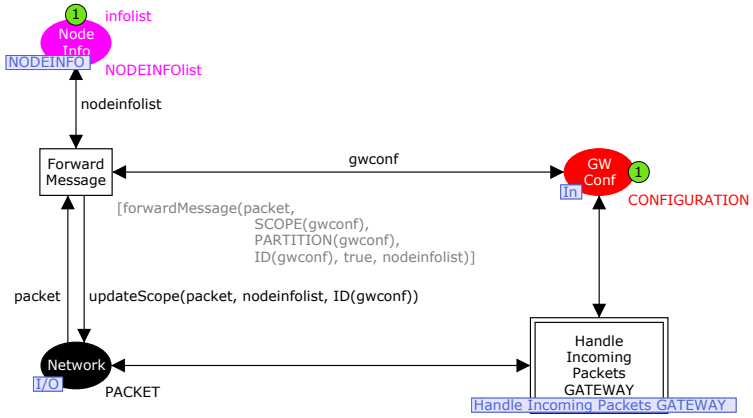

Figure 4.34: The *Gateway* module.

With the exception of a few function names, handling incoming neighbor requests at the gateway is no different from the procedure described in section 4.5.18. The fourth module, *Manage Prefixes*, is used for managing allocated prefixes and is described in section 4.5.23.

4.5.21 Create Address Response

The structure of *Create Address Response* (figure 4.36) is similar to the *Proxy Module* described in 4.5.17, except for the management of occupied prefixes.

Firing transition *CreateResponse* removes the first available prefix from the list stored in *AvailPrefix*, and returns the rest back to the place. This prefix is combined with the global network prefix, and a host ID of all 0's to make a globally routable IP address. An address reply is created containing the newly

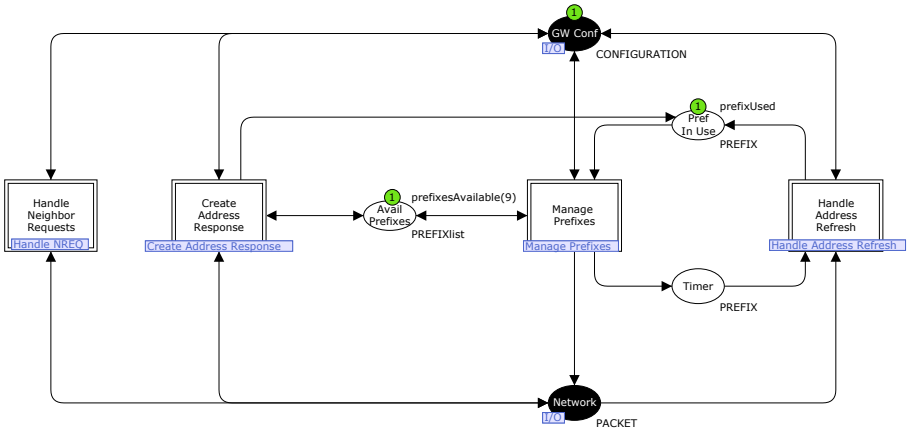

Figure 4.35: The *Handle Incoming Packets GATEWAY* module.

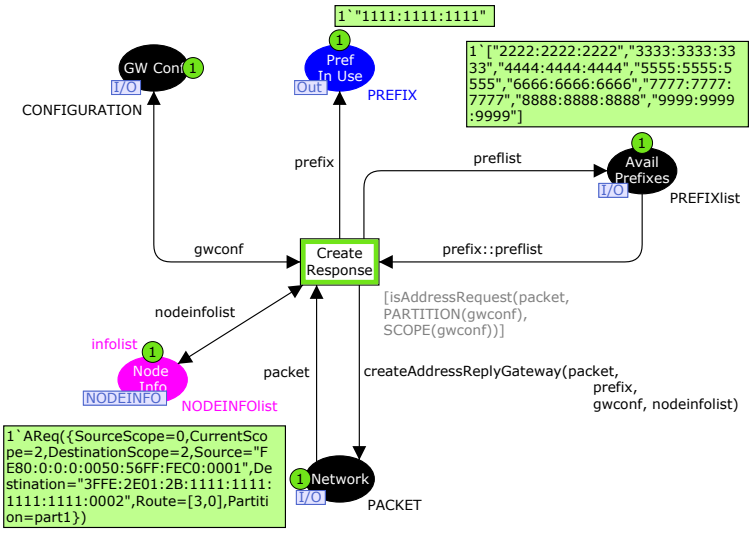

Figure 4.36: The *Create address response* module.

constructed IP, and is sent towards the requester, assuming that it still is part of the gateway's partition. The prefix chosen for the requester is sent to *PrefInUse*, which will be made available on page *Manage Prefixes* (section 4.5.23). After creating an address response, the module is depicted in figure 4.37.

4.5.22 Handle Address Refresh

The *Handle Address Refresh* module is shown in figure 4.38. Recall that the gateway sends refresh requests address to all nodes carrying a particular prefix if the associated proxy node does not refresh its prefix. Handling these incoming replies is the purpose of the module depicted in figure 4.38.

When a packet arrives at the module, it can take one of two paths. If the

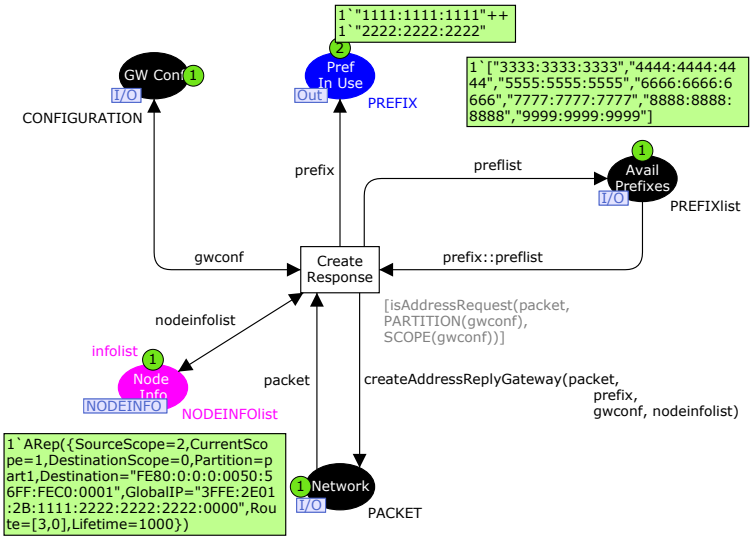

Figure 4.37: *Create Address Reponse* after creating the address response at the gateway.

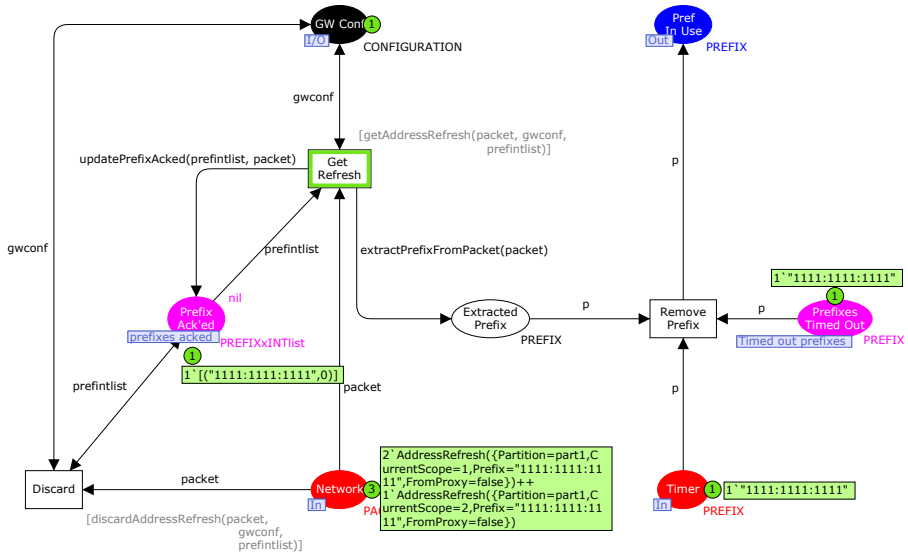

Figure 4.38: The *Handle Address Refresh* module.

received refresh message has not already been acknowledged, i.e., the gateway has not yet received a refresh for a particular prefix, *GetRefresh* will remove the reply from *Network*. While firing *GetRefresh*, the counter associated with the prefix is updated in the list of acknowledged prefixes in place *PrefixAck'ed*. The counter associated with a prefix p_1 keeps track of whether or not a refresh message has been received for p_1. If this counter has a non-zero value, the gateway can ignore subsequent replies for p_1 until its timer expires again. The prefix list stored at

PrefixAck'ed is also used in *Manage prefixes* (section 4.5.23). *GetRefresh* extracts the prefix from the packet, and adds it to *ExtractedPrefix*. After extracting the prefix, it must be removed from the fusion place of prefixes that has timed out. Transition *RemovePrefix* removes the extracted prefix p_1 from *PrefixesTimedOut* and *Timer*, ensuring that p_1 is not made available in the *Manage prefixes*-module. If multiple incoming arcs share the same arc inscription, e.g., the arcs from *PrefixExtracted* and *PrefixesTimedOut* both have p as inscription, the token bound to p is the same from both places. This means that whatever extracted prefix resides at *PrefixExtracted*, a token of the same colour will be removed from *PrefixesTimedOut* and *Timer* when transition *RemovePrefix* is fired. If tokens of the same colour are not available on these places, the transition is not enabled. Figure 4.39 shows *Handle Address Refresh* after having received an address refresh message for prefix `1111:1111:1111`. The *RemovePrefix* has fired, and removed the prefix from *PrefixesTimedOut*, and enabled *Discard* for discarding subsequent replies for that prefix.

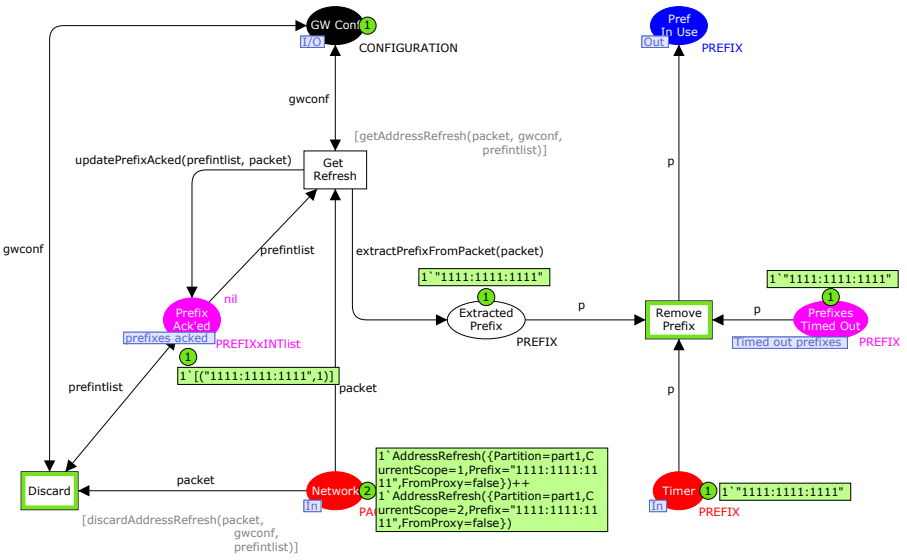

Figure 4.39: The *Handle Address Refresh* module after handling an incoming refresh message.

If the address refresh is the periodically refresh message sent by a proxy node, no prefix has timed out. Since there are no actual timers associated with the prefix in this model, we do not have to pass the prefix to *Manage Prefixes*, and update its lifetime. Instead we can just discard the refresh message if it originated from a proxy node. Only if an address refresh is received due to a prefix timeout do we need to pass the prefix to *Manage Prefixes*, in order to update the list of occupied/available prefixes.

4.5.23 Manage Prefixes

Manage prefixes is shown in figure 4.40. The purpose of this module is to keep track of which prefixes are currently assigned to proxy nodes, and which are available on incoming address requests.

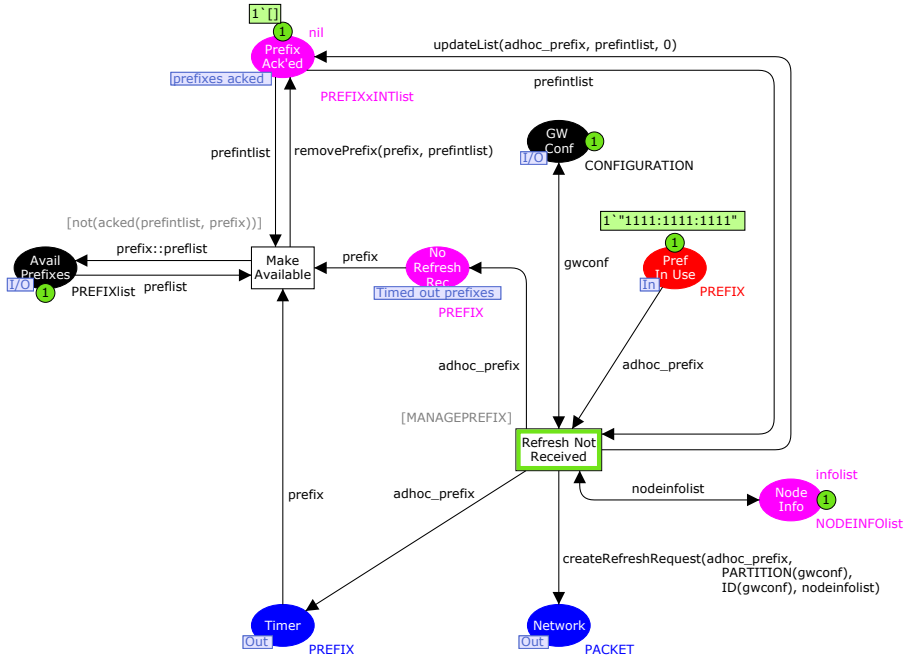

Figure 4.40: The *Manage Prefixes* module.

The place *PrefInUse* holds a set of prefixes already assigned. The initial marking has one occupied prefix `1111:1111:1111`. The transition *RefreshNotReceived* is enabled if the gateway has not received the periodically address refresh message for a given prefix located at *PrefInUse*. Firing this transition adds the prefix to *Timer*, and *NoRefreshRec* that holds prefixes for which no refresh has been received. The prefix is also added to the list of prefixes that needs to be acknowledged. This list is instantaneously made available on *Handle Address Refresh* (figure 4.38) via the fusion place *PrefixAck'ed*. Finally, refresh requests are generated and sent through *Network* to all nodes using the prefix in question. Figure 4.41 shows the *Manage Prefix* module after having sent refresh requests for prefix `1111:1111:1111`.

Receiving a refresh reply from *Handle Address Refresh* will enable *RefreshReceived*, removing the corresponding token from *Timer*, ensuring that it is not made available. If an address refresh pakcet is not received, *MakeAvailable* is enabled, and adds the prefix at *Timer* to the list of available prefixes. If a reply is received, it is added to *PrefInUse*, and the cycle of receiving/not receiving address refresh messages continues. As mentioned earlier, a node is limited in the

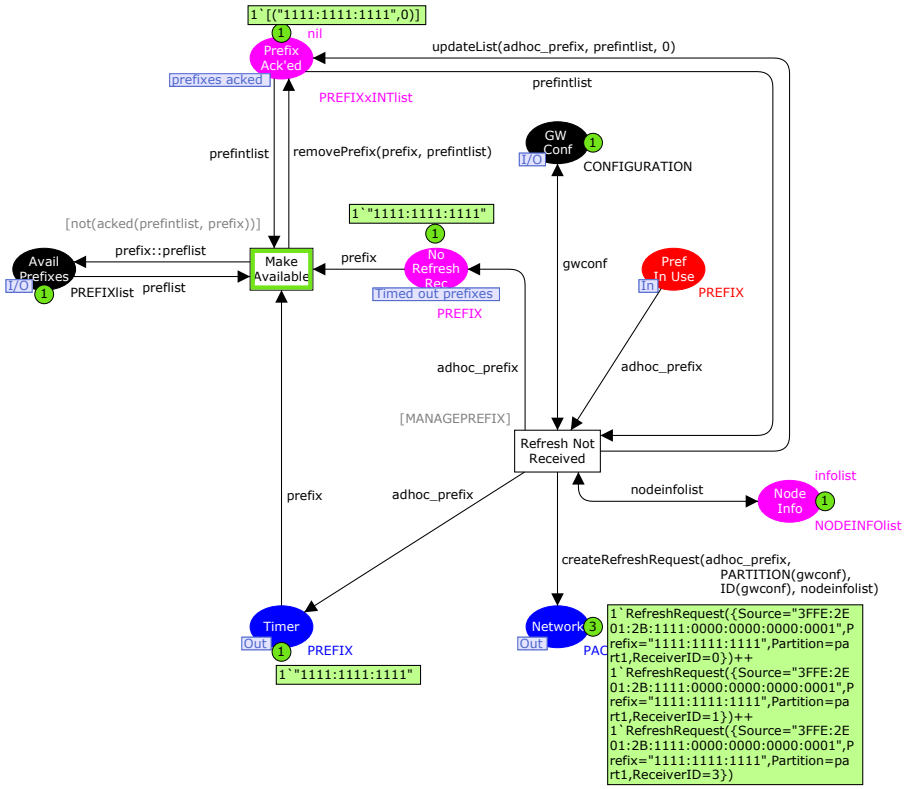

Figure 4.41: The *Manage Prefixes* module after sending refresh requests.

number of address refresh messages it is allowed to send. Therefore, prefixes will always timeout on a successful simulation, because the gateway will not receive a reply for its latest refresh request.

4.6 Summary

This chapter provided detailed descriptions of the different components comprising the constructed CPN model. The chapter started by discussing the simplifications and assumptions made in order to turn the specification into a model. Next followed an introduction to different CPN constructs created to emulate topology changes throughout the simulation. This was achieved by introducing the concept of *scope* and *partition*. Section 4.3 presented an overview of the model topology, and also the hierarchically structure of the CPN model. The definitions of the messages exchanged throughout the autoconfiguration process, was introduced in section 4.4. A detailed description of each module composing the model was given in section 4.5, introducing key CPN concepts to the reader with limited knowledge of Coloured Petri Nets.

Chapter 5

Validation by means of simulation

As described in section 1.3, there are two simulation types of CPN models; either interactively or automatic. Using interactive simulation, the user is completely in control of the simulation process, selecting which enabled transition fires next. Interactive simulation enables the user to investigate how each step affects the system state, by observing the token flow and colours. The graphical nature of CPN models enables the user to easily observe the changes made by firing a certain transition.

The automatic simulation, on the other hand, keeps selecting enabled events at random until no such exists, or a stop criterion has been satisfied. The user has no control of which events occur next, and even at a slow execution speed, it is difficult to observe and process all system changes. Therefore, automatic simulations are often only conducted when the user is interested in the end result, and not the individual changes throughout model execution. Even though a CPN model has a graphically representation, interacting and interpreting the result of a system change can be challenging for users with no prior CPN Tools experience. In a complex CPN model, the states can be scattered throughout an immense number of modules, making it troublesome to achieve the desired overview of all tokens. Also, a non-CPN expert may not truly understand how the model works, and therefore less capable of proposing corrections. Representing the simulation result in a more widely known format can assist non-CPN experts in getting a better understanding of the model behavior. One such format is *Message Sequence Charts* or MSCs, which is a language for describing interaction between independent processes. The benefit of using MSCs is that it too is a versatile graphical language, with many application areas. Due to this versatility, MSCs are more widely used and understood by many people.

This chapter starts with a very brief introduction to message sequence charts in section 5.1. A presentation of the properties investigated in the simulation is presented in section 5.2. Section 5.3 shows simulation runs of the model covering various scenarios, and documented as message sequence charts. Finally, section 5.4 summarizes the simulation results.

5.1 Message Sequence Charts

Message Sequence Charts (MSCs) describes interaction between processes or entities in a graphically manner. MSCs are standardized by the ITU-TS[38] (the Telecommunication Standardization section of the International Telecommunication Union) in Recommendation Z.120[39].

A Message Sequence Chart is a finite set of entities, which is an abstraction of a process may input/output messages or perform local actions. An entity is represented by a vertical axis with the name of the entity written in a box located at the top of the axis. Moving from top to bottom along this vertical axis represents progression of time. The individual events in an entity are totally ordered in time, but no global time is assumed.
A small box drawn on the vertical line, with "action text" placed beside it, portrays a local action. Local actions are events such as *Address generated*, *Solicitation received* or *Setup complete*. Messages sent between entities are represented by horizontal arrows starting at the current time position at the sender, and drawn across the chart, ending at the receiver's time line at the current time position. Figure 5.1 shows an example of a message sequence chart with two entities A and B.

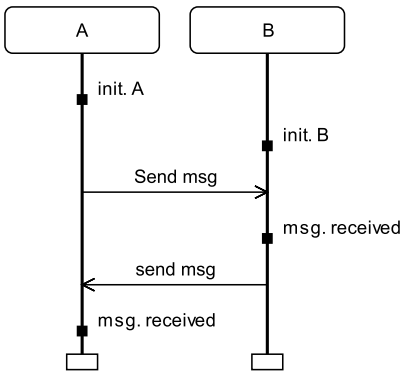

Figure 5.1: An example of a MSC containing two entities A and B.

The initial event indicated by the action text *init.A* is an example of a local action. After both entities have been initialized, A sends a message to B indicated by the horizontal arrow with the text *Send msg*. Another local action

occurs at B, before it sends a message to A. A more thorough introduction to MSCs can be found in [40].

An animation layer can be added to CP-Nets, allowing us to generate MSCs based on state changes. The tool for creating these MSCs is called *BRITNeY suite*[41], which is an acronym for ***B**asic **R**eal-time **I**nteractive **T**ool for **Ne**t-based animation*. To benefit from this automatic generation of MSCs, we must annotate transitions in the CPN model with ML code that generates an event (either local action or sending of a message). An example of the annotation of a transition is listed below:

```
input (mac, conf);
action
msc.addInternalEvent(idToProcess(ID(conf)), "Select MAC: "^mac);
```

This code is taken from the *SelectMAC* transition of the *Requesters Select MAC* module shown in figure 5.2.

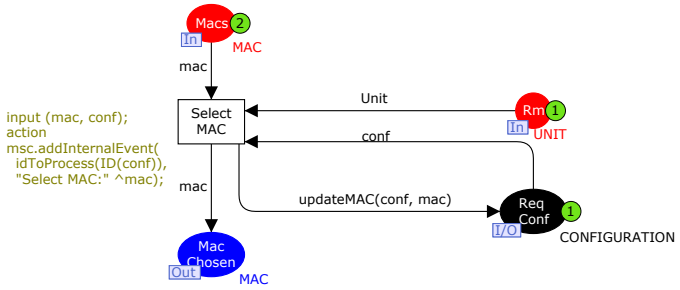

Figure 5.2: The *Select MAC* module annotated with ML code for generating an internal BRITNeY event.

This code adds an *internal event*, which is BRITNeY suites name for *local actions*, to the requester entity. The first parameter to *addInternalEvent* is the process/entity at which the event occurs, and the second parameter is the action text. The function *idToProcess* converts an ID from the configuration file to a process/entity name. To send messages from one process to another requires adding an event. This event must specify from which process it originates and which process is the intended target. The code segment below is taken from the *Nodes Handle AREQ* module (figure 5.3). It is attached to the *ForwardRequest* transition, and will forward a message from it self to the *Gateway*-process with label *"Forward AREQ"*.

```
input (conf);
action
msc.addEvent(idToProcess(ID(conf)), "Gateway", "Forward AREQ")
```

Section 5.3 shows various execution scenarios described as message sequence charts.

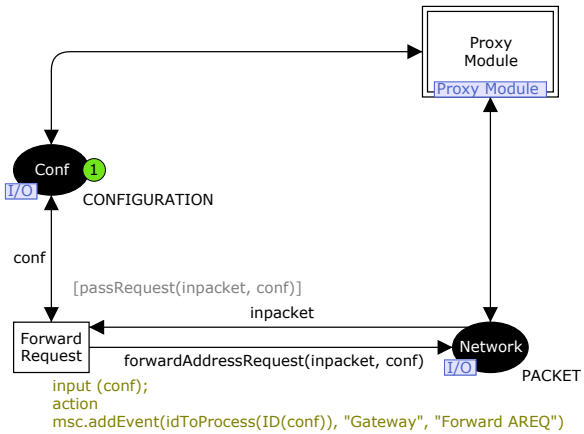

Figure 5.3: The *Handle AREQ* module annotated with ML code for generating a BRITNeY event.

5.2 Properties investigated

To prove or disprove correctness of the model, we must first define what this means in terms of autoconfiguration. Important aspects of the autoconfiguration process that must be investigated is listed below.

Address uniqueness. The overall concern of address configuration is ensuring addresses uniqueness for each requester. To guarantee uniqueness of the addresses breaks down to several sub tasks that must be handled correctly.

Topology changes. Can the protocol adapt to topology changes during the autoconfiguration process? For example, can the protocol cope with a MANET partitioning during the configuration process? Is prefixes handled correctly when connectivity changes occur?

Prefix management. A crucial task for the gateway, besides supplying the MANET with Internet access, is handling prefixes. If malfunctioning modules cause glitches in the prefix management, several proxy nodes may be assigned identical prefixes, inevitable leading to duplicate addresses.

Address management. Proxy node management of host IDs is another crucial task. Poor host ID management could potentially lead to duplicate addresses by assigning an identical host ID to multiple requesters.

5.3 Model Simulation

This section describes six autoconfiguration scenarios, presenting the course of action as message sequence charts. The chosen scenarios gradually involves more and more features of the model. These scenarios are designed to systematically test the constructed model, investigating the properties listed in section 5.2.

5.3.1 Scenario 1: One requester

In the first scenario we simulate the autoconfiguration process for a single requester. The MANET consists of one requester, a node N, a proxy node P, and a gateway G, which is used throughout the remainder of this chapter to identify non-requesting nodes. The topology considered in this simulation is depicted in figure 5.4.

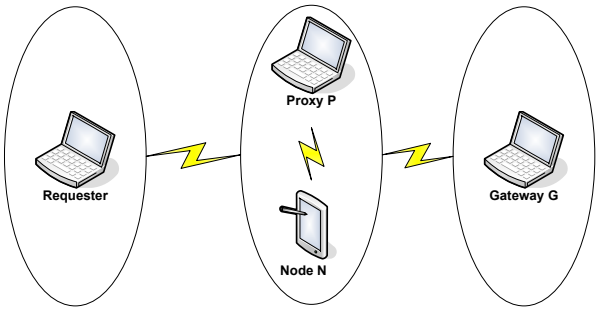

Figure 5.4: The network topology considered in the simulation with one requester.

This simulation tests the very basic features of the model, focusing on address management at the proxy node. The topology consists of one non-configured requester, attempting to acquire an address from P. P manages the ad hoc prefix `1111:1111:1111`, and N has been assigned an address containing this prefix. In this simple setup, we ignore the periodic address refresh messages sent by P. Therefore, no communication with the gateway is expected, as the proxy node is solely responsible for handling address assignments.

Figure 5.5 shows the message sequence chart generated for the autoconfiguration process in the above-described scenario. In an attempt to acquire an address, the requester selects a MAC address and generates its link-local address. The address undergoes duplicate address detection, but no advertisements are received due to a topology with only one requester (phase 1). Recall that this model assumes that only requesting nodes may be involved in conflicts, and having one requester ensures a conflict-free assignment. Broadcast of the neighbor request is received by N and P, enabling the requester to choose one of the incoming replies. Since the topology setup contains a proxy node, regardless of the choice of initiator,

the proxy intercepts the address request. It generates a new address and sends a reply to the requester (phase 2). The host ID contained in the address reply is expected to be `0003` as `0000` is reserved for multicasting purpose, with `0001` and `0002` assigned to P and N respectively.

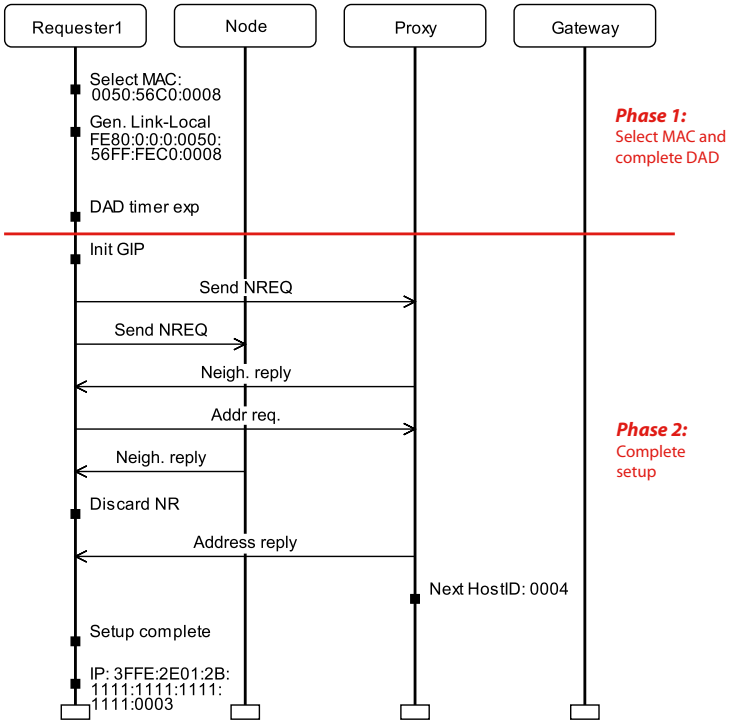

Figure 5.5: Message sequence chart with one requester.

The first four events in the message sequence chart are events at the requester, leading up to the broadcast of neighbor requests (NREQs). The requester broadcasts a neighbor request to all nodes in a one-hop range, i.e., N and P. The requester chooses the proxy as neighbor and initiates the request for a global address. The neighbor reply from N is correctly discarded, as stated in the internal event *Discard NR*. The proxy sends an address reply, and correctly increments the next available host ID to `0004`. The setup completes at the requester by updating its IP to `3FFE:2E01:2B:1111:1111:1111:1111:0003`.

5.3.2 Scenario 2: Two requesters - no conflict

In this next simulation we consider the original topology (see figure 3.1 on page 30), with two non-configured requesters, a configured node, a proxy, and a gateway. The two requesting nodes select different MAC addresses in the initial step hence no link-local address conflict is expected. The focus point of this simu-

lation is address management at the proxy when handling multiple concurrent requests. P is expected to allocate addresses with host ID `0003` and `0004` to the requesting nodes. Figure 5.6 shows the message sequence chart generated based on the described scenario.

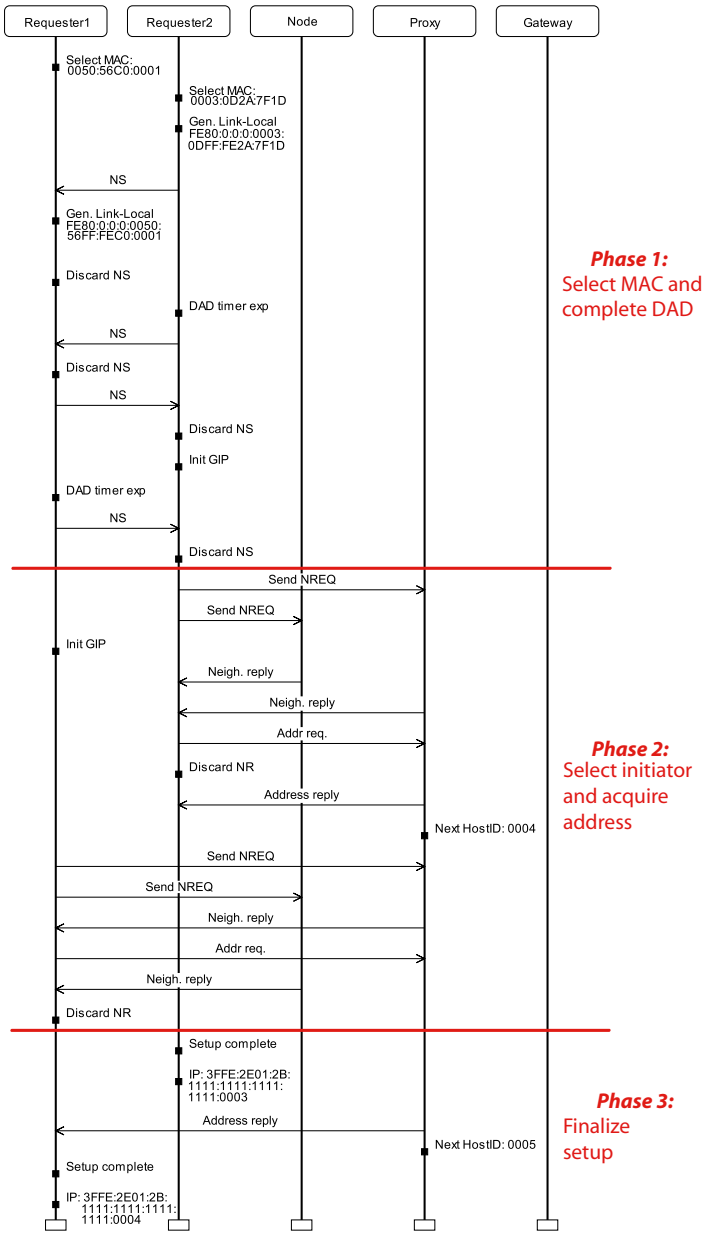

Figure 5.6: Message sequence chart with two non-conflicting requesters.

As expected, the two requesters generate their link-local addresses and exchange neighbor solicitations, which are correctly discarded (phase 1). The send-

ing and receiving of neighbor requests/replies is also handled as expected, discarding any obsolete replies. The interesting part of this simulation is to check whether or not P can handle concurrent requests, correctly updating the host ID. As shown in the internal events, *Next HostID*, the host ID is updated correctly (phase 2). This is also reflected in the global IP addresses assigned to each of the requesters. Each address has different host IDs, thus the uniqueness of addresses is fulfilled (phase 3).

5.3.3 Scenario 3: Two requesters - duplicate addresses

In this next simulation, we consider the same topology as above, but with an address conflict between the requesters. Both requesters initial select identical MAC addresses leading to identical link-local address. In this simulation there is no restriction regarding the choice of initiator. Selecting N as initiator implies that the address request is forwarded to the gateway. Even though N is in direct reach of P, the request will be forwarded to G instead of P. This simulates that N has recently become a member of the MANET, and has yet to receive a proxy advertisement. At some point during the simulation it is expected that the requesters become aware of this address clash. This forces them to reset their configuration, and start the configuration process from scratch. Two key issues are investigated in this simulation. The first and foremost concern investigated is the requesters handling of duplicate addresses, examining whether or not their parameters are properly reset. Secondly, we want examine the gateways management of the newly occupied prefix. The simulation, however, does not include the sending and receiving of address refresh messages. This issue is examined in section 5.3.4. Figure 5.7 shows the generated message sequence chart.

As figure 5.7 shows, both requesters selected `0050:56C0:0008` as MAC address. *Requester2* discards the first neighbor solicitation since it has not yet generated the link-local address. *Requester1* handles the incoming solicitation by generating an advertisement, which is sent to *Requester2* via *Send NA* (phase 1). Handling this incoming advertisement forces the requester to select a new MAC address, restarting the configuration process. Both requesters generate a new link-local addresses based on a new MAC addresses, and complete the duplicate address detection procedure without problems (phase 2). They proceed to broadcast neighbor requests and selecting an initiator (phase 3). *Requester1* selects N, while *Requester2* selects P, both discarding any pending neighbor replies. As expected, *Requester2* receives an address reply containing an IP address with host ID `0003` from P. Before the address request from *Requester1* can reach G, it is forwarded via N as stated in event *Forward AREQ*. This also illustrates the concept of scope prevents messages from reaching nodes not directly connected, without being forwarded through intervening nodes. G selects an available prefix, `2222:2222:2222`, and transmit respond via N to *Requester1*. *Requester1* receives

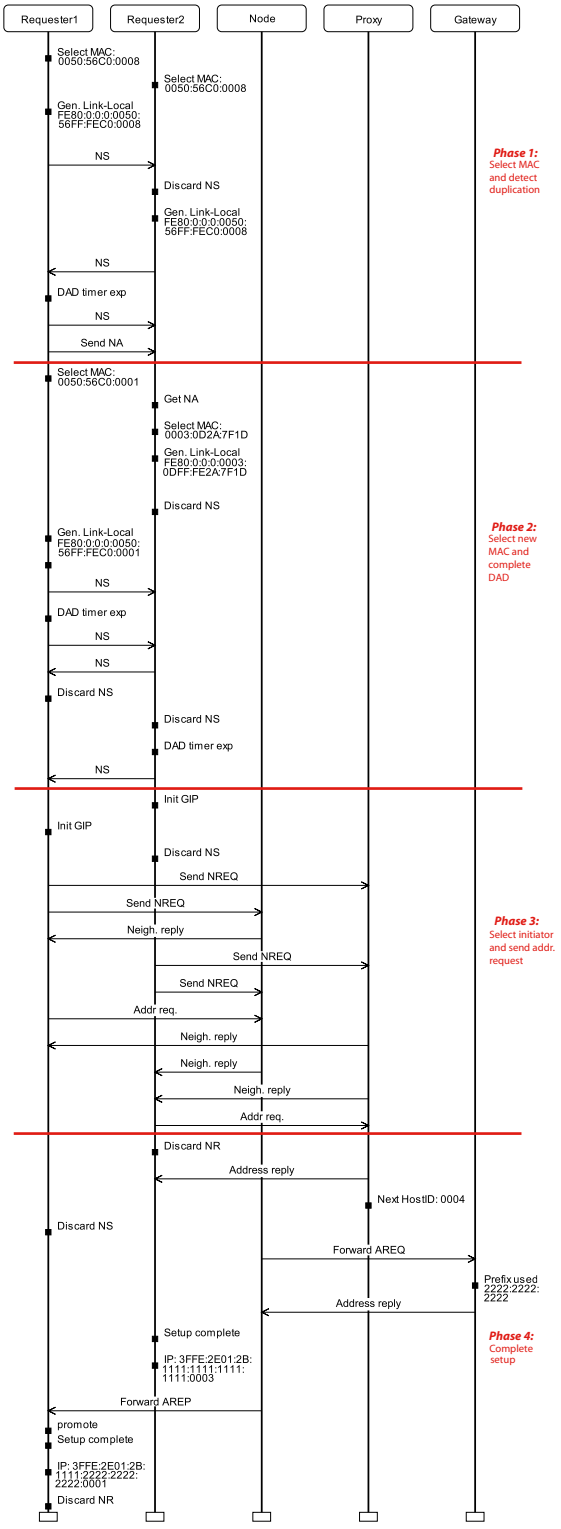

Figure 5.7: Message sequence chart with two conflicting requesters.

the address reply, and is correctly promoted to proxy (phase 4).

5.3.4 Scenario 4: Managing prefixes

The forth simulation scenario investigates the management of prefixes at the gateway, and the handling of address refresh packets. The topology used for this simulation consist of four configured nodes, each located in the same network setup as the previous scenario. Two fully configured nodes have replaced the requesting nodes, one having received its address from the proxy, and the other from the gateway. This means that *Node1*, *Node2* and *Proxy2* occupy prefix `1111:1111:1111`, while *Proxy1* manages `2222:2222:2222`. The simulation starts with a proxy node's transmission of *address refresh messages* to the gateway. The number of retransmission is limited to prevent an infinite sequence of address refresh cycles. After reaching this limit, the proxy nodes become disconnected from the network, and are no longer able to refresh its prefix. This have the effect that its prefix lifetime expire, and G starts sending refresh requests. All nodes, beside the disconnected nodes, carrying this prefix respond with a refresh reply (RREP). An upper limit on the number of refresh replies is also implemented, which prevents G from receiving replies for its last refresh request. Receiving no reply for the final request allows the gateway to reenter the prefix to the pool of available prefixes. Figure 5.8 shows the generated message sequence chart. The four internal events simply lists the IP address occupied by the four non-gateway nodes to aid the readability of the chart.

The proxies start sending address refresh messages to the gateway as expected. The gateway "acknowledges" the reception of these refresh packet illustrated as the internal event *ACK* (phase 1). After comleting two of these refresh/acknowledgment cycles, the proxies have reached the upper limit on address refresh messages. The gateway "multicast" address requests to all nodes in the MANET as started in the specification[15]. Since a genuine multicast algorithm has not been implemented, multicasting consist of generating packets for every node, who act according to their own configuration parameters. *Node1* and *Node2* send responses to G, which acknowledges the first incoming message, discarding the subsequent reply. As expected, G never receives any reply for prefix `2222:2222:2222` because the node carrying this prefix has become disconnected from the network. These requests are correctly discarded at the remaining nodes. *Node1* and *Node2* also reach the maximum number of allowed refresh replies (event *RREP limit exc.*) and no reply is sent. The gateway returns both prefixes to the pool of available prefixes (phase 2).

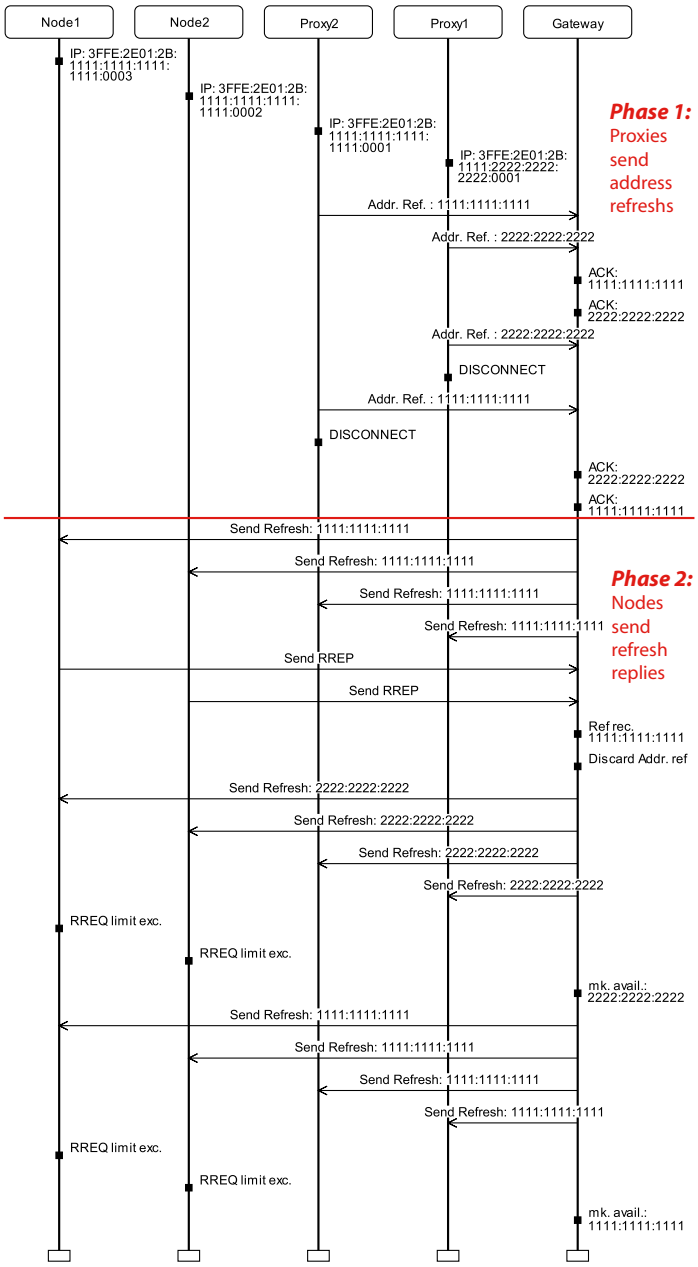

Figure 5.8: Message sequence chart for simulating prefix management.

5.3.5 Scenario 5: Two requester - scope change

Simulating topology changes during the setup phase is investigated next, concentrating on scope changes. We exploit the same topology as scenario 2 and 3, but *Requester1* may changes its connectivity during the setup attempt. Place *NewScope1* of the *Update Configuration* module now contains a token with token colour *3*. This enables the requester to change its connectivity such that its closest neighbor becomes the gateway. Recall that if a requester changes connectivity *before* it has received a global IP, it must retransmit neighbor solicitations. The focus of this simulation is investigating how the model react to topology changes while configuring a requester. The periodic address refresh messages are ignored in this simulation as well. Figure 5.9 shows the generated message sequence chart.

As figure 5.9 shows, the requesters choose distinct MAC addresses and *Requester1* completes the duplicate address detection procedure (phase 1). After broadcasting neighbor requests to N and P, *Requester1* receives a reply from P. Before it has a chance to initiate the requester for a global address, *Requester1* changes it connectivity, and is no longer within direct contact with neither N nor P. This connectivity change implicate that its closest neighbor becomes the gateway in scope 2. *Requester1* rebroadcasts neighbor requests in its new setting (phase 2). *Requester1* correctly replies to the neighbor response from the gateway (phase 3). The setup completes by promoting *Requester1* to proxy, while the configuration procedure has completed at *Requester2* by an address reply from P (phase 4).

5.3.6 Scenario 6: Two requester - partition change

This final simulation focuses on how the requesting and fully configured nodes react to partitioning. The requesting node *Requester1* changes partition to *part2* during the simulation. This illustrates the case where the requester looses MANET connectivity before it has been properly configured. From the point of partitioning, *Requester1* is no longer able to receive any packets originating from partition *part1*. Whenever a packet cannot reach the intended target, an event is added to the source of the message, e.g., if a neighbor request cannot reach its intended target, an event from sender to receiver is added with the text *NR LOST*. Figure 5.10 shows the generated message sequence chart.

The change of partition for *Requester1* occurs right after it has sent the first neighbor solicitation to *Requester2* (phase 1). *Requester2* generates a link-local address and initiates the duplicate address detection procedure. It transmit neighbor solicitations, but they are lost as indicated by the event *NS LOST*. *Requester1* keeps sending neighbor solicitations until the *DAD limit* is reached, and initiates the request for a global address. The neighbor requests sent are

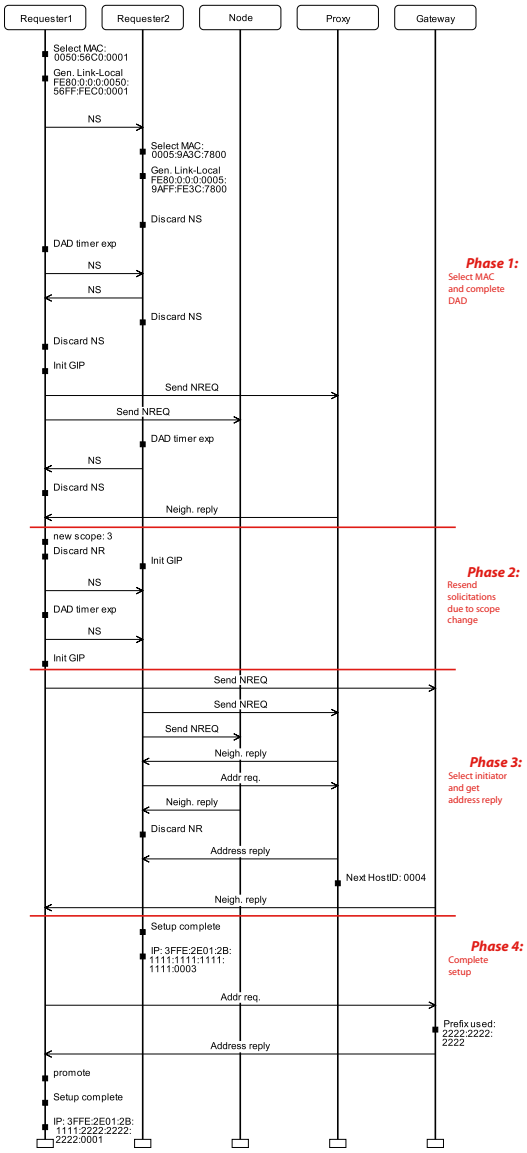

Figure 5.9: Message sequence chart for the simulation containing a change of scope at *Requester1*.

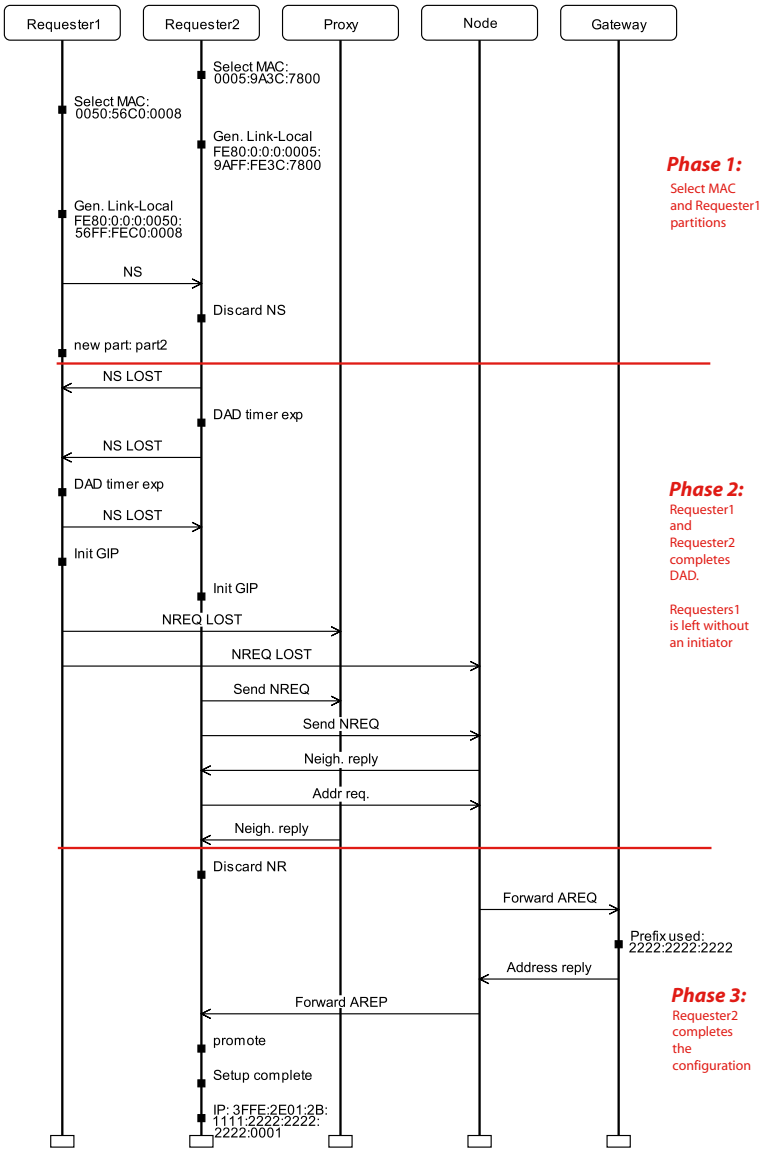

Figure 5.10: Message sequence chart for the simulation containing a change of partition at *Requester1*.

never received by any node belonging to *part1*, thus the setup fails for *Requester1* as expected (phase 2). Meanwhile, the setup procedure continues for *Requester2*, which ends up with an address promoting it to proxy (phase 3).

5.4 Summary

This chapter presented six simulation covering various aspects of the model documented as message sequence charts. The simulations gradually became more and more elaborate, until all aspects of the model had been covered. Simulation began at the simplest setting with only one requesting node, receiving its global address from a proxy node. Next was a simulation with two concurrent non-conflicting requesters. The simulation was repeated with an address conflict, investigating a correct reset of the involved parties. Managing occupied prefixes at the gateway was tested in scenario 4. The final two simulations focused on coping with topology changes, one focusing on the requester's scope change, while the other investigated the models behavior at partitioning. All simulation results reflected the anticipated behavior, but an issue was discovered during the simulation.

The simulation discovered an unwanted situation that could lead to duplicate global addresses. The specification[15] does not describe in great detail its policy about reusing timed out prefixes:

> *If the Gateway does not receive any reply message for the Refresh_Request message before timeout, it decides that there is no node using that address space, and removes the proxy address space from the table. This address space can be allocated to the other nodes later.*

A prefix for which no refresh reply is received can be returned to the pool of available prefixes, but it is not mentioned when it is permitted to reallocate this prefix. A problematic scenario discovered during simulation is as follows. A requesting node R_1 has receives a global address with a particular ad hoc prefix P_1. If the proxy node responsible for P_1 becomes disconnected, the gateway starts sending refresh requests. If R_1 temporary, too, becomes disconnected, the refresh requests are left unanswered, enabling the gateway to release P_1. Another requester R_2 joins the network, acquiring an address from that gateway. Since P_1 is now available, R_2 may be handed the responsibility for P_1. Requester R_1 could rejoin the network before its address lifetime has expired. This leaves us with a proxy node handling P_1, and a node R_1 with P_1 as prefix, that R_2 is unaware of. Any subsequent requests for global address to R_2 could potentially reuse the address of R_1, leaving us with a duplicate global IP address. The likelihood of this scenario depends on the timing of refresh requests, and the validity period of an assigned address. If the lifetime of an address is longer than the refresh

request/reply cycle, one could experience this undesirable state. Figure 5.11 shows the message sequence chart sketching the scenario in question.

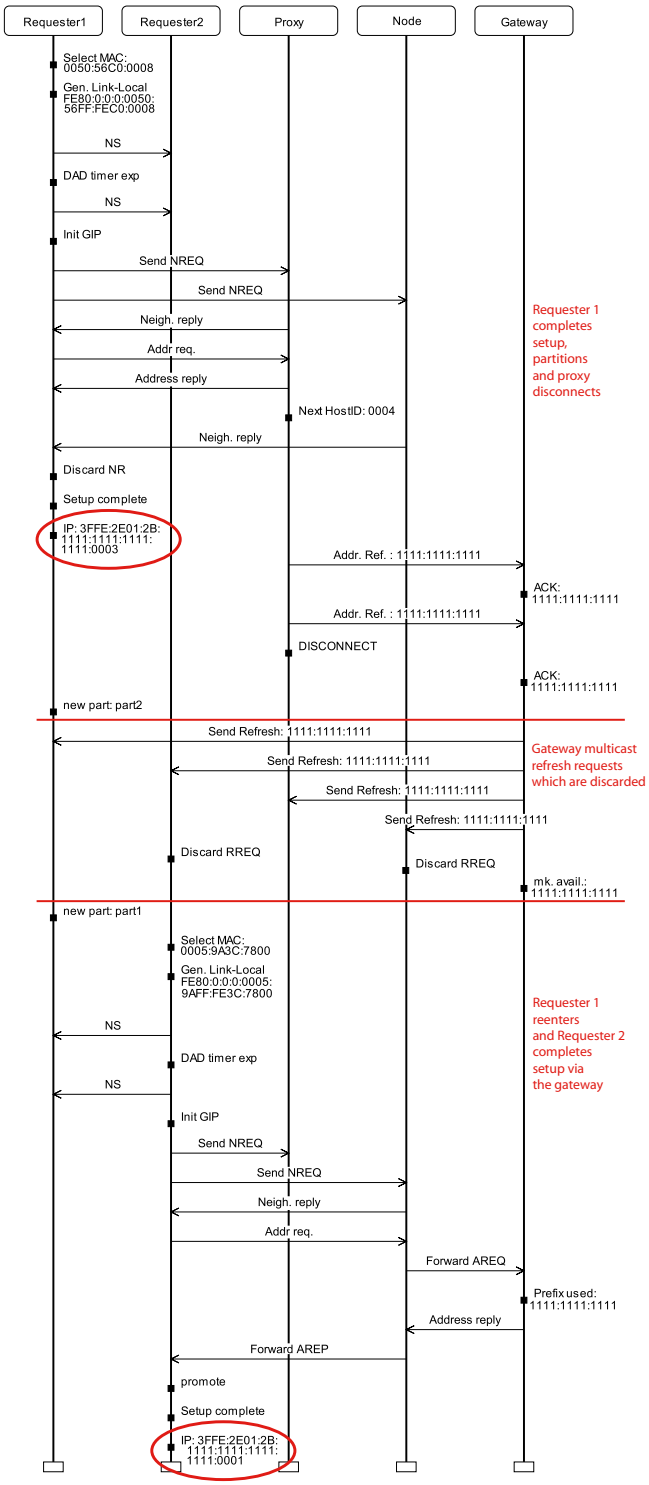

Figure 5.11: Message sequence chart depicting a problematic scenario with potential duplicated global addresses.

Chapter 6

State space analysis

The simulations documented in chapter 5 gave us an indication that the model behaved according to the specification. However, there may still be undesirable properties left undiscovered by the simulations. To ensure that these otherwise undiscovered properties are brought to the designers attention, a *formal verification* of the model must be completed. This chapter describes the formal verification of the CPN model, introducing the basic concepts behind this verification. Section 6.1 introduces the basic concept of a *state space* utilized for formal verification. CPN Tools[13] is capable of generating a *state space report* containing various standard properties. The structure of state space reports is discussed in section 6.2. Section 6.3 presents the formal verification in different settings. Each subsection of 6.3 formally verifies the scenarios introduced in section 5.3. Finally, section 6.4 summarizes.

6.1 Introduction

The simulations conducted and documented in chapter 5 investigated certain state configurations and state changes during simulation execution. Only a subset of all possible states and state changes were inspected. To ensure that these untouched states do not lead to undesirable system configurations, CPN introduce the concept of a *full state space*. The idea behind a full state space, or state space for short, is to calculate **all** reachable states (markings) and state changes of the CPN model. A state space is presented as a directed graph[42] with nodes corresponding to the set of reachable markings, and arcs representing occurring binding elements. An arc with binding element (t, b) as label, exist from a node representing a marking M_1 to a node representing a marking M_2 if and only if (t, b) is enabled in M_1, and the occurrence of (t, b) leads to marking M_2. State

spaces are computed fully automatic, and can be used to derive a set of standard behavioral properties, which are independent of the model and its construction. These standard properties are in some cases sufficient to analyze a system, but often more model specific properties needs investigation.

Prior to initiating the state space calculations, it is often necessary to make minor adjustments to the model. If transitions can occur an infinite number of times, we may have an infinite number of reachable markings, hence a finite state space cannot be constructed.

The construction of the state space for a given CPN model starts by creating a node representing the initial marking as shown in figure 6.1.

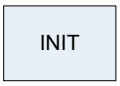

Figure 6.1: The initial state space graph.

The CPN model in this thesis has six enabled binding elements in the initial marking. To ease the readability, the binding for the configuration tokens is not shown:

$$
\begin{aligned}
SM_{11} &= (Requester1'SelectMAC, \langle mac = 0050:56C0:0001, Unit = ()\rangle)\\
SM_{12} &= (Requester1'SelectMAC, \langle mac = 0050:56C0:0008, Unit = ()\rangle)\\
SM_{21} &= (Requester2'SelectMAC, \langle mac = 0005:9A3C:7800, Unit = ()\rangle)\\
SM_{22} &= (Requester2'SelectMAC, \langle mac = 0050:56C0:0008, Unit = ()\rangle)\\
SAR &= (Node2'SendAddressRefresh,\\
&\quad \langle count = 0, nodeinfolist = [(0,0,part1,true),(1,0,part1,true),\\
&\qquad\qquad (2,1,part1,true),(3,1,part1,true),\\
&\qquad\qquad (4,2,part1,true)]\\
&\quad \rangle)\\
RNR &= (Gateway'RefreshNotReceived,\\
&\quad \langle prefintlist = [], adhoc\_prefix = \text{"}1111:1111:1111\text{"}\rangle)
\end{aligned}
$$

Prefixing the transition name is the page instance at which the transition resides. Each of these six binding elements leads to a new system state. Six new nodes in the state space graph are created and connected to the initial state. The updated state space graph is shown in figure 6.2.

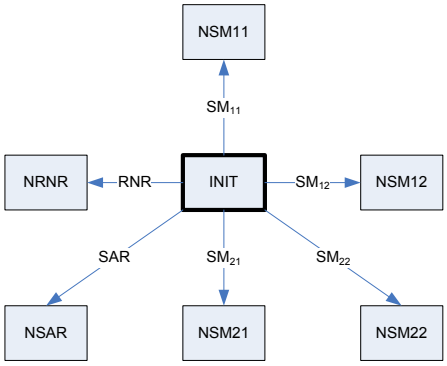

Figure 6.2: The partial state space after processing the initial state.

The arcs connecting the initial state to each of the six new states are annotated with the name of the binding element. The thick border surrounding *Init* indicates that all calculations of enabled binding elements have been completed for that node. In CPN terms, the node has been *processed*. Next, an unprocessed node is selected at random, and calculations are completed for this node. This calculation may lead to the creation of new nodes and/or connecting already existing state space nodes. When all nodes have been processed a full state space has been constructed. Following the creation of a full state space is often the generation of a *Strongly Connected Component Graph* (SCC graph). SCC graphs are used by CPN Tools to derive standard behavioral properties. Nodes in a SCC graph are cyclic *sub-graphs* of the state space graph called *strongly connected components*. These nodes are obtained by making a disjoint division of the state space nodes, such that the following property holds. Two nodes n_1 and n_2 of the state space graph are in same SCC if and only if they are mutually reachable, i.e., a path connects n_1 to n_2 and vice versa. Strongly connected component S_1 has an arc to S_2 if and only if the state space has an arc leading from a node in S_1 to a node in S_2. Strongly connected components without any *outgoing* arcs are called *terminal SCCs*. When the creation of state space and SCC graph has been completed, the usual step is to generate a *state space report* based on these graphs. The state space report contains basic information about the generated graphs and some standard behavioral properties. These properties are introduced throughout the rest of this chapter. Beside the standard properties generated by CPN Tools, the modeler may formulate model specific questions that need to be investigated. Questions are formulated as *query functions* operating on the state space and SCC graphs. CPN Tools has a broad variety of predefined query- and auxiliary functions, aiding the modeler to construct custom query functions. The query functions used in the verification of this model are presented throughout the chapter.

A state space is always relative to the configuration of the model in question. Therefore, it is not sufficient to select one configuration and complete the state space analysis, believing correctness has been achieved based on this. Several configurations must be tested, often starting with the simplest case, and gradually extending the complexity of the configuration. Changing the configuration parameters can often result in a dramatically change in the size, and calculation time of the state space and SCC graph. One of the changeable parameters in this model is the number of MAC addresses available at the requester. Adding just one more MAC address to the place *Macs* could result in an exponential growth in the state space size and calculation time. This is known as *state space explosion* due to this rapid growth in size. Each of the six simulation scenarios described in chapter 5 will form a model configuration on which the state space is calculated.

6.2 State space report structure

The standard state space report is divided into four sections. The first section contains statistics about the state space graph and the strongly connected component graph. These statistics include the number of nodes and arcs in the graphs, and the time it took to construct them. Other statistics contain information about the status of state space calculations. If calculations are terminated before it has processed all the nodes, the report lists the status as *partial*. There are a number of situations where only a partial state space can be obtained. A partial state space may be obtained when the complexity of the model results in a state space that takes too long to calculate, or a graph that is unable to fit in the computer memory. Even though a full state space often is preferred, a partial state may still be useful to identify errors. If the partial state space contains an undesirable property, then the full state space will contain the same unwanted property.

Next in the state space report is the *boundedness properties*. Boundedness properties lists the token multiplicity of a given place, and which colour they may have, when all reachable markings are considered. These properties are divided into two sub categories, namely best upper and lower bound. The *best upper integer bound* specifies the maximal number of tokens a given place has when considering any reachable marking. The converse hold for *best lower integer bound*. When the best lower and upper integer bounds are equal, a place always contains the same number of tokens, but their colour may vary. An example of such a place is *ReqConf* of the requester, always holding one configuration token of different colours. Moving to the *best upper multi-set bound* and *best*

lower multi-set bound, specifies for each colour of the place, the maximal number of tokens present *and* which colour they have.

The final portion of the state space report contains information about the *home* and *liveness properties*. The home properties tell us if a *home marking* exists. A home marking M_{home} is a marking that can be reached from **any** reachable marking. In other words, we cannot produce an occurrence sequence that makes it impossible to reach M_{home}. If a home marking exists for our model, we can conclude that no matter how the connectivity changes, and how many address conflicts occur etc., we can always reach the system state represented by M_{home}. A home marking implies that it is always *possible* to reach M_{home}, but there are no guarantees that it will be reached. If a home marking does not exist, there may exist a *home space* instead. A home space M^*_{home} is a generalization of a home marking. A home space is a *set* of markings such that at least one marking in M^*_{home} can be reached from any reachable marking. In other words, it is impossible to have an occurrence sequence that cannot reach one of the markings in M^*_{home}.

The liveness properties in the state space report include listing *dead markings*. A dead marking is a marking in which no enabled binding element exists. If a model has a dead marking, which also is a *home marking*, then we can conclude that the model is *partially correct*. In other words, if the execution terminates, then we have the correct result, provided that the home marking is the desired result. Partial correctness can also be obtained by having a set of dead markings comprising a home space. Furthermore, if the home marking is the only dead marking, then we are guaranteed that regardless of model execution, it is always possible to terminate with the correct result.

A transition T is said to be *live* if from any reachable marking, we can always find an occurrence sequence in which T is contained. The final part of the report contains information about three *fairness properties*. A transition T is *impartial* if it occurs infinitely often in all infinite occurrence sequences, i.e., if T is removed, then all infinite occurrence sequences are removed. If T is *fair*, then it occurs infinite often in all infinite occurrence sequences where it is infinite often enabled. Finally, T is *just* if it cannot remain enabled forever without occurring.

6.3 Formal verification

As the simulations described in section 5.3, each setting for the formal verification explores different aspects of the protocol. The majority of test scenarios directly correspond to the simulations of section 5.3.

6.3.1 One requester

The formal verification starts at the most basic configuration, namely a topology with only one requester (scenario 1 from chapter 5). The state space statistics is shown in listing 6.1.

```
 Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  34
     Arcs:   48
     Secs:   0
     Status: Full

  Scc Graph
     Nodes:  34
     Arcs:   48
     Secs:   0
```

Listing 6.1: Statistics for state space generation with one requester

The *fully* generated state space is relatively small with only 34 nodes and 48 arcs, and was computed "instantaneously". Comparing the number of nodes and arcs in the two graphs, we can conclude that there are no cycles in the model. Recall that the SCC nodes consist of cyclic sub-graphs of the state space. Since there are equal amount of nodes in both the SCC and state space graph, all SCC components consist of a single state space node. If there were *fewer* nodes in the SCC graph at least one cycle would exist. A cycle in the SCC graph indicate that we can have an infinite occurrence sequence, and thereby no guarantee of protocol termination.

Next in the report are the boundedness properties. Due to the complexity of the model, only a few selected places are commented on during this analysis. The full state space report for this, and the remaining five scenarios, can be found in appendix B. All upper- and lower integer bounds are as expected. The upper integer bound in the *Network*-place revealed that at most two packets are in transit at any given time. This reflects the anticipated as the only message sent in multiple quantities is the neighbor requests. The upper multi-set bound for the *Gateway*'s *AvailPrefixes*-place of available prefixes confirmed the simulation conducted in chapter 5. The management of prefixes correctly removes the newly occupied prefix from the list of available prefixes. This is shown in the two lists at *AvailPrefixes*, where the latter has removed the prefix `2222:2222:2222` correctly. Inspecting the set of occupied prefixes at the gateway's *PrefInUse*-place reveals a correct handling of occupied prefixes. The upper multi-set bounds for these

places are listed below.

```
Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1
   1`["2222:2222:2222","3333:3333:3333","4444:4444:4444",
      "5555:5555:5555","6666:6666:6666","7777:7777:7777",
      "8888:8888:8888","9999:9999:9999"]++
   1`["3333:3333:3333","4444:4444:4444","5555:5555:5555",
      "6666:6666:6666","7777:7777:7777","8888:8888:8888",
      "9999:9999:9999"]

Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1
   1`"1111:1111:1111"++1`"2222:2222:2222"
```

The multi-set bound for the requesters *Req1Conf*-place revealed that up to eight different tokens reside on the place throughout the execution. The upper integer bound, however, states that only one is present on the place at any given time. These tokens correspond correctly to the initial marking, and a token for each step in the autoconfiguration process. The remaining two tokens are the different final configuration tokens depending on which initiator the requester chose. Comparing the final IP address of the requester to each of the configured nodes, gave no conflict in the global IP address assigned.

The third part of the state space report is concerned with the liveness- and home properties, and is shown in listing 6.2.

```
 Home Properties
------------------------------------------------------------------------

    Home Markings
     None

 Liveness Properties
------------------------------------------------------------------------

  Dead Markings
     [31,34]
```

Listing 6.2: The home and liveness properties for the model containing one requester.

The home properties state that there is no home marking in this model. This is expected because the requester can choose between two initiators, leading to different final IP addresses. The liveness properties lists two dead markings, represented by state space node N_{31} and N_{34} respectively. As mentioned earlier, CPN Tools is equipped with a set of standard query functions that operates on these constructed graphs. A function, *ListDeadMarkings*, returns a list of all dead markings, which in this case contains two elements. Recall that a *home space* is a set of markings that is reachable from any reachable marking. CPN

Tools contain a function *HomeSpace* that given a list of nodes return true if these nodes comprise a home space. Evaluating this function on the list of dead markings returned true. In the remainder of this chapter, this procedure has been used when discussing home spaces. CPN Tools has a facility that, given a node number, displays the corresponding marking in the model. Applying this tool to node N_{31} returns a marking in which the requester has completed an autoconfiguration successfully, while having received its IP from the proxy. The marking corresponding to node N_{34} has successfully configured the requester with an IP received from the gateway. Combining this information with the fact that the dead markings comprise a home space, we can conclude that no matter how the execution of the protocol progresses, it is always possible to reach the state where the requester has been successfully configured. Furthermore, the model contain no cycles, thus we are guaranteed that one of the successful states will eventually be reached. In other words, we have obtained *correctness* of the model. The liveness properties also show a list of dead transition instances. All these transitions are, however, expected to be disabled throughout the model execution as they are part of the prefix management, which is not activated in this scenario.

6.3.2 Two requesters - no conflict

The second configuration has a topology with two requesters residing at scope 0 (scenario 2 from chapter 5). Adding the second requester increased the size and calculation time for the state space as shown in listing 6.3.

```
 Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  10078
     Arcs:   41724
     Secs:   72
     Status: Full

  Scc Graph
     Nodes:  10078
     Arcs:   41724
     Secs:   1
```

Listing 6.3: State space statistics for two non-conflicting requesters.

The state space consists of more than 10.000 nodes and 40.000 arcs. The SCC graph consists of the same amount of nodes and arcs, resulting in a cycle-free configuration. The upper and lower integer and multi-set bounds revealed no surprises with regard to the token multiplicity and colours of each place.

The liveness and home properties of the model is shown in listing 6.4.

```
 Home Properties
------------------------------------------------------------------------

  Home Markings
     None

 Liveness Properties
------------------------------------------------------------------------

  Dead Markings
     6 [9797,9789,10078,10077,10052,...]
```

Listing 6.4: Liveness properties for two non-conflicting requesters.

As in the previous scenario, this configuration has no home marking. This reflects the anticipated as the requesters can choose different initiators, leading to diverse IP addresses. The number of dead markings has changed from two to six. Listing the complete set of dead markings using function *ListDeadMarkings*, resulted in the following six state space nodes:

`9797, 9789, 10078, 10077, 10052, 10024`

These six dead markings correspond to the six combinations of assigning IP addresses to the requesters. Node N_{9797} correspond to the scenario in which both requesters have received an IP from the proxy, and that *Requester2* got the first address response, i.e., has the lowest host ID. The marking in node N_{9789} is the opposite case with *Requester1* having the lowest host ID. Similar, the other four markings correspond to the remaining combinations of receiving the IP from the proxy and the gateway. The six dead markings comprise a home space, implying correctness of the model, and a guaranteed successful setup.

The multi-set bound for the requesters *GlobalIP*-place is listed below:

```
Requester'Global_IP 1
 1`"3FFE:2E01:2B:1111:1111:1111:1111:0003"++
 1`"3FFE:2E01:2B:1111:1111:1111:1111:0004"++
 1`"3FFE:2E01:2B:1111:2222:2222:2222:0001"++
 1`"3FFE:2E01:2B:1111:3333:3333:3333:0001"++1`"::"

Requester'Global_IP 2
 1`"3FFE:2E01:2B:1111:1111:1111:1111:0003"++
 1`"3FFE:2E01:2B:1111:1111:1111:1111:0004"++
 1`"3FFE:2E01:2B:1111:2222:2222:2222:0001"++
 1`"3FFE:2E01:2B:1111:3333:3333:3333:0001"++1`"::"
```

The two multi-sets are identical. However, multi-set bounds does not state whether or not the two requesters had the same IP address in the same marking,

only which addresses were in use during the state space generation. To ensure that the requesters did not concurrently possess identical IP addresses, the query function in listing 6.5 was applied to all dead markings.

```
fun detectDuplications(list) =
let
  fun check(n) =
   let
      val ip1 = ms_to_col(Mark.Requester'Global_IP 1 n))
      val ip2 = ms_to_col(Mark.Requester'Global_IP 2 n))
   in
      (ip1 = ip2)                    andalso
      (ip1 <> unspecifiedAddress)    andalso
      (ip2 <> unspecifiedAddress)
   end

   fun aux(nil) = nil
   |   aux(x::xs) = if check(x) then x::aux(xs) else aux(xs)
in
    aux(list)
end
```

Listing 6.5: Query function for detecting duplicate addresses.

The function *detectDuplications* consists of two auxiliary functions *aux* and *check*. The multi-set containing a single element on place *GlobalIP* is converted into an element using *ms_to_col*. Checking for duplications consists of a simple comparison, while ensuring that none of the IP addresses is the unspecified address. If the addresses are equal while different from the unspecified address, the function *check* adds the node number representing the marking to a list of markings. When all nodes have been process, the list of markings fulfilling the Boolean predicate is returned. Applying *detectDuplications* to the dead markings for this configuration revealed no such duplicate addresses.

6.3.3 Two requesters - conflict

The considered topology has two requesters, a proxy node and a gateway. Distinguishing this setup from the previous topology is a conflict in link-local addresses. The purpose of this test is to investigate the protocols ability to cope with such duplicate link-local addresses. As mentioned in section 6.1, adding additional steps to the configuration process, could potentially result in a state space explosion, dramatically increasing the size and calculation time. To reduce the risk of such an explosion, minor changes are made to the model. Recall that the requester broadcasts a neighbor request after a successful completion of the duplicate address detection. The receivers transmit a reply, and the requester chooses one of these as initiator. The requester then initiates a request for a

global address. Since the topology only contains one fully configured node within a one-hop distance, this node is automatically chosen as initiator. Instead of responding to the *neighbor reply*, the proxy node allocates an available address directly on a neighbor request. This saves us from a series of unnecessary steps in this model setup, and reduces the calculation time and state space size. This change in the model does not affect the verification. Assume that we do perform the additional steps of choosing an initiator. This results in additional states in which the requesters can detect the duplication, i.e., receive a neighbor solicitation or advertisement. But since the process resets to an unconfigured state regardless of when this detection occurs, disregarding the initiator selection does not change the final verification result. The state space statistics for this setup is shown in listing 6.6.

```
 Statistics
----------------------------------------------------------------------

  State Space
     Nodes:  142317
     Arcs:   551948
     Secs:   17550
     Status: Full

  Scc Graph
     Nodes:  142317
     Arcs:   551948
     Secs:   28
```

Listing 6.6: State space statistics for a topology with two conflicting requesters.

The addition of an address conflict dramatically increased the size and calculation time of the state space. It consists of more than 140.000 nodes and 500.000 arcs, but still, the state space and SCC graph contains equal amount of nodes and arcs, translating into a cycles-free model.
The liveness and home properties of the model is shown in listing 6.7.

```
 Home Properties
----------------------------------------------------------------------

  Home Markings
     None

 Liveness Properties
----------------------------------------------------------------------

  Dead Markings
     282 [96442,96441,96440,96439,95236,...]
```

Listing 6.7: Liveness properties for two conflicting requesters.

The large increase in state space nodes also lead to a major increase in dead markings. These dead markings, still, comprise a home space. To ensure that none of the dead markings has left either requester without a globally routable IP, the query function in listing 6.8 was applied to the dead markings.

```
fun notAddressAcquired(list)=
let
  fun check(n) =
  (
   (ms_to_col(Mark.Requester'Global_IP 1 n) <> "::" andalso
    CS(ms_to_col(Mark.MANET'Req1_Conf 1 n)) = FullyConfigured)
     orelse
   (ms_to_col(Mark.Requester'Global_IP 2 n)  <> "::" andalso
    CS(ms_to_list(Mark.MANET'Req2_Conf 1 n)) = FullyConfigured)
  )

  fun aux(nil) = nil
  |   aux(x::xs) = if check(x) then aux(xs) else x::aux(xs)
in
  aux(list)
end
```

Listing 6.8: Query function for checking global IPs.

The function *notAddressAcquired* takes the list of dead markings and invokes the auxiliary function *aux* to check the IP at the requesters *GlobalIP*-place. The function *aux* uses another function, *check*, to test the current marking for a lack of globally routable IP at either requester. The function tests whether or not the IP address residing at place *GlobalIP* is the unspecified address. To avoid flagging state in which the requester is not fully configured, the function also checks that the configuration stage is *FullyConfigured*. If both requesters have acquired an IP, the function recursively analyze the rest of the markings. If a check fails, the state space node number is added to the list of nodes *notAddressAcquired* return. Applying the function on the list of dead markings returned the empty list. In other words, at all dead markings, the requesters is fully configured, and have acquired a globally reachable IP address. To ensure the uniqueness of these addresses, the function *detectDuplications* of listing 6.5 was applied, and it too returned the empty list. To summarize, the two requesters have different IP addresses in all dead markings, and in none of the dead markings were either requester left with the unspecified address as global IP. Because the dead markings comprise a home space, and the model is cycle-free, we have obtained correctness of the model.

6.3.4 Manage prefix

Minor topology changes are made to this setup as well to reduce the risk of a state space explosion. The setup consists of one fully configured node, one proxy node and a gateway. The *Node* and *Proxy* both occupy IP addresses with ad hoc prefix `1111:1111:1111`.

The generated state space is relatively small as shown in listing 6.9.

```
 Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  325
     Arcs:   710
     Secs:   1
     Status: Full

  Scc Graph
     Nodes:  325
     Arcs:   710
     Secs:   0
```

Listing 6.9: State space statistics for prefix management.

As in all previous scenarios there are the same number of nodes and arcs in the generated graphs, which translates into a cycle-free model. The upper and lower integer and multi-set bounds revealed no surprises with regard to the token multiplicity and colours of each place.

The liveness and home properties for this setup is shown in listing 6.10.

```
 Home Properties
------------------------------------------------------------------------

  Home Markings
     None

 Liveness Properties
------------------------------------------------------------------------

  Dead Markings
     16 [61,325,322,321,316,...]
```

Listing 6.10: Liveness and home properties for prefix management.

The full list of dead markings is shown in figure 6.3.

```
59, 150, 156, 159, 252, 273, 278, 290, 295
300, 307, 312, 313, 316

244, 245
```

Figure 6.3: The full list of dead markings in scenario 4.

The model contains no home marking, but the dead markings in figure 6.3 comprise a home space. Using the *State space to sim* tool on node number 159 resulted in a situation in which the gateway has prematurely released a prefix. By applying a standard query function, CPN Tools can provide the shortest occurrence sequence leading to this state. The function in listing 6.11 uses this query function to write the occurrence sequence to a log file.

```
fun WriteListToFile(nil) = ()
|     WriteListToFile(x::xs) =
let
  fun writeaux(nil) = ()
  |     writeaux(x::xs) =
   let
      val strlist =
          List.map st_BE (List.map ArcToBE (ArcsInPath(1,x)))
   in
     WriteToFile(strlist, Int.toString(x));
     writeaux(xs)
   end
in
   writeaux(x::xs)
end
```

Listing 6.11: A function for mapping a list of state space nodes to occurrence sequences.

The function uses *ArcsInPath* to construct the list of arcs on the path from the initial marking to x. The CPN ML function *Map* applies *ArcToBE* to each entry in the list in order to construct a list of *binding elements*. This list of binding elements is converted into its string representation by mapping the function *st_BE* onto all entries. The string-list of binding elements is passed to function *WriteToFile*, which writes the list to a file. The chain of events that leads to this undesirable state corresponding to state space node 159, is listed below as $\langle page \rangle' \langle transition \rangle \ \langle page\ instance \rangle$:

```
Send_Address_Refresh'Send_Address_Refresh 2
Manage_Prefixes'Refresh_Not_Received 1
Handle_Address_Refresh'Get_Refresh 1
Handle_Address_Refresh'Remove_Prefix 1
```

```
Manage_Prefixes'Refresh_Not_Received 1
Manage_Prefixes'Make_Available 1
Handle_RREQ'Send_Reply 1
Handle_Address_Refresh'Get_Refresh 1
Handle_RREQ'Discard 1
```

A correct prefix management depends heavily on the prefix timers, since the gateway cannot distinguish if a periodic address refresh does not arrive due to long delays or a departured proxy node. If a address refresh packet or the replies to refresh requests are delayed or lost, a prefix can prematurely become available. Occurrence sequences from the initial marking to nodes in the first and second row of figure 6.3 are all examples of prematurely releasing a prefix. Theses situations correspond to the simulations conducted in section 5.3.4, where the prefix is made available, not due to node partitioning as in the simulation, but due to long delays in refresh replies.

Only nodes N_{244} and N_{245} correspond to occurrence sequences in which the gateway correctly handles prefixes. The gateway cannot be held accountable for prematurely releasing a prefix due to long network delays, because it does not maintain information aiding it to estimate the time of a round trip. Even retransmitting refresh messages a fixed number of times does not provide any guarantee that an occupied prefix is not released due to delays or packet losses. Releasing the prefix in the above-mentioned occurrence sequences cannot be categorized as a design error, but as an unfortunate property of network transmission.

To ensure that the prefix is not made available while the node and/or proxy are still connected, the query function in listing 6.12 was applied to the dead markings.

```
fun detectPrefix(list : Node list)=
let
 fun inPrefixList(pre, nil) = false
 |   inPrefixList(pre, x::xs) = if x = pre then true else
                                inPrefixList(pre, xs)
 fun check(n) =
  let
    val reqconf    = ms_to_col(Mark.MANET'Req1_Conf 1 n)
    val proxyconf  = ms_to_com(Mark.MANET'Conf1 1 n)
    val prefixlist =
    ms_to_col(
              Mark.Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1 n)
  in
    (CONNECTED(reqconf) orelse CONNECTED(proxyconf)) andalso
    inPrefixList(extractPrefix(IP(reqconf)), prefixlist)
  end

 fun aux(nil) = nil
 |   aux(x::xs) = if check(x) then x::aux(xs) else aux(xs)

in
```

```
  aux(list)
end
```

Listing 6.12: A query function for checking prefix management.

The function *detectPrefix* is similar in structure to the previous listed customized query functions. If function *check* returns true, then the state space node is added to the list returned by *detectPrefix*. The function *check* returns true if either the node or proxy is still connected to the MANET, while the prefix has been made available. The list of state space nodes returned by *detectPrefix* contain all states from figure 6.3 where the gateway has released a prefix prematurely. To summarize, we have encountered states in which a prefix has been released while still occupied by connected nodes. Duplication may occur if the gateway reallocates this prefix to newly arriving nodes.

6.3.5 Change scope

The topology for this setup consists of two requesters, a proxy node and a gateway. The purpose of this configuration is to investigate the behavior of the model if the requester changes connectivity during the setup attempt. Recall that if the requester changes connectivity, i.e., scope *before* the address reply has been received, it must retransmit neighbor solicitations. As in scenario 5 presented in section 5.3.5, the scope to which the requester changes is 3, making it a directly reachable only from the gateway.

The statistics for the generated state space is shown in listing 6.13.

```
 Statistics
----------------------------------------------------------------------

  State Space
     Nodes:  5754
     Arcs:   16917
     Secs:   15
     Status: Full

  Scc Graph
     Nodes:  5754
     Arcs:   16917
     Secs:   0
```

Listing 6.13: State space statistics for the topology with a connectivity change.

Similar to the previous scenarios, the number of state space nodes and SCC nodes are identical, hence no cycles exist in this setup either. More interesting is the liveness and home properties of the model shown in listing 6.14.

```
 Home Properties
-------------------------------------------------------------------

  Home Markings
     None

 Liveness Properties
-------------------------------------------------------------------

  Dead Markings
     21 [5754,5753,5747,5744,5721,...]
```

Listing 6.14: Liveness and home properties for the topology with a connectivity change.

The model contains 21 dead markings in this setting, which comprise a home space. If each of these dead markings represent a state in which the moving requester successfully has complete a configuration, we have obtained correctness due to a cycle-free configuration.

The query function *notAddressAcquired* from listing 6.8 was applied to the dead markings, and returned the empty list. Given that the dead markings comprise a home space, and that all of these markings contains a proper global address at the requesters, means that we cannot have an occurrence sequence, which makes it impossible to complete a successful setup. Furthermore, the function *detectDuplications* from listing 6.5 likewise returned the empty list. In other words, in all dead markings both requesters have unique globally reachable IP addresses.

6.3.6 Change partition

As opposed to changing connectivity, i.e. scope, changing partition will inevitable lead to states in which the requester is left without an IP address. The time of departure from the MANET decides whether or not a successful setup is completed. Similar to the topology in section 6.3.5, it consists of two requesters, a proxy node and a gateway. *Requester1* is set to change partition to *part2* during the configuration attempt. All communication with the other nodes is no longer possible at that point.
The statistics for this setup is shown in listing 6.15.

```
 Statistics
-------------------------------------------------------------------

  State Space
     Nodes:  3028
     Arcs:   9272
```

```
    Secs:   5
    Status: Full

  Scc Graph
    Nodes:  3028
    Arcs:   9272
    Secs:   0
```

Listing 6.15: State space statistics for the topology with a partitioning requester.

This model setup contains no cycles as implied by the state space and SCC graph size. The liveness and home properties are shown in listing 6.16.

```
 Home Properties
------------------------------------------------------------------------

  Home Markings
    None

 Liveness Properties
------------------------------------------------------------------------

  Dead Markings
    8 [3000,2999,2996,2995,2909,...]
```

Listing 6.16: Liveness and home properties for the topology with a partitioning at the requester.

Calculating the state space for scenario 6 of chapter 5, resulted in 24 dead markings, which comprise a home space. As mentioned in the beginning of this section, it is inevitable that the requester is left without a proper global IP. Using a slightly modified version of *notAddressAcquired* from listing 6.8 revealed that 18 of the 24 dead markings has left the requester without a proper address. The modification consisted of removing the branch checking *Requester2*'s IP address, since only *Requester1* experiences the topology change. More interesting is whether or not the remaining six states has left both requesters with individual global IP. Function *detectDuplications* of listing 6.5 was evaluated on the dead markings, and returned the empty list. In other words, if *Requester1* is configured properly, then the addresses are unique. Since we have a non-cyclic model setup, and a home space, which contain unique addresses at both requesters, we have obtained partial correctness. A successful validation of this model setup does not necessary mean that *Requester1* must be properly configured. The loss of MANET connection, simulated by the partitioning, prevents *Requester1* from receiving the address reply. Therefore, the criteria for successfully validating scenario 6 must be uniqueness of requester addresses, which we have achieved.

6.4 Summary

This chapter presented the formal verification of the constructed CPN model. Section 6.1 introduced the basic concepts of state space analysis, including a short description of how state spaces are constructed. The structure of a standard state space report was discussed in section 6.2, and an introduction to key concepts such as *home space* and *partial correctness*. The formal verification of the scenarios described in chapter 5 was given in section 6.3. Correctness of each scenario was established via standard CPN queries and customized query functions, tailored for capturing essential model properties necessary for verification. To avoid a state space explosion, minor changes was made to some of the configurations. These modification, however, had no effect on the outcome of the verification.

Chapter 7

Conclusion

In this thesis we have investigated distributed autoconfiguration of mobile ad hoc networks. The investigation was a four-phase exploration, starting by analyzing the problem domain, creating a model that reflected the specification, conducting simulations, and finally a formal verification.

The purpose of this thesis was to prove or disprove *correctness* of distributed autoconfiguration as described by Lee et al.[15]. Correctness consists of four key properties that must be fulfilled at any given time. The overall concern of an autoconfiguration scheme is *address uniqueness*, ensuring that no two current MANET members occupy identical global address. Correct address handling at the proxy nodes, and prefix management at the gateway are paramount in order to obtain this uniqueness. Due to the mobile nature of MANET, the proposition must be able to cope with connectivity changes during the configuration phase. These properties were investigated throughout a series of informal simulations, and a formal verification using standard state space functionally in combination with customized query functions.

The simulations was conducted on the model prior to the formal verification, and documented as message sequence charts. Six independent scenarios were constructed, covering different aspects of the model, as presented in chapter 5. Interpreting the constructed message sequence charts helped us conduct an informal verification of the model behavior. By analyzing the message sequence charts for scenarios with one and two non-conflicting requesters, we can conclude that correct model behavior was achieved in these settings. Correct model behavior was also accomplished in case of two conflicting requesters, and in case of topology changes (MANET partitioning and scope changes).

The simulations revealed at least one potential problem with prefix management at the gateway. If the periodic address refresh messages from a proxy do not reach the gateway, it assumes that the associated prefix is unoccupied. However, if a node temporary becomes disconnected or experiences long transmission delays, the multicast refresh requests may wrongfully remain unanswered. The temporary disconnected node may reenter the MANET while its address is still valid, and the gateway has prematurely released the prefix. A newly arriving node may be configured using this prefix, and duplicate addresses can occur as a result.

Finally, state space analysis was conducted on the CPN model as means of a formal verification. The six representative scenarios outlined in chapter 5 formed the model configuration for state space analysis. To avoid a state space explosion, minor model changes was made to some configurations, without affecting the final result. The formal verification succeeded in establishing the four key properties of the autoconfiguration proposal examined, with the exception of prefix management at the gateway. Prefix management failed due to prematurely making a prefix available for reallocation as outlined above.

7.1 Future work

The constructed model investigated the core functionality of distributed autoconfiguration as suggested by Lee et al. A gradually extension of the model to capture some (or all) of the assumptions and simplifications introduced in section 4.1, is considered the next step. The most severe restriction made is the assumption regarding 100% network reliability, and is a top candidate for model improvement. Given the complications following packets loss, packet corruption, and/or overtaking, there is a high probability that scenarios uncovered by this model may discover further undesired properties currently not covered. Adding retransmission of packets could result in a complicated model, thus a simplification is needed to avoid a state space explosion. As proposed in section 6.3.3, several steps in the configuration process can be merged to reduce the state space. After improving the model by adding retransmission, omitted optional operations should be implemented, such as selection of tentative proxies on incoming refresh replies.

Efficiency and scalability is a great concern for the majority of MANET related technologies. A MANET mechanism, whether it is a routing protocol or an autoconfiguration scheme, must operate correctly while being as efficient as possible. Extending the model with timing would allow us to engage in a performance analysis of the proposal. A key property of the proposal is a fast setup by

delegating address assignments to proxy nodes. Conducting performance analysis between various suggestions comparing it to Lee et al.[15], is considered a part of the future work. It would be interesting to compare the average setup time and scalability for this distributed approach as opposed to using, e.g., PACMAN[1] or a centralized solution[19], determining if it is beneficial to utilize this distributed approach.

Bibliography

[1] Kilian Weniger. Pacman: Passive autoconfiguration for mobile ad hoc networks. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 23(3):507–519, March 2005. [cited at p. iv, 16, 17, 127]

[2] Charles E. Perkins. *Ad Hoc Networking*. Addison Wesley Professional, 2001. Page 1-27. [cited at p. 1]

[3] S. Corson and J. Macker. Mobile ad hoc networking (manet): Routing protocol performance issues and evaluation considerations. http://www.faqs.org/rfcs/rfc2501.html, January 1999. Request for comment (RFC) 2501. [cited at p. 2]

[4] University of Southern California Information Sciences Institute. Internet protocol. http://tools.ietf.org/html/rfc791, September 1981. Request for comment (RFC) 791. [cited at p. 2]

[5] S. Deering and R. Hinden. Internet protocol, version 6 (ipv6) specifications. http://www.faqs.org/rfcs/rfc2460.html, December 1998. Request for comment (RFC) 2460. [cited at p. 2, 33]

[6] R. Droms, J. Bound, B. Volz, T. Lemon, C. Perkins, and M. Carney. Dynamic host configuration protocol for ipv6 (dhcpv6). http://www.ietf.org/rfc/rfc3315.txt, July 2003. Request for comment (RFC) 3315. [cited at p. 4, 9]

[7] S. Thomson and T. Narten. Ipv6 stateless address autoconfiguration. http://www.ietf.org/rfc/rfc2462.txt, December 1998. Request for comment (RFC) 2462. [cited at p. 4, 9, 29, 32, 34]

[8] Charles Perkins, Elizabeth Belding-Royer, and Samir Das. Ad hoc on-demand distance vector (aodv) routing. http://www.ietf.org/rfc/rfc3561.txt, July 2003. Request for comment (RFC) 3561. [cited at p. 4, 17, 25, 43]

[9] T. Narten, E. Nordmark, and W. Simpson. Neighbor discovery for ip version 6 (ipv6). http://www.ietf.org/rfc/rfc2461.txt, December 1998. Request for comment (RFC) 2461. [cited at p. 4, 21, 34, 36]

[10] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems.* Springer, January 2006. Draft manuscript. [cited at p. 5]

[11] W. Reisig. *Petri Nets, An Introduction.* Springer Verlag, 1985. [cited at p. 5]

[12] Standard ML. www.smlnj.org. [cited at p. 5]

[13] CPN tools. http://wiki.daimi.au.dk/cpntools/cpntools.wiki. [cited at p. 5, 105]

[14] Alloy. http://alloy.mit.edu/. [cited at p. 5]

[15] Dongkeun Lee, Jaepil Yoo, Hyunsik Kang, Keecheon Kim, and Kyunglim Kang. Distributed ipv6 addressing technique for mobile ad-hoc networks. *Proceedings of the 2006 ACM symposium on Applied computing SAC '06*, pages 1156–1160, April 2006. [cited at p. 6, 7, 10, 29, 32, 43, 79, 96, 101, 125, 127]

[16] Ross Finlayson, Timothy Mann, Jeffrey Mogul, and Marvin Theimer. A reverse address resolution protocol. http://www.ietf.org/rfc/rfc0903.txt, June 1984. Request for comment (RFC) 903. [cited at p. 9]

[17] R. Droms. Dynamic host configuration protocol. http://www.faqs.org/rfcs/rfc2131.html, March 1997. Request for comment (RFC) 2131. [cited at p. 9]

[18] Ryuji Wakikawa, Antti Tuimonen, and Thomas Clausen. Ipv6 support on mobile ad-hoc network. http://tools.ietf.org/html/draft-wakikawa-manet-ipv6-support-02, March 2006. Internet draft. [cited at p. 9, 44]

[19] F. Templin, S. Russert, and K. Grace. Network localized mobility management using dhcp. http://www.potaroo.net/ietf/idref/draft-templin-autoconf-netlmm-dhcp/, September 2006. Internet Draft. [cited at p. 10, 127]

[20] Sanket Nesargi and Ravi Prakash. Manetconf: configuration of hosts in a mobile ad hoc network. *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 2:1059–1068, June 2002. [cited at p. 11]

[21] Brandon Bielstein. The osi reference model. http://nsgn.net/osi_reference_model. [cited at p. 16]

[22] Thomas Clausen and Philippe Jacquet. Optimized link state routing protocol (olsr). http://www.ietf.org/rfc/rfc3626.txt, October 2003. Request for comment (RFC) 3626. [cited at p. 17, 43]

[23] Run length encoding. http://en.wikipedia.org/wiki/Run-length_encoding. [cited at p. 18]

[24] IEEE. Part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. http://standards.ieee.org/getieee802/download/802.11-1999.pdf, June 2003. [cited at p. 18]

[25] D. Johnson, C. Perkins, and J. Arkko. Mobility support in ipv6. http://www.ietf.org/rfc/rfc3775.txt, June 2004. Request for comment (RFC) 3775. [cited at p. 19]

[26] Kilian Weniger and Martina Zitterbart. Ipv6 autoconfiguration in large scale mobile ad-hoc networks, 2002. In Proceedings of European Wireless 2002. [cited at p. 20]

[27] Institute of Electrical and Electronics Engineers. http://www.ieee.org. [cited at p. 20]

[28] Christophe Jelger and Thomas Noel. Proactive address autoconfiguration and prefix continuity in ipv6 hybrid ad hoc networks. *Sensor and Ad Hoc Communications and Networks, 2005. IEEE SECON 2005. 2005 Second Annual IEEE Communications Society Conference*, pages 107–117, September 2005. [cited at p. 23]

[29] Ryuji Wakikawa, Jari T. Malinen, Charles E. Perkins, Anders Nilsson, and Antti J. Tuominen. Global connectivity for ipv6 mobile ad hoc networks. http://tools.ietf.org/wg/manet/draft-wakikawa-manet-globalv6-05.txt, March 2006. Internet draft. [cited at p. 25]

[30] T. Clausen, C. Dearlove, and P. Jacquet. The optimized link-state routing protocol version 2. http://tools.ietf.org/html/draft-ietf-manet-olsrv2-02, June 2006. Internet draft. [cited at p. 26]

[31] I. Chakeres and C. Perkins. Dynamic manet on-demand (dymo) routing. http://www.ietf.org/internet-drafts/draft-ietf-manet-dymo-06.txt, October 2006. Internet draft. [cited at p. 27]

[32] Greg Daley, Nick Moore, and Erik Nordmark. Tentative source link-layer address options for ipv6 neighbour discovery. http://tools.ietf.org/html/draft-daley-ipv6-tsllao-00.txt, June 2004. Internet draft. [cited at p. 34]

[33] N. Moore. Optimistic duplicate address detection (dad) for ipv6. http://www.ietf.org/rfc/rfc4429.txt, April 2006. Request for Comments (RFC) 4429. [cited at p. 34]

[34] Edward Cheng. On-demand multicast routing in mobile ad hoc networks. Master's thesis, Carleton University, Ontario, Canada, January 2001. [cited at p. 40]

[35] Yunjung Yi, Sung-Ju Lee, William Su, and Mario Gerla. On-demand multicast routing protocol (odmrp) for ad hoc networks. http://tools.ietf.org/id/draft-ietf-manet-odmrp-04.txt, November 2002. Internet draft. [cited at p. 40]

[36] Thomas Kunz. Multicasting in mobile ad-hoc networks: achieving high packet delivery ratios. In *CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 156–170. IBM Press, 2003. [cited at p. 40]

[37] The Network Simulator − ns2. http://www.isi.edu/nsnam/ns/. [cited at p. 45]

[38] International Telecommunication Union. http://www.itu.int/home/. [cited at p. 88]

[39] International Telecommunication Union. Recommendation z.120 (04/04). http://www.itu.int/rec/T-REC-Z.120-200404-P/en, April 2004. Prepublished. [cited at p. 88]

[40] M.A. Reniers. Introduction to message sequence charts (msc'92). http://www.win.tue.nl/∼michelr/mscintro/mscintro.html. [cited at p. 89]

[41] Michael Westergaard. Britney suite. http://wiki.daimi.au.dk/tincpn/. [cited at p. 89]

[42] Directed graph. http://www.nist.gov/dads/HTML/directedGraph.html. [cited at p. 105]

Appendices

Appendix A

ML code

Colour set definitions

```
colset UNIT = unit;
colset INT = int;
colset BOOL = bool;
colset STRING = string;
colset CONFIGURATIONSTAGE =
with
Initial    |
SelectMAC  |
LinkLocal  |
DAD        |
Global     |
FullyConfigured;

colset CONNECTED = BOOL;
colset ID = INT;
colset IDlist = list ID;
colset INTlist = list INT;
colset IP     = STRING;
colset MAC = STRING;
colset PARTITION = with part1 | part2;
colset PREFIX = STRING;
colset PREFIXlist = list PREFIX;
colset PREFIXxINT = product PREFIX*INT;
colset PREFIXxINTlist = list PREFIXxINT;
colset SCOPE = INT;

colset ARep = (*Address reply*)
record
SourceScope : SCOPE *
CurrentScope : SCOPE *
DestinationScope : SCOPE *
```

```
Partition : PARTITION *
Destination : IP *
GlobalIP : IP *
Route : IDlist *
Lifetime : INT;

colset AReq= (*Address request *)
record
SourceScope : SCOPE *
CurrentScope : SCOPE *
DestinationScope : SCOPE *
Source : IP *
Destination : IP *
Route : IDlist *
Partition : PARTITION;

colset NA = (* Neighbor adv. *)
record
SourceScope : SCOPE *
CurrentScope : SCOPE *
DestinationScope : SCOPE *
Partition : PARTITION *
Source : IP *
Target : IP *
Destination : IP *
Route : IDlist *
ReceiverID : ID;

colset Error =
record
msg : STRING;

colset NR = (* Neighbor requests. *)
record
SourceScope : SCOPE *
Partition : PARTITION *
Source : IP *
Route : IDlist *
ReceiverID : ID;

colset NRep = (* Neighbor reply *)
record
SourceScope : SCOPE *
CurrentScope : SCOPE *
DestinationScope : SCOPE *
Partition : PARTITION *
Source : IP *
Route : IDlist *
Destination : IP;

colset NS = (*Neighbor solicitation*)
```

```
record
SourceScope   : SCOPE*
CurrentScope : SCOPE *
DestinationScope : SCOPE *
Partition : PARTITION *
Source : IP *
Target : IP *
Destination : IP *
Route : IDlist *
ReceiverID :ID;

colset REFRESH = (*Refresh requests*)
record
Source : IP *
Prefix    : PREFIX *
Partition : PARTITION *
ReceiverID : ID;

colset AR = (*Address refresh*)
record
Partition : PARTITION *
CurrentScope      :INT *
Prefix : PREFIX *
FromProxy : BOOL;

colset PACKET =
union
NS               : NS +
NA               : NA +
NR               : NR +
NRep             : NRep +
AReq             : AReq +
ARep             : ARep +
ERROR            : Error +
RefreshRequest : REFRESH +
AddressRefresh : AR;

colset IDxSCOPE = product ID * SCOPE;
colset IDxSCOPElist = list IDxSCOPE;
colset NODEINFO = product ID * SCOPE * PARTITION * CONNECTED;
colset NODEINFOlist = list NODEINFO;

colset CONFIGURATION =
record
Scope : SCOPE *
Partition : PARTITION *
IP : IP *
isProxy : BOOL *
ID : ID *
MAC : MAC *
SolAddr : IP *
```

```
ConfigurationStage : CONFIGURATIONSTAGE *
NRcount : INT *
Connected : BOOL;
```

Variable definitions

```
(*REQUESTER*)
var conf : CONFIGURATION;
var count : INT;
var gap : STRING;
var id : INT;
var nodeinfolist : NODEINFOlist;
var ip, newip : IP;
var isProxy : BOOL;
var mac, oldmac : MAC;
var packet : PACKET;
var partition : PARTITION;
var prefix : PREFIX;
var scope : SCOPE;
var sol_addr : IP;
var Unit : UNIT;

(*NODE*)
var addr : INT;

(*Gateway*)
var adhoc_prefix : PREFIX;
var gwconf : CONFIGURATION;
var p : PREFIX;
var preflist : PREFIXlist;
var prefintlist : PREFIXxINTlist;
```

Values

```
(*MISC*)
val hex = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
           "A", "B", "C", "D", "E", "F"];

val IPsize = 37
val linklocalprefix = "FE80:0:0:0:"
val linklocalgap = "FF:FE"
val solicitationPrefix = "FF02:0:0:0:0:1:FF";

(*GATEWAY*)
val globalIP_valid = 1000;
val globalNetworkPrefix = "3FFE:2E01:2B:1111"
val gatewayIP = globalNetworkPrefix^":0000:0000:0000:0001" : IP;
val maxPA = 1;
val maxRefresh = 1;
val maxAR = 1;
```

```
val prefixlist = ["2222:2222:2222","3333:3333:3333",
                  "4444:4444:4444","5555:5555:5555",
                  "6666:6666:6666","7777:7777:7777",
                  "8888:8888:8888","9999:9999:9999"]

(*REQUESTER*)
val DADlimit = 2;
val requestermacs = ["0050:56C0:0008","0050:56C0:0001",
                     "0005:9A3C:7800", "0011:0907:DD45"];

val unspecifiedAddress = "::";
val initialtimer = ERROR({msg=""})

(*CHANGED WHEN TOPOLOGY CHANGES*)
val MANAGEPREFIX = true;
val broadcastRec = [2, 3, 4];
val prefixUsed = "1111:1111:1111"
val maxID = 4
val proxyID = 2;
val req1scope = 0;
val req2scope = 0;
val node1scope = 1;
val node2scope = 1;
val gwscope = 2;
val infolist = [(0, req1scope, part1, true),
                (1, req2scope, part1, true),
                (proxyID, node1scope, part1, true),
                (3, node2scope, part1, true),
                (maxID, gwscope, part1, true)] : NODEINFOlist

val requesterscopelist = [(0,req1scope),
                          (1, req2scope)] : IDxSCOPElist

val numrequesters = List.length requesterscopelist;
val requestconf1 = {NRcount = 0,Scope = req1scope,
                    Partition = part1, SolAddr = "::",
                    isProxy = false,IP = "::", ID = 0, MAC = "::",
                    ConfigurationStage=Initial,
                    Connected = true} : CONFIGURATION

val requestconf2 = {NRcount = 0, Scope = req2scope,
                    Partition = part1, SolAddr = "::",
                    isProxy = false,IP = "::", ID = 1, MAC = "::",
                    ConfigurationStage=Initial,
                    Connected = true} : CONFIGURATION

val config1 = {Scope = node1scope, Partition = part1,
               IP = globalNetworkPrefix ^":"^prefixUsed^":0001",
               isProxy = true, ID = proxyID,
               ConfigurationStage = FullyConfigured, MAC = "::",
               SolAddr="::", NRcount = 0,
```

```
                Connected = true} : CONFIGURATION;

val config2 = {Scope = node2scope, Partition = part1,
               IP = globalNetworkPrefix ^":"^prefixUsed^":0002",
               isProxy = false, ID = proxyID+1,
               ConfigurationStage = FullyConfigured, MAC = "::",
               SolAddr="::", NRcount = 0,
               Connected = true} : CONFIGURATION;

val gwconfig = {Scope = gwscope, Partition = part1, IP = gatewayIP,
                isProxy = false, ID = maxID,
                ConfigurationStage = FullyConfigured, MAC = "::",
                SolAddr="::", NRcount = 0,
                Connected = true} : CONFIGURATION;
```

Functions

Requester functions

MISC

```
fun createAddressRequest(NRep{Route = r, Source = s,
                              SourceScope = ss,
                              Destination = d,...},
                         conf as {Scope = scope, Partition = part,
                                  ID = id,...} : CONFIGURATION,
                         list : NODEINFOlist) =
if notSamePartition(conf, list, List.last(r)) then empty else
1`AReq{SourceScope = scope,
       CurrentScope = updateScopeAux(scope, scope, ss),
       DestinationScope = ss,
       Source = d,
       Destination = s,
       Route = [id],
       Partition = part}
|   createAddressRequest(P : PACKET, _,_) = 1`P
```

```
fun createGlobalIPfromReply(ARep{GlobalIP = gip,...}) =
let
  val gipsize = String.size gip
  val firstpart = if gipsize >= 4 then
                    String.substring(gip, 0, gipsize - 4) else ""
in
  firstpart ^ "0001"
end
|   createGlobalIPfromReply(_) = "ERROR"
```

```
fun createNeighborRequests(conf as {IP = ip, Partition =
                                    part, Scope = s,
                                    ConfigurationStage = cs,
                                    ID = sid,...} : CONFIGURATION,
```

```
                              list : NODEINFOlist) =
let

  fun inList(nil,_) = false
  |   inList((i,s)::xs, id : ID) =
      if i = id then true else inList(xs, id)

  fun inBroadcastList(nil, _) = false
  |   inBroadcastList(i::xs, id) =
      if i = id then true else inBroadcastList(xs, id)

  val withinscopelist =
      List.filter(fn (id, scope, part, con) =>
                  (id <> sid) andalso withinscope(scope, s) andalso
                   not(inList(requesterscopelist, id)) andalso con
                   andalso inBroadcastList(broadcastRec, id)) list

  fun NRaux(nil) = empty
  |   NRaux((i,src, p, con)::xs) =
  if notSamePartition(conf, list, i) then NRaux(xs) else
     1`NR({Route = [sid], Source = ip, Partition = part,
           SourceScope = s, ReceiverID = i})++NRaux(xs)
in
  NRaux(withinscopelist)
end
```

```
fun NRaux(nil) = empty
|   NRaux((i,src, p, con)::xs) =
    1`NR({Route = [sid], Source = ip, Partition = part,
          SourceScope = s, ReceiverID = i})++NRaux(xs)
in
  NRaux(withinscopelist)
end
```

```
fun createSingleNS({IP = linklocal, SolAddr = sol_addr,
                    Partition = part, Scope = scope,
                    ID = sid,...} : CONFIGURATION,
                    id : ID, destscope : SCOPE) =
1`NS({Target            = linklocal,
      Partition         = part,
      Source            = unspecifiedAddress,
      Destination       = sol_addr,
      SourceScope       = scope,
      CurrentScope      = updateScopeAux(scope, scope, destscope),
      DestinationScope  = destscope,
      Route             = [sid],
      ReceiverID        = id})
```

```
fun createSingleNA({Scope = scope, IP = ip, Partition = part,
                    ID = srcid,...} : CONFIGURATION,
                    destscope : SCOPE, target : IP, dest : IP,
```

```
                    id : ID)=
1`NA{SourceScope = scope,
     CurrentScope = updateScopeAux(scope, scope, destscope),
     DestinationScope = destscope,
     Partition = part,
     Source = ip,
     Target  = target,
     Destination = dest,
     Route         = [srcid],
     ReceiverID = id
}
```

```
fun createMultipleNS(c as {IP = ip, SolAddr = sa, ID = id,
                           Partition = part,...} : CONFIGURATION,
                     list ) =
let
  fun NSaux(nil) = empty
  |   NSaux((i, s,p,con)::xs : NODEINFOlist) =
        if i = id orelse (notSamePartition(c, list, i)) then
          NSaux(xs)
        else
          createSingleNS(c, i, s) ++ NSaux(xs)
in
  if ip = unspecifiedAddress then empty else
     NSaux(List.take(infolist, numrequesters))
end
```

```
fun createNStimeout(conf as {MAC = mac,...} : CONFIGURATION,
                    NS{Target = t,...}) =
let
  val firstpart  = if t <> unspecifiedAddress then
                   String.substring(t, 11, 7) else ""
  val secondpart = if t <> unspecifiedAddress then
                   String.substring(t, 23, 7) else ""
in
  if ((firstpart ^ secondpart) = mac) then createMultipleNS(conf)
     else empty
end
|   createNStimeout(_) = empty
```

```
fun generateLinkLocal(prefix : PREFIX, gap : STRING, mac : MAC) =
let
  val partOne = if String.size mac >= 8 then
                String.substring(mac : STRING, 0, 7)  else ""
  val partTwo = if String.size mac >= 8 then
                String.extract(mac : STRING, 7, NONE) else ""
in
  if String.size mac >= 8 then
  prefix ^ partOne ^ gap ^ partTwo : IP else "ERROR"
end
```

```
fun generateNAs(NS{Source = src, SourceScope = ss, Target = t,
                   Destination = d, ...},
                c as {Partition = part, IP = ip, ID = id,
                      Scope = scope,...} : CONFIGURATION) =
let
  fun NAaux(nil) = empty
  |   NAaux((i,s,p,con)::xs : NODEINFOlist) =
        if i = id orelse notSamePartition(c, list, i) then
          NAaux(xs)
        else
          createSingleNA(c, s, t,src, i) ++ NAaux(xs)
in
  NAaux(List.take(infolist, numrequesters))
end
|   generateNAs(P as PACKET, _) = 1`P
```

```
fun initiateGIP({ConfigurationStage = cs,
                 MAC = mac,...} : CONFIGURATION) =
if cs = DAD then 1`mac else empty
```

```
fun makeMAClist(intlist : INTlist) =
let
   fun aux(nil) = empty
   |     aux(x::xs) = if (x >= List.length(requestermacs) orelse
                         (x < 0))
                     then aux(xs) else
                         1`List.nth(requestermacs, x) ++ aux(xs)
in
  aux(intlist)
end
```

```
fun updateCount(count : INT, NS{Target = t, ...},
                {MAC= mac,...} :CONFIGURATION) =
let
  val newcount = if String.size t > String.size unspecifiedAddress
                 then count+1 else 0
in
  newcount
end
|   updateCount(_) = 0
```

```
fun updPart({Scope = s, IP = ip, isProxy = isp, ID = id, MAC = mac,
             SolAddr = sa, ConfigurationStage = c, NRcount = nrc,
             Connected= con,...} : CONFIGURATION,
            npart : PARTITION) =
{Scope = s, IP = ip, isProxy = isp, ID = id, Partition = npart,
 MAC = mac, SolAddr = sa, ConfigurationStage = c, NRcount = nrc,
 Connected = con } : CONFIGURATION
```

```
fun resetFullyConfigured(conf : CONFIGURATION) =
  if CS(conf) = FullyConfigured then resetConfig(conf) else conf
```

```
fun upReqScope(list : NODEINFOlist, id : ID, nscope : SCOPE) =
let
  val arr = Array.fromList(list)
  val (uid, us, up, ucon) = List.nth(list, id)
  val n as (nid, ns,np,ncon) = (uid, nscope, up, ucon)
  val array = Array.update(arr, id, n)
in
  Array.foldr (op ::) [] arr
end
```

```
fun upReqPart(list : NODEINFOlist, id : ID, part : PARTITION) =
let
  val arr = Array.fromList(list)
  val (uid, us, _, ucon) = List.nth(list, id)
  val n as (nid, ns, np, ncon) = (uid, us, part, ucon)
  val array = Array.update(arr, id, n)
in
  Array.foldr (op ::) [] arr
end
```

Extracting functions

```
fun extractGIP(ARep{GlobalIP = gip,...}) = gip
|   extractGIP(_) = "ERROR"
```

```
fun TARGET(NS{Target = t,...}) = t
|   TARGET(NA{Target = t,...}) = t
|   TARGET(_) = "ERROR"
```

```
fun extractIP(NS{Target = target, ...}) = target
|   extractIP(_) = "ERROR_EXTRACTING_IP" : IP
```

Guard functions

```
fun canCreateNS(mac : MAC, conf : CONFIGURATION) =
mac<>unspecifiedAddress andalso
MAC(conf) = mac andalso CS(conf) = LinkLocal
```

```
fun canSendNRs(conf, mac) =
mac<>unspecifiedAddress andalso mac = MAC(conf) andalso
CS(conf) = Global
```

```
fun canInitiateGIP(conf, count)=
count = DADlimit
  andalso
CS(conf) = DAD
```

```
fun discardNS(NS{ReceiverID = rid, Target = target,
                 Partition = part, CurrentScope = cs,...},
              conf : CONFIGURATION) =
let
```

```
  val id = ID(conf)
  val ip = generateLinkLocal(linklocalprefix, linklocalgap,
                             MAC(conf))
  val p  = PARTITION(conf)
  val con = CONNECTED(conf)
in
(con andalso (cs = SCOPE(conf)) andalso (rid = id)) andalso
 (not(target = ip) andalso (part = p) )
end
|    discardNS(_) = false
```

```
fun discardNA(NA{ReceiverID = rid, Destination = dest,
                 Partition = part,...},
              {ID = id, IP = ip, Partition =  p,
               ConfigurationStage = cs,
               Connected = con,...} : CONFIGURATION)=
(id = rid) andalso (cs = Initial) andalso con
|   discardNA(_) = false
```

```
fun enableTE(count : INT, conf : CONFIGURATION) =
count < DADlimit andalso count > 0 andalso(CS(conf) = DAD)
```

```
fun isIPrange(ip: IP) =
let
  val stringsize = String.size ip
  val lastpart = if stringsize >= 4 then
                 String.substring(ip, stringsize-4, 4) else ""
in
  lastpart = "0000"
end
```

```
fun isNS(NS{ReceiverID = rid, Destination = dest,
            CurrentScope = cs, ...}, {ID = id, SolAddr = ip,
            Scope = scope, Connected = con,...} : CONFIGURATION) =
(rid = id) andalso (ip = dest) andalso (cs = scope) andalso con
|   isNS(_)  = false
```

```
fun isNA(NA{ReceiverID = rid,Target =target, CurrentScope = cs,...},
         {ID = id, IP = ip, Scope = scope, MAC = mac,
          Connected = con,...} : CONFIGURATION) =
let
  val linklocal = if mac <> unspecifiedAddress then
                  generateLinkLocal(linklocalprefix,
                                    linklocalgap, mac)
                  else ""
in
  (rid = id) andalso ( (ip = target) orelse (target = linklocal) )
  andalso (cs = scope) andalso con
end
|   isNA(_,_) = false
```

```
fun getNR(packet as NRep{CurrentScope = cs, Partition = part,
                         Destination =dest,DestinationScope=ds,...},
          {Connected = con, MAC = mac, Scope = s, SolAddr = sol_ip,
           Partition = p, ConfigurationStage = cstage,
           NRcount = nrc,...} : CONFIGURATION) =
(part = p)  andalso (cs = s)
  andalso
(dest = generateLinkLocal(linklocalprefix, linklocalgap, mac))
  andalso nrc = 0 andalso (cstage = Global) andalso con
|   getNR(_) = false
```

```
fun discardNR(NRep{CurrentScope = cs, Partition = part,
                   Destination = dest,DestinationScope=ds,...},
              {Connected = con, MAC = mac, Scope = s,
               SolAddr = sol_ip, Partition = p,
               ConfigurationStage = cstage,
               NRcount = nrc,...} : CONFIGURATION) =
(dest = generateLinkLocal(linklocalprefix, linklocalgap, mac))
  andalso
((ds <> s) orelse (nrc <> 0) )
  andalso con
|   discardNR(_) = false
```

```
fun isARep(ARep{CurrentScope = cs, Partition = part,
                Destination = dest, ...}, {Scope = s, Partition = p,
                IP = ip, Connected = con,...} : CONFIGURATION) =
(s = cs) andalso (part = p) andalso (dest = ip) andalso con
|   isARep(_) = false
```

```
fun keepRR(RefreshRequest{Prefix=pref, ReceiverID = rid,...},
           {IP = ip, ConfigurationStage=c, ID = id,
            Connected = con,...} : CONFIGURATION) =
let
  val extractedPrefix = if String.size ip < IPsize then "::" else
                        extractPrefix(ip)
in
  (pref = extractedPrefix) andalso (c = FullyConfigured) andalso
  (id = rid) andalso con
end
|   keepRR(_) = false
```

```
fun promoteToProxy(packet : PACKET, conf : CONFIGURATION)=
isIPrange(extractGIP(packet)) andalso isARep(packet, conf)
```

```
fun getAddressReply(packet as ARep{Route=r,...},
                    conf : CONFIGURATION) =
 isARep(packet, conf)  andalso not(isIPrange(extractGIP(packet)))
 andalso (CS(conf) = Global) andalso CONNECTED(conf) andalso
 List.hd r = ID(conf)
|   getAddressReply(_) = false
```

Node functions

MISC

```
fun padIP(ip, 0) = ""
|   padIP(ip, padding) = padIP(ip, padding-1) ^"0"
```

```
fun createAddressReply(AReq{SourceScope=ss, CurrentScope = cs,
                            DestinationScope = ds, Source =src,
                            Route= r,...},
                       conf as {IP = manetnodesIP,
                                Partition = partition,
                                Scope = scope  ,... }:CONFIGURATION,
                       constructedIP : IP, list : NODEINFOlist) =
let
  val size = String.size constructedIP
  val slack = 4 - size
  val prefix = extractPrefix(manetnodesIP)
  val correctedIP = if String.size manetnodesIP >= 17
  then
    String.substring(manetnodesIP, 0, 18 ) ^
                     prefix ^":" ^ padIP(constructedIP,
                                         slack) ^ constructedIP
  else "ERROR"
in
 if notSamePartition(conf, list, List.last(r)) then empty else
     1`ARep{DestinationScope = ss,
            CurrentScope = updateScopeAux(scope,scope,ss),
            SourceScope = scope, Partition = partition,
            Destination = src, GlobalIP = correctedIP,
            Lifetime = globalIP_valid, Route = r}
end
|   createAddressReply(_) = 1`ERROR{msg = "NOT AN ADDRESS REQUEST"}
```

```
fun createNeighborReply(NR{Source = src, SourceScope = ss,
                           Route = r,...},
                        conf as {Partition = part, IP = ip,
                                 Scope = scope,
                                 ID = id,... }: CONFIGURATION,
                        list : NODEINFOlist) =
if notSamePartition(conf, list, List.last(r)) then empty else
1`NRep{DestinationScope = ss,
       CurrentScope = updateScopeAux(scope, scope, ss) ,
       SourceScope = scope, Partition = part, Source = ip,
       Route = [id], Destination = src}
|   createNeighborReply(_) = 1`ERROR{msg = "NOT A NEIGHBOR REQUEST"}
```

```
fun createAddressRefresh(conf as {IP : IP, Partition = part,
                                  Scope = s,...} : CONFIGURATION,
                         RefreshRequest{Prefix = pref,...},
                         isProxy : BOOL, list) =
```

```
if notSamePartition(conf, list, maxID) then empty else
1`AddressRefresh{Prefix =pref ,
                 CurrentScope = updateScopeAux(s, s, gwscope) ,
                 Partition = part, FromProxy = isProxy}
|   createAddressRefresh(_,_,_,_) = empty
```

```
fun createAddressRefreshProxy(conf as {IP : IP, Partition = part,
                                       Scope = s,...}:CONFIGURATION,
                              list : NODEINFOlist) =
if notSamePartition(conf, list, maxID) then empty else
1`AddressRefresh{Prefix = extractPrefix(IP),
                 CurrentScope = updateScopeAux(s, s, gwscope) ,
                 Partition = part, FromProxy = true}
```

```
fun differentPartitions(NR{Partition = p,...}, part : PARTITION) =
(p <> part)
|   differentPartitions(_) = false
```

```
fun forwardAddressRequest(AReq{SourceScope = ss, CurrentScope = cs,
                               DestinationScope = ds, Source = src,
                               Destination = d, Route = r,
                               Partition = p},
                          {Scope = s,ID = id,...} : CONFIGURATION)=
AReq{SourceScope = ss,
     CurrentScope = updateScopeAux(s, cs, gwscope),
     DestinationScope = gwscope, Source = src, Destination = d,
     Route = id::r, Partition = p}
|   forwardAddressRequest(_) =
     ERROR{msg = "error forwarding address request"}
```

```
fun makeAddr(old : INT) =
let
    fun toHex(d) =
    let
      val r = d mod 16
    in
      if (d-r) = 0 then List.nth(hex, d) else
          toHex( (d-r) div 16) ^ List.nth(hex,r)
    end
 in
  toHex(old)
 end
```

```
fun updateScope(ARep{Route = r, SourceScope = ss, CurrentScope = cs,
                     DestinationScope = ds, Partition = p,
                     Destination = dest, GlobalIP = gip,
                     Lifetime = lt}, list : NODEINFOlist, id : ID) =
let
  val (userid, userscope,_,_) = List.nth(list, List.last r)
  val newroute = if hd r = id then List.tl r else r
in
```

```
 ARep{Route = newroute, SourceScope = ss,
      DestinationScope = userscope,
      CurrentScope = updateScopeAux(ss,cs,userscope),
      Destination = dest, Partition = p, GlobalIP = gip,
      Lifetime = lt}
end
|   updateScope(AReq{Route = r, Source = src, Destination = dest,
                      Partition = p, SourceScope = ss,
                      CurrentScope = cs, DestinationScope = ds},
                 list : NODEINFOlist, id : ID)=
let
  val (userid, userscope,_,_) = List.nth(list, List.last r)
  val newroute = if hd r = id then List.tl r else id::r
in
 AReq{Route = newroute, SourceScope = ss,
      DestinationScope = updateScopeAux(ss,cs,ds), Source = src,
      CurrentScope = updateScopeAux(ss,cs,ds), Destination = dest,
      Partition = p}
end
|   updateScope(_) = ERROR{msg = "FAILED TO UPDATE SCOPE"}
```

Guard functions

```
fun forwardAux(current : SCOPE, dest : SCOPE, myScope : SCOPE) =
(current <> dest) andalso (current = myScope)
```

```
fun userMoved(userid : ID, list : NODEINFOlist, ds : SCOPE) =
let
  val (id, scope,_,_) = List.nth(list, userid)
in
  scope <> ds
end
```

```
fun forwardMessage(ARep{CurrentScope = cs, DestinationScope = ds,
                        Partition = part, Route = r, ...},
                   myscope : SCOPE, p : PARTITION, id : ID,
                   con : BOOL, userlist : NODEINFOlist) =
forwardAux(cs, ds, myscope) andalso (part = p) andalso
not(userMoved(List.last r, userlist, ds)) andalso (hd r = id)
andalso con
|   forwardMessage(_) = false
```

```
fun isNREQ(NR{ReceiverID = idr, Partition = part,
              SourceScope = ss,...}, id : ID) =
(id = idr)
|   isNREQ(_) = false
```

```
fun withinreach(NR{SourceScope = ss, ...}, scope : SCOPE) =
((ss = scope) orelse (ss = scope-1) orelse (ss=scope+1))
|   withinreach(_) = false
```

```
fun isRefreshRequest(RefreshRequest{Partition = part,
```

```
                                        ReceiverID = rid,...},
                        p : PARTITION, id : ID)=
(rid = id) andalso (p = part)
|    isRefreshRequest(_,_,_) = false
```

```
fun sendReply(inpacket : PACKET, conf : CONFIGURATION) =
isNREQ(inpacket, ID(conf))
  andalso
withinreach(inpacket, SCOPE(conf))
  andalso
not(differentPartitions(inpacket, PARTITION(conf)))
  andalso
CONNECTED(conf)
```

```
fun discardNREQ(inpacket : PACKET, conf : CONFIGURATION) =
(
 isNREQ(inpacket, ID(conf))
   andalso
 (not(withinreach(inpacket,SCOPE(conf))))
   orelse
 differentPartitions(inpacket, PARTITION(conf))
)
andalso
  CONNECTED(conf)
```

Gateway functions

MISC

```
fun createAddressReplyGateway(AReq{SourceScope=ss, Source =src,
                                   Route = x::xs,...},
                              prefix,
                              conf as {Partition = partition,
                              Scope = scope,
                              ID = sid,...}:CONFIGURATION,
                              list : NODEINFOlist) =
if notSamePartition(conf, list, List.last(x::xs)) then empty else
 1`ARep{DestinationScope = ss,
        CurrentScope = updateScopeAux(scope, scope,ss),
        SourceScope = scope, Partition = partition,
        Destination = src,
        GlobalIP = globalNetworkPrefix^":"^prefix^":0000",
        Lifetime = globalIP_valid, Route = x::xs}
|   createAddressReplyGateway(_) =
     1`ERROR{msg = "NOT AN ADDRESS REQUEST"}
```

```
fun createRefreshRequest(adhoc_prefix : PREFIX,
                         conf : CONFIGURATION, id : ID, list) =
let
  fun RRaux(~1) = empty
  |   RRaux(a : ID) =
   let
      val (_,_,_,con) = List.nth(list,a)
   in
      if a = id orelse not(con) orelse
         notSamePartition(conf, list, a)
      then
        RRaux(a-1)
      else
        1`RefreshRequest{Source = gatewayIP, Prefix = adhoc_prefix,
                         Partition = PARTITION(conf),
                         ReceiverID = a} ++ RRaux(a-1)
  end
in
  RRaux(maxID)
end
```

```
fun updateList(prefix : PREFIX, list : PREFIXxINTlist, cnt : INT) =
let
  val (match, nomatch)=List.partition(fn (p,i) => p = prefix) list;

   fun updatelist(nil) = (prefix, 0)::nomatch
   |   updatelist(x::xs : PREFIXxINTlist) =
         if cnt = 0 then (prefix, 0)::nomatch
          else (prefix, #2(x)+1)::nomatch
in
```

```
  updatelist(match)
end
```

```
fun updatePrefixAcked(prefintlist : PREFIXxINTlist, packet) =
if prefintlist = nil then
  nil
else
  updateList(extractPrefixFromPacket(packet) , prefintlist , ~1)
```

```
fun removePrefix(p : PREFIX, list : PREFIXxINTlist) =
let
  val neg = List.filter(fn (pre, count) => pre <> p) list
in
  neg
end
```

Extracting functions

```
fun extractPrefixFromPacket(AddressRefresh{Prefix = pre, ...})
                                                    : PREFIX =
 pre
|   extractPrefixFromPacket(_) = "ERROR"
```

Guard functions

```
fun acked(pl : PREFIXxINTlist, pref : PREFIX) =
let
  val (pos, neg) = List.partition(fn (p,i) => p=pref) pl
in
   if pos <> nil then #2(List.hd(pos)) > 0 else false
end
```

```
fun isAddressRequest(AReq{Partition = part, CurrentScope = cs,
                          Destination = dest,...},
                     partition : PARTITION, scope : SCOPE) =
(part = partition) andalso (cs = scope)
|   isAddressRequest(_) = false;
```

```
fun isAddressRefresh(AddressRefresh{Partition = p,
                                    CurrentScope = cs,...},
                     {Scope = s,
                      Partition = gwpart,...} : CONFIGURATION) =
(p = gwpart)
|   isAddressRefresh(_,_) = false;
```

```
fun DiscardfromProxy(AddressRefresh{Partition = p,
                                    FromProxy = fp,...},
                     {Partition = part,...} : CONFIGURATION) =
(p = part) andalso fp
|   DiscardfromProxy(_) = false
```

```
fun receivedAddressRefresh(AddressRefresh{Prefix = prefix,...},
```

```
                              list : PREFIXxINTlist) =
List.exists(fn (x,y) => ((x = prefix) andalso (y<>0))) list
|    receivedAddressRefresh(_) = false
```

```
fun discardAddressRefresh(packet : PACKET, gwconf : CONFIGURATION,
                          prefintlist : PREFIXxINTlist)=
receivedAddressRefresh(packet, prefintlist)
  orelse
(
 DiscardfromProxy(packet, gwconf) andalso
 receivedAddressRefresh(packet, prefintlist)
)
  orelse
(DiscardfromProxy(packet, gwconf) andalso prefintlist = nil)
```

```
fun getAddressRefresh(packet : PACKET, gwconf : CONFIGURATION,
                      prefintlist : PREFIXxINTlist) =
let
  val prefix = extractPrefixFromPacket(packet)
  val prefixinlist = List.filter(fn (p, count) => (p = prefix)
                     andalso (count = 0)) prefintlist
in
  prefixinlist <> nil
     orelse
(
  not(DiscardfromProxy(packet, gwconf))
    andalso
  isAddressRefresh(packet, gwconf)
    andalso
  not(receivedAddressRefresh(packet, prefintlist))
)
end
```

MISC functions

```
fun canReceiveRRs(packet : PACKET, conf : CONFIGURATION) =
 isRefreshRequest(packet,  PARTITION(conf), ID(conf))
   andalso
 CONFIGUREDIP(conf)
   andalso
 CONNECTED(conf)
```

```
fun createReply(packet as AReq{Destination = dest,...},
                conf : CONFIGURATION) =
isAddressRequest(packet, PARTITION(conf), SCOPE(conf))
  andalso
ISPROXY(conf)
  andalso
dest = IP(conf)
  andalso
CONNECTED(conf)
|    createReply(_) = false
```

```
fun discardNonAcked(packet : PACKET, conf : CONFIGURATION)=
isRefreshRequest(packet, PARTITION(conf), ID(conf)) andalso
not(CONFIGUREDIP(conf))
```

```
fun discardRRs(packet as RefreshRequest{ReceiverID = rid,...},
               conf as {ID = id,Connected = con,...}: CONFIGURATION,
               count : INT)=
if not(keepRR(packet, conf)) then
   (id = rid) andalso con
else
  (not(keepRR(packet, conf)) orelse discardNonAcked(packet, conf))
    orelse
  count >= maxRefresh
|    discardRRs(_) = false
```

```
fun passRequest(inpacket as AReq{DestinationScope = ds,
                                 CurrentScope = cs,
                                 Destination = dest,...},
                conf : CONFIGURATION) =
not(ISPROXY(conf))
  andalso
isAddressRequest(inpacket, PARTITION(conf),SCOPE(conf))
  andalso
dest = IP(conf)
  andalso
CONNECTED(conf)
|    passRequest(_) = false
```

```
fun sendRRs(packet : PACKET, conf : CONFIGURATION, count : INT)=
if CONFIGUREDIP(conf) andalso CONNECTED(conf) then
(keepRR(packet, conf)
  andalso
 count < maxRefresh)
else
  false
```

BRITNeY

Initialization - scenario 4

```
structure msc = MSC(val name = "Manage prefix")
val _ = msc.addProcess("Node1");
val _ = msc.addProcess("Node2");
val _ = msc.addProcess("Proxy2");
val _ = msc.addProcess("Proxy1");
val _ = msc.addProcess("Gateway");
val processList = ["Node1", "Node2","Proxy2", "Proxy1", "Gateway"];
```

Functions

```
fun idToProcess(id : ID) =
```

```
 if id >= List.length processList then "" else
                                        List.nth(processList, id)
```

```
fun partToStr(p : PARTITION) =
  case p of part1 => "part1"
          | part2 => "part2"
```

```
fun britneyAREP(conf : CONFIGURATION, AReq{Route = r,...},
                addr : INT, list : NODEINFOlist) =
let
   val recv = hd r
   val newhostid = Int.toString(addr+1)
   val slack = 4 - String.size newhostid
   val nexthostid = padIP(newhostid, slack) ^ newhostid
   val notsame = notSamePartition(conf, list, recv)
in

 if notsame then
 (msc.addEvent(idToProcess(ID(conf)), idToProcess(recv),
               "Address reply LOST"))

 else
 (msc.addEvent(idToProcess(ID(conf)), idToProcess(recv),
               "Address reply"));
  msc.addInternalEvent(idToProcess(ID(conf)),
                       "Next HostID: " ^ nexthostid)
end
|   britneyAREP(_) = ()
```

```
fun britneyAREPGW(conf : CONFIGURATION, AReq{Route = r,...}) =
     msc.addEvent(idToProcess(ID(conf)), idToProcess(hd r),
                  "Address reply")
|   britneyAREPGW(_) = ()
```

```
fun britneyCreateAR(conf, NRep{Route = r,...}, list : NODEINFOlist)=
let
   val recv = hd r
   val notsame = notSamePartition(conf, list, recv)
in
  if notsame then
   msc.addEvent(idToProcess(ID(conf)), idToProcess(recv),
                "Addr req. LOST")
else
  msc.addEvent(idToProcess(ID(conf)), idToProcess(recv),
               "Addr req.")
end
|   britneyCreateAR(_)= ()
```

```
fun britneyGWAddressRefresh(AddressRefresh{Prefix = pref,...}) =
msc.addInternalEvent("Gateway", "Ref rec. " ^pref)
|   britneyGWAddressRefresh(_) = ()
```

```
fun britneyCreateNR(NR{Route = r,...}, conf : CONFIGURATION,
                    list : NODEINFOlist) =
let
  val recv = hd r : ID
  val notsame = notSamePartition(conf, list, recv)
in
 if notsame then
    msc.addEvent(idToProcess(ID(conf)), idToProcess(recv),
                 "Neigh. reply LOST")
else
  msc.addEvent(idToProcess(ID(conf)), idToProcess(recv),
               "Neigh. reply")
end
|   britneyCreateNR(_) = ()
```

```
fun britneyForward(id : ID , srcscope : SCOPE, ARep{Route = r,...})=
  msc.addEvent(idToProcess(id), idToProcess(hd (List.tl r)),
               "Forward AREP")
|   britneyForward(id : ID , srcscope : SCOPE, AReq{Route = r,...})=
  msc.addEvent(idToProcess(id), idToProcess(hd (List.tl r)),
               "Forward AREQ")
|   britneyForward(id : ID , srcscope : SCOPE, NRep{Route = r, ...})=
  msc.addEvent(idToProcess(id), idToProcess(hd (List.tl r)),
               "Forward NRep")
|   britneyForward(_) = ()
```

```
fun britneyNotReceived(conf : CONFIGURATION, bc : NODEINFOlist,
                       prefix : PREFIX) =
let
  fun Aux(nil) = ()
  |   Aux((x,s, p, c)::xs : NODEINFOlist) =
   (
    if ID(conf) = x orelse notSamePartition(conf, bc, x) then
      Aux(xs)
    else
      (msc.addEvent(idToProcess(ID(conf)), idToProcess(x),
                    "Send Refresh: " ^ prefix) ; Aux(xs) )
   );
in
  Aux(bc)
end
```

```
fun britneyDiscardAR(AddressRefresh{Prefix = pre,...}) =
 msc.addInternalEvent("Gateway", "ACK: " ^ pre)
|   britneyDiscardAR(_) = ()
```

```
fun britneyGIP(conf : CONFIGURATION, bc : NODEINFOlist) =
let
   fun GIPAux(nil) = ()
   |   GIPAux((x,s, p, c)::xs : NODEINFOlist) =
```

```
    (
     if ID(conf) = x orelse not(withinscope(SCOPE(conf), s)) then
       GIPAux(xs)
     else
      if notSamePartition(conf, bc, x) then
         (msc.addEvent(idToProcess(ID(conf)), idToProcess(x),
                       "NREQ LOST") ; GIPAux(xs) )
         else (msc.addEvent(idToProcess(ID(conf)), idToProcess(x),
                            "Send NREQ") ; GIPAux(xs) )
   );
in
 GIPAux(List.drop(infolist, numrequesters))
end
```

```
fun britneyPromote(conf : CONFIGURATION, packet) =
(msc.addInternalEvent(idToProcess(ID(conf)), "promote");
 msc.addInternalEvent(idToProcess(ID(conf)),"Setup complete");
 msc.addInternalEvent(idToProcess(ID(conf)),
                      "IP: "^createGlobalIPfromReply(packet))
);
```

```
fun britneySendNA(conf, NS{Source = src, Route = r,...},
                  list : NODEINFOlist) =
let
  val recv = List.last r
  val notsame = notSamePartition(conf, list, recv)
in
  if notsame then
     msc.addEvent(idToProcess(ID(conf)), idToProcess(recv),
                  "NA LOST")
  else
     msc.addEvent(idToProcess(ID(conf)), idToProcess(recv),
                  "Send NA")
end
|   britneySendNA(_) = ()
```

```
fun britneySendAddressRefresh(conf : CONFIGURATION, count : INT,
                              list : NODEINFOlist) =
let
  val notsame = notSamePartition(conf, list, maxID)
in
  if notsame then
     msc.addEvent(idToProcess(ID(conf)), "Gateway",
                  "Address refresh LOST")
  else
   (
    if count < maxAR then
       msc.addEvent(idToProcess(ID(conf)), "Gateway",
                    "Addr. Ref. : " ^extractPrefix(IP(conf)) )
    else
      (msc.addEvent(idToProcess(ID(conf)), "Gateway",
```

```
                    "Addr. Ref. : " ^extractPrefix(IP(conf)) );
        msc.addInternalEvent(idToProcess(ID(conf)),
                             "DISCONNECT"))
)
end
```

```
fun britneyNS(conf : CONFIGURATION, list : NODEINFOlist) =
let
  fun  NSaux(nil) = ()
   |   NSaux((i,s,p,con)::xs : NODEINFOlist) =
       (
        if ID(conf) = i orelse notSamePartition(conf, list, i) then
           NSaux(xs)
        else
           (msc.addEvent(idToProcess(ID(conf)), idToProcess(i)  ,
                         "NS");
         NSaux(xs))
       );
in
  NSaux(List.take(infolist, numrequesters))
end
```

```
fun britneySendRR(conf : CONFIGURATION, list : NODEINFOlist) =
let
  val notsame = notSamePartition(conf, list, maxID)
in
  if notsame then
   msc.addEvent(idToProcess(ID(conf)),  "Gateway", "RREP LOST")
else
  msc.addEvent(idToProcess(ID(conf)),  "Gateway", "Send RREP")
end
```

State space functions

```
fun WriteToFile(list, name : STRING) =
let
  val off = TextIO.openOut(outpath ^"/output/" ^ name ^".txt")

  fun write(nil) = (TextIO.output(off, "\n"); TextIO.closeOut(off))
  |   write(x::xs) = (TextIO.output(off, x^"\n"); write(xs));
in
  write(list)
end
```

```
fun WriteLog(dir : string, msg : string) =
let
  val outfile = TextIO.openAppend(dir)
in
  TextIO.output(outfile,msg);
  TextIO.closeOut(outfile)
end
```

```
(*Write the shortst path from the initial marking to the markings*)
fun WriteListToFile(nil) = ()
|   WriteListToFile(x::xs) =
let
  fun writeaux(nil) = ()
  |   writeaux(x::xs) =
    let
      val index = x;
      val strlist = List.map st_BE(List.map
                                   ArcToBE(ArcsInPath(1,index)))
    in
      WriteToFile(strlist, Int.toString(index));
      writeaux(xs)
    end
in
  writeaux(x::xs)
end
```

```
fun writeDup(nil)   = WriteLog(outdir, "NIL")
|   writeDup(x::xs) = (WriteLog(outdir, Int.toString(x)^"\n");
                       writeDup(xs))
```

```
fun GenStateSpaceReport(p : string, name : string) =
(
  WriteLog(outdir, "Initializing OCC graph construction...\n");
  CalculateOccGraph();
  if OGSaveReport.FullOG() = "1" then
    WriteLog(outdir, "FULL OCC graph complete ...\n")
  else
    WriteLog(outdir, "PARTIAL OCC graph complete ...\n");
```

```
  WriteLog(outdir, "Initilizing SCC graph construction...\n");
  CalculateSccGraph();
  WriteLog(outdir, "SCC graph complete...\n");

  WriteLog(outdir, "Saving report...\n");
  OGSaveReport.SaveReport(true,true,true,true,
                          true,true,true,true, p^name);
  WriteLog(outdir, "Report saved...\n\n");

  WriteLog(outdir, "QUERYS.............\n\n");
  WriteLog(outdir, "Deadmarkings is home space: " ^
                Bool.toString(HomeSpace(ListDeadMarkings()))^"\n");
  WriteLog(outdir, "detectDuplicationsList: \n");
  writeDup(detectDuplications(ListDeadMarkings()));
  WriteLog(outdir, " \n");

  WriteLog(outdir, "Address NOT acquired: \n");
  writeDup(notAddressAcquired(ListDeadMarkings()));

 WriteLog(outdir, "\n---------------------- \n\n\n")
 )
```

```
(*Detects if two requesters have the same IP address at the same
  time*)
fun detectDuplications(list) =
let
  fun check(n) =
    let
      val ip1 = ms_to_col(Mark.Requester'Global_IP 1 n)
      val ip2 = ms_to_col(Mark.Requester'Global_IP 2 n)
    in
      (ip1 = ip2)
        andalso
      (ip1 <> unspecifiedAddress)
        andalso
      (ip2 <> unspecifiedAddress)
   end

  fun aux(nil) = nil
  |   aux(x::xs) = if check(x) then x::aux(xs) else aux(xs)
in
  aux(list)
end
```

```
fun detectPrefix(list : Node list)=
let
  fun inPrefixList(pre, nil) = false
  |   inPrefixList(pre, x::xs) = if x = pre then true else
                                 inPrefixList(pre, xs)

  fun check(n) =
```

```
    let
      val reqconf   = ms_to_col(Mark.MANET'Req1_Conf 1 n)
      val proxyconf = ms_to_com(Mark.MANET'Conf1 1 n)
      val prefixlist =
      ms_to_col(
            Mark.Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1 n)
  in
    (CONNECTED(reqconf) orelse CONNECTED(proxyconf)) andalso
    inPrefixList(extractPrefix(IP(reqconf)), prefixlist)
  end

fun aux(nil) = nil
|   aux(x::xs) = if check(x) then aux(xs) else x::aux(xs)

in
  aux(list)
end
```

```
(*Checks if any of the markings given in "list" has a requester
  without an IP*)
fun notAddressAcquired(list)=
let
  fun check(n) =
  (
    (ms_to_col(Mark.Requester'Global_IP 1 n) <> "::" andalso
     CS(ms_to_col(Mark.MANET'Req1_Conf 1 n)) = FullyConfigured)
      orelse
    (ms_to_col(Mark.Requester'Global_IP 2 n)  <> "::" andalso
     CS(ms_to_col(Mark.MANET'Req2_Conf 1 n)) = FullyConfigured)
   )

fun aux(nil) = nil
|   aux(x::xs) = if check(x) then aux(xs) else x::aux(xs)
in
  aux(list)
end
```

Appendix B

State space reports

One requester

```
CPN Tools state space report for:
C:\SPECIALE\1-OneRequester.cpn
Report generated: Wed Jun 13 19:09:30 2007

 Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  34
     Arcs:   48
     Secs:   0
     Status: Full

  Scc Graph
     Nodes:  34
     Arcs:   48
     Secs:   0

 Boundedness Properties
------------------------------------------------------------------------


  Best Integer Bounds                                 Upper
      Lower
     Create_Address_Response'Node_Info 1               1          1
     Duplicate_Address_Detection'NS_Sent 1             1          1
     Duplicate_Address_Detection'Node_Info 1           1          1
     Duplicate_Address_Detection'Timer 1               1          1
     Gateway'Node_Info 1                               1          1
     Global_IP_Request'Node_Info 1                     1          1
```

```
   Handle_Address_Refresh'Extracted_Prefix 1        0          0
   Handle_Address_Refresh'Prefix_Ack'ed 1           1          1
   Handle_Address_Refresh'Prefixes_Timed_Out 1      0          0
   Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1 1          1
   Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1    2          1
   Handle_Incoming_Packets_GATEWAY'Timer 1          0          0
   Handle_Incoming_Packets_NODE'Node_Info     1         1
               1
   Handle_Incoming_Packets_NODE'Node_Info 2         1          1
   Handle_NR'Node_Info 1                            1          1
   Handle_NREQ'Node_Info 1                          1          1
   Handle_NREQ'Node_Info 2                          1          1
   Handle_NREQ'Node_Info 3                          1          1
   Handle_NS'Node_Info 1                            1          1
   Handle_RREQ'Node_Info 1                          1          1
   Handle_RREQ'Node_Info 2                          1          1
   Handle_RREQ'Node_Info 3                          1          1
   Handle_RREQ'Refresh_Counter 1                    1          1
   Handle_RREQ'Refresh_Counter 2                    1          1
   Handle_RREQ'Refresh_Counter 3                    1          1
   MANET'Conf1 1                                    1          1
   MANET'Conf2 1                                    1          1
   MANET'GW_Conf 1                                  1          1
   MANET'Macs_1 1                                   1          0
   MANET'Network 1                                  2          0
   MANET'New_Part1 1                                0          0
   MANET'New_Scope1 1                               0          0
   MANET'Req1_Conf 1                                1          1
   Manage_Prefixes'No_Refresh_Rec 1                 0          0
   Manage_Prefixes'Node_Info 1                      1          1
   Manage_Prefixes'Prefix_Ack'ed 1                  1          1
   Proxy_Module'New_Addr 1                          1          1
   Proxy_Module'New_Addr 2                          1          1
   Proxy_Module'Node_Info 1                         1          1
   Proxy_Module'Node_Info 2                         1          1
   Requester'Global_IP 1                            1          1
   Requester'Init_DAD 1                             1          1
   Requester'Init_Global_REQ 1                      1          1
   Requester'Mac_Chosen 1                           1          0
   Requester'Rm 1                                   1          0
   Send_Address_Refresh'Counter 1                   1          1
   Send_Address_Refresh'Counter 2                   1          1
   Send_Address_Refresh'Counter 3                   1          1
   Send_Address_Refresh'Node_Info 1                 1          1
   Send_Address_Refresh'Node_Info 2                 1          1
   Send_Address_Refresh'Node_Info 3                 1          1
   Update_Configuration'Node_Info 1                 1          1

Best Upper Multi-set Bounds
   Create_Address_Response'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
```

```
    (3,1,part1,true),(4,2,part1,true)]

Duplicate_Address_Detection'NS_Sent 1  1`0++1`1++1`2++1`3
Duplicate_Address_Detection'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Duplicate_Address_Detection'Timer 1
 1`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0050:56FF:FEC0:0008",Route=[0],
       Destination="FF02:0:0:0:0:1:FFC0:0008",ReceiverID=0})++
 1`ERROR({msg=""})

Gateway'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Global_IP_Request'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_Address_Refresh'Extracted_Prefix 1        empty
Handle_Address_Refresh'Prefix_Ack'ed 1           1`[]
Handle_Address_Refresh'Prefixes_Timed_Out 1      empty
Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1
 1`["2222:2222:2222","3333:3333:3333","4444:4444:4444",
    "5555:5555:5555","6666:6666:6666","7777:7777:7777",
    "8888:8888:8888","9999:9999:9999"]++
 1`["3333:3333:3333","4444:4444:4444","5555:5555:5555",
    "6666:6666:6666","7777:7777:7777","8888:8888:8888",
    "9999:9999:9999"]

Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1
 1`"1111:1111:1111"++1`"2222:2222:2222"

Handle_Incoming_Packets_GATEWAY'Timer 1          empty
Handle_Incoming_Packets_NODE'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_Incoming_Packets_NODE'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NR'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
```

```
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NS'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Refresh_Counter 1  1`0
Handle_RREQ'Refresh_Counter 2  1`0
Handle_RREQ'Refresh_Counter 3  1`0
MANET'Conf1 1
 1`{Scope=1,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0001",isProxy=true,
    ID=2,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'Conf2 1
 1`{Scope=1,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0002",isProxy=false,
    ID=3,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'GW_Conf 1
 1`{Scope=2,Partition=part1,
    IP="3FFE:2E01:2B:1111:0000:0000:0000:0001",isProxy=false,
    ID=4,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'Macs_1 1      1`"0050:56C0:0008"
MANET'Network 1
 1`NR({SourceScope=0,Partition=part1,Route=[0],
```

```
      Source="FE80:0:0:0:0050:56FF:FEC0:0008",ReceiverID=2})
         ++
 1`NR({SourceScope=0,Partition=part1,Route=[0],
      Source="FE80:0:0:0:0050:56FF:FEC0:0008",ReceiverID=3})
         ++
 1`NRep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[2],
        Source="3FFE:2E01:2B:1111:1111:1111:1111:0001",
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008"})++
 1`NRep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[3],
        Source="3FFE:2E01:2B:1111:1111:1111:1111:0002",
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008"})++
 1`AReq({SourceScope=0,CurrentScope=1,DestinationScope=1,
        Source="FE80:0:0:0:0050:56FF:FEC0:0008",
        Destination="3FFE:2E01:2B:1111:1111:1111:1111:0001",
        Route=[0],Partition=part1})++
 1`AReq({SourceScope=0,CurrentScope=1,DestinationScope=1,
        Source="FE80:0:0:0:0050:56FF:FEC0:0008",
        Destination="3FFE:2E01:2B:1111:1111:1111:1111:0002",
        Route=[0],Partition=part1})++
 1`AReq({SourceScope=0,CurrentScope=2,DestinationScope=2,
        Source="FE80:0:0:0:0050:56FF:FEC0:0008",
        Destination="3FFE:2E01:2B:1111:1111:1111:1111:0002",
        Route=[3,0],Partition=part1})++
 1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0003",
        Route=[0],Lifetime=1000})++
 1`ARep({SourceScope=2,CurrentScope=0,DestinationScope=0,
        Partition=part1,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008",
        GlobalIP="3FFE:2E01:2B:1111:2222:2222:2222:0000",
        Route=[0],Lifetime=1000})++
 1`ARep({SourceScope=2,CurrentScope=1,DestinationScope=0,
        Partition=part1,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008",
        GlobalIP="3FFE:2E01:2B:1111:2222:2222:2222:0000",
        Route=[3,0],Lifetime=1000})

MANET'New_Part1 1   empty
MANET'New_Scope1 1  empty
MANET'Req1_Conf 1
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0003",isProxy=false,
    ID=0,MAC="0050:56C0:0008",
    SolAddr="FF02:0:0:0:0:1:FFC0:0008",NRcount=1,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:2222:2222:2222:0001",isProxy=true,
```

```
    ID=0,MAC="0050:56C0:0008",
    SolAddr="FF02:0:0:0:0:1:FFC0:0008",NRcount=1,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1‘{Scope=0,Partition=part1,IP="::",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="::",NRcount=0,
    ConfigurationStage=SelectMAC,Connected=true}++
 1‘{Scope=0,Partition=part1,IP="::",isProxy=false,ID=0,
    MAC="::",SolAddr="::",ConfigurationStage=Initial,NRcount
        =0,
    Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=LinkLocal,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=DAD,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=Global,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=Global,NRcount=1,Connected=true}

Manage_Prefixes’No_Refresh_Rec 1  empty
Manage_Prefixes’Node_Info 1
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Manage_Prefixes’Prefix_Ack’ed 1   1‘[]
Proxy_Module’New_Addr 1           1‘3
Proxy_Module’New_Addr 2           1‘3++1‘4
Proxy_Module’Node_Info 1
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Proxy_Module’Node_Info 2
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Requester’Global_IP 1
 1‘"3FFE:2E01:2B:1111:1111:1111:1111:0003"++
 1‘"3FFE:2E01:2B:1111:2222:2222:2222:0001"++1‘"::"

Requester’Init_DAD 1              1‘"0050:56C0:0008"++1‘"::"
Requester’Init_Global_REQ 1       1‘"0050:56C0:0008"++1‘"::"
Requester’Mac_Chosen 1            1‘"0050:56C0:0008"
Requester’Rm 1                    1‘()
```

```
   Send_Address_Refresh'Counter 1   1`0
   Send_Address_Refresh'Counter 2   1`0
   Send_Address_Refresh'Counter 3   1`0
   Send_Address_Refresh'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Send_Address_Refresh'Node_Info 2
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Send_Address_Refresh'Node_Info 3
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Update_Configuration'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

Best Lower Multi-set Bounds
   Create_Address_Response'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Duplicate_Address_Detection'NS_Sent 1    empty
   Duplicate_Address_Detection'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Duplicate_Address_Detection'Timer 1      empty
   Gateway'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Global_IP_Request'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Handle_Address_Refresh'Extracted_Prefix 1         empty
   Handle_Address_Refresh'Prefix_Ack'ed 1            1`[]
   Handle_Address_Refresh'Prefixes_Timed_Out 1       empty
   Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1  empty
   Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1
    1`"1111:1111:1111"

   Handle_Incoming_Packets_GATEWAY'Timer 1           empty
   Handle_Incoming_Packets_NODE'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Handle_Incoming_Packets_NODE'Node_Info 2
```

```
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NR'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NS'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Refresh_Counter 1    1`0
Handle_RREQ'Refresh_Counter 2    1`0
Handle_RREQ'Refresh_Counter 3    1`0
MANET'Conf1 1
 1`{Scope=1,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0001",isProxy=true,
    ID=2,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'Conf2 1
 1`{Scope=1,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0002",isProxy=false,
    ID=3,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'GW_Conf 1
```

```
 1`{Scope=2,Partition=part1,
    IP="3FFE:2E01:2B:1111:0000:0000:0000:0001",isProxy=false,
    ID=4,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'Macs_1 1                      empty
MANET'Network 1                     empty
MANET'New_Part1 1                   empty
MANET'New_Scope1 1                  empty
MANET'Req1_Conf 1                   empty
Manage_Prefixes'No_Refresh_Rec 1    empty
Manage_Prefixes'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Manage_Prefixes'Prefix_Ack'ed 1     1`[]
Proxy_Module'New_Addr 1             1`3
Proxy_Module'New_Addr 2             empty
Proxy_Module'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Proxy_Module'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Requester'Global_IP 1               empty
Requester'Init_DAD 1                empty
Requester'Init_Global_REQ 1         empty
Requester'Mac_Chosen 1              empty
Requester'Rm 1                      empty
Send_Address_Refresh'Counter 1      1`0
Send_Address_Refresh'Counter 2      1`0
Send_Address_Refresh'Counter 3      1`0
Send_Address_Refresh'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Send_Address_Refresh'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Send_Address_Refresh'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Update_Configuration'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Home Properties
```

```
------------------------------------------------------------------------

    Home Markings
     None

 Liveness Properties
------------------------------------------------------------------------

  Dead Markings
     [31,34]

  Dead Transition Instances
     Gateway'Forward_Message 1
     Handle_AREQ'Forward_Request 2
     Handle_Address_Refresh'Discard 1
     Handle_Address_Refresh'Get_Refresh 1
     Handle_Address_Refresh'Remove_Prefix 1
     Handle_Incoming_Packets_NODE'Forward_Message 2
     Handle_NA'Discard_NA 1
     Handle_NA'Get_NA 1
     Handle_NREQ'Create_Reply 1
     Handle_NREQ'Discard 1
     Handle_NREQ'Discard 2
     Handle_NREQ'Discard 3
     Handle_NS'Discard_NS 1
     Handle_NS'Generate_NA 1
     Handle_RREQ'Discard 1
     Handle_RREQ'Discard 2
     Handle_RREQ'Discard 3
     Handle_RREQ'Send_Reply 1
     Handle_RREQ'Send_Reply 2
     Handle_RREQ'Send_Reply 3
     Manage_Prefixes'Make_Available 1
     Manage_Prefixes'Refresh_Not_Received 1
     Proxy_Module'Create_Reply 1
     Send_Address_Refresh'Send_Address_Refresh 1
     Send_Address_Refresh'Send_Address_Refresh 2
     Send_Address_Refresh'Send_Address_Refresh 3
     Update_Configuration'Change_Partition 1
     Update_Configuration'Change_Scope 1

  Live Transition Instances
     None


 Fairness Properties
------------------------------------------------------------------------

     No infinite occurrence sequences.
------------------------------------------------------------------------
```

Two requesters - no conflict

```
CPN Tools state space report for:
C:\SPECIALE\2-TwpRequestersNoConflict.cpn
Report generated: Wed Jun 13 19:15:30 2007

 Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  10078
     Arcs:   41724
     Secs:   72
     Status: Full

  Scc Graph
     Nodes:  10078
     Arcs:   41724
     Secs:   1

 Boundedness Properties
------------------------------------------------------------------------


  Best Integer Bounds                                  Upper
      Lower
     Create_Address_Response'Node_Info 1                 1           1
     Duplicate_Address_Detection'NS_Sent 1               1           1
     Duplicate_Address_Detection'NS_Sent 2               1           1
     Duplicate_Address_Detection'Node_Info 1             1           1
     Duplicate_Address_Detection'Node_Info 2             1           1
     Duplicate_Address_Detection'Timer 1                 1           1
     Duplicate_Address_Detection'Timer 2                 1           1
     Gateway'Node_Info 1                                 1           1
     Global_IP_Request'Node_Info 1                       1           1
     Global_IP_Request'Node_Info 2                       1           1
     Handle_Address_Refresh'Extracted_Prefix 1           0           0
     Handle_Address_Refresh'Prefix_Ack'ed 1              1           1
     Handle_Address_Refresh'Prefixes_Timed_Out 1         0           0
     Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1 1              1
     Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1       3           1
     Handle_Incoming_Packets_GATEWAY'Timer 1             0           0
     Handle_Incoming_Packets_NODE'Node_Info 1            1           1
     Handle_Incoming_Packets_NODE'Node_Info 2            1           1
     Handle_NR'Node_Info 1                               1           1
     Handle_NR'Node_Info 2                               1           1
     Handle_NREQ'Node_Info 1                             1           1
     Handle_NREQ'Node_Info 2                             1           1
     Handle_NREQ'Node_Info 3                             1           1
     Handle_NS'Node_Info 1                               1           1
     Handle_NS'Node_Info 2                               1           1
```

```
   Handle_RREQ'Node_Info 1                          1          1
   Handle_RREQ'Node_Info 2                          1          1
   Handle_RREQ'Node_Info 3                          1          1
   Handle_RREQ'Node_Info 4                          1          1
   Handle_RREQ'Refresh_Counter 1                    1          1
   Handle_RREQ'Refresh_Counter 2                    1          1
   Handle_RREQ'Refresh_Counter 3                    1          1
   Handle_RREQ'Refresh_Counter 4                    1          1
   MANET'Conf1 1                                    1          1
   MANET'Conf2 1                                    1          1
   MANET'GW_Conf 1                                  1          1
   MANET'Macs_1 1                                   1          0
   MANET'Macs_2 1                                   1          0
   MANET'Network 1                                  8          0
   MANET'New_Part1 1                                0          0
   MANET'New_Part2 1                                0          0
   MANET'New_Scope1 1                               0          0
   MANET'New_Scope2 1                               0          0
   MANET'Req1_Conf 1                                1          1
   MANET'Req2_Conf 1                                1          1
   Manage_Prefixes'No_Refresh_Rec 1                 0          0
   Manage_Prefixes'Node_Info 1                      1          1
   Manage_Prefixes'Prefix_Ack'ed 1                  1          1
   Proxy_Module'New_Addr 1                          1          1
   Proxy_Module'New_Addr 2                          1          1
   Proxy_Module'Node_Info 1                         1          1
   Proxy_Module'Node_Info 2                         1          1
   Requester'Global_IP 1                            1          1
   Requester'Global_IP 2                            1          1
   Requester'Init_DAD 1                             1          1
   Requester'Init_DAD 2                             1          1
   Requester'Init_Global_REQ 1                      1          1
   Requester'Init_Global_REQ 2                      1          1
   Requester'Mac_Chosen 1                           1          0
   Requester'Mac_Chosen 2                           1          0
   Requester'Rm 1                                   1          0
   Requester'Rm 2                                   1          0
   Send_Address_Refresh'Counter 1                   1          1
   Send_Address_Refresh'Counter 2                   1          1
   Send_Address_Refresh'Counter 3                   1          1
   Send_Address_Refresh'Counter 4                   1          1
   Send_Address_Refresh'Node_Info 1                 1          1
   Send_Address_Refresh'Node_Info 2                 1          1
   Send_Address_Refresh'Node_Info 3                 1          1
   Send_Address_Refresh'Node_Info 4                 1          1
   Update_Configuration'Node_Info 1                 1          1
   Update_Configuration'Node_Info 2                 1          1

Best Upper Multi-set Bounds
   Create_Address_Response'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
```

```
    (3,1,part1,true),(4,2,part1,true)]

Duplicate_Address_Detection'NS_Sent 1 1`0++1`1++1`2++1`3
Duplicate_Address_Detection'NS_Sent 2 1`0++1`1++1`2++1`3
Duplicate_Address_Detection'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Duplicate_Address_Detection'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Duplicate_Address_Detection'Timer 1
 1`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0050:56FF:FEC0:0001",Route=[0],
       Destination="FF02:0:0:0:0:1:FFC0:0001",ReceiverID=0})++
 1`ERROR({msg=""})

Duplicate_Address_Detection'Timer 2
 1`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0005:9AFF:FE3C:7800",Route=[1],
       Destination="FF02:0:0:0:0:1:FF3C:7800",ReceiverID=1})++
 1`ERROR({msg=""})

Gateway'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Global_IP_Request'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Global_IP_Request'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_Address_Refresh'Extracted_Prefix 1        empty
Handle_Address_Refresh'Prefix_Ack'ed 1           1`[]
Handle_Address_Refresh'Prefixes_Timed_Out 1      empty
Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1
 1`["2222:2222:2222","3333:3333:3333","4444:4444:4444",
    "5555:5555:5555","6666:6666:6666","7777:7777:7777",
    "8888:8888:8888","9999:9999:9999"]++
 1`["3333:3333:3333","4444:4444:4444","5555:5555:5555",
    "6666:6666:6666","7777:7777:7777","8888:8888:8888",
    "9999:9999:9999"]++
 1`["4444:4444:4444","5555:5555:5555","6666:6666:6666",
    "7777:7777:7777","8888:8888:8888","9999:9999:9999"]
```

```
Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1
 1`"1111:1111:1111"++1`"2222:2222:2222"++1`"3333:3333:3333"

Handle_Incoming_Packets_GATEWAY'Timer 1   empty
Handle_Incoming_Packets_NODE'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_Incoming_Packets_NODE'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NR'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NR'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NS'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NS'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]
```

```
Handle_RREQ'Node_Info 4
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Refresh_Counter 1  1`0
Handle_RREQ'Refresh_Counter 2  1`0
Handle_RREQ'Refresh_Counter 3  1`0
Handle_RREQ'Refresh_Counter 4  1`0
MANET'Conf1 1
 1`{Scope=1,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0001",isProxy=true,
    ID=2,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'Conf2 1
 1`{Scope=1,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0002",isProxy=false,
    ID=3,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'GW_Conf 1
 1`{Scope=2,Partition=part1,
    IP="3FFE:2E01:2B:1111:0000:0000:0000:0001",isProxy=false,
    ID=4,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'Macs_1 1      1`"0050:56C0:0001"
MANET'Macs_2 1      1`"0005:9A3C:7800"
MANET'Network 1
 2`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0005:9AFF:FE3C:7800",Route=[1],
       Destination="FF02:0:0:0:0:1:FF3C:7800",ReceiverID=0})++
 2`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0050:56FF:FEC0:0001",Route=[0],
       Destination="FF02:0:0:0:0:1:FFC0:0001",ReceiverID=1})++
 1`NR({SourceScope=0,Partition=part1,Route=[1],
       Source="FE80:0:0:0:0005:9AFF:FE3C:7800",ReceiverID=2})
          ++
 1`NR({SourceScope=0,Partition=part1,Route=[1],
       Source="FE80:0:0:0:0005:9AFF:FE3C:7800",ReceiverID=3})
          ++
 1`NR({SourceScope=0,Partition=part1,Route=[0],
       Source="FE80:0:0:0:0050:56FF:FEC0:0001",ReceiverID=2})
          ++
 1`NR({SourceScope=0,Partition=part1,Route=[0],
       Source="FE80:0:0:0:0050:56FF:FEC0:0001",ReceiverID=3})
          ++
 1`NRep({SourceScope=1,CurrentScope=0,DestinationScope=0,
```

```
        Partition=part1,Route=[2],
        Source="3FFE:2E01:2B:1111:1111:1111:1111:0001",
        Destination="FE80:0:0:0:0005:9AFF:FE3C:7800"})++
1`NRep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[2],
        Source="3FFE:2E01:2B:1111:1111:1111:1111:0001",
        Destination="FE80:0:0:0:0050:56FF:FEC0:0001"})++
1`NRep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[3],
        Source="3FFE:2E01:2B:1111:1111:1111:1111:0002",
        Destination="FE80:0:0:0:0005:9AFF:FE3C:7800"})++
1`NRep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[3],
        Source="3FFE:2E01:2B:1111:1111:1111:1111:0002",
        Destination="FE80:0:0:0:0050:56FF:FEC0:0001"})++
1`AReq({SourceScope=0,CurrentScope=1,DestinationScope=1,
        Source="FE80:0:0:0:0005:9AFF:FE3C:7800",
        Destination="3FFE:2E01:2B:1111:1111:1111:1111:0001",
        Route=[1],Partition=part1})++
1`AReq({SourceScope=0,CurrentScope=1,DestinationScope=1,
        Source="FE80:0:0:0:0005:9AFF:FE3C:7800",
        Destination="3FFE:2E01:2B:1111:1111:1111:1111:0002",
        Route=[1],Partition=part1})++
1`AReq({SourceScope=0,CurrentScope=1,DestinationScope=1,
        Source="FE80:0:0:0:0050:56FF:FEC0:0001",
        Destination="3FFE:2E01:2B:1111:1111:1111:1111:0001",
        Route=[0],Partition=part1})++
1`AReq({SourceScope=0,CurrentScope=1,DestinationScope=1,
        Source="FE80:0:0:0:0050:56FF:FEC0:0001",
        Destination="3FFE:2E01:2B:1111:1111:1111:1111:0002",
        Route=[0],Partition=part1})++
1`AReq({SourceScope=0,CurrentScope=2,DestinationScope=2,
        Source="FE80:0:0:0:0005:9AFF:FE3C:7800",
        Destination="3FFE:2E01:2B:1111:1111:1111:1111:0002",
        Route=[3,1],Partition=part1})++
1`AReq({SourceScope=0,CurrentScope=2,DestinationScope=2,
        Source="FE80:0:0:0:0050:56FF:FEC0:0001",
        Destination="3FFE:2E01:2B:1111:1111:1111:1111:0002",
        Route=[3,0],Partition=part1})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[1],Lifetime=1000,
        Destination="FE80:0:0:0:0005:9AFF:FE3C:7800",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0003"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[1],Lifetime=1000,
        Destination="FE80:0:0:0:0005:9AFF:FE3C:7800",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0004"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[0],Lifetime=1000,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0001",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0003"})++
```

```
 1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
         Partition=part1,Route=[0],Lifetime=1000,
         Destination="FE80:0:0:0:0050:56FF:FEC0:0001",
         GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0004"})++
 1`ARep({SourceScope=2,CurrentScope=0,DestinationScope=0,
         Partition=part1,Route=[1],Lifetime=1000,
         Destination="FE80:0:0:0:0005:9AFF:FE3C:7800",
         GlobalIP="3FFE:2E01:2B:1111:2222:2222:2222:0000"})++
 1`ARep({SourceScope=2,CurrentScope=0,DestinationScope=0,
         Partition=part1,Route=[1],Lifetime=1000,
         Destination="FE80:0:0:0:0005:9AFF:FE3C:7800",
         GlobalIP="3FFE:2E01:2B:1111:3333:3333:3333:0000"})++
 1`ARep({SourceScope=2,CurrentScope=0,DestinationScope=0,
         Partition=part1,Route=[0],Lifetime=1000,
         Destination="FE80:0:0:0:0050:56FF:FEC0:0001",
         GlobalIP="3FFE:2E01:2B:1111:2222:2222:2222:0000"})++
 1`ARep({SourceScope=2,CurrentScope=0,DestinationScope=0,
         Partition=part1,Route=[0],Lifetime=1000,
         Destination="FE80:0:0:0:0050:56FF:FEC0:0001",
         GlobalIP="3FFE:2E01:2B:1111:3333:3333:3333:0000"})++
 1`ARep({SourceScope=2,CurrentScope=1,DestinationScope=0,
         Partition=part1,Route=[3,1],Lifetime=1000,
         Destination="FE80:0:0:0:0005:9AFF:FE3C:7800",
         GlobalIP="3FFE:2E01:2B:1111:2222:2222:2222:0000"})++
 1`ARep({SourceScope=2,CurrentScope=1,DestinationScope=0,
         Partition=part1,Route=[3,1],Lifetime=1000,
         Destination="FE80:0:0:0:0005:9AFF:FE3C:7800",
         GlobalIP="3FFE:2E01:2B:1111:3333:3333:3333:0000"})++
 1`ARep({SourceScope=2,CurrentScope=1,DestinationScope=0,
         Partition=part1,Route=[3,0],Lifetime=1000,
         Destination="FE80:0:0:0:0050:56FF:FEC0:0001",
         GlobalIP="3FFE:2E01:2B:1111:2222:2222:2222:0000"})++
 1`ARep({SourceScope=2,CurrentScope=1,DestinationScope=0,
         Partition=part1,Route=[3,0],Lifetime=1000,
         Destination="FE80:0:0:0:0050:56FF:FEC0:0001",
         GlobalIP="3FFE:2E01:2B:1111:3333:3333:3333:0000"})

MANET'New_Part1 1   empty
MANET'New_Part2 1   empty
MANET'New_Scope1 1  empty
MANET'New_Scope2 1  empty
MANET'Req1_Conf 1
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0003",isProxy=false,
    ID=0,MAC="0050:56C0:0001",
    SolAddr="FF02:0:0:0:0:1:FFC0:0001",NRcount=1,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0004",isProxy=false,
    ID=0,MAC="0050:56C0:0001",
    SolAddr="FF02:0:0:0:0:1:FFC0:0001",NRcount=1,
```

```
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:2222:2222:2222:0001",isProxy=true,
    ID=0,MAC="0050:56C0:0001",
    SolAddr="FF02:0:0:0:0:1:FFC0:0001",NRcount=1,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:3333:3333:3333:0001",isProxy=true,
    ID=0,MAC="0050:56C0:0001",
    SolAddr="FF02:0:0:0:0:1:FFC0:0001",NRcount=1,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,IP="::",isProxy=false,ID=0,
    MAC="0050:56C0:0001",SolAddr="::",
    ConfigurationStage=SelectMAC,NRcount=0,Connected=true}++
 1`{Scope=0,Partition=part1,IP="::",isProxy=false,ID=0,
    MAC="::",SolAddr="::",ConfigurationStage=Initial,
    NRcount=0,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0001",isProxy=false,ID=0,
    MAC="0050:56C0:0001",SolAddr="FF02:0:0:0:0:1:FFC0:0001",
    ConfigurationStage=LinkLocal,NRcount=0,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0001",isProxy=false,ID=0,
    MAC="0050:56C0:0001",SolAddr="FF02:0:0:0:0:1:FFC0:0001",
    ConfigurationStage=DAD,NRcount=0,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0001",isProxy=false,ID=0,
    MAC="0050:56C0:0001",SolAddr="FF02:0:0:0:0:1:FFC0:0001",
    ConfigurationStage=Global,NRcount=0,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0001",isProxy=false,ID=0,
    MAC="0050:56C0:0001",SolAddr="FF02:0:0:0:0:1:FFC0:0001",
    ConfigurationStage=Global,NRcount=1,Connected=true}

MANET'Req2_Conf 1
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0003",isProxy=false,
    ID=1,MAC="0005:9A3C:7800",
    SolAddr="FF02:0:0:0:0:1:FF3C:7800",NRcount=1,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0004",isProxy=false,
    ID=1,MAC="0005:9A3C:7800",
    SolAddr="FF02:0:0:0:0:1:FF3C:7800",NRcount=1,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:2222:2222:2222:0001",isProxy=true,
    ID=1,MAC="0005:9A3C:7800",
    SolAddr="FF02:0:0:0:0:1:FF3C:7800",NRcount=1,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
```

```
    IP="3FFE:2E01:2B:1111:3333:3333:3333:0001",isProxy=true,
    ID=1,MAC="0005:9A3C:7800",
    SolAddr="FF02:0:0:0:0:1:FF3C:7800",NRcount=1,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1‘{Scope=0,Partition=part1,IP="::",isProxy=false,ID=1,
    MAC="0005:9A3C:7800",SolAddr="::",
    ConfigurationStage=SelectMAC,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,IP="::",isProxy=false,ID=1,
    MAC="::",SolAddr="::",ConfigurationStage=Initial,NRcount
        =0,
    Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0005:9AFF:FE3C:7800",isProxy=false,ID=1,
    MAC="0005:9A3C:7800",SolAddr="FF02:0:0:0:0:1:FF3C:7800",
    ConfigurationStage=LinkLocal,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0005:9AFF:FE3C:7800",isProxy=false,ID=1,
    MAC="0005:9A3C:7800",SolAddr="FF02:0:0:0:0:1:FF3C:7800",
    ConfigurationStage=DAD,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0005:9AFF:FE3C:7800",isProxy=false,ID=1,
    MAC="0005:9A3C:7800",SolAddr="FF02:0:0:0:0:1:FF3C:7800",
    ConfigurationStage=Global,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0005:9AFF:FE3C:7800",isProxy=false,ID=1,
    MAC="0005:9A3C:7800",SolAddr="FF02:0:0:0:0:1:FF3C:7800",
    ConfigurationStage=Global,NRcount=1,Connected=true}

Manage_Prefixes’No_Refresh_Rec 1  empty
Manage_Prefixes’Node_Info 1
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Manage_Prefixes’Prefix_Ack’ed 1   1‘[]
Proxy_Module’New_Addr 1           1‘3
Proxy_Module’New_Addr 2           1‘3++1‘4++1‘5
Proxy_Module’Node_Info 1
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Proxy_Module’Node_Info 2
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Requester’Global_IP 1
 1‘"3FFE:2E01:2B:1111:1111:1111:1111:0003"++
 1‘"3FFE:2E01:2B:1111:1111:1111:1111:0004"++
 1‘"3FFE:2E01:2B:1111:2222:2222:2222:0001"++
 1‘"3FFE:2E01:2B:1111:3333:3333:3333:0001"++1‘"::"

Requester’Global_IP 2
```

```
     1`"3FFE:2E01:2B:1111:1111:1111:1111:0003"++
     1`"3FFE:2E01:2B:1111:1111:1111:1111:0004"++
     1`"3FFE:2E01:2B:1111:2222:2222:2222:0001"++
     1`"3FFE:2E01:2B:1111:3333:3333:3333:0001"++1`"::"

   Requester'Init_DAD 1            1`"0050:56C0:0001"++1`"::"
   Requester'Init_DAD 2            1`"0005:9A3C:7800"++1`"::"
   Requester'Init_Global_REQ 1     1`"0050:56C0:0001"++1`"::"
   Requester'Init_Global_REQ 2     1`"0005:9A3C:7800"++1`"::"
   Requester'Mac_Chosen 1          1`"0050:56C0:0001"
   Requester'Mac_Chosen 2          1`"0005:9A3C:7800"
   Requester'Rm 1                  1`()
   Requester'Rm 2                  1`()
   Send_Address_Refresh'Counter 1 1`0
   Send_Address_Refresh'Counter 2 1`0
   Send_Address_Refresh'Counter 3 1`0
   Send_Address_Refresh'Counter 4 1`0
   Send_Address_Refresh'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Send_Address_Refresh'Node_Info 2
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Send_Address_Refresh'Node_Info 3
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Send_Address_Refresh'Node_Info 4
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Update_Configuration'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Update_Configuration'Node_Info 2
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

Best Lower Multi-set Bounds
   Create_Address_Response'Node_Info 1
     1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
        (3,1,part1,true),(4,2,part1,true)]

   Duplicate_Address_Detection'NS_Sent 1          empty
   Duplicate_Address_Detection'NS_Sent 2          empty
   Duplicate_Address_Detection'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]
```

```
Duplicate_Address_Detection'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Duplicate_Address_Detection'Timer 1              empty
Duplicate_Address_Detection'Timer 2              empty
Gateway'Node_Info 1 1`[(0,0,part1,true),(1,0,part1,true),(2,1,
   part1,true),(3,1,part1,true),(4,2,part1,true)]
Global_IP_Request'Node_Info 1
                    1`[(0,0,part1,true),(1,0,part1,true),(2,1,
                        part1,true),(3,1,part1,true),(4,2,part1
                        ,true)]
Global_IP_Request'Node_Info 2
                    1`[(0,0,part1,true),(1,0,part1,true),(2,1,
                        part1,true),(3,1,part1,true),(4,2,part1
                        ,true)]
Handle_Address_Refresh'Extracted_Prefix 1        empty
Handle_Address_Refresh'Prefix_Ack'ed 1           1`[]
Handle_Address_Refresh'Prefixes_Timed_Out 1      empty
Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1 empty
Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1
 1`"1111:1111:1111"

Handle_Incoming_Packets_GATEWAY'Timer 1          empty
Handle_Incoming_Packets_NODE'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_Incoming_Packets_NODE'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NR'Node_Info 1
  1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
     (3,1,part1,true),(4,2,part1,true)]

Handle_NR'Node_Info 2
  1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
     (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
```

```
    (3,1,part1,true),(4,2,part1,true)]

Handle_NS'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NS'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 4
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Refresh_Counter 1   1`0
Handle_RREQ'Refresh_Counter 2   1`0
Handle_RREQ'Refresh_Counter 3   1`0
Handle_RREQ'Refresh_Counter 4   1`0
MANET'Conf1 1
 1`{Scope=1,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0001",isProxy=true,
    ID=2,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'Conf2 1
 1`{Scope=1,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0002",isProxy=false,
    ID=3,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'GW_Conf 1
 1`{Scope=2,Partition=part1,
    IP="3FFE:2E01:2B:1111:0000:0000:0000:0001",isProxy=false,
    ID=4,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'Macs_1 1                  empty
MANET'Macs_2 1                  empty
MANET'Network 1                 empty
```

```
MANET'New_Part1 1                 empty
MANET'New_Part2 1                 empty
MANET'New_Scope1 1                empty
MANET'New_Scope2 1                empty
MANET'Req1_Conf 1                 empty
MANET'Req2_Conf 1                 empty
Manage_Prefixes'No_Refresh_Rec 1  empty
Manage_Prefixes'Node_Info 1
  1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
     (3,1,part1,true),(4,2,part1,true)]

Manage_Prefixes'Prefix_Ack'ed 1   1`[]
Proxy_Module'New_Addr 1           1`3
Proxy_Module'New_Addr 2           empty
Proxy_Module'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Proxy_Module'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Requester'Global_IP 1             empty
Requester'Global_IP 2             empty
Requester'Init_DAD 1              empty
Requester'Init_DAD 2              empty
Requester'Init_Global_REQ 1       empty
Requester'Init_Global_REQ 2       empty
Requester'Mac_Chosen 1            empty
Requester'Mac_Chosen 2            empty
Requester'Rm 1                    empty
Requester'Rm 2                    empty
Send_Address_Refresh'Counter 1    1`0
Send_Address_Refresh'Counter 2    1`0
Send_Address_Refresh'Counter 3    1`0
Send_Address_Refresh'Counter 4    1`0
Send_Address_Refresh'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Send_Address_Refresh'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Send_Address_Refresh'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Send_Address_Refresh'Node_Info 4
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]
```

```
    Update_Configuration'Node_Info 1
     1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
        (3,1,part1,true),(4,2,part1,true)]

    Update_Configuration'Node_Info 2
     1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
        (3,1,part1,true),(4,2,part1,true)]

 Home Properties
------------------------------------------------------------------------

  Home Markings
     None

 Liveness Properties
------------------------------------------------------------------------

  Dead Markings
     6 [9797,9789,10078,10077,10052,...]

  Dead Transition Instances
     Gateway'Forward_Message 1
     Handle_AREQ'Forward_Request 2
     Handle_Address_Refresh'Discard 1
     Handle_Address_Refresh'Get_Refresh 1
     Handle_Address_Refresh'Remove_Prefix 1
     Handle_Incoming_Packets_NODE'Forward_Message 2
     Handle_NA'Discard_NA 1
     Handle_NA'Discard_NA 2
     Handle_NA'Get_NA 1
     Handle_NA'Get_NA 2
     Handle_NREQ'Create_Reply 1
     Handle_NREQ'Discard 1
     Handle_NREQ'Discard 2
     Handle_NREQ'Discard 3
     Handle_NS'Generate_NA 1
     Handle_NS'Generate_NA 2
     Handle_RREQ'Discard 1
     Handle_RREQ'Discard 2
     Handle_RREQ'Discard 3
     Handle_RREQ'Discard 4
     Handle_RREQ'Send_Reply 1
     Handle_RREQ'Send_Reply 2
     Handle_RREQ'Send_Reply 3
     Handle_RREQ'Send_Reply 4
     Manage_Prefixes'Make_Available 1
     Manage_Prefixes'Refresh_Not_Received 1
     Proxy_Module'Create_Reply 1
     Send_Address_Refresh'Send_Address_Refresh 1
     Send_Address_Refresh'Send_Address_Refresh 2
```

```
    Send_Address_Refresh'Send_Address_Refresh 3
    Send_Address_Refresh'Send_Address_Refresh 4
    Update_Configuration'Change_Partition 1
    Update_Configuration'Change_Partition 2
    Update_Configuration'Change_Scope 1
    Update_Configuration'Change_Scope 2

  Live Transition Instances
    None

 Fairness Properties
------------------------------------------------------------------------

    No infinite occurrence sequences.
```

Two requesters - conflict

```
CPN Tools state space report for:
C:\SPECIALE\3-TwoRequestesConflict.cpn
Report generated: Thu Jun 14 19:44:46 2007

 Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  142317
     Arcs:   551948
     Secs:   17550
     Status: Full

  Scc Graph
     Nodes:  142317
     Arcs:   551948
     Secs:   28

 Boundedness Properties
------------------------------------------------------------------------


  Best Integer Bounds                                  Upper
      Lower
     Create_Address_Response'Node_Info 1                 1          1
     Duplicate_Address_Detection'NS_Sent 1               1          1
     Duplicate_Address_Detection'NS_Sent 2               1          1
     Duplicate_Address_Detection'Node_Info 1             1          1
     Duplicate_Address_Detection'Node_Info 2             1          1
     Duplicate_Address_Detection'Timer 1                 1          1
     Duplicate_Address_Detection'Timer 2                 1          1
     Gateway'Node_Info 1                                 1          1
     Global_IP_Request'Node_Info 1                       1          1
     Global_IP_Request'Node_Info 2                       1          1
     Handle_Address_Refresh'Extracted_Prefix 1           0          0
     Handle_Address_Refresh'Prefix_Ack'ed 1              1          1
     Handle_Address_Refresh'Prefixes_Timed_Out 1         0          0
     Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1 1  1          1
     Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1       1          1
     Handle_Incoming_Packets_GATEWAY'Timer 1             0          0
     Handle_Incoming_Packets_NODE'Node_Info 1            1          1
     Handle_NR'Node_Info 1                               1          1
     Handle_NR'Node_Info 2                               1          1
     Handle_NREQ'New_Addr 1                              1          1
     Handle_NREQ'New_Addr 2                              1          1
     Handle_NREQ'Node_Info 1                             1          1
     Handle_NREQ'Node_Info 2                             1          1
     Handle_NS'Node_Info 1                               1          1
     Handle_NS'Node_Info 2                               1          1
```

```
   Handle_RREQ'Node_Info 1                          1          1
   Handle_RREQ'Node_Info 2                          1          1
   Handle_RREQ'Node_Info 3                          1          1
   Handle_RREQ'Refresh_Counter 1                    1          1
   Handle_RREQ'Refresh_Counter 2                    1          1
   Handle_RREQ'Refresh_Counter 3                    1          1
   MANET'Conf1 1                                    1          1
   MANET'GW_Conf 1                                  1          1
   MANET'Macs_1 1                                   1          1
   MANET'Macs_2 1                                   1          1
   MANET'Network 1                                  16         0
   MANET'New_Part1 1                                0          0
   MANET'New_Part2 1                                0          0
   MANET'New_Scope1 1                               0          0
   MANET'New_Scope2 1                               0          0
   MANET'Req1_Conf 1                                1          1
   MANET'Req2_Conf 1                                1          1
   Manage_Prefixes'No_Refresh_Rec 1                 0          0
   Manage_Prefixes'Node_Info 1                      1          1
   Manage_Prefixes'Prefix_Ack'ed 1                  1          1
   Proxy_Module'New_Addr 1                          1          1
   Proxy_Module'Node_Info 1                         1          1
   Requester'Global_IP 1                            1          1
   Requester'Global_IP 2                            1          1
   Requester'Init_DAD 1                             1          1
   Requester'Init_DAD 2                             1          1
   Requester'Init_Global_REQ 1                      1          1
   Requester'Init_Global_REQ 2                      1          1
   Requester'Mac_Chosen 1                           1          0
   Requester'Mac_Chosen 2                           1          0
   Requester'Rm 1                                   1          0
   Requester'Rm 2                                   1          0
   Send_Address_Refresh'Counter 1                   1          1
   Send_Address_Refresh'Counter 2                   1          1
   Send_Address_Refresh'Counter 3                   1          1
   Send_Address_Refresh'Node_Info 1                 1          1
   Send_Address_Refresh'Node_Info 2                 1          1
   Send_Address_Refresh'Node_Info 3                 1          1
   Update_Configuration'Node_Info 1                 1          1
   Update_Configuration'Node_Info 2                 1          1

Best Upper Multi-set Bounds
   Create_Address_Response'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Duplicate_Address_Detection'NS_Sent 1    1`0++1`1++1`2
   Duplicate_Address_Detection'NS_Sent 2    1`0++1`1++1`2
   Duplicate_Address_Detection'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]
```

```
Duplicate_Address_Detection'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Duplicate_Address_Detection'Timer 1
 1`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0050:56FF:FEC0:0001",Route=[0],
       Destination="FF02:0:0:0:0:1:FFC0:0001",ReceiverID=0})++
 1`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0050:56FF:FEC0:0008",Route=[0],
       Destination="FF02:0:0:0:0:1:FFC0:0008",ReceiverID=0})++
 1`ERROR({msg=""})

Duplicate_Address_Detection'Timer 2
 1`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0005:9AFF:FE3C:7800",Route=[1],
       Destination="FF02:0:0:0:0:1:FF3C:7800",ReceiverID=1})++
 1`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0050:56FF:FEC0:0008",Route=[1],
       Destination="FF02:0:0:0:0:1:FFC0:0008",ReceiverID=1})++
 1`ERROR({msg=""})

Gateway'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Global_IP_Request'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Global_IP_Request'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_Address_Refresh'Extracted_Prefix 1    empty
Handle_Address_Refresh'Prefix_Ack'ed 1       1`[]
Handle_Address_Refresh'Prefixes_Timed_Out 1  empty
Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1
 1`["2222:2222:2222","3333:3333:3333","4444:4444:4444",
    "5555:5555:5555","6666:6666:6666","7777:7777:7777",
    "8888:8888:8888","9999:9999:9999"]

Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1
 1`"1111:1111:1111"

Handle_Incoming_Packets_GATEWAY'Timer 1      empty
```

```
Handle_Incoming_Packets_NODE'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NR'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NR'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'New_Addr 1  1`3
Handle_NREQ'New_Addr 2  1`3++1`4++1`5++1`6++1`7
Handle_NREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NS'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NS'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Refresh_Counter 1  1`0
Handle_RREQ'Refresh_Counter 2  1`0
Handle_RREQ'Refresh_Counter 3  1`0
MANET'Conf1 1
 1`{Scope=1,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0001",isProxy=true,
    ID=2,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}
```

```
MANET'GW_Conf 1
 1`{Scope=2,Partition=part1,
    IP="3FFE:2E01:2B:1111:0000:0000:0000:0001",isProxy=false,
    ID=4,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'Macs_1 1
 1`[]++1`["0050:56C0:0001"]++1`["0050:56C0:0008","0050:56C0
     :0001"]

MANET'Macs_2 1
 1`[]++1`["0005:9A3C:7800"]++1`["0050:56C0:0008","0005:9A3C
     :7800"]

MANET'Network 1
 2`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0005:9AFF:FE3C:7800",Route=[1],
       Destination="FF02:0:0:0:0:1:FF3C:7800",ReceiverID=0})++
 2`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0050:56FF:FEC0:0001",Route=[0],
       Destination="FF02:0:0:0:0:1:FFC0:0001",ReceiverID=1})++
 2`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0050:56FF:FEC0:0008",Route=[0],
       Destination="FF02:0:0:0:0:1:FFC0:0008",ReceiverID=1})++
 2`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0050:56FF:FEC0:0008",Route=[1],
       Destination="FF02:0:0:0:0:1:FFC0:0008",ReceiverID=0})++
 1`NA({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,
       Source="3FFE:2E01:2B:1111:1111:1111:1111:0003",
       Target="FE80:0:0:0:0050:56FF:FEC0:0008",
       Destination="::",Route=[0],ReceiverID=1})++
 1`NA({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,
       Source="3FFE:2E01:2B:1111:1111:1111:1111:0003",
       Target="FE80:0:0:0:0050:56FF:FEC0:0008",
       Destination="::",Route=[1],ReceiverID=0})++
 1`NA({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,
       Source="3FFE:2E01:2B:1111:1111:1111:1111:0004",
       Target="FE80:0:0:0:0050:56FF:FEC0:0008",
       Destination="::",Route=[0],ReceiverID=1})++
 1`NA({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,
       Source="3FFE:2E01:2B:1111:1111:1111:1111:0004",
       Target="FE80:0:0:0:0050:56FF:FEC0:0008",
       Destination="::",Route=[1],ReceiverID=0})++
```

```
1`NA({SourceScope=0,CurrentScope=0,DestinationScope=0,
      Partition=part1,
      Source="3FFE:2E01:2B:1111:1111:1111:1111:0005",
      Target="FE80:0:0:0:0050:56FF:FEC0:0008",
      Destination="::",Route=[0],ReceiverID=1})++
1`NA({SourceScope=0,CurrentScope=0,DestinationScope=0,
      Partition=part1,
      Source="3FFE:2E01:2B:1111:1111:1111:1111:0005",
      Target="FE80:0:0:0:0050:56FF:FEC0:0008",
      Destination="::",Route=[1],ReceiverID=0})++
1`NA({SourceScope=0,CurrentScope=0,DestinationScope=0,
      Partition=part1,Source="FE80:0:0:0:0050:56FF:FEC0:0008"
          ,
      Target="FE80:0:0:0:0050:56FF:FEC0:0008",
      Destination="::",Route=[0],ReceiverID=1})++
1`NA({SourceScope=0,CurrentScope=0,DestinationScope=0,
      Partition=part1,Source="FE80:0:0:0:0050:56FF:FEC0:0008"
          ,
      Target="FE80:0:0:0:0050:56FF:FEC0:0008",
      Destination="::",Route=[1],ReceiverID=0})++
1`NR({SourceScope=0,Partition=part1,Route=[1],
      Source="FE80:0:0:0:0005:9AFF:FE3C:7800",ReceiverID=2})
          ++
1`NR({SourceScope=0,Partition=part1,Route=[1],
      Source="FE80:0:0:0:0005:9AFF:FE3C:7800",ReceiverID=3})
          ++
1`NR({SourceScope=0,Partition=part1,Route=[0],
      Source="FE80:0:0:0:0050:56FF:FEC0:0001",ReceiverID=2})
          ++
1`NR({SourceScope=0,Partition=part1,Route=[0],
      Source="FE80:0:0:0:0050:56FF:FEC0:0001",ReceiverID=3})
          ++
1`NR({SourceScope=0,Partition=part1,Route=[0],
      Source="FE80:0:0:0:0050:56FF:FEC0:0008",ReceiverID=2})
          ++
1`NR({SourceScope=0,Partition=part1,Route=[0],
      Source="FE80:0:0:0:0050:56FF:FEC0:0008",ReceiverID=3})
          ++
1`NR({SourceScope=0,Partition=part1,Route=[1],
      Source="FE80:0:0:0:0050:56FF:FEC0:0008",ReceiverID=2})
          ++
1`NR({SourceScope=0,Partition=part1,Route=[1],
      Source="FE80:0:0:0:0050:56FF:FEC0:0008",ReceiverID=3})
          ++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[1],Lifetime=1000,
        Destination="FE80:0:0:0:0005:9AFF:FE3C:7800",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0003"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[1],Lifetime=1000,
        Destination="FE80:0:0:0:0005:9AFF:FE3C:7800",
```

```
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0004"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[1],Lifetime=1000,
        Destination="FE80:0:0:0:0005:9AFF:FE3C:7800",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0005"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[1],Lifetime=1000,
        Destination="FE80:0:0:0:0005:9AFF:FE3C:7800",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0006"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[0],Lifetime=1000,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0001",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0003"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[0],Lifetime=1000,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0001",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0004"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[0],Lifetime=1000,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0001",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0005"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[0],Lifetime=1000,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0001",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0006"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[0],Lifetime=1000,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0003"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[1],Lifetime=1000,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0003"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[0],Lifetime=1000,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0004"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[1],Lifetime=1000,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0004"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[0],Lifetime=1000,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0005"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[1],Lifetime=1000,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0005"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[0],Lifetime=1000,
```

```
         Destination="FE80:0:0:0:0050:56FF:FEC0:0008",
         GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0006"})++
 1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
         Partition=part1,Route=[1],Lifetime=1000,
         Destination="FE80:0:0:0:0050:56FF:FEC0:0008",
         GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0006"})

MANET'New_Part1 1   empty
MANET'New_Part2 1   empty
MANET'New_Scope1 1  empty
MANET'New_Scope2 1  empty
MANET'Req1_Conf 1
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0003",isProxy=false,
    ID=0,MAC="0050:56C0:0001",
    SolAddr="FF02:0:0:0:0:1:FFC0:0001",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0003",isProxy=false,
    ID=0,MAC="0050:56C0:0008",
    SolAddr="FF02:0:0:0:0:1:FFC0:0008",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0004",isProxy=false,
    ID=0,MAC="0050:56C0:0001",
    SolAddr="FF02:0:0:0:0:1:FFC0:0001",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0004",isProxy=false,
    ID=0,MAC="0050:56C0:0008",
    SolAddr="FF02:0:0:0:0:1:FFC0:0008",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0005",isProxy=false,
    ID=0,MAC="0050:56C0:0001",
    SolAddr="FF02:0:0:0:0:1:FFC0:0001",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0005",isProxy=false,
    ID=0,MAC="0050:56C0:0008",
    SolAddr="FF02:0:0:0:0:1:FFC0:0008",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0006",isProxy=false,
    ID=0,MAC="0050:56C0:0001",
    SolAddr="FF02:0:0:0:0:1:FFC0:0001",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,IP="::",isProxy=false,ID=0,
    MAC="0050:56C0:0001",SolAddr="::",
    ConfigurationStage=SelectMAC,NRcount=0,Connected=true}++
 1`{Scope=0,Partition=part1,IP="::",isProxy=false,ID=0,
```

```
    MAC="0050:56C0:0008",SolAddr="::",
    ConfigurationStage=SelectMAC,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,IP="::",isProxy=false,ID=0,
    MAC="::",SolAddr="::",ConfigurationStage=Initial,
    NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0001",isProxy=false,ID=0,
    MAC="0050:56C0:0001",SolAddr="FF02:0:0:0:0:1:FFC0:0001",
    ConfigurationStage=LinkLocal,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0001",isProxy=false,ID=0,
    MAC="0050:56C0:0001",SolAddr="FF02:0:0:0:0:1:FFC0:0001",
    ConfigurationStage=DAD,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0001",isProxy=false,ID=0,
    MAC="0050:56C0:0001",SolAddr="FF02:0:0:0:0:1:FFC0:0001",
    ConfigurationStage=Global,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=LinkLocal,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=DAD,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=Global,NRcount=0,Connected=true}

MANET'Req2_Conf 1
 1‘{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0003",isProxy=false,
    ID=1,MAC="0005:9A3C:7800",
    SolAddr="FF02:0:0:0:0:1:FF3C:7800",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0003",isProxy=false,
    ID=1,MAC="0050:56C0:0008",
    SolAddr="FF02:0:0:0:0:1:FFC0:0008",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0004",isProxy=false,
    ID=1,MAC="0005:9A3C:7800",
    SolAddr="FF02:0:0:0:0:1:FF3C:7800",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0004",isProxy=false,
    ID=1,MAC="0050:56C0:0008",
    SolAddr="FF02:0:0:0:0:1:FFC0:0008",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
```

```
 1‘{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0005",isProxy=false,
    ID=1,MAC="0005:9A3C:7800",
    SolAddr="FF02:0:0:0:0:1:FF3C:7800",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0005",isProxy=false,
    ID=1,MAC="0050:56C0:0008",
    SolAddr="FF02:0:0:0:0:1:FFC0:0008",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0006",isProxy=false,
    ID=1,MAC="0005:9A3C:7800",
    SolAddr="FF02:0:0:0:0:1:FF3C:7800",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1‘{Scope=0,Partition=part1,IP="::",isProxy=false,ID=1,
    MAC="0005:9A3C:7800",SolAddr="::",
    ConfigurationStage=SelectMAC,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,IP="::",isProxy=false,ID=1,
    MAC="0050:56C0:0008",SolAddr="::",
    ConfigurationStage=SelectMAC,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,IP="::",isProxy=false,ID=1,
    MAC="::",SolAddr="::",ConfigurationStage=Initial,
    NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0005:9AFF:FE3C:7800",isProxy=false,ID=1,
    MAC="0005:9A3C:7800",SolAddr="FF02:0:0:0:0:1:FF3C:7800",
    ConfigurationStage=LinkLocal,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0005:9AFF:FE3C:7800",isProxy=false,ID=1,
    MAC="0005:9A3C:7800",SolAddr="FF02:0:0:0:0:1:FF3C:7800",
    ConfigurationStage=DAD,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0005:9AFF:FE3C:7800",isProxy=false,ID=1,
    MAC="0005:9A3C:7800",SolAddr="FF02:0:0:0:0:1:FF3C:7800",
    ConfigurationStage=Global,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=1,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=LinkLocal,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=1,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=DAD,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=1,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=Global,NRcount=0,Connected=true}

Manage_Prefixes'No_Refresh_Rec 1      empty
Manage_Prefixes'Node_Info 1
```

```
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Manage_Prefixes'Prefix_Ack'ed 1       1`[]
Proxy_Module'New_Addr 1               1`3
Proxy_Module'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Requester'Global_IP 1
 1`"3FFE:2E01:2B:1111:1111:1111:1111:0003"++
 1`"3FFE:2E01:2B:1111:1111:1111:1111:0004"++
 1`"3FFE:2E01:2B:1111:1111:1111:1111:0005"++
 1`"3FFE:2E01:2B:1111:1111:1111:1111:0006"++1`"::"

Requester'Global_IP 2
 1`"3FFE:2E01:2B:1111:1111:1111:1111:0003"++
 1`"3FFE:2E01:2B:1111:1111:1111:1111:0004"++
 1`"3FFE:2E01:2B:1111:1111:1111:1111:0005"++
 1`"3FFE:2E01:2B:1111:1111:1111:1111:0006"++1`"::"

Requester'Init_DAD 1
 1`"0050:56C0:0001"++1`"0050:56C0:0008"++1`"::"

Requester'Init_DAD 2
 1`"0005:9A3C:7800"++1`"0050:56C0:0008"++1`"::"

Requester'Init_Global_REQ 1        1`"::"
Requester'Init_Global_REQ 2        1`"::"
Requester'Mac_Chosen 1
 1`"0050:56C0:0001"++1`"0050:56C0:0008"

Requester'Mac_Chosen 2
 1`"0005:9A3C:7800"++1`"0050:56C0:0008"

Requester'Rm 1                     1`()
Requester'Rm 2                     1`()
Send_Address_Refresh'Counter 1     1`0
Send_Address_Refresh'Counter 2     1`0
Send_Address_Refresh'Counter 3     1`0
Send_Address_Refresh'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Send_Address_Refresh'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Send_Address_Refresh'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]
```

```
   Update_Configuration'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Update_Configuration'Node_Info 2
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

Best Lower Multi-set Bounds
   Create_Address_Response'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Duplicate_Address_Detection'NS_Sent 1            empty
   Duplicate_Address_Detection'NS_Sent 2            empty
   Duplicate_Address_Detection'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Duplicate_Address_Detection'Node_Info 2
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Duplicate_Address_Detection'Timer 1              empty
   Duplicate_Address_Detection'Timer 2              empty
   Gateway'Node_Info 1
     1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
        (3,1,part1,true),(4,2,part1,true)]

   Global_IP_Request'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Global_IP_Request'Node_Info 2
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Handle_Address_Refresh'Extracted_Prefix 1        empty
   Handle_Address_Refresh'Prefix_Ack'ed 1           1`[]
   Handle_Address_Refresh'Prefixes_Timed_Out 1      empty
   Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1
    1`["2222:2222:2222","3333:3333:3333","4444:4444:4444",
       "5555:5555:5555","6666:6666:6666","7777:7777:7777",
       "8888:8888:8888","9999:9999:9999"]

   Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1
    1`"1111:1111:1111"

   Handle_Incoming_Packets_GATEWAY'Timer 1          empty
   Handle_Incoming_Packets_NODE'Node_Info 1
```

```
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NR'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NR'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'New_Addr 1                          1`3
Handle_NREQ'New_Addr 2                          empty
Handle_NREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NS'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NS'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Refresh_Counter 1   1`0
Handle_RREQ'Refresh_Counter 2   1`0
Handle_RREQ'Refresh_Counter 3   1`0
MANET'Conf1 1
 1`{Scope=1,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0001",isProxy=true,
    ID=2,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'GW_Conf 1
```

```
 1`{Scope=2,Partition=part1,
    IP="3FFE:2E01:2B:1111:0000:0000:0000:0001",isProxy=false,
    ID=4,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'Macs_1 1                    empty
MANET'Macs_2 1                    empty
MANET'Network 1                   empty
MANET'New_Part1 1                 empty
MANET'New_Part2 1                 empty
MANET'New_Scope1 1                empty
MANET'New_Scope2 1                empty
MANET'Req1_Conf 1                 empty
MANET'Req2_Conf 1                 empty
Manage_Prefixes'No_Refresh_Rec 1  empty
Manage_Prefixes'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Manage_Prefixes'Prefix_Ack'ed 1   1`[]
Proxy_Module'New_Addr 1           1`3
Proxy_Module'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Requester'Global_IP 1             empty
Requester'Global_IP 2             empty
Requester'Init_DAD 1              empty
Requester'Init_DAD 2              empty
Requester'Init_Global_REQ 1       1`"::"
Requester'Init_Global_REQ 2       1`"::"
Requester'Mac_Chosen 1            empty
Requester'Mac_Chosen 2            empty
Requester'Rm 1                    empty
Requester'Rm 2                    empty
Send_Address_Refresh'Counter 1    1`0
Send_Address_Refresh'Counter 2    1`0
Send_Address_Refresh'Counter 3    1`0
Send_Address_Refresh'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Send_Address_Refresh'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Send_Address_Refresh'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Update_Configuration'Node_Info 1
```

```
      1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
         (3,1,part1,true),(4,2,part1,true)]

     Update_Configuration'Node_Info 2
      1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
         (3,1,part1,true),(4,2,part1,true)]

 Home Properties
--------------------------------------------------------------------------

  Home Markings
     None

 Liveness Properties
--------------------------------------------------------------------------

  Dead Markings
     282 [96442,96441,96440,96439,95236,...]

  Dead Transition Instances
     Create_Address_Response'Create_Response 1
     Duplicate_Address_Detection'Initiate_GIP_REQ 1
     Duplicate_Address_Detection'Initiate_GIP_REQ 2
     Gateway'Forward_Message 1
     Global_IP_Request'Initiate_Global_IP_Request 1
     Global_IP_Request'Initiate_Global_IP_Request 2
     Handle_AR'Promote_To_Proxy 1
     Handle_AR'Promote_To_Proxy 2
     Handle_AREQ'Forward_Request 1
     Handle_Address_Refresh'Discard 1
     Handle_Address_Refresh'Get_Refresh 1
     Handle_Address_Refresh'Remove_Prefix 1
     Handle_Incoming_Packets_NODE'Forward_Message 1
     Handle_NR'Discard_NR 1
     Handle_NR'Discard_NR 2
     Handle_NR'Get_NR 1
     Handle_NR'Get_NR 2
     Handle_NREQ'Create_Reply 1
     Handle_NREQ'Discard 1
     Handle_NREQ'Discard 2
     Handle_RREQ'Discard 1
     Handle_RREQ'Discard 2
     Handle_RREQ'Discard 3
     Handle_RREQ'Send_Reply 1
     Handle_RREQ'Send_Reply 2
     Handle_RREQ'Send_Reply 3
     Manage_Prefixes'Make_Available 1
     Manage_Prefixes'Refresh_Not_Received 1
     Proxy_Module'Create_Reply 1
     Send_Address_Refresh'Send_Address_Refresh 1
     Send_Address_Refresh'Send_Address_Refresh 2
```

```
    Send_Address_Refresh'Send_Address_Refresh 3
    Update_Configuration'Change_Partition 1
    Update_Configuration'Change_Partition 2
    Update_Configuration'Change_Scope 1
    Update_Configuration'Change_Scope 2

  Live Transition Instances
    None

 Fairness Properties
------------------------------------------------------------------------

    No infinite occurrence sequences.
------------------------------------------------------------------------
```

Prefix management

```
CPN Tools state space report for:
C:\SPECIALE\4-ManagePrefix.cpn
Report generated: Sat Jun 16 11:04:01 2007

 Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  325
     Arcs:   710
     Secs:   1
     Status: Full

  Scc Graph
     Nodes:  325
     Arcs:   710
     Secs:   0

 Boundedness Properties
------------------------------------------------------------------------


  Best Integer Bounds                                  Upper
      Lower
     Create_Address_Response'Node_Info 1                 1          1
     Gateway'Node_Info 1                                 1          1
     Handle_Address_Refresh'Extracted_Prefix 1           2          0
     Handle_Address_Refresh'Prefix_Ack'ed 1              1          1
     Handle_Address_Refresh'Prefixes_Timed_Out 1         1          0
     Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1    1          1
     Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1       1          0
     Handle_Incoming_Packets_GATEWAY'Timer 1             1          0
     Handle_Incoming_Packets_NODE'Node_Info 1            1          1
     Handle_Incoming_Packets_NODE'Node_Info 2            1          1
     Handle_NREQ'Node_Info 1                             1          1
     Handle_NREQ'Node_Info 2                             1          1
     Handle_NREQ'Node_Info 3                             1          1
     Handle_RREQ'Node_Info 1                             1          1
     Handle_RREQ'Node_Info 2                             1          1
     Handle_RREQ'Refresh_Counter 1                       1          1
     Handle_RREQ'Refresh_Counter 2                       1          1
     MANET'Conf1 1                                       1          1
     MANET'Conf3 1                                       1          1
     MANET'GW_Conf 1                                     1          1
     MANET'Network 1                                     6          0
     Manage_Prefixes'No_Refresh_Rec 1                    1          0
     Manage_Prefixes'Node_Info 1                         1          1
     Manage_Prefixes'Prefix_Ack'ed 1                     1          1
     Proxy_Module'New_Addr 1                             1          1
```

```
   Proxy_Module'New_Addr 2                              1          1
   Proxy_Module'Node_Info 1                             1          1
   Proxy_Module'Node_Info 2                             1          1
   Send_Address_Refresh'Counter 1                       1          1
   Send_Address_Refresh'Counter 2                       1          1
   Send_Address_Refresh'Node_Info 1                     1          1
   Send_Address_Refresh'Node_Info 2                     1          1

Best Upper Multi-set Bounds
   Create_Address_Response'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,false),
       (3,1,part1,false),(4,2,part1,true)]++
    1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
       (3,1,part1,false),(4,2,part1,true)]

   Gateway'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,false),
       (3,1,part1,false),(4,2,part1,true)]++
    1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
       (3,1,part1,false),(4,2,part1,true)]

   Handle_Address_Refresh'Extracted_Prefix 1
    2`"1111:1111:1111"

   Handle_Address_Refresh'Prefix_Ack'ed 1
    1`[]++1`[("1111:1111:1111",0)]++1`[("1111:1111:1111",1)]

   Handle_Address_Refresh'Prefixes_Timed_Out 1
    1`"1111:1111:1111"

   Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1
    1`["1111:1111:1111","3333:3333:3333","4444:4444:4444",
       "5555:5555:5555","6666:6666:6666","7777:7777:7777",
       "8888:8888:8888","9999:9999:9999"]++
    1`["3333:3333:3333","4444:4444:4444","5555:5555:5555",
       "6666:6666:6666","7777:7777:7777","8888:8888:8888",
       "9999:9999:9999"]

   Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1
    1`"1111:1111:1111"

   Handle_Incoming_Packets_GATEWAY'Timer 1
    1`"1111:1111:1111"

   Handle_Incoming_Packets_NODE'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,false),
       (3,1,part1,false),(4,2,part1,true)]++
    1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
       (3,1,part1,false),(4,2,part1,true)]

   Handle_Incoming_Packets_NODE'Node_Info 2
```

```
 1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,false),
    (3,1,part1,false),(4,2,part1,true)]++
 1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
    (3,1,part1,false),(4,2,part1,true)]

Handle_NREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,false),
    (3,1,part1,false),(4,2,part1,true)]++
 1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
    (3,1,part1,false),(4,2,part1,true)]

Handle_NREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,false),
    (3,1,part1,false),(4,2,part1,true)]++
 1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
    (3,1,part1,false),(4,2,part1,true)]

Handle_NREQ'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,false),
    (3,1,part1,false),(4,2,part1,true)]++
 1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
    (3,1,part1,false),(4,2,part1,true)]

Handle_RREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,false),
    (3,1,part1,false),(4,2,part1,true)]++
 1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
    (3,1,part1,false),(4,2,part1,true)]

Handle_RREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,false),
    (3,1,part1,false),(4,2,part1,true)]++
 1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
    (3,1,part1,false),(4,2,part1,true)]

Handle_RREQ'Refresh_Counter 1  1`0++1`1
Handle_RREQ'Refresh_Counter 2  1`0++1`1
MANET'Conf1 1
 1`{Scope=1,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0001",isProxy=true,
    ID=2,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=false}++
 1`{Scope=1,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0001",isProxy=true,
    ID=2,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'Conf3 1
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0002",isProxy=false,
    ID=0,MAC="::",SolAddr="::",NRcount=0,
```

```
    ConfigurationStage=FullyConfigured,Connected=false}++
 1‘{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0002",isProxy=false,
    ID=0,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET’GW_Conf 1
 1‘{Scope=2,Partition=part1,
    IP="3FFE:2E01:2B:1111:0000:0000:0000:0001",isProxy=false,
    ID=4,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET’Network 1
 3‘RefreshRequest(
   {Source="3FFE:2E01:2B:1111:0000:0000:0000:0001",
   Prefix="1111:1111:1111",Partition=part1,ReceiverID=0})++
 3‘RefreshRequest(
   {Source="3FFE:2E01:2B:1111:0000:0000:0000:0001",
   Prefix="1111:1111:1111",Partition=part1,ReceiverID=2})++
 1‘AddressRefresh({Partition=part1,CurrentScope=1,
                   Prefix="1111:1111:1111",FromProxy=false})++
 1‘AddressRefresh({Partition=part1,CurrentScope=2,
                   Prefix="1111:1111:1111",FromProxy=false})++
 1‘AddressRefresh({Partition=part1,CurrentScope=2,
                   Prefix="1111:1111:1111",FromProxy=true})

Manage_Prefixes’No_Refresh_Rec 1   1‘"1111:1111:1111"
Manage_Prefixes’Node_Info 1
 1‘[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
    (3,1,part1,false),(4,2,part1,true)]

Manage_Prefixes’Prefix_Ack’ed 1
 1‘[]++1‘[("1111:1111:1111",0)]++1‘[("1111:1111:1111",1)]

Proxy_Module’New_Addr 1    1‘3
Proxy_Module’New_Addr 2    1‘3
Proxy_Module’Node_Info 1
 1‘[(0,0,part1,true),(1,0,part1,false),(2,1,part1,false),
    (3,1,part1,false),(4,2,part1,true)]++
 1‘[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
    (3,1,part1,false),(4,2,part1,true)]

Proxy_Module’Node_Info 2
 1‘[(0,0,part1,true),(1,0,part1,false),(2,1,part1,false),
    (3,1,part1,false),(4,2,part1,true)]++
 1‘[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
    (3,1,part1,false),(4,2,part1,true)]

Send_Address_Refresh’Counter 1       1‘1
Send_Address_Refresh’Counter 2       1‘1++1‘2
Send_Address_Refresh’Node_Info 1
```

```
     1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,false),
        (3,1,part1,false),(4,2,part1,true)]++
     1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
        (3,1,part1,false),(4,2,part1,true)]

   Send_Address_Refresh'Node_Info 2
     1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,false),
        (3,1,part1,false),(4,2,part1,true)]++
     1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
        (3,1,part1,false),(4,2,part1,true)]

Best Lower Multi-set Bounds
   Create_Address_Response'Node_Info 1                 empty
   Gateway'Node_Info 1                                 empty
   Handle_Address_Refresh'Extracted_Prefix 1           empty
   Handle_Address_Refresh'Prefix_Ack'ed 1              empty
   Handle_Address_Refresh'Prefixes_Timed_Out 1         empty
   Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1    empty
   Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1       empty
   Handle_Incoming_Packets_GATEWAY'Timer 1             empty
   Handle_Incoming_Packets_NODE'Node_Info 1            empty
   Handle_Incoming_Packets_NODE'Node_Info 2            empty
   Handle_NREQ'Node_Info 1                             empty
   Handle_NREQ'Node_Info 2                             empty
   Handle_NREQ'Node_Info 3                             empty
   Handle_RREQ'Node_Info 1                             empty
   Handle_RREQ'Node_Info 2                             empty
   Handle_RREQ'Refresh_Counter 1                       empty
   Handle_RREQ'Refresh_Counter 2                       empty
   MANET'Conf1 1                                       empty
   MANET'Conf3 1                                       empty
   MANET'GW_Conf 1
     1`{Scope=2,Partition=part1,
        IP="3FFE:2E01:2B:1111:0000:0000:0000:0001",isProxy=false,
        ID=4,MAC="::",SolAddr="::",NRcount=0,
        ConfigurationStage=FullyConfigured,Connected=true}

   MANET'Network 1                                     empty
   Manage_Prefixes'No_Refresh_Rec 1                    empty
   Manage_Prefixes'Node_Info 1
     1`[(0,0,part1,true),(1,0,part1,false),(2,1,part1,true),
        (3,1,part1,false),(4,2,part1,true)]

   Manage_Prefixes'Prefix_Ack'ed 1                     empty
   Proxy_Module'New_Addr 1                             1`3
   Proxy_Module'New_Addr 2                             1`3
   Proxy_Module'Node_Info 1                            empty
   Proxy_Module'Node_Info 2                            empty
   Send_Address_Refresh'Counter 1                      1`1
   Send_Address_Refresh'Counter 2                      empty
   Send_Address_Refresh'Node_Info 1                    empty
```

```
    Send_Address_Refresh'Node_Info 2                    empty

 Home Properties
-------------------------------------------------------------------------

  Home Markings
    None

 Liveness Properties
-------------------------------------------------------------------------

  Dead Markings
    16 [61,325,322,321,316,...]

  Dead Transition Instances
    Create_Address_Response'Create_Response 1
    Gateway'Forward_Message 1
    Handle_AREQ'Forward_Request 1
    Handle_AREQ'Forward_Request 2
    Handle_Incoming_Packets_NODE'Forward_Message 1
    Handle_Incoming_Packets_NODE'Forward_Message 2
    Handle_NREQ'Create_Reply 1
    Handle_NREQ'Create_Reply 2
    Handle_NREQ'Create_Reply 3
    Handle_NREQ'Discard 1
    Handle_NREQ'Discard 2
    Handle_NREQ'Discard 3
    Proxy_Module'Create_Reply 1
    Proxy_Module'Create_Reply 2
    Send_Address_Refresh'Send_Address_Refresh 1

  Live Transition Instances
    None

 Fairness Properties
-------------------------------------------------------------------------

    No infinite occurrence sequences.
```

Scope change

```
CPN Tools state space report for:
C:\SPECIALE\5-TwoRequestersChangeScope.cpn
Report generated: Wed Jun 13 19:27:51 2007

 Statistics
------------------------------------------------------------------------

  State Space
     Nodes:  5754
     Arcs:   16917
     Secs:   15
     Status: Full

  Scc Graph
     Nodes:  5754
     Arcs:   16917
     Secs:   0

 Boundedness Properties
------------------------------------------------------------------------


  Best Integer Bounds                                  Upper
      Lower
     Create_Address_Response'Node_Info 1                 1          1
     Duplicate_Address_Detection'NS_Sent 1               1          1
     Duplicate_Address_Detection'NS_Sent 2               1          1
     Duplicate_Address_Detection'Node_Info 1             1          1
     Duplicate_Address_Detection'Node_Info 2             1          1
     Duplicate_Address_Detection'Timer 1                 1          1
     Duplicate_Address_Detection'Timer 2                 1          1
     Gateway'Node_Info 1                                 1          1
     Global_IP_Request'Node_Info 1                       1          1
     Global_IP_Request'Node_Info 2                       1          1
     Handle_Address_Refresh'Extracted_Prefix 1           0          0
     Handle_Address_Refresh'Prefix_Ack'ed 1              1          1
     Handle_Address_Refresh'Prefixes_Timed_Out 1         0          0
     Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1 1  1          1
     Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1       2          1
     Handle_Incoming_Packets_GATEWAY'Timer 1             0          0
     Handle_Incoming_Packets_NODE'Node_Info 1            1          1
     Handle_NR'Node_Info 1                               1          1
     Handle_NR'Node_Info 2                               1          1
     Handle_NREQ'Node_Info 1                             1          1
     Handle_NREQ'Node_Info 2                             1          1
     Handle_NS'Node_Info 1                               1          1
     Handle_NS'Node_Info 2                               1          1
     Handle_RREQ'Node_Info 1                             1          1
     Handle_RREQ'Node_Info 2                             1          1
```

```
   Handle_RREQ'Node_Info 3                          1          1
   Handle_RREQ'Refresh_Counter 1                    1          1
   Handle_RREQ'Refresh_Counter 2                    1          1
   Handle_RREQ'Refresh_Counter 3                    1          1
   MANET'Conf1 1                                    1          1
   MANET'GW_Conf 1                                  1          1
   MANET'Macs_1 1                                   1          0
   MANET'Macs_2 1                                   1          0
   MANET'Network 1                                  9          0
   MANET'New_Part1 1                                0          0
   MANET'New_Part2 1                                0          0
   MANET'New_Scope1 1                               1          1
   MANET'New_Scope2 1                               0          0
   MANET'Req1_Conf 1                                1          1
   MANET'Req2_Conf 1                                1          1
   Manage_Prefixes'No_Refresh_Rec 1                 0          0
   Manage_Prefixes'Node_Info 1                      1          1
   Manage_Prefixes'Prefix_Ack'ed 1                  1          1
   Proxy_Module'New_Addr 1                          1          1
   Proxy_Module'Node_Info 1                         1          1
   Requester'Global_IP 1                            1          1
   Requester'Global_IP 2                            1          1
   Requester'Init_DAD 1                             1          1
   Requester'Init_DAD 2                             1          1
   Requester'Init_Global_REQ 1                      1          1
   Requester'Init_Global_REQ 2                      1          1
   Requester'Mac_Chosen 1                           1          0
   Requester'Mac_Chosen 2                           1          0
   Requester'Rm 1                                   1          0
   Requester'Rm 2                                   1          0
   Send_Address_Refresh'Counter 1                   1          1
   Send_Address_Refresh'Counter 2                   1          1
   Send_Address_Refresh'Counter 3                   1          1
   Send_Address_Refresh'Node_Info 1                 1          1
   Send_Address_Refresh'Node_Info 2                 1          1
   Send_Address_Refresh'Node_Info 3                 1          1
   Update_Configuration'Node_Info 1                 1          1
   Update_Configuration'Node_Info 2                 1          1

Best Upper Multi-set Bounds
   Create_Address_Response'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]++
    1`[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]

   Duplicate_Address_Detection'NS_Sent 1  1`0++1`1++1`2++1`3
   Duplicate_Address_Detection'NS_Sent 2  1`0++1`1++1`2++1`3
   Duplicate_Address_Detection'Node_Info 1
    1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
       (3,1,part1,true),(4,2,part1,true)]++
```

```
 1`[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Duplicate_Address_Detection'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1`[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Duplicate_Address_Detection'Timer 1
 1`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0050:56FF:FEC0:0008",Route=[0],
       Destination="FF02:0:0:0:0:1:FFC0:0008",ReceiverID=0})++
 1`NS({SourceScope=3,CurrentScope=3,DestinationScope=3,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0050:56FF:FEC0:0008",Route=[0],
       Destination="FF02:0:0:0:0:1:FFC0:0008",ReceiverID=0})++
 1`ERROR({msg=""})

Duplicate_Address_Detection'Timer 2
 1`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0005:9AFF:FE3C:7800",Route=[1],
       Destination="FF02:0:0:0:0:1:FF3C:7800",ReceiverID=1})++
 1`ERROR({msg=""})

Gateway'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1`[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Global_IP_Request'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1`[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Global_IP_Request'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1`[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_Address_Refresh'Extracted_Prefix 1        empty
Handle_Address_Refresh'Prefix_Ack'ed 1           1`[]
Handle_Address_Refresh'Prefixes_Timed_Out 1      empty
Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1
 1`["2222:2222:2222","3333:3333:3333","4444:4444:4444",
    "5555:5555:5555","6666:6666:6666","7777:7777:7777",
```

```
    "8888:8888:8888","9999:9999:9999"]++
 1‘["3333:3333:3333","4444:4444:4444","5555:5555:5555",
    "6666:6666:6666","7777:7777:7777","8888:8888:8888",
    "9999:9999:9999"]

Handle_Incoming_Packets_GATEWAY’Pref_In_Use 1
 1‘"1111:1111:1111"++1‘"2222:2222:2222"

Handle_Incoming_Packets_GATEWAY’Timer 1   empty
Handle_Incoming_Packets_NODE’Node_Info 1
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1‘[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NR’Node_Info 1
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1‘[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NR’Node_Info 2
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1‘[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ’Node_Info 1
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1‘[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NREQ’Node_Info 2
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1‘[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NS’Node_Info 1
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1‘[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_NS’Node_Info 2
1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1‘[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]
```

```
Handle_RREQ'Node_Info 1
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1`[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 2
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1`[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Node_Info 3
 1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1`[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Handle_RREQ'Refresh_Counter 1   1`0
Handle_RREQ'Refresh_Counter 2   1`0
Handle_RREQ'Refresh_Counter 3   1`0
MANET'Conf1 1
 1`{Scope=1,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0001",isProxy=true,
    ID=2,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'GW_Conf 1
 1`{Scope=2,Partition=part1,
    IP="3FFE:2E01:2B:1111:0000:0000:0000:0001",isProxy=false,
    ID=4,MAC="::",SolAddr="::",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}

MANET'Macs_1 1      1`"0050:56C0:0008"
MANET'Macs_2 1      1`"0005:9A3C:7800"
MANET'Network 1
 2`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0005:9AFF:FE3C:7800",Route=[1],
       Destination="FF02:0:0:0:0:1:FF3C:7800",ReceiverID=0})++
 2`NS({SourceScope=0,CurrentScope=0,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0050:56FF:FEC0:0008",Route=[0],
       Destination="FF02:0:0:0:0:1:FFC0:0008",ReceiverID=1})++
 2`NS({SourceScope=3,CurrentScope=2,DestinationScope=0,
       Partition=part1,Source="::",
       Target="FE80:0:0:0:0050:56FF:FEC0:0008",Route=[0],
       Destination="FF02:0:0:0:0:1:FFC0:0008",ReceiverID=1})++
 1`NR({SourceScope=0,Partition=part1,Route=[1],
       Source="FE80:0:0:0:0005:9AFF:FE3C:7800",ReceiverID=2})
           ++
```

```
1`NR({SourceScope=0,Partition=part1,Route=[0],
      Source="FE80:0:0:0:0050:56FF:FEC0:0008",ReceiverID=2})
         ++
1`NR({SourceScope=3,Partition=part1,Route=[0],
      Source="FE80:0:0:0:0050:56FF:FEC0:0008",ReceiverID=4})
         ++
1`NRep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[2],
        Source="3FFE:2E01:2B:1111:1111:1111:1111:0001",
        Destination="FE80:0:0:0:0005:9AFF:FE3C:7800"})++
1`NRep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1Route=[2],,
        Source="3FFE:2E01:2B:1111:1111:1111:1111:0001",
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008"})++
1`NRep({SourceScope=2,CurrentScope=3,DestinationScope=3,
        Partition=part1,Route=[4],
        Source="3FFE:2E01:2B:1111:0000:0000:0000:0001",
       Destination="FE80:0:0:0:0050:56FF:FEC0:0008"})++
1`AReq({SourceScope=0,CurrentScope=1,DestinationScope=1,
        Source="FE80:0:0:0:0005:9AFF:FE3C:7800",
        Destination="3FFE:2E01:2B:1111:1111:1111:1111:0001",
        Route=[1],Partition=part1})++
1`AReq({SourceScope=0,CurrentScope=1,DestinationScope=1,
        Source="FE80:0:0:0:0050:56FF:FEC0:0008",
        Destination="3FFE:2E01:2B:1111:1111:1111:1111:0001",
        Route=[0],Partition=part1})++
1`AReq({SourceScope=3,CurrentScope=2,DestinationScope=2,
        Source="FE80:0:0:0:0050:56FF:FEC0:0008",
        Destination="3FFE:2E01:2B:1111:0000:0000:0000:0001",
        Route=[0],Partition=part1})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[1],Lifetime=1000,
        Destination="FE80:0:0:0:0005:9AFF:FE3C:7800",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0003"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[1],Lifetime=1000,
        Destination="FE80:0:0:0:0005:9AFF:FE3C:7800",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0004"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[0],Lifetime=1000,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0003"})++
1`ARep({SourceScope=1,CurrentScope=0,DestinationScope=0,
        Partition=part1,Route=[0],Lifetime=1000,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008",
        GlobalIP="3FFE:2E01:2B:1111:1111:1111:1111:0004"})++
1`ARep({SourceScope=2,CurrentScope=3,DestinationScope=3,
        Partition=part1,Route=[0],Lifetime=1000,
        Destination="FE80:0:0:0:0050:56FF:FEC0:0008",
        GlobalIP="3FFE:2E01:2B:1111:2222:2222:2222:0000"})
```

```
MANET'New_Part1 1    empty
MANET'New_Part2 1    empty
MANET'New_Scope1 1   1`[]++1`[3]
MANET'New_Scope2 1   empty
MANET'Req1_Conf 1
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0003",isProxy=false,
    ID=0,MAC="0050:56C0:0008",
    SolAddr="FF02:0:0:0:0:1:FFC0:0008",NRcount=1,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0004",isProxy=false,
    ID=0,MAC="0050:56C0:0008",
    SolAddr="FF02:0:0:0:0:1:FFC0:0008",NRcount=1,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,IP="::",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="::",
    ConfigurationStage=SelectMAC,NRcount=0,Connected=true}++
 1`{Scope=0,Partition=part1,IP="::",isProxy=false,ID=0,
    MAC="::",SolAddr="::",ConfigurationStage=Initial,
    NRcount=0,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=LinkLocal,NRcount=0,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=DAD,NRcount=0,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=Global,NRcount=0,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=Global,NRcount=1,Connected=true}++
 1`{Scope=3,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0003",isProxy=false,
    ID=0,MAC="0050:56C0:0008",
    SolAddr="FF02:0:0:0:0:1:FFC0:0008",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=3,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0004",isProxy=false,
    ID=0,MAC="0050:56C0:0008",
    SolAddr="FF02:0:0:0:0:1:FFC0:0008",NRcount=0,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=3,Partition=part1,
    IP="3FFE:2E01:2B:1111:2222:2222:2222:0001",isProxy=true,
    ID=0,MAC="0050:56C0:0008",
    SolAddr="FF02:0:0:0:0:1:FFC0:0008",NRcount=1,
```

```
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=3,Partition=part1,IP="::",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="::",
    ConfigurationStage=SelectMAC,NRcount=0,Connected=true}++
 1`{Scope=3,Partition=part1,IP="::",isProxy=false,ID=0,
    MAC="::",SolAddr="::",ConfigurationStage=Initial,
    NRcount=0,Connected=true}++
 1`{Scope=3,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=LinkLocal,NRcount=0,Connected=true}++
 1`{Scope=3,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",
    SolAddr="FF02:0:0:0:0:1:FFC0:0008",ConfigurationStage=DAD,
    NRcount=0,Connected=true}++
 1`{Scope=3,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=Global,NRcount=0,Connected=true}++
 1`{Scope=3,Partition=part1,
    IP="FE80:0:0:0:0050:56FF:FEC0:0008",isProxy=false,ID=0,
    MAC="0050:56C0:0008",SolAddr="FF02:0:0:0:0:1:FFC0:0008",
    ConfigurationStage=Global,NRcount=1,Connected=true}

MANET'Req2_Conf 1
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0003",isProxy=false,
    ID=1,MAC="0005:9A3C:7800",
    SolAddr="FF02:0:0:0:0:1:FF3C:7800",NRcount=1,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="3FFE:2E01:2B:1111:1111:1111:1111:0004",isProxy=false,
    ID=1,MAC="0005:9A3C:7800",
    SolAddr="FF02:0:0:0:0:1:FF3C:7800",NRcount=1,
    ConfigurationStage=FullyConfigured,Connected=true}++
 1`{Scope=0,Partition=part1,IP="::",isProxy=false,ID=1,
    MAC="0005:9A3C:7800",SolAddr="::",
    ConfigurationStage=SelectMAC,NRcount=0,Connected=true}++
 1`{Scope=0,Partition=part1,IP="::",isProxy=false,ID=1,
    MAC="::",SolAddr="::",ConfigurationStage=Initial,NRcount
        =0,
    Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0005:9AFF:FE3C:7800",isProxy=false,ID=1,
    MAC="0005:9A3C:7800",SolAddr="FF02:0:0:0:0:1:FF3C:7800",
    ConfigurationStage=LinkLocal,NRcount=0,Connected=true}++
 1`{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0005:9AFF:FE3C:7800",isProxy=false,ID=1,
    MAC="0005:9A3C:7800",SolAddr="FF02:0:0:0:0:1:FF3C:7800",
    ConfigurationStage=DAD,NRcount=0,Connected=true}++
```

```
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0005:9AFF:FE3C:7800",isProxy=false,ID=1,
    MAC="0005:9A3C:7800",SolAddr="FF02:0:0:0:0:1:FF3C:7800",
    ConfigurationStage=Global,NRcount=0,Connected=true}++
 1‘{Scope=0,Partition=part1,
    IP="FE80:0:0:0:0005:9AFF:FE3C:7800",isProxy=false,ID=1,
    MAC="0005:9A3C:7800",SolAddr="FF02:0:0:0:0:1:FF3C:7800",
    ConfigurationStage=Global,NRcount=1,Connected=true}

Manage_Prefixes'No_Refresh_Rec 1    empty
Manage_Prefixes'Node_Info 1
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Manage_Prefixes'Prefix_Ack'ed 1     1‘[]
Proxy_Module'New_Addr 1             1‘3++1‘4++1‘5
Proxy_Module'Node_Info 1
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1‘[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Requester'Global_IP 1
 1‘"3FFE:2E01:2B:1111:1111:1111:1111:0003"++
 1‘"3FFE:2E01:2B:1111:1111:1111:1111:0004"++
 1‘"3FFE:2E01:2B:1111:2222:2222:2222:0001"++1‘"::"

Requester'Global_IP 2
 1‘"3FFE:2E01:2B:1111:1111:1111:1111:0003"++
 1‘"3FFE:2E01:2B:1111:1111:1111:1111:0004"++1‘"::"

Requester'Init_DAD 1             1‘"0050:56C0:0008"++1‘"::"
Requester'Init_DAD 2             1‘"0005:9A3C:7800"++1‘"::"
Requester'Init_Global_REQ 1      1‘"0050:56C0:0008"++1‘"::"
Requester'Init_Global_REQ 2      1‘"0005:9A3C:7800"++1‘"::"
Requester'Mac_Chosen 1           1‘"0050:56C0:0008"
Requester'Mac_Chosen 2           1‘"0005:9A3C:7800"
Requester'Rm 1                   1‘()
Requester'Rm 2                   1‘()
Send_Address_Refresh'Counter 1   1‘0
Send_Address_Refresh'Counter 2   1‘0
Send_Address_Refresh'Counter 3   1‘0
Send_Address_Refresh'Node_Info 1
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
 1‘[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]

Send_Address_Refresh'Node_Info 2
 1‘[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
    (3,1,part1,true),(4,2,part1,true)]++
```

```
     1`[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
        (3,1,part1,true),(4,2,part1,true)]

   Send_Address_Refresh'Node_Info 3
     1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
        (3,1,part1,true),(4,2,part1,true)]++
     1`[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
        (3,1,part1,true),(4,2,part1,true)]

   Update_Configuration'Node_Info 1
     1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
        (3,1,part1,true),(4,2,part1,true)]++
     1`[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
        (3,1,part1,true),(4,2,part1,true)]

   Update_Configuration'Node_Info 2
      1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
        (3,1,part1,true),(4,2,part1,true)]++
     1`[(0,3,part1,true),(1,0,part1,true),(2,1,part1,true),
        (3,1,part1,true),(4,2,part1,true)]

Best Lower Multi-set Bounds
   Create_Address_Response'Node_Info 1              empty
   Duplicate_Address_Detection'NS_Sent 1            empty
   Duplicate_Address_Detection'NS_Sent 2            empty
   Duplicate_Address_Detection'Node_Info 1          empty
   Duplicate_Address_Detection'Node_Info 2          empty
   Duplicate_Address_Detection'Timer 1              empty
   Duplicate_Address_Detection'Timer 2              empty
   Gateway'Node_Info 1                              empty
   Global_IP_Request'Node_Info 1                    empty
   Global_IP_Request'Node_Info 2                    empty
   Handle_Address_Refresh'Extracted_Prefix 1        empty
   Handle_Address_Refresh'Prefix_Ack'ed 1           1`[]
   Handle_Address_Refresh'Prefixes_Timed_Out 1      empty
   Handle_Incoming_Packets_GATEWAY'Avail_Prefixes 1 empty
   Handle_Incoming_Packets_GATEWAY'Pref_In_Use 1
    1`"1111:1111:1111"

   Handle_Incoming_Packets_GATEWAY'Timer 1          empty
   Handle_Incoming_Packets_NODE'Node_Info 1         empty
   Handle_NR'Node_Info 1                            empty
   Handle_NR'Node_Info 2                            empty
   Handle_NREQ'Node_Info 1                          empty
   Handle_NREQ'Node_Info 2                          empty
   Handle_NS'Node_Info 1                            empty
   Handle_NS'Node_Info 2                            empty
   Handle_RREQ'Node_Info 1                          empty
   Handle_RREQ'Node_Info 2                          empty
   Handle_RREQ'Node_Info 3                          empty
   Handle_RREQ'Refresh_Counter 1                    1`0
```

```
    Handle_RREQ'Refresh_Counter 2                    1`0
    Handle_RREQ'Refresh_Counter 3                    1`0
    MANET'Conf1 1
     1`{Scope=1,Partition=part1,
        IP="3FFE:2E01:2B:1111:1111:1111:1111:0001",isProxy=true,
        ID=2,MAC="::",SolAddr="::",NRcount=0,
        ConfigurationStage=FullyConfigured,Connected=true}

    MANET'GW_Conf 1
     1`{Scope=2,Partition=part1,
        IP="3FFE:2E01:2B:1111:0000:0000:0000:0001",isProxy=false,
        ID=4,MAC="::",SolAddr="::",NRcount=0,
        ConfigurationStage=FullyConfigured,Connected=true}

    MANET'Macs_1 1                                   empty
    MANET'Macs_2 1                                   empty
    MANET'Network 1                                  empty
    MANET'New_Part1 1                                empty
    MANET'New_Part2 1                                empty
    MANET'New_Scope1 1                               empty
    MANET'New_Scope2 1                               empty
    MANET'Req1_Conf 1                                empty
    MANET'Req2_Conf 1                                empty
    Manage_Prefixes'No_Refresh_Rec 1                 empty
    Manage_Prefixes'Node_Info 1
     1`[(0,0,part1,true),(1,0,part1,true),(2,1,part1,true),
        (3,1,part1,true),(4,2,part1,true)]

    Manage_Prefixes'Prefix_Ack'ed 1                  1`[]
    Proxy_Module'New_Addr 1                          empty
    Proxy_Module'Node_Info 1                         empty
    Requester'Global_IP 1                            empty
    Requester'Global_IP 2                            empty
    Requester'Init_DAD 1                             empty
    Requester'Init_DAD 2                             empty
    Requester'Init_Global_REQ 1                      empty
    Requester'Init_Global_REQ 2                      empty
    Requester'Mac_Chosen 1                           empty
    Requester'Mac_Chosen 2                           empty
    Requester'Rm 1                                   empty
    Requester'Rm 2                                   empty
    Send_Address_Refresh'Counter 1                   1`0
    Send_Address_Refresh'Counter 2                   1`0
    Send_Address_Refresh'Counter 3                   1`0
    Send_Address_Refresh'Node_Info 1                 empty
    Send_Address_Refresh'Node_Info 2                 empty
    Send_Address_Refresh'Node_Info 3                 empty
    Update_Configuration'Node_Info 1                 empty
    Update_Configuration'Node_Info 2                 empty

Home Properties
```

```
------------------------------------------------------------------------

  Home Markings
     None

 Liveness Properties
------------------------------------------------------------------------

  Dead Markings
     21 [5754,5753,5747,5744,5721,...]

  Dead Transition Instances
     Gateway'Forward_Message 1
     Handle_AR'Promote_To_Proxy 2
     Handle_AREQ'Forward_Request 1
     Handle_Address_Refresh'Discard 1
     Handle_Address_Refresh'Get_Refresh 1
     Handle_Address_Refresh'Remove_Prefix 1
     Handle_Incoming_Packets_NODE'Forward_Message 1
     Handle_NA'Discard_NA 1
     Handle_NA'Discard_NA 2
     Handle_NA'Get_NA 1
     Handle_NA'Get_NA 2
     Handle_NR'Discard_NR 2
     Handle_NREQ'Discard 1
     Handle_NREQ'Discard 2
     Handle_NS'Generate_NA 1
     Handle_NS'Generate_NA 2
     Handle_RREQ'Discard 1
     Handle_RREQ'Discard 2
     Handle_RREQ'Discard 3
     Handle_RREQ'Send_Reply 1
     Handle_RREQ'Send_Reply 2
     Handle_RREQ'Send_Reply 3
     Manage_Prefixes'Make_Available 1
     Manage_Prefixes'Refresh_Not_Received 1
     Send_Address_Refresh'Send_Address_Refresh 1
     Send_Address_Refresh'Send_Address_Refresh 2
     Send_Address_Refresh'Send_Address_Refresh 3
     Update_Configuration'Change_Partition 1
     Update_Configuration'Change_Partition 2
     Update_Configuration'Change_Scope 2

  Live Transition Instances
     None


 Fairness Properties
------------------------------------------------------------------------

     No infinite occurrence sequences.
------------------------------------------------------------------------
```