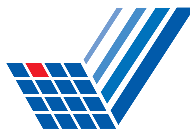
Integrated Formal Modeling and Automated Analysis of Computer Network Attacks

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik

von
Gerrit Rothmaier



Dortmund
2006

Tag der mündlichen Prüfung: 22.02.2007

Dekan: Prof. Dr. Peter Buchholz

Gutachter: Prof. Dr. Heiko Krumm, Prof. Dr. Joachim Biskup

Abstract

In the field of formal modeling and analysis as related to computer network security, existing approaches are highly specialized towards either a protocol, node, or network view. Typically, they are even further specialized towards a specific subset of one view (e.g., a certain class of protocols, interactions of local node components, or network propagation of predefined vulnerabilities). Thus, each approach covers only a small part of the aspects related to practical computer network attack scenarios. Often, further restrictions with respect to the dynamics allowed for the model, properties supported or user guidance required during analysis, have to be observed. Multiple approaches, and thus models, formalisms, and analysis tools, need to be employed to provide a more complete view of computer network attack scenarios. Both the modeling task and the analysis task have to be done multiple times and it is hard to ensure the consistency of the models and analysis results.
We present a novel approach that comprehensively integrates the protocol, node, and network view on a middle level of detail. Furthermore, the models are expressive enough to support dynamic changes. A wide range of properties can be specified using different mechanisms. As integrated models naturally are of higher complexity than more specialized models limited to a single view, analysis is particularly challenging. Generally, automated analysis approaches quickly fail due to state space explosion effects. Nevertheless, by careful modeling, considering optimization possibilities at all stages, modeling using an object-oriented and compositional yet simple structured language, and employing a state of the art analysis tool we are able to achieve automated analysis.
Our approach is based on the high-level specification language cTLA 2003, a framework for modeling computer network attack scenarios, a scheme for translating cTLA 2003 to PROMELA, the cTLA2PC translation and optimization tool, and the powerful model checker SPIN. For demonstrating the feasibility of our approach, the modeling and analysis of three case studies involving multi-node dynamic network scenarios is presented. In these case studies, precise attack sequences are automatically predicted as violations of abstract security properties.

Keywords:
Computer Networks, Formal Methods, Integrated Modeling, Automated Analysis, Protocol, Node, Network, Security, Attacks, SPIN, cTLA

Zusammenfassung

Die vorhandenen Ansätze zur formalen Modellierung und Analyse von Computernetzwerksicherheit sind entweder auf eine Protokoll-, Knoten-, oder Netzwerksicht ausgerichtet. Meist beschränken sie sich sogar auf einen speziellen Teilbereich einer dieser Sichten (z.B. eine bestimmte Art von Protokollen, die Interaktion zwischen den lokalen Komponenten eines Knotens, oder die Ausbreitung vordefininierter Verletzlichkeiten). Insgesamt wird von jedem Ansatz jeweils nur ein kleiner Teil der Aspekte, die in praktischen Computernetzwerkangriffsszenarien vorkommen, abgedeckt. Hinzu kommen oft weitere Einschränkungen in Bezug auf Unterstützung dynamischer Änderungen, modellier- und untersuchbare Eigenschaften, benötigte Unterstützung der Analyse durch den Benutzer, usw. Um eine vollständigere Sicht auf Computernetzwerkangriffsszenarien zu erhalten, müssen daher mehrere Ansätze, und damit auch Modelle, Formalismen und Werkzeuge, eingesetzt werden. Sowohl die Modellierungs- als auch die Analysearbeit fallen damit mehrfach an und Konsistenz zwischen den verschiedenen Modellen und Analyseergebnissen lässt sich nur sehr schwer erreichen.

In dieser Arbeit wird ein neuartiger Ansatz vorgestellt, der die Protokoll-, Knoten- und Netzwerksicht auf mittlerer Detailebene übergreifend integriert. Die Modelle sind ausdrucksstark genug, um dynamische Änderungen zu beinhalten. Vielfältige Eigenschaften können über unterschiedliche Mechanismen spezifiziert werden. Da integrierte Modelle deutlich komplexer als eingeschränkte Modelle für einen Teilbereich sind, ist die Analyse besonders schwierig. Im Allgemeinen schlagen Ansätze zur automatischen Analyse schnell durch Zustandsraumexplosion fehl. Durch eine intelligente Modellierung, die Berücksichtigung von Optimierungsmöglichkeiten auf allen Ebenen, die Modellierung mit einer objektorientieren und kompositionalen, aber trotzdem auf einer einfachen Struktur basierenden Sprache, und dem Einsatz eines dem aktuellen Stand der Forschung entsprechenden Analysewerkzeuges sind wir trotzdem in der Lage, erfolgreich automatisiert zu analysieren.

Unser Ansatz basiert auf der Spezifikationshochsprache cTLA 2003, einem Framework zur Modellierung von Computernetzwerkangriffsszenarien, einem Übersetzungsschema von cTLA 2003 nach PROMELA, dem cTLA2PC Übersetzungs- und Optimierungswerkzeug, und dem mächtigen Modellchecker SPIN. Die Durchführbarkeit unseres Ansatzes wird durch die Modellierung und Analyse von drei dynamischen Netzwerkszenarien zunehmender Komplexität aufgezeigt. In diesen Szenarien werden konkrete Angriffsfolgen als Verletzungen vorgegebener Sicherheitseigenschaften automatisch aufgedeckt.

Schlagwörter:
Rechnernetze, Formale Methoden, Integrierte Modellierung, Automatische Analyse, Protokoll, Knoten, Netzwerk, Sicherheit, Angriffe, SPIN, cTLA

Acknowledgment

Thanks to my advisor, Prof. Dr. Heiko Krumm, for his support and advice over the years and the feedback on my thesis work. Furthermore, I am very grateful to Prof. Dr. Joachim Biskup for the opportunity to present and discuss ideas at the "Kolleg Sicherheit".

I also thank Esther Bantle for her cheerful encouragement and proofreading of this thesis. I am grateful to fellow Ph.D. student Frank Müller for fruitful discussions and pointing me to Jorge Cham's funny reflections on becoming a Ph.D.

My thesis greatly benefited from collaborations with my graduate students, including Andre Pohl, Tobias Kneiphoff, Marc Malik, and Helge Konetzka. Particularly, Andre Pohl spent a lot of time working on the cTLA2PC tool during his master's thesis and continuously supported the tool afterwards.

Dedicated to Heidrun, Karin, and Esther

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Goal Statement . . . 2
1.3 Thesis Outline . . . 2
1.4 Publications . . . 4

2 Related Work **7**
2.1 Classification Scheme . . . 7
2.2 Protocol-Oriented Approaches . . . 8
2.3 Node-Oriented Approaches . . . 10
2.4 Network-Oriented Approaches . . . 12
2.5 Network Security Tools . . . 16
2.6 Discussion . . . 18

3 Spin, Promela, TLA, and cTLA **19**
3.1 Spin . . . 19
3.2 Promela . . . 24
3.3 Temporal Logic of Actions (TLA) . . . 29
3.4 Compositional Temporal Logic of Actions (cTLA) . . . 31

4 An Integrated, Formal Modeling and Automated Analysis Approach **35**
4.1 Objectives . . . 35
4.2 Implementation . . . 36
4.3 Workflow . . . 38
4.4 Modeling Steps . . . 40

5 The cTLA 2003 Modeling Language **43**
5.1 Comparison to cTLA 2000 . . . 43
5.2 Specification Structure . . . 46
5.3 Process Types . . . 51
5.4 Grammar . . . 56

6 Translation, cTLA2PC, and Eclipse Integration 59
6.1 Motivation . . . 59
6.2 Translation Scheme . . . 59
6.3 The cTLA2PC Translation Tool . . . 68
6.4 Eclipse Integration . . . 73

7 Computer Network Modeling Framework 78
7.1 Frameworks . . . 78
7.2 Networking Concepts . . . 79
7.3 Domain View . . . 82
7.4 Packages & Elements . . . 83

8 Optimization Strategies 96
8.1 Motivation . . . 96
8.2 Scenario . . . 97
8.3 Model Design . . . 98
8.4 cTLA Model . . . 101
8.5 Promela Model . . . 105
8.6 Verifier Compilation & Run-Time Options . . . 109

9 Case Study: IP-ARP 111
9.1 Introduction . . . 111
9.2 Modeling . . . 112
9.3 Analysis . . . 117
9.4 Discussion . . . 122

10 Case Study: IP-RIP 125
10.1 Introduction . . . 125
10.2 Modeling . . . 126
10.3 Analysis . . . 132
10.4 Discussion . . . 136

11 Case Study: IP-OSPF 139
11.1 Introduction . . . 139
11.2 Modeling . . . 140
11.3 Analysis . . . 147
11.4 Discussion . . . 153

12 Conclusion 157
12.1 Summary of Contributions . . . 157
12.2 Future Work . . . 158
12.3 Looking Ahead . . . 159

Bibliography 160

A cTLA 2003 Grammar 167

B IP-RIP cTLA 2003 Model 179

List of Figures, Tables, and Listings

List of Figures

2.1 Classification Scheme 7

3.1 Spin Analysis Workflow 20

4.1 Ideal Workflow of Our Approach 39

5.1 Graphical Representation of System SimpleSys 56

6.1 Transforming a Compositional cTLA System to Promela 60
6.2 cTLA2PC Translation Process 71
6.3 Plug-in Architecture of the Eclipse Integration 74
6.4 Interactive Simulation of a Translated Specification in Eclipse . . . 76

7.1 Layers of the TCP/IP Reference Model 79
7.2 Threefold Internet Routing Architecture 80
7.3 Large-Scale Network View 82
7.4 Small-Scale Network View 83
7.5 Framework overview 84

8.1 Modeling Stages & Optimizations 97
8.2 Layered Packet Processing and the Activity Thread Approach . . . 100

9.1 IP-ARP Scenario 111
9.2 Layers and Protocols in the IP-ARP Scenario 112
9.3 Compositional Structure of the IP-ARP Model 116
9.4 Framework and Specific Process Types of the IP-ARP Model 117
9.5 Violating Sequence 1 in the IP-ARP model 121
9.6 Violating Sequence 2 in the IP-ARP model 122

10.1 IP-RIP Scenario 125
10.2 Layers and Protocols in the IP-RIP Scenario 126
10.3 Compositional Structure of the IP-RIP Model 130

10.4 Framework and Specific Process Types of the IP-RIP Model 131
10.5 Example Attack Sequence in the IP-RIP model 136

11.1 IP-OSPF Scenario 139
11.2 Layers and Protocols in the IP-OSPF Scenario 141
11.3 Process Type OSPFRouter 143
11.4 Compositional Structure of the IP-OSPF Model 148
11.5 Framework and Specific Process Types of the IP-OSPF Model . . . 149
11.6 Example Attack Sequence in the OSPF model 154

List of Tables

3.1 Promela Built-In Data Types 25

5.1 cTLA Basic Data Types 49

9.1 Optimization Effects on State-Vector Size in the IP-ARP Example . . 119

10.1 Initial routing table of R1 131
10.2 Optimization Effects on State-Vector Size in the IP-RIP Example . . . 133
10.3 Effects of the Unroll Actions Optimization on a Benchmark Sequence in the IP-RIP Example 134

11.1 Initial routing table of R1 146
11.2 Optimization Effects on State-Vector Size in the IP-OSPF Example . . 150
11.3 IP-OSPF Model File Size Comparison 151

List of Listings

5.1 cTLA Specification Outline 47
5.2 cTLA Simple Process Type 52
5.3 cTLA Extending Process Type 52
5.4 cTLA Example Extending Process Type (Adding) 53
5.5 cTLA Example Extending Process Type (Constraining) 53
5.6 cTLA Subsystem Process Type 54
5.7 cTLA Example Subsystem Process Type 55

6.1 Compositional and Expanded Form of a cTLA Action 61
6.2 Generated Promela Specification Outline 69

8.1 snd_h1 cTLA System Action 101

8.2 Action snd_h1 in the Flat System 102
8.3 Action snd_h1 After Paramodulation 102
8.4 Equalities After Splitting of Parameter pkt 103
8.5 Action snd_h1b After Paramodulation 103
8.6 Action snd_r1 104
8.7 Action snd_r1 After Unroll (Excerpt) 104
8.8 Bit Array Mapping BVSET and BVGET Macros 106
8.9 PacketT with dat Array in the IP ARP Scenario 106
8.10 Assignment Implementation in Promela Without Bit Array Mapping 106
8.11 Assignment Implementation in Promela With Bit Array Mapping . . 107
8.12 Excerpt of Action fwd's cTLA Guard Expression 107
8.13 Part of Action fwd's Promela Guard Expression after Macro Expansion 108
8.14 Part of Action fwd's Promela Guard Expression after Macro Expansion, with Reduce Function Nesting Optimization 109

9.1 Assertion 1: IP-ARP Example, Send Actions 118
9.2 Assertion 2: IP-ARP Example, Receive Actions 118
9.3 Spin Verifier Output in the IP-ARP Example (Assertion 1) 121

10.1 Assertion in the IP-RIP Example 132
10.2 Spin Verifier Output in the IP-RIP Example 135

11.1 Assertion in the IP-OSPF Example 149
11.2 Spin Verifier Output in the IP-OSPF Example 153

1 Introduction

1.1 Background

In recent years, computer networks have been connected worldwide based on open standards to form the *Internet*. Internet-based applications and protocols have been developed rapidly and found wide-spread use both in the private and public sectors. On the one hand, new possibilities for international information exchange and collaboration are opening up. On the other hand, attack opportunities are increasing, too. Concepts in the area of IT security seem to be lagging behind, however. Attack traffic makes up a significant part of all Internet traffic [PYB$^+$04], and the financial impact of successful attacks is significant [GCH03].

Meaningful countermeasures can only be taken with a proper understanding of the attack possibilities, sequences and their impact. Formal methods are generally very well suited in this situation. In the context of computer network attacks, three views have to be considered. First, the processing of packets according to protocols (*protocol view*). Typically, several layered protocols are involved. Second, the nodes with their local initialization and configuration items (*node view*). Furthermore, the nodes may support administrative actions (e.g., change of IP address). Third, topology and connectivity aspects of the networks over which the packets are transmitted (*network view*). These aspects influence packet propagation and routing. Due to the inherent dynamics of computer networks, even small models often exhibit significant complexity. Thus, both formal modeling and analysis are hard to do.

Existing approaches related to computer network security are typically restricted to a single view. Many approaches exist for generic protocol (e.g., [RRCQ03]) and security protocol verification (e.g., [Mea96]). They cover one single protocol in a specialized way, but no node or network aspects. Other approaches take a node-oriented view (e.g., [RS98]), allowing to check for vulnerabilities arising from the interaction of local system components. The modeling of network or protocol related aspects is not considered. Recently, approaches for network vulnerability analysis have appeared (e.g., [OGA05]). These approaches consider attacks combining predefined vulnerabilities on multiple nodes linked according to a connectivity matrix. Thus, they support both a limited node and network view. The protocol view, however, is hardly supported at all. Furthermore, the models are largely static and the analysis is restricted to monotone properties.

Altogether, the approaches lack in flexibility and expressiveness. Particularly,

they are not able to *integrate multiple* views of a computer network scenario in one single model. Instead, multiple approaches are required for considering a single scenario. Even worse, the approaches use different formalisms and require multiple models to be devised. As a result, careful work has to go towards ensuring consistency between the models. Similarly, different tools and mechanisms are necessary for analysis. Attacks involving multiple views cannot be analyzed and dynamics is severely restricted. Overall, the efforts required for modeling and analysis are multiplied at best and the scenario cannot be handled otherwise at all. To sum it up, existing approaches are not satisfactory.

1.2 Goal Statement

The main goal of this thesis is the development of a new integrated approach for the formal modeling and automated analysis of computer network attack models. The approach must accomplish the following subgoals:

- Formal Modeling: Models have to describe systems in a clear and precise way.
- Integration of Multiple Views: Protocol, node, and network views of a scenario have to be integrated in a single, consistent model.
- Executable Models: Models have to be executable so that their behavior can be traced and validated (e.g., by interfacing with an interactive simulation tool).
- Support for Dynamics: Both the modeling and the analysis must be able to cope with dynamic changes (e.g., logical connectivity between nodes is not static but depends on dynamic routing decisions).
- Automated Analysis: Attack sequences have to be found automatically (i.e., by a tool not requiring user input) by checking for violations of properties specified with the model.
- Ease of Use: The approach shall facilitate both the modeling and analysis task (e.g., by providing libraries, tool support etc).

Finally, the practical feasibility of the approach will be demonstrated by its application to several case studies.

1.3 Thesis Outline

Beginning with the introduction (chapter 1), the thesis is divided into three parts and twelve chapters. The main part of the thesis introduces our approach for integrated formal modeling and automated analysis of computer network attacks. The

approach combines cTLA 2003, a computer network framework, a compiler, optimization strategies and the SPIN tool for automated analysis. Furthermore, an ECLIPSE [OTI03] based modeling and analysis environment and a workflow for applying the approach is provided.

The *first part*, chapters 2 and 3, provides background material required for understanding the thesis. Chapter 2 classifies the related work into protocol-, node-, and network-oriented approaches. Selected approaches from each area are considered with respect to their suitability for integrated modeling and analysis of computer network attack models.

Chapter 3 gives an overview of SPIN, PROMELA, TLA, and cTLA. SPIN [Hol03] is a powerful model checker for analyzing models written in PROMELA. PROMELA TLA [Lam94], and cTLA [HK00] are specification languages for distributed systems. Furthermore, the underlying machine models defining the semantics of the specifications are outlined.

The main or *second part* of the thesis, chapters 4 to 8, describes our integrated formal modeling and automated analysis approach. Chapter 4 gives a bird's eye view of the approach. The chapter details the objectives of our approach, then outlines implementation, workflow, and modeling steps.

Chapter 5 describes cTLA 2003 [RK03], which we use as the modeling language for our approach. In contrast to cTLA, cTLA 2003 provides executable specifications and adds modeling enhancements to foster reuse and to ease the modeling task. Furthermore, the chapter explains cTLA 2003 specification structure, semantics, and grammar.

Chapter 6 explains the scheme for translating cTLA specifications to the more low-level PROMELA specifications. Then, the architecture of the cTLA2PC compiler tool, which implements the scheme, is described. The chapter concludes with a brief overview of the ECLIPSE integration we engineered for cTLA2PC and SPIN.

Chapter 7 covers the computer network modeling framework we devised to greatly simplify the modeling task. After a short introduction to frameworks and an overview of the networking concepts related to our application domain, the network, node, and protocol views taken by the framework are described. Finally, the packages and classes of the cTLA framework are presented.

Chapter 8 deals with optimization strategies for models and their implementation in the approach in order to alleviate state space explosion effects. Optimizations have to be considered at all stages, from the scenario level to the PROMELA level.

The *third part*, chapters 9 to 11, demonstrates the feasibility of our approach. To this aim, three cases studies following our approach are presented. The case studies include routing at different levels (cf. section 7.2.2), in order to show that our approach is able to deal with the inherent dynamics.

Chapter 9 describes the modeling and analysis of a LAN scenario involving several nodes and the IP and ARP protocols. ARP is a low-level "routing protocol"

employed inside a LAN. The analysis shows an interesting similarity between ARP attacks and certain administrative actions.

Chapter 10 presents a multi-LAN IP and RIP scenario. RIP is a distance-vector type routing protocol used for routing between LANs belonging to the same organization (interior-gateway). The scenario contains multiple LANs, host nodes, and router nodes. Attack sequences are influenced by the packet propagation which in turn depends on the network topology.

Chapter 11 applies our approach to the largest scenario so far involving IP and OSPF. OSPF is a complex link-state interior-gateway routing protocol. The scenario includes different network and router types supported by OSPF besides the host nodes. During analysis, we encountered interesting limitations of the SPIN and GCC tools. Furthermore, to the best of our knowledge, our analysis is the first formal consideration of OSPF security properties.

The thesis concludes with chapter 12, which summarizes the main contributions. Furthermore, an outlook for future work is given.

1.4 Publications

Parts of the research and results of this thesis have already been published, in particular as conference proceedings. In this section, we list the publications and the respective contributions of the author of this thesis.

cTLA 2003 Description This technical report [RK03], written by the author of this thesis and Heiko Krumm, contains a preliminary version of the syntax and semantics of CTLA 2003 as well as the relationship to TLA and CTLA 2000. While CTLA 2000 was developed by Peter Herrmann, Heiko Krumm et al. [HK00], the author invented and contributed the sections on CTLA 2003.

Chapter 5 gives an updated and more detailed description of CTLA 2003. The relationship to TLA, CTLA 2000 is explained in sections 3.3, 3.4, respectively, of chapter 3.

Analyzing Network Management Effects with Spin and cTLA The paper [RPK04], published in collaboration with Andre Pohl and Heiko Krumm, reports on the application of an early version of the integrated formal modeling and automated analysis approach to an IP-ARP LAN scenario. All key parts of the paper, particularly the description of the approach (cf. section 4)), the generic model structure, the example scenario and its modeling, optimizations, and analysis, were contributed by the author. The scheme for translating CTLA to PROMELA (cf. section 6.2) was devised by the author as well; however, Andre Pohl provided a detailed implementation of the scheme with the CTLA2PC translation tool.

Chapter 9 contains a detailed description of the IP-ARP modeling and analysis. The translation scheme and the cTLA2PC tool are described in chapter 6. Optimizations are summarized and categorized together with those found during the modeling and analysis of other scenarios in chapter 8.

cTLA Computer Network Specification Framework This online document [Rot04], written by the author, describes the cTLA computer network modeling framework. The framework is a refined version of the generic model structure introduced in [RPK04].

In chapter 7, an updated and extended presentation of the framework together with the key networking concepts relevant to the application area is given.

Formale Modellierung und Analyse protokollbasierter Angriffe in TCP/IP Netzwerken am Beispiel von ARP und RIP The paper [RK05a] (in German), by the author and Heiko Krumm, introduces a new IP-RIP scenario and compares its modeling and analysis to the IP-ARP scenario. Furthermore, a brief overview of the framework is given. As with the IP-ARP scenario, the key parts regarding modeling and analysis of the scenarios were written by the author.

A detailed description of the IP-RIP scenario is contained in chapter 10.

Using Spin and Eclipse for Optimized High-Level Modeling and Analysis of Computer Network Attack Models The paper [RKK05], written by the author, Tobias Kneiphoff, and Heiko Krumm, focuses on the optimized translation of cTLA code to PROMELA as required by the model checker SPIN. Furthermore, the composition of both the IP-ARP and IP-RIP scenarios from framework types and model-specific types is shown. Last but not least, the plug-ins for integrating the approach into the ECLIPSE environment are presented. Again, the author of this thesis contributed the key parts regarding the translation, optimization, and composition of the scenarios. Regarding the plug-ins, the author did the initial research on how to integrate SPIN and cTLA2PC with ECLIPSE. The detailed design and implementation of the plug-ins, however, were done by Tobias Kneiphoff as part of his master's thesis [Kne04].

The ECLIPSE integration is outlined in section 6.4. Details of the translation from cTLA to PROMELA are contained in section 6.2.

A Framework-based Approach for Formal Modeling and Analysis of Multi-Level Attacks in Computer Networks This paper [RK05b], published in collaboration with Heiko Krumm, explains the IP-RIP scenario modeling and the framework in more detail. Furthermore, the new unroll action parameters optimization and its effect on the scenario are examined. As before, all key parts of the paper were contributed by the author of this thesis.

The unroll action parameters optimization is described in section 8.4.2. Chapter 10 gives an in-depth view of the IP-RIP scenario.

2 Related Work

Several formal modeling and analysis approaches that are related to the context of computer network security exist. In this chapter, we give an overview of the key approaches. We begin with a short explanation of our classification scheme. Then, we outline the related work according to this scheme. After that, we give a short survey of practical security tools. Finally, we briefly summarize the advantages and limitations of the existing approaches.

2.1 Classification Scheme

Existing approaches largely differ in the areas covered and level of detail provided for these areas. To better compare and classify the approaches, we devised a decomposition into three views naturally related to computer network attack models (cf. Fig. 2.1).

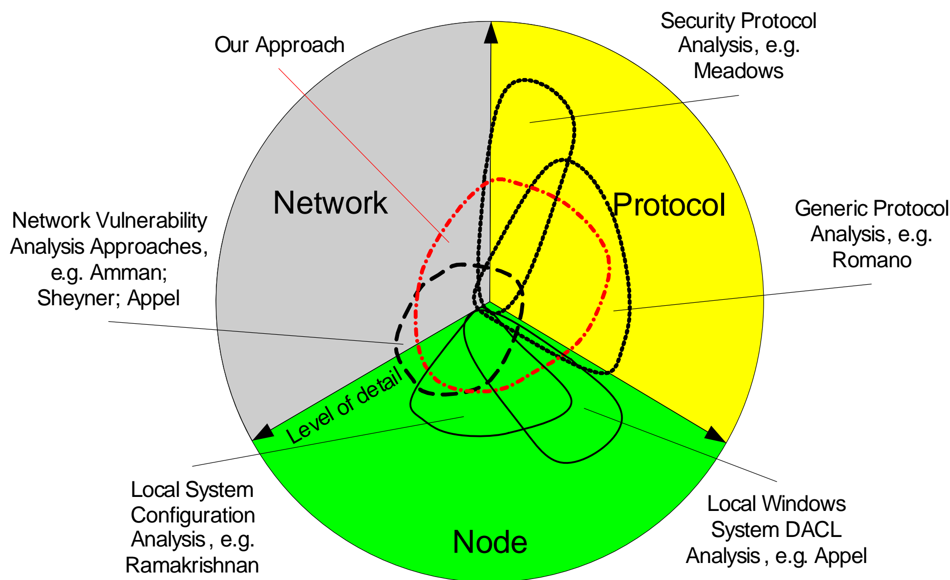


Figure 2.1: Classification Scheme

First, the *protocol view* defines the packet processing of the nodes and the types of packets exchanged between the nodes. Typically, several layered protocols are

running on top of each node. Second, the *node view* takes into account the local configuration (e.g., IP address, routing table, services) as well as the initialization and administration actions (e.g., for setting up interfaces) of a node. Third, the *network view* deals with topology and connectivity aspects of the logical or physical media. These aspects define broadcast zones, influence packet propagation and routes etc.

We depict these views as sectors of a circle. The different types of approaches are symbolized by areas inside the circle, which extend inside one or multiple sectors according to the extent the different views are covered. The distance from the center gives a hint about the level of detail that can be provided by an approach. For example, security protocol analysis approaches can model and analyze security protocols from a low to a very high level of detail. However, the coverage reached for aspects related to the network and node views, is very low and highly abstract (i.e., low detail). Furthermore, in contrast to generic protocol analysis, security protocol analysis covers a smaller fraction of all protocols.

Most approaches are very specialized towards one view. *Network vulnerability analysis (NVA)* approaches take a broader view. Thus, they are particularly interesting to us and we carefully consider several of them. Our classification scheme helps to highlight the similarities and differences despite the varying formalisms. Some approaches even changed formalisms as they evolved. For example, the approach by Amman et al. [AR00; RON02; JNO03] started with a logical model representation and the SMV model checker as the analysis tool and turned into an approach based on a graph based model with a custom analysis engine.

2.2 Protocol-Oriented Approaches

Generally, in *protocol verification* a protocol is analyzed with respect to a certain property (e.g., the reliable delivery of messages) under reasonable assumptions (e.g., media loss is bounded). *Security protocol verification* is specialized on analyzing security properties of cryptographic protocols (e.g., authentication protocols).

A variety of methods is applied in both fields, including classic logic and algebraic calculi (e.g., [KKN+03]), special calculi (e.g., [BAN90]), and process system modeling techniques (e.g., [RRCQ03]). Different kinds of analysis tools are used, including logic programming environments, expert system shells, theorem provers, algebraic term rewriting systems, and model checkers. Often, these tools are highly specialized to support their approach. With respect to our classification, node and network related aspects are mostly out of scope for protocol verification approaches.

As there are many approaches and whole conferences dedicated to the subject of protocol verification, we can only exemplify the work done in this area. We first outline a protocol verification approach based on process system modeling, then two

security protocol verification approaches combining multiple analysis techniques.

Romano et al.: Protocol Verification Romano et al. [RRCQ03] suggest a protocol verification approach based on a client, server, and network process for each protocol command. Modeling is done using the PROMELA specification language; for analysis the SPIN (cf. section 3.1) model checker is employed. As an example, they describe the modeling and verification of the sessionless mode of the *reliable HTTP* (HTTPR) messaging protocol.

In *sessionless mode*, no state information is kept between commands. Thus, for each protocol command an independent PROMELA model can be built. The client and server processes deal with the processing of the protocol commands; the network process models a lossy channel between client and server. The property considered is reliable message delivery, i.e., each message is delivered exactly once. Their analysis is greatly simplified by the sessionless mode: Each command model can be analyzed separately. Furthermore, all variables not involved in the property are removed prior to analysis. Using the SPIN model checker, the property is successfully verified.

Regarding our classification, the protocol view is covered with a medium to high level of detail, taking into account packets and protocol commands. Node and network aspects are covered to a small extent by e.g., representing the network with a channel.

Meadows (NRL): Cryptographic Protocol Verification Following the approach by Catherine Meadows [Mea96], cryptographic protocols are specified as FSMs with algebraic reduction rules. The approach is based on an extended *Dolev-Yao* [DY83] model. Analysis is done in an interactive way by combining different analysis techniques using a custom tool, the *NRL protocol analyzer*. The approach has been successfully applied to multiple protocols. For instance, a draft version of the *Internet key exchange protocol (IKE)* [HC98] was analyzed and several ambiguities and omissions were found [Mea99].

The NRL tool, written in PROLOG, particularly supports the following techniques:

- Backwards search (interactive and heuristic) from a user specified goal state (e.g., intruder knows the session key) to find preceding states
- Rewriting based on algebraic reduction rules that capture abstract properties of cryptographic algorithms, e.g., $D_K(E_K(w)) = w$ (decrypting and encrypting based on the same key are inverse operations)
- Pruning of states based on a database (facts) of reachable states

- Pruning of the set of producible words by a user specified language of unreachable words. An algorithm that can prove unreachability in many cases is integrated.

Both the intruder goal and the protocol modeling are more flexible than with the Dolev-Yao model: Any state can be specified as the intruder goal; and intruder actions like the decryption and encryption of already known words can be added as transitions rules to the protocol modeling.

In comparison to the approach by Romano et al., this approach is on the one hand very specialized on cryptographic protocols and (typically) requires user interaction during analysis. On the other hand, the approach has helped to uncover flaws in several different cryptographic protocols.

Armando et al. (AVISPA): Security Protocol Verification The *automated validation of Internet security protocols and applications* (AVISPA) [ABB+05] project by Armando et al. is a large-scale public-private partnership, funded by the EU. AVISPA is based on its own high-level security protocol modeling language, which can be translated for use with different back-end analysis tools.

The *high level protocol specification language* (HLPSL) is used to model security protocols together with their desired security properties. HLPSL supports different process types (called *roles*) which can be composed to build the system. Furthermore, special primitives (e.g., hash function, symmetric key) which ease the modeling of cryptographic protocols are included. Security properties (called *goals*) are defined using predefined constructs (e.g., authentication, secrecy). Using a translator, HLPSL specifications are transformed to the formalism required by the analysis back-end. Particularly, different model checkers are supported. A library of security problems (i.e., security protocols with properties) analyzed using AVISPA is available.

In comparison to the approach by Catherine Meadows, AVISPA, on the one hand, has a more powerful language, is more flexible regarding the choice of the back-end tool, and provides a higher degree of automation (depending on the back-end). Common operations of security and cryptographic protocols are supported by built-in constructs of the modeling language. One the other hand, these high-level constructs have to be adequately translated to the formalism used by the respective back-end. This might not be possible for all back-ends and hinder analysis.

2.3 Node-Oriented Approaches

Node-oriented approaches consider properties like the role (e.g., server, client, attacker), local configuration (e.g., IP address, routing table, installed system components or services), initialization and administration actions (e.g., for setting up

interfaces) of a node. They typically check for vulnerabilities arising from the interaction of local configuration items.

Ramakrishnan and Sekar: Local System Component Vulnerabilities The work by Ramakrishnan and Sekar [RS98] describes the analysis of attack sequences resulting from the combined behavior of local system components of a single Unix host. Modeling is done using a PROLOG variant and analysis is based on the XSB [SSW+05] logic programming environment.

The system model consists of process models for system components and channels. Process models for a file system, mail send program, mail display program, and printer spooler are designed. The user is defined to always execute the sequence: read a file, write a file, print, send mail. All process models are very simple, e.g., the file system model only includes two operations that read or write to a channel variable after checking the `access` predicate. The mail display program has a vulnerability that can be exploited in connection with the mail send program that allows an attacker to overwrite files (e.g., the `passwd` file).

Analysis is done by using appropriate queries. For example, a query of the form `fs.write(passwd,_)` is resolved to sequences resulting in a write operation on the `passwd` file. The queries are analyzed using XSB, a PROLOG programming environment employing tabled resolution. *Tabled resolution* improves efficiency and allows the system to terminate where the depth first search based resolution employed by standard PROLOG fails.

In [RS02], the approach is slightly extended and described in more detail. This time, first a high-level model of the system is built (using a custom language based on CSP). To use the high-level model with XSB, it is translated manually. The composition of the system model from the process models is described in more detail. Furthermore, the process models are extended with further operations. Particularly, the file system model supports an operation for resolving symbolic links.

The system model describes a single node with a few highly simplified system components. Furthermore, the model seems quite geared towards the file vulnerability. As this is a local vulnerabilities approach, network and protocol related aspects are not considered.

Appel, Govindavajhala et al.: Local DACL Vulnerabilities Recently, Appel, Govindavajhala et al. [GA06] have come up with another approach based on a PROLOG model and the XSB logic programming environment. Their model deals with specific rights given by entries of discretionary access control lists (DACLs) on files and services as defined for Microsoft Windows systems. Analysis is done for local privilege escalation attacks due to improper ACL configurations.

Their model has a small set of rules (i.e., PROLOG clauses) around the `service_change_config` and `write_dac` rights. A principal with `service_-`

change_config right on a service may substitute the file executed by the system to provide the service with its own file. In this case, if the service is run under a privileged account (i.e., LocalSystem), the principal succeeded in extending its privileges. The write_dac right allows to give arbitrary rights on a resource to a principal, especially the service_change_config right. Furthermore, the model includes a rule stating that if a principal is a member of a group, and the group has some right, then the principal has the right as well.

Besides the rules, the model contains a set of facts. These facts excerpt the rights given by the DACLs on services and files. By using a custom scanner tool on a real Windows XP host, the facts are populated automatically. This allows them to analyze this specific case of local configuration vulnerabilities easily.

For analysis, some account, e.g., the guest account or an account from the Authenticated Users group, is assumed to be compromised by the attacker. Then, analysis is done using queries. For example, a query of the form compromised(localSystem) results in sequences listing steps of how accounts can gain localSystem access. In the attack graph shown in the paper, these sequences appear to be two-step at most: first group membership is applied, then a privilege escalation through some access control entry.

This approach deals with a very specific case of local system vulnerabilities: improper access control lists. For that case, the approach is highly automated and quite elegant. It does not cover network or protocol related aspects, however. The authors state that the DACL approach is a special case of the MulVAL approach described below (cf. section 2.4). Indeed, both approaches are based on the same modeling language (PROLOG) and logic programming environment (XSB). Apart from that, the approaches have little in common, however. Particularly, the models are quite different and are not based on an integrated set of clauses and facts. Furthermore, the properties stated for MulVAL – multihost and multistage – do not apply well to a local host DACL approach with few steps for a successful attack.

2.4 Network-Oriented Approaches

Formal network vulnerability analysis (NVA) approaches have emerged quite recently. They try to find sequences of attacks on single nodes that finally lead to the violation of a security property (e.g., no root access on another node). The attacker starts from a certain node and may switch to other nodes after a successful attack.

NVA approaches generally take an abstract, global network and node view. Typically, the network view is reduced to a static connectivity matrix. Physical and logical network layers are not distinguished; the transfer of packets is not considered. The node view consists of a set of constants symbolizing the services and/or vulnerabilities of the node. Moreover, a set of attack or exploit rules depending on the vulnerabilities contained in the modeling is defined. Regarding protocols, they are

at most "modeled" by a protocol specific constant value in the connectivity matrix.

In short, NVA approaches offer some integration between different views, particularly network and node. The view is very abstract, however. Furthermore, analysis can only uncover *new combinations* of the already *known* vulnerabilities listed for each node. In the following paragraphs, we present the most important NVA approaches to date.

Amman, Ritchie et al.: Model Checking for NVA The approach by Amman, Ritchie et al. [AR00] was one of the first NVA approaches. Their modeling is based on a logical representation which is analyzed using the SMV [CMU01] model checker.

The node modeling consists of a set of booleans indicating which of the predefined vulnerabilities do exist and of an integer representing the attacker access level (none, user, root) on the node. A boolean connectivity matrix is used to model the network. To represent the exploitation of a vulnerability, matching exploit rules have to be included. An *exploit rule* consists of preconditions (connectivity, existing vulnerabilities and access level) and effects (increased access level, new vulnerabilities). Finally, a security property, e.g., no root access level on a specific node, is stated and the model is analyzed using SMV. SMV tries possible combinations of the vulnerabilities defined in the model as allowed by the exploit rules. If the property is violated, the corresponding attack sequence is put out.

In [RON02], the approach – now called *topological vulnerability analysis* (TVA) – is extended and described in more detail. The connectivity matrix underlying the network model may now contain integer values instead of boolean. These values symbolize the protocol or service accessible through the connection. The authors suggest a constant naming scheme for these values with a prefix corresponding to the layer of the TCP/IP reference model (cf. section 7.2.1) the protocol belongs to. For example, the constant `TRANS_WU_FTPD` represents a transport layer connection to a WuFTP daemon.

Due to the model-checker based analysis, complex properties can be examined. The simplicity of the model, however, limits the usefulness of complex properties a bit. A node's configuration and components are only represented by booleans. Protocols are symbolized through the values already described in the connectivity matrix. No sending, receiving, or processing of packets or protocol commands is modeled. Thus, the level of detail of the protocol modeling is very low. Regarding network aspects, the connectivity matrix allows modeling on a medium level of detail, however.

Jajodia, Noel et al.: Dependency Graph Based NVA In [JNO03], Jajodia, Noel et al. present an extended and partially changed version of the TVA approach. Now, a specific dependency graph [AWK02] is generated for analysis. Accordingly, instead

of the SMV model checker, a custom graph analysis engine is used. Furthermore, the modeling is based on XML files which are partially created automatically.

In the *dependency graph*, exploits (e) and conditions (c) are expressed as vertices, dependencies as edges. An edge from a c-vertex to an e-vertex is labeled with the preconditions of the exploit. An edge from a e-vertex to a c-vertex is labeled with the postconditions (i.e., effects) of the exploit. The graph is generated prior to analysis by custom translation tools from XML input files.

The network model (connectivity matrix) is encoded in `network.xml`; the vulnerabilities of the nodes are listed in `attack.xml` and the exploit rules and security conditions in `conditions.xsl`. The vulnerabilities file is created automatically based on the output of the NESSUS network vulnerability scanner. Furthermore, the `network.xml` may be created automatically based on the output of network discovery tools. The exploit rules for the included vulnerabilities and the security condition, however, still have to be manually defined by an expert.

In comparison, on the one hand, this graph based approach scales better to larger networks. It has a polynomial run-time. On the other hand, the older model checker based approach is more expressive. Particularly, the graph-based approach assumes *monotonicity*, i.e., if a condition is true once it is assumed that it stays true forever. Of course this simplifies analysis greatly. Models involving routing changes, dynamic address assignment or administrator actions disabling services violate this assumption. Likewise, the old approach allowed security conditions to involve the full set of mechanism supported by SMV (e.g., temporal properties). The type of security conditions supported by the new approach is not described in detail, but seems to be restricted to propositional predicates.

Sheyner, Wing et al.: Attack Graphs for NVA The approach by Sheyner, Wing et al. [SHJ+02] is based on XML files describing the network model, vulnerabilities of the nodes, atomic attacks and a security property. These input files are encoded into a finite state machine (FSM) model. The FSM model is analyzed by a variant of SMV which automatically generates an attack graph as result. All paths violating the security property are depicted in the attack graph.

An *attack graph* is a directed graph where each vertex is labeled with an attack identifier, source host, and target host. The attack identifier represents an atomic attack. An *atomic attack* stands for the exploitation of a vulnerability on the target host from the source host. The attack graph is both exhaustive and succinct, i.e., it shows exactly those vertices contributing to the violation of the security property. Root nodes correspond to atomic attacks possible from the initial state. An edge between two vertices means that the atomic attacks can be executed in the order indicated by the edge. Thus, the paths from the root nodes to the leaf nodes are sequences of atomic attacks finally leading to the violation of the security property.

The network model is similar to the connectivity matrix known from Amman.

Additionally, the node model, i.e., the services running on the nodes are included in the connectivity matrix as well. Again, attacks consist of preconditions and effects. From all these elements, the FSM model is generated. It has state variables for the attack identifier, source host, and target host. During each step, these variables are nondeterministically set to a value out of the defined values for the model. Then, if the preconditions for the atomic attack corresponding to the attack identifier are satisfied, the attack is executed, i.e., the state of the FSM model is changed according to the effects. This way, all possible combinations of atomic attacks are enumerated.

Due to the use of the SMV model checker, security properties can be more complex than with the graph-based approach by Amman. On the downside, if complex (e.g., temporal) properties are used, the construction of the attack graph can take exponential time.

Appel, Govindavajhala et al (MulVAL): Logic Programming for NVA This work is by the same authors as the Windows DACL local configuration vulnerabilities approach already described (cf. section 2.3). The approach is called MulVAL [OGA05], meaning multihost, multistage, vulnerability analysis. The network and node model are in principle very similar to the NVA approaches described above. Like in the dependency graph approach by Amman and the Windows DACL approach, the creation of the node model can be partially automated using a vulnerability scanner. Additionally, this approach partially automates the generation of attack preconditions and effects.

All modeling is done in a PROLOG variant. The node model includes a list (i.e., PROLOG facts) of services and vulnerabilities on the node. Vulnerabilities include two attributes stating the scope (either `localExploit` or `remoteExploit`) and effect (either `privilegeEscalation` or `denialOfService`) of a successful exploit in a very abstract way. The authors provide a modified vulnerability scanner which can – in most cases automatically – populate the node model. Furthermore, additional information like client programs can be added. A connectivity matrix is used for the network modeling. This matrix includes protocol and port constants.

The exploit rules are in more generic form than with the other NVA approaches. They are not dependent on a specific vulnerability but on the attributes of a vulnerability. For example, a remote-code execution rule can be defined for all vulnerabilities that have the `remoteExploit` attribute and are contained in a service that is running on a host the attacker can access.

Analysis is done using PROLOG queries. Thus, no complex security properties are possible. Like the dependency graph NVA approach, monotonicity is assumed and analysis takes polynomial time. The approach has its limitations but is overall well-rounded. In comparison to Sheyner, Wing et al., the node modeling is more detailed and highly automated. Furthermore, the attack rules are slightly more generic.

2.5 Network Security Tools

There is a range of tools for practical network security testing. These tools can be grouped into different categories: vulnerability scanners, penetration testing tools, and intrusion detection systems.

Vulnerability Scanners These tools are used to detect known weaknesses (e.g., missing patches) on hosts. Typically, vulnerability scanning tools are network-based. NESSUS [Nes06] is a prime example. Using NESSUS, the scan can be done from a daemon running on a central server; no host-based agent is required. The detection capability is realized via plug-ins. NESSUS provides, for instance, a set of plug-ins corresponding to the security bulletins published by MICROSOFT. This way, wide-spread vulnerabilities existing on the nodes can be recognized and remedied manually (e.g., by patching) later on.

Some NVA approaches (cf. section 2.4) populate their node models using the output of vulnerability scanning tools. Unfortunately, the recognition of weaknesses is not always reliable. One reason is that the weakness may depend on complex conditions that are not considered by the vulnerability scanner (e.g., configuration of the local host or its environment). Furthermore, a reliable scan may impact the host being scanned too much.

Penetration Testing Tools After possible weaknesses have been identified using a vulnerability scanner, penetration testing tools can be used to check if these weaknesses are indeed exploitable. The METASPLOIT tool [Met06] provides a library of exploits for common vulnerabilities. These exploits can be combined with payloads (e.g., remote command shell) depending on the operating system of the target etc. Some exploits are architecture independent, e.g., routing protocol exploits, and do not require a payload specific to an operating system.

NEMESIS [NS04] is a more generic tool that, however, requires expert knowledge. It allows to assemble custom packets (e.g., from the command line or via scripts) for many different protocols. This is particularly useful for testing protocol-related weaknesses. Often, sequences of packets containing unusual, reserved, or random values are used. Besides weaknesses, deviations from the standard may be discovered. Of course, penetration testing tools can only show that a weakness is exploitable; they cannot prove that a weakness is not exploitable, even if the penetration testing tool fails.

Intrusion Detection Systems (IDS) An IDS typically comprises sensors, a rule engine, and a management console. The *sensors* generate events which are processed by the *rule engine*. If a rule matches, an alert is triggered and displayed on the *management console*.

IDS can be classified according to the type of event history analysis: attack signatures or anomaly detection. *Attack signatures* are patterns representing known bad behavior. They are stored in a library which has to be updated regularly as new attacks become evident. In order to achieve some flexibility, regular expressions may be included in the attack signatures. *Anomaly detection* works by comparing selected properties of the current behavior with previously learned regular behavior. Typically, to establish the properties of regular behavior, network traffic is recorded for a certain period of time. The recorded traffic is then analyzed for the type and distribution of the packets occuring. More advanced methods like machine learning techniques may be used, too.

Some IDS support both attack signatures and anomaly detection. On the one hand, IDS supporting anomaly detection have greater chances to recognize that "something bad" happened. On the other hand, they produce much higher rates of false positives than attack signature based IDS. A *false positive* is an alert even though no attack has occured. Furthermore, performance is a serious concern of anomaly detection based IDS in high volume environments.

The location and scope of the sensors offers an alternative way to distinguish three types of IDS: First, *network intrusion detection systems (NIDS)* use sensors to watch the network traffic and to monitor multiple hosts. NIDS are the most widespread type of IDS, with SNORT [Sno05] and BRO [LBN04] two well-known open source implementations exist. Second, *host-based intrusion detection systems (HIDS)* employ sensors that are located on the hosts and may watch log files, system files, and system calls. Third, *hybrid intrusion detection systems* combine both host and network based sensors and try to correlate the events. In contrast to the tools discussed above, IDS are typically only able to detect attacks after their occurence. They cannot be used to analyze certain aspects of a node, network or protocol beforehand.

Intrusion prevention systems (IPS) try to stop attacks before they happen. IPS combine multiple techniques in one system, for instance access control (firewall), intrusion detection, and prevention of certain types of buffer overflows. An example of an IPS with these capabilities is the commercial product BLINK [eEy06] by EEYE. On the one hand, combining multiple techniques in principle protects against more attacks than a single technique. On the other hand, this combination of different techniques into one system greatly increases the complexity. Thus, the IPS has to be designed and implemented very carefully to not introduce weaknesses of its own. As Bruce Schneier puts it [Sch00]: *Complexity is the worst enemy of security. Secure systems should be cut to the bone and made as simple as possible.*

2.6 Discussion

On the one hand, there are many approaches for modeling and analysis of node, protocol, or network security aspects. Additionally, tools for practical network se-

curity testing and intrusion detection are available. On the other hand, existing approaches are very specialized towards one of the node, protocol, or network views. Thus, these approaches cannot provide an integrated view of computer network attack scenarios. Depending on the approach, further limitations often apply during analysis. The user must, for instance, guide analysis interactively, specify a somewhat arbitrary goal state as a starting point, or may be restricted to monotone properties.

Protocol-related approaches (cf. section 2.2) take a very abstract network and node view. Approaches covering the node view (cf. section 2.3) hardly incorporate a protocol or network view. NVA approaches (cf. section 2.4) model network (e.g., connectivity matrix) and node aspects (e.g., list of services or vulnerabilities), but only with a low to medium level of detail. The protocol view (e.g., processing of packets and protocol commands) is hardly modeled at all. The models are largely *static*: They do not support dynamic services, routing or firewall rules. Analysis with logic programming tools is restricted by the monotonicity property and has to be done using queries. Finally, only new combinations of the already known vulnerabilities listed for each node are found.

We aim to develop a new approach that integrates the node, protocol, and network views with a medium to high level of detail in a single consistent model and can predict attack sequences in dynamic scenarios. Analysis shall be automatic and allow dynamic properties to be specified. None of the existing approaches combines these qualities. For realizing this aim, a general, more expressive modeling language is required. Of course, the resulting models will be significantly more complex than models limited to a single view. Analysis of complex models is hard to achieve due to state space explosion effects. Furthermore, general analysis tools cannot make use of special case shortcuts that apply to more specialized models. Automatic analysis not aided by interactive user guidance is even harder. Thus, our aim is very challenging. Certainly, an eye has to be kept on the scenario size and attacker verboseness. Moreover, optimization possibilities have to be considered at all stages.

3 Spin, Promela, TLA, and cTLA

In this chapter, we give a brief overview of SPIN, PROMELA, TLA and CTLA. First, we describe the SPIN model checker. Then, we introduce SPIN's specification language, PROMELA. Finally, we explain the structure and key concepts of TLA and its compositional variant CTLA 2000.

3.1 Spin

In the following paragraphs, we outline SPIN's most noteworthy properties. A comprehensive presentation of SPIN is contained in Holzmann [Hol03].

3.1.1 Overview

SPIN is a tool for the automated verification of distributed software systems, also called a *model checker*. Its development started in 1980 at BELL LABS. From 1991 on, it has been freely available as *open source*. Throughout the years, SPIN was continuously adapted to new developments and extended with new features. Nowadays, SPIN is maintained at NASA JPL.

SPIN is widely recognized as one of the most powerful and most popular model checkers. In 2002, SPIN was decorated with the prestigious ACM SOFTWARE SYSTEMS AWARD. Applications of the SPIN tool include mission-critical software and call processing software. For example, SPIN was used to verify selected algorithms of NASA space missions. Regarding industrial applications, large parts of LUCENT'S PATHSTAR call server software were verified using SPIN.

Analysis Workflow The basic analysis workflow with SPIN (cf. Figure 3.1) is as follows: First, a system description (or *specification*) is written in SPIN's input language, PROMELA (cf. section 3.2). This description includes claims (cf. section 3.1.3) about the system. The description is then parsed and checked for syntax errors. If no syntax errors are detected by the PROMELA parser, the specification can be analyzed using two basic basic modes of operation: simulation and verification.

In *simulation mode*, the specification is executed by SPIN until no more statements are executable, an assertion fails, or the simulation is stopped by the user. Executed statements and current values of global variables are printed to the console. As

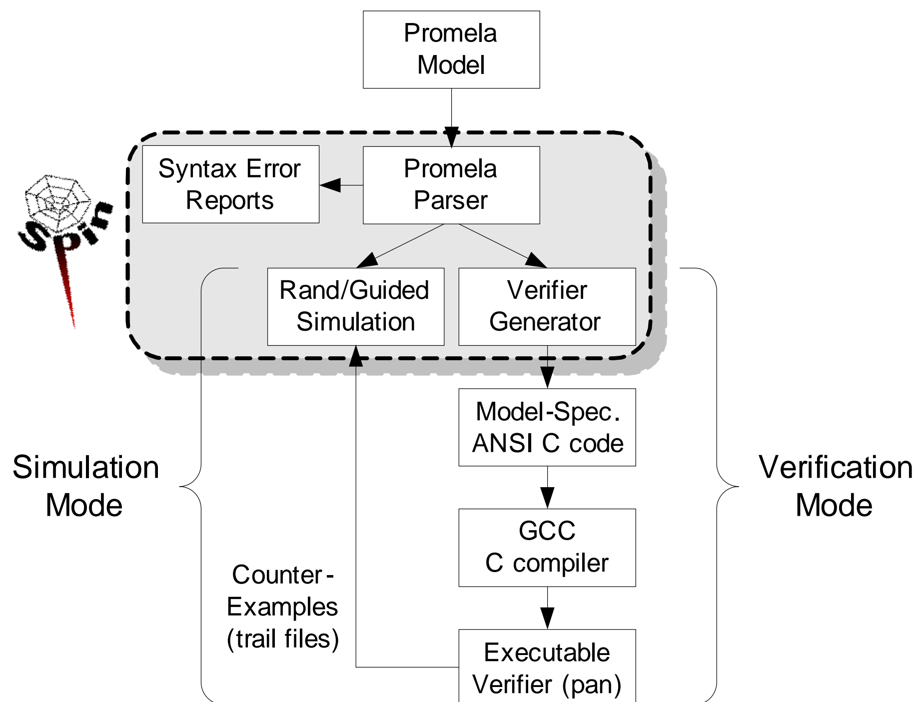


Figure 3.1: Spin Analysis Workflow

SPIN specifications are non-deterministic, there may be several possible (i.e., non-blocked) statements which could be executed next. Selection between these choices depends on the specific simulation mode. The user interactively selects the next statement in *interactive simulation* mode. In *random simulation* mode, one of the possible next statements is selected randomly. Finally, in *guided simulation* mode, the statements are chosen according to the sequence stored in a trail file. Typically, trail files are generated from violating sequences during verification.

In *verification mode*, SPIN hands the PROMELA specification to the verifier generator. This module generates the ANSI C code for a verifier. Depending on the given options (cf. section 3.1.4), a verifier for either exhaustive verification or approximative verification is generated. The verifier source code is then handed over to the GCC C compiler. GCC compiles the verifier source code and creates an executable verifier. This *executable verifier* is a stand-alone program which does neither depend on SPIN nor on the PROMELA model specification anymore. By running the verifier, the original model is checked against its claims.

3.1.2 System Representation & Optimization

SPIN creates an internal representation of the specified system and the claims to be checked as follows: First, for all processes of the system, SPIN constructs a labeled

finite state machine P_i (cf. section 3.2). Then, the asynchronous product of the P_i is calculated yielding the system finite state machine (FSM). An *asynchronous product* follows *interleaving semantics*, i.e., all transitions of the product FSM correspond to a transition of only one of the component FSMs. All other component FSMs perform a stuttering step during that transition.

Furthermore, a Büchi automaton is constructed which accepts the *negation* of the claims. A *Büchi automaton* extends the concept of FSMs to accept infinite sequences. The synchronous product of the system FSM and the Büchi automaton is calculated, yielding the final automaton. In a *synchronous product*, all transitions are joint transitions of the component automata. The final automaton is then analyzed for its acceptance set. If the acceptance set is non-empty, the claims can be violated.

Partial Order Reduction The generated automatons tend to be very large. SPIN by default performs partial order reductions to reduce the number of states and transitions. The basic idea of *partial order reduction* is that if two adjacent operations o_1, o_2 are independent, their execution order does not matter, i.e., both o_1, o_2 and o_2, o_1 yield the same result. Then, any fixed ordering of the operations can be chosen as a representative and the other one can be removed from the state graph. This idea can be extended to n operations easily. The main difficulty is to determine which operations are independent. Operations that are *data independent*, i.e., do not use the same variable, are typically independent. The specified claims, however, may relate the otherwise independent variables. For example, if o_1 modifies v_1 and o_2 modifies v_2, they are not independent if a claim contains $v_1 < v_2$.

SPIN implements a conservative partial order reduction algorithm which determines data independence statically. On the one hand, this is advantageous because the run-time cost of applying the optimization is quite low. On the other hand, new dynamic partial order reduction techniques maintain to be significantly more efficient [FG05].

3.1.3 Specifying Claims

Historically, in distributed systems, the distinction between safety and liveness properties has been made [AS85; Kin94]. As explained by Lamport [Lam77], a *safety property* states that something *bad* will not happen during a system execution, while a *liveness property* states that something *good* must happen eventually. A typical safety property is mutual exclusion in multi-process systems, i.e., that two processes will never enter the critical section at the same time. Such properties are expressed using *invariants*. Liveness properties particularly concern *fairness* and *progress* requirements, e.g., the absence of starvation. For example, if a packet is sent it will eventually be received. Fairness properties are only supported in a limited way by SPIN. SPIN subsumes safety and liveness properties under the notion

of *correctness claims*. For specifying correctness claims, PROMELA supports various methods, e.g., assertions, `never` claims, and linear time logic (LTL) expressions.

Assertions In PROMELA, *assertions* take the form `assert(expr)`. The expression `expr` gives a boolean expression which evaluates to true. As a PROMELA statement, the assertion is always executable. If execution reaches the assertion, it is evaluated. If the expression evaluates to false, the verification (or simulation) is stopped and the sequence of statements executed up to the assertion is written to a trail file. An assertion is only evaluated at those points during an execution sequence where the assertion itself is executed. Thus, to check an invariant after every step, the assertion, for example, has to be inserted in a monitor process which runs concurrently to the other system processes.

Never Claims For stating more complicated global invariants, SPIN offers never claims. A *never claim* describes behavior that should never happen and should be checked at each execution step. Internally, a never claim corresponds to a Büchi automaton for the *negated* property. The Büchi automaton is synchronously factored into the final automaton describing the system and the claims (cf. section 3.1.2).

Linear Time Logic (LTL) Formulas Directly writing never claims may be tedious and error-prone. As any *Linear Time Logic (LTL)* formula can be expressed as a Büchi automation, SPIN supports the automated conversion to never claims. Using the `-f` parameter, SPIN uses its built-in converter to generate a never claim matching the given LTL expression.

As we are most concerned with security and not with fairness considerations, we will focus on safety properties for our models. Furthermore, as the interesting properties are typically related to the exchange of packets (i.e., send and receive actions), claims can typically be checked using assertions.

3.1.4 Verifier Compilation & Run-Time Options

SPIN recognizes several options that can be given either for verifier compilation or during verifier run-time. In the following paragraphs, we outline the most important options during both stages.

Verifier Compilation Options The source code for the model specific verifier generated by SPIN contains parts which can be included selectively (C `#ifdef symbol` statements) in the compilation by defining the appropriate symbols. Using the GCC compiler, a symbol is defined by adding a parameter `-D<symbol>` to its command line. The following symbols are often useful:

- `-DBFS`: By default, SPIN employs depth first search to explore the state-space. The `-DBFS` option changes the search algorithm to breadth first search.
- `-DSAFETY`: By default, SPIN includes code to be able to check for liveness properties (e.g., no progress cycles). The option `-DSAFETY` disables that code. For analysis of safety properties, this results in a smaller and faster verifier.

  `-DREDUCE` This default option includes SPIN's partial order reduction algorithm. In special cases, partial order reductions can be disabled by giving the option `-DNOREDUCE`.
- `-DCOLLAPSE=n` With the `-DCOLLAPSE` option, the state-vector is divided into smaller components. Instead of storing the values of each component, an index is stored for each one. Every time a new value is encountered, the index has to be incremented. This works well if the component's values only cover their range partially.
- `-DMA=n` Usually, each state is directly stored in a lookup table. With the `-DMA` (deterministic minimized automaton) option, a state descriptor is stored instead. This state descriptor is used as input to a finite state automaton (FSA) which then outputs the corresponding state. The FSA is dynamically extended if new states have to be represented. More details about this method for storing states are described in [HP99].
- `-DBITSTATE` This option enables approximative verification using supertrace or bitstate hashing instead of the default exhaustive state space search.
- `-DMEMLIM=n` Using option `-DMEMLIM`, the maximum amount of memory to be allocated by SPIN can be fixed. After the specified amount is exceeded, the verification halts with an error and SPIN prints statistics concerning search depth, number of states and transitions etc. This is useful for comparing models with different optimizations and SPIN settings.

The set of verifier compilation options typically used for analysis of our models is described in section 8.6.

Verifier Run-Time Options Verifier run-time options are much less versatile than the compile time options. Thus, we seldomly use verifier run-time options. The following options are sometimes helpful, however:

- `-mn` This option sets the maximum search depth to n. After depth n is exceeded, the verification halts similar to the `-DMEMLIM` option.
- `-d` Using option `-d`, SPIN outputs its internal state tables for the model representation as contained in the verifier.

3.2 Promela

SPIN comes with its own specification language, called PROMELA (a Process Meta Language). PROMELA has been developed especially for the description of concurrent process systems. The focus lies on synchronization and communication aspects, not on implementation or computational aspects. In the following sections, we first outline the underlying semantics and syntax of PROMELA. Then, we give an outline of the key PROMELA constructs. A detailed presentation of all PROMELA constructs is contained in the manual pages section of [Hol03].

Semantics A PROMELA specification describes a set of communicating finite state machines (cf. [Hol03, ch. 7]). Each PROMELA process (keyword `proctype`, cf. section 3.2.2) describes a labeled finite state machine.

A *labeled finite state machine (LFSM)* is a 5-tuple (S, s_0, L, T, F) consisting of a finite set of states S, an initial state $s_0 \in S$, a finite set of labels L, a set of transitions $T \subset S \times L \times S$, and a set of final states $F \subset S$.

For each process of the PROMELA specification, the set of states S corresponds to the control points of the `proctype`'s body. The initial state corresponds to the entry point (first statement) of the body of the `proctype` declaration, the set of final states is made up of the exit point and the control points corresponding to statements labeled with the `end` keyword.

The transitions relation T defines the possible flow of control (e.g., derived from the statements separated by `;`, and the keywords `atomic`, `if`, `goto` etc). Each transition's label represents the basic PROMELA statement governing the transition (e.g. `(x > y)` or `x=x-y`). This way, the preconditions and effects of executing the transition are determined.

Syntax The basic syntax of PROMELA resembles a stripped-down version of the C programming language combined with non-deterministic control structures (e.g., non-deterministic selection). Furthermore, PROMELA adds constructs for defining process types, processes, communication, and synchronization.

3.2.1 Variables and Types

Variables can be declared using either a built-in data type or a user-defined type.

Variables Local variables are declared inside the context of a process type (cf. section 3.2.2). Global variables are declared at specification level. In any case, the declaration of variables looks like this:

```
varType simpleVarName;
varType arrayVarName[n];
```

The second declaration defines an array variable, i.e., a fixed size ordered collection of n elements, of data type `varType`. For data type, both built-in and user-defined types can be used.

Built-in and User-Defined Types Built-in data types are `bit`, `bool`, `byte`, `short`, `int`, and `unsigned`. Their typical ranges are illustrated in Table 3.1. However, depending on the C compiler that is used to translate SPIN generated verifiers, these ranges may differ.

Type	Range
bit	{ 0, 1 }
bool	{ true, false }
byte	0 . . . 255
short	-32768 . . . 32767
int	$-2^{31} \ldots 2^{31} - 1$

Table 3.1: Promela Built-In Data Types

Furthermore, user data types may be defined using keyword `typedef`. A user-defined type is a record type, i.e., a fixed size collection of n elements of (possibly) different types which are accessed via their field name. Consider the following declaration of a user-defined type:

```
typedef userType {
        type1 fieldName1;
        ...
        typeN fieldNameN;
}
```

After this declaration `userType` may be used like a built-in type. Its elements are accessed using the "`.`" operator and the field name, i.e.,

```
userType userVar;
userVar.fieldN = ...
```

Of course, user-defined types may contain further user-defined types. This way, complicated types can be recursively assembled.

3.2.2 Process Types and Processes

The behavior of a PROMELA specification is determined by its processes, which are instantiated from process types.

Process Types A process type is defined using the keyword `proctype`:

```
proctype procTypeName( ppar1:... ) {
  /* local declarations */
  ...
  /* body statements */
  ...
}
```

Furthermore, initialization parameters (`ppar1`, ...) can be defined. These formal parameters have to be replaced with actual parameters (i.e., fixed values) for instantiating the process. Each process type may contain local declarations for variables and message channels. Furthermore, the process body typically contains PROMELA statements like control structures (e.g., loops), guards, and assignments.

The process body is monolithic. It is not possible to define multiple procedures or methods inside the body. After the execution of the last body statement, a process terminates. Furthermore, in contrast to object-oriented programming languages and CTLA process types (cf. section 5.3), PROMELA process types cannot be composed.

Processes Processes are created by instantiation of process types. Instantiation automatically causes the process to be run as well. Multiple processes in a system are run concurrently following interleaving semantics. Process instantiation and execution is typically done using the keyword `run` from the special `init` process:

```
init {
        run procTypeName( cpar1, ... )
        ...
}
```

This causes an anonymous process of process type `procTypeName` to be instantiated and run.

If only a single instance of a process type is required, no `run` statement is needed. In this case, it suffices to add the keyword `active` to the process type definition:

```
active proctype procTypeName( ppar1:... ) {
...
```

If all processes can be instantiated using the `active` keyword, no `init` process is required. This is more efficient, as one process is removed from the system.

3.2.3 Communication

For communication between processes, channels and shared global variables can be used.

Channels Messages can be passed between processes using a *channel*. A channel is defined using the keyword `chan`:

```
chan chanName = [chanSize] of { typeName };
```

This statement creates a channel with a capacity of `chanSize` elements. Particularly, if the capacity is zero, a *synchronous* (rendezvous) communication channel is created. For passing a message over a synchronous channel, a send operation has to be immediately followed by a receive operation. Non-zero capacities are used for *asynchronous* (buffered) communication. A buffered channel works like a *FIFO queue*. Messages, i.e., elements of the specified type (e.g., `int`), are sent to and received from a channel using special built-in commands: "`!`" sends and "`?`" receives.

Shared Global Variables An alternative and often more efficient method for communication between processes is to use global variables. Global variables are always shared, they can be accessed by all processes of the system.

A process can read and write a global variable by assignment similar to a local variable. Single reads and writes of global variables are *atomic*. No synchronization (e.g., critical sections) is provided, however.

3.2.4 Synchronization & Atomicity

Synchronization In PROMELA, conceptually, all statements are *guarded*. That means they are only *executable* if they evaluate to true. Otherwise, they *block*.

For example, the expression `x == y` is executable if and only if *x* and *y* have the same value. The expression `true` is always executable and the expression `false` (or `(0)`) always blocks. Assignment statements, e.g., `x = y`, are always executable.

If guarded statements are used in combination with global variables (semaphores) between processes, they provide an easy means to build synchronization constructs. For example, busy wait loops can be implemented simply by stating e.g., `(turn != MY_TURN)`. Channels can be similarly used to build synchronizers.

Atomicity Single statements are atomic. Statements are separated by "`;`" or "`->`", both are equivalent. For clarity, however, it is a convention to use "`->`" after statements that may potentially block (i.e., that are true guards).

Furthermore, PROMELA offers a method for marking a sequence of statements as an indivisible, atomic unit. This is done by enclosing the statements in `atomic` as follows:

```
atomic { guard; statement2; ...; statementN; }
```

The first statement is the guard for the whole sequence. If one of the other statements inside `atomic` blocks, atomicity is lost, however. The statements are allowed to make use of non-determinism.

If all statements inside an `atomic` sequence are deterministic and no statement will ever block, a `d_step` sequence can be used instead:

```
d_step { guard; statement2; ...; statementN; }
```

In this case, verification can be done much more efficiently than with an `atomic` sequence.

3.2.5 Non-Deterministic Control Structures

PROMELA supports non-determinism by non-deterministic control structures.

do-loop The do-loop is a non-deterministic repetition construct. Consider the following outline:

```
do
:: (guard1) -> ...; // statements for option 1
...
:: (guardN) -> ...; // statements for option n
od;
```

In this outline, n options are enclosed by the do-loop. The first statement of each option is taken as the guard for the executability of the option. Each time the loop is executed, one of the executable options is chosen non-deterministically. The loop itself is repeated until a `break` statement is encountered.

if-selection The if-selection construct resembles the do-loop. Consider the following outline:

```
if
:: (guard1) -> ...; // statements for option 1
...
:: (guardN) -> ...; // statements for option n
:: else -> ...
fi;
```

As in the do-loop, one of the executable options is chosen non-deterministically. This happens only once, however; after that, the if-selection construct is exited. If none of the other options is executable, the (optional) else option is executed.

3.3 Temporal Logic of Actions (TLA)

The *temporal logic of actions* (TLA) was developed by Leslie Lamport [Lam94]. It is a specification logic used for the modeling of discrete event dynamic systems. In a *discrete event dynamic system*, the state changes are driven by asynchronous events. In contrast, in classical system theory, systems are time-driven.

Leslie Lamport derives TLA by combining a custom *logic of actions*, which formalizes the concept of actions, with a *linear-time temporal logic*, which enables reasoning about sequences of states. As this deduction is quite technical, we take a different approach based on the TLA canonical formula and the correspondence to state transition systems.

3.3.1 Basics

In TLA systems, the state components of real systems are represented by state variables. A *state variable* (or variable in short) has a name (e.g., x), a type (e.g., Nat for natural numbers), and a current value (e.g., 3). The *state* of the model is determined by the current values of all state variables.

A *primed variable*, e.g., x', refers to the value of the variable in the next state. If s refers to the current state and t refers to the next state, and if x has the value a in state s and value b in state t, then $x = a$ and $x' = b$.

An *action* is a predicate about a pair of states. The predicate is assembled from variables, primed variables, and constant symbols, e.g., $x' * 2 = y$. This predicate is true for all state pairs (s, t), where the value of x in state t is twice the amount of the value of y in state s. Thus, an action defines a relation between states.

Furthermore, an action $\mathcal{A}$ is called *enabled* for a state s, if a state t exists so that $\mathcal{A}$ is true for (s, t). In this case, if the state is indeed changed from s to t according to $\mathcal{A}$, the action is said to be *executed*.

3.3.2 TLA Canonical Formula

The *canonical formula* Φ for a TLA system is given by

$$\Phi := \mathsf{Init} \wedge \Box [\mathcal{N}]_f \wedge \mathsf{FA}$$

where

- Init is the *initialization predicate*, defining the initial states of the system
- $\Box [\mathcal{N}]_f$ is the always subformula, defining the system steps
- FA specifies fairness requirements for some subset of the actions

The *always subformula*, $\Box[\mathcal{N}]_f$, is made up of the *always operator* $\Box$, the next-state predicate $\mathcal{N}$ and the stutter-function f. The *next-state predicate* $\mathcal{N}$ combines the actions into system steps (e.g., by disjunction). Thus, each *system step* corresponds to the execution of one or more actions. To support the composition of a system from subsystems by conjunction properly, stuttering steps have to be allowed. During a *stuttering step*, the state of a (sub)system remains unchanged. Formally, if $x_1, \ldots, x_n$ are the state variables of the (sub)system, this is realized by defining the stutter-function f as $x_1 = x'_1 \vee x_2 = x'_2 \vee \cdots \vee x_n = x'_n$. Thus, the subformula makes sure that always either a system step or a stuttering step occurs.

The optional *fairness requirements* FA are given as a conjunction of weak fairness $\mathsf{WF}(\mathcal{A})$ and strong fairness $\mathsf{SF}(\mathcal{A})$ formulas, where $\mathcal{A}$ is an action. A *weak fairness* assumption $\mathsf{WF}(\mathcal{A})$ assumes that the action $\mathcal{A}$ has to be executed in situations, where the action is enabled and continuously will be enabled until its execution. A strong *fairness assumption* $\mathsf{SF}(\mathcal{A})$ assumes that the action $\mathcal{A}$ has to be executed if the action will be enabled again and again until its execution.

3.3.3 Correspondence to State Transition Systems

A *state transition system (*STS*)* is a 3-tuple STS $::= \langle S, S_0, T \rangle$ consisting of a set of states S, a set of initial states $S_0 \in S$, and a set of transitions $T \subset S \times S$.

In the case of a TLA system description, the set of states S is spanned by the set of variables V in the system. The set of initial states S_0 is determined by the Init predicate. The transitions are given implicitly by the always subformula, $\Box[\mathcal{N}]_f$, which defines a relation between states.

Let STS be a state transition system. Then, the set of all state sequences (so-called *behaviors*) of the system is given by:

$$B_{\mathrm{STS}} ::= \{b : b = (s_0, s_1, \ldots) \in S^\infty, s_0 \in S_0, \forall n \in \mathbf{N} : ((s_i, s_{i+1}) \in T \vee s_i = s_{i+1})\}$$

As described by Alpern and Schneider [AS85], all properties of the system can be formulated as intersections of safety and liveness properties. Safety properties can be checked via reachability analysis in B_{STS}.

The characteristic of liveness properties is that they cannot be violated by partial executions of the system (otherwise the partial execution would constitute a "bad thing" and belong to a safety property, not a liveness property). Instead, there has to be an infinite execution that always stutters with respect to the action(s) that would fulfill the liveness property. Liveness properties are expressed by fairness assumptions for actions as described above. This way, liveness properties do not contain unintended safety parts that conflict with the STS. The set B_{STS} can be restricted to the behaviors satisfying the fairness assumptions.

3.4 Compositional Temporal Logic of Actions (cTLA)

cTLA extends TLA with explicit notions of processes, process types and process composition. Furthermore, canonical parts of specifications are not explicitly written down in cTLA. In the following paragraphs, we outline the key properties of cTLA, version 2000, in relation to TLA. Further details about cTLA 2000 are described in [HK00]. The scenarios considered in this thesis (cf. chapters 9 to 11) are modeled using the refined version cTLA 2003, which is presented in chapter 5.

3.4.1 Processes

cTLA introduces the notion of processes and process types. A *process* is a state transition system that is instantiated from a *process type*. Process types are specified in a programming language like syntax. The following sections are contained in a cTLA *simple process type* specification:

- Header (`PROCESS ProcName( ppar1: tpar1; ... )`): In this section, the name of the process type and a list of generic parameters are given. Upon instantiation, the generic parameters are replaced with actual parameters.
- Import (`IMPORT`): The import section is optional and allows the inclusion of constants, data types, and functions defined in other modules.
- Initialization (`INIT`): The initial state of the processes instantiated from this process type is defined through the initialization predicate.
- Variables (`VARIABLES`): In the variables section, the variables which span the state space of the process are declared. cTLA variables are private to the process and cannot be accessed by other processes.
- Actions (`ACTIONS`): The process body is defined by the actions section. This section lists the actions of the process type.

Correspondence to the TLA Canonical Formula An instance of a simple process is a state transition system directly corresponding to the TLA canonical formula (cf. section 3.3.2). The initialization predicate Init is the initialization predicate `INIT` of the process. The next-state predicate is given by the disjunction $\mathsf{Next} = \mathsf{act}_1 \vee \mathsf{act}_2 \vee \mathsf{act}_3 \vee \cdots \vee \mathsf{act}_m$, where the act_i are the actions listed below the `ACTIONS` keyword. Furthermore, for the actions marked with fairness requirements, FA has to be set to the conjunction of these markings (i.e., $\mathsf{FA} = \mathsf{WF}(\mathsf{act}_{i1}) \wedge \cdots \wedge \mathsf{WF}(\mathsf{act}_{ik}) \wedge \mathsf{SF}(\mathsf{act}_{j1}) \wedge \cdots \wedge \mathsf{SF}(\mathsf{act}_{jl})$). Finally, the set V of variables of the state transition system corresponds to the local variables of the process listed below the `VARIABLES` keyword.

Example: Simple Process A cTLA system instantiated from the simple process type `Relais`

```
PROCESS Relais();
VAR
  b: Buffer;              // state space
INIT ::=
  b.c = ST_READY          // initial states
ACTIONS
  in( m : Mtype ) ::=     // put message into buffer
    b.c = ST_READY
    AND b.c' = ST_BUSY
    AND b.b' = m;

  out( m : Mtype ) ::=    // get message from buffer
    b.c = ST_BUSY
    AND b.b = m
    AND b.c' = ST_READY
    AND b.b' = b.b;       // unchanged(b.b)
END;
```

corresponds to the TLA formula Φ:

$$
\begin{aligned}
\Phi &:= \mathsf{Init} \wedge \Box\, [\mathcal{N}]_f \\
\mathsf{Init} &:= bc = \mathtt{ST_READY} \\
\mathcal{N} &:= \mathcal{A}_1 \vee \mathcal{A}_2 \\
\mathcal{A}_1 &:= bc = \mathtt{ST_READY} \wedge bc' = \mathtt{ST_BUSY} \wedge bb' = m \\
\mathcal{A}_2 &:= bc = \mathtt{ST_BUSY} \wedge bb = m \wedge bc' = \mathtt{ST_READY} \wedge \mathsf{Unchanged}(bb) \\
f &:= bc' = bc \wedge bb' = bb
\end{aligned}
$$

The Unchanged predicate requires that the value of its arguments remains unchanged, e.g., $\mathsf{Unchanged}(x) := x = x'$.

3.4.2 Process Types & Process Composition

Besides the simple process type, cTLA supports composed or subsystem process types. As the name suggests, a *subsystem process type* is composed of other processes.

The constituting processes $P_1, P_2, \ldots, P_n$ are listed in an additional section (`PROCESSES`). As the initialization predicate is derived from the initialization of the constituting processes, the corresponding section is not used. Furthermore, the actions section lists a special type of actions, system actions.

System Actions In the context of process composition, *system actions* are joint (i.e., synchronous) actions of the constituting processes. They determine the interaction between the processes. Each system action couples actions from the processes by logical conjunction. Thus, a system action sact has the form sact $= P_1.\mathsf{act_}j_1 \wedge \cdots \wedge P_n.\mathsf{act_}j_n$. Each $P_i.\mathsf{act}_{ji}$ is either a real action of P_i or the pseudo-action `stutter`. If a process performs the `stutter` pseudo-action, all its state variables remain unchanged during the execution of the system action.

Data communication between processes can also be realized through system actions and their parameters. For example, if a data item d shall be exchanged between two processes P_1, P_2, a system action can be defined in the following way: `sact(d) ::= P1.act1(d) AND P2.act2(d)`. Both P_1 and P_2 are then able to read d in the respective actions.

Correspondence to the TLA Canonical Formula As with the simple process type, an instance of a subsystem process type corresponds to the TLA canonical formula. The correspondence is a bit more complicated, though. For a subsystem process type S composed of process instances $P_1, \ldots, P_n$, the Init predicate is given by the conjunction of the individual initialization predicates, i.e., Init $= P_1.\mathsf{Init} \wedge \cdots \wedge P_n.\mathsf{Init}$. The next-state predicate is given by the disjunction of the system actions. The set of variables V is the union of the sets of variables V_i of the P_i. Regarding fairness, under some restrictions related to the system action coupling, FA is given by the conjunction $P_1.\mathsf{FA} \wedge \cdots \wedge P_n.\mathsf{FA}$. Thus, an instance of a composed process type defines a state transition system and is available for further composition.

Example: Subsystem Process A cTLA system instantiated from the subsystem process type `TransferSys` – which is composed of three `Relais` instances –

```
PROCESS TransferSys
CONTAINS
  SR: Relais;               // source relais
  TR: Relais;               // transfer relais
  DR: Relais;               // destination relais
ACTIONS
  put( m: Mtype ) ::=       // put message into source relais
    SR.in(m) AND TR.stutter AND DR.stutter;
  send( m: Mtype ) ::=      // send message from s. to t. relais
    SR.out(m) AND TR.in(m) AND DR.stutter;
  receive( m: Mtype ) ::=   // receive message from t. in d. relais
    SR.stutter AND TR.out(m) AND DR.in(m);
  get( m: Mtype ) ::=       // get message from destination relais
    SR.stutter AND TR.stutter AND DR.out(m);
END;
```

corresponds to the TLA formula Ψ:

$$
\begin{aligned}
\Psi &:= \mathsf{Init} \land \Box\ [\mathcal{N}]_f \\
\mathsf{Init} &:= srbc = \texttt{ST_READY} \land trbc = \texttt{ST_READY} \land drbc = \texttt{ST_READY} \\
\mathcal{N} &:= \mathcal{N}_1 \lor \mathcal{N}_2 \lor \mathcal{N}_3 \lor \mathcal{N}_4 \\
\mathcal{N}_1 &:= srbc = \texttt{ST_READY} \land srbc' = \texttt{ST_BUSY} \land srbb' = m \\
&\quad \land \mathsf{Unchanged}(trbc, trbb, drbc, drbb) \\
\mathcal{N}_2 &:= srbc = \texttt{ST_BUSY} \land srbb = m \land srbc' = \texttt{ST_READY} \\
&\quad \land trbc = \texttt{ST_READY} \land trbc' = \texttt{ST_BUSY} \land trbb' = m \\
&\quad \land \mathsf{Unchanged}(srbb, drbc, drbb) \\
\mathcal{N}_3 &:= trbc = \texttt{ST_BUSY} \land trbb = m \land trbc' = \texttt{ST_READY} \\
&\quad \land drbc = \texttt{ST_READY} \land drbc' = \texttt{ST_BUSY} \land drbb' = m \\
&\quad \land \mathsf{Unchanged}(srbc, srbb, trbb) \\
\mathcal{N}_4 &:= drbc = \texttt{ST_BUSY} \land drbb = m \land drbc' = \texttt{ST_READY} \\
&\quad \land \mathsf{Unchanged}(srbc, srbb, trbc, trbb, drbb) \\
f &:= \mathsf{Unchanged}(srbc, srbb, trbc, trbb, drbc, drbb)
\end{aligned}
$$

Superposition The composition of processes as described above has the character of *superposition*, i.e., a property of a subsystem is a property of the system as a whole. For safety properties, which constrain the initial states and state transitions only, the superposition property holds. This is due to all state variables being private to their respective process or subsystem and the logical conjunction used for assembling the Init predicate and the system actions.

Regarding liveness properties, an action may be blocked due to its environment, e.g., the coupling with another action in a system action. This may violate the original action's fairness requirements. Thus, the fairness requirements in cTLA are *conditional* and have to fulfill certain restrictions. They refer to periods of time where the action is not blocked by its environment.

Under these restrictions, the equivalence between the direct canonical subsystem formula and the compositional subsystem formula gained from conjugating the canonical formulas for the P_i can be shown. A detailed examination is contained in [Her98].

4 An Integrated, Formal Modeling and Automated Analysis Approach

As explained in chapter 2, existing approaches for formal modeling and automated analysis of computer network models are not sufficient. Particularly, the modeling formalisms are quite restricted and do not offer integrated modeling of protocol, node, and network aspects. Furthermore, the models are largely static and the analysis often assumes monotonicity.

Thus, we develop a new approach based on a high-level modeling language, a computer network modeling framework, a translator, optimization strategies, and an analysis tool. In the following sections, we briefly present the objectives, implementation and workflow of the approach. Furthermore, we give an overview of the steps required to build a model for a scenario.

4.1 Objectives

The following objectives are key for our integrated formal modeling and automated analysis approach.

Formal Modeling Systems shall be modeled using a *formal language*. Formal modeling provides for a clear and precise description of the system. A further advantage is that well-established techniques and tools for transforming and analyzing formal specifications exist. Thus, we do not have to develop all tools and techniques from scratch but can adapt the existing ones to our approach.

Integration of Multiple Views Existing approaches are heavily geared towards a single view (e.g., the node view) and are not expressive enough to integrate aspects from other views (e.g., the protocol or network view). Our approach shall be able to integrate the protocol, node, and network view into *one consistent model*. Thus, we do not have to use different formalisms, models, and analysis tools for each of the views. Instead, a single formalism, model, and analysis tool shall support all three views with a medium to high level of detail.

Executable Models The formal system models shall be *executable*, i.e., an environment for running the specification needs to be provided. The environment has to support step-by-step simulation both in an interactive and a guided way. Such an environment is essential for validating a model, i.e., to make sure that a model reflects the "real world" in an adequate way.

Dynamic Models The models shall not be restricted to static views of protocol, node, and network aspects. Instead, they shall be able to express *dynamics* like command dependent replies, node address changes, and network routing updates. Particularly, physical and logical layers of the network model have to be distinguished, a connectivity matrix is not enough.

Automated Analysis Automated, tool-supported analysis of properties of a model shall be possible. The supported mechanisms for stating properties shall range from simple to complex. Attack sequences have to be found *automatically* by checking for violation of security properties. No user guidance in the form of supplying appropriate queries or intermediate lemmas shall be necessary. Furthermore, the sequences discovered have to correspond to model-level operations, not to a low-level internal representation dependent on the analysis tool.

Ease of Use The approach shall *facilitate* all steps of its workflow. Particularly, both the modeling and the analysis tasks need appropriate support. The formal language has to enable the reuse and extension of existing models. A concept for modeling computer networks has to be provided. Regarding the analysis task, suitable tool support must be provided for transformation and analysis of models.

Finally, the practical feasibility of the approach shall be demonstrated by several case studies.

4.2 Implementation

For realizing our objectives (cf. section 4.1), we employ the following languages, methods, and tools.

Compositional TLA 2003 As a formal modeling language, we use compositional TLA 2003 (cTLA 2003). cTLA is a based on the formal language TLA (cf. section 3.3). In comparison to TLA, cTLA offers support for compositional modeling based on process types and process composition (cf. section 3.4).

cTLA *2003*, a refined version of cTLA, adds a new extending process type that supports object-oriented modeling and facilitates framework based modeling. Furthermore, cTLA 2003 stresses efficient executability of specifications e.g., by pro-

viding finite data types which can be directly mapped to common machine-level data types. Overall, cTLA 2003's features are balanced between expressiveness, abstraction level and implementation efficiency.

Computer Network Modeling Framework To facilitate the modeling task and support an integrated view of network, node, and protocol-related aspects, we devised the *computer network modeling framework* (cf. chapter 7). It evolved from preliminary models and is used for the case studies described in chapters 9 to 11. The framework is written in cTLA 2003 and provides key elements like media, nodes, and types for packets, protocols, interfaces etc.

Due to cTLA 2003's object-oriented features, the existing elements can be extended and specialized with ease. This allows for framework-based scenario modeling, i.e., models largely reuse the framework's code and only add code as required for a specific scenario.

Spin Model Checker For the simulation and analysis of our models, we employ SPIN (cf. section 3.1). SPIN is one of the most well-known and powerful tools for the automated verification of distributed systems. It has been continuously adapted to new developments and algorithms. SPIN's input language, PROMELA (cf. section 3.2) comprises a restricted subset of the C programming language and adds constructs for non-determinism as well as communication and synchronization between processes. PROMELA is quite a low-level language not supporting e.g., process-type composition. Furthermore, PROMELA does not foster a clear model structure by providing e.g., a concept of actions.

SPIN can both simulate and analyze PROMELA models in a number of different ways. Regarding simulation, both interactive and guided modes exist. For analysis, properties can be stated e.g., using a wide range of mechanisms. Analysis models are translated to executable verifiers using a C compiler which increases verification performance. The well-known state space explosion problem, however, still makes the analysis of most models hard.

Translation & cTLA2PC Tool As SPIN supports only PROMELA models, cTLA models have to be translated. Thus, we designed a translation scheme (cf. section 6.2) for transforming cTLA models to PROMELA. Two key steps characterize the translation from cTLA to PROMELA: First, the process composition is resolved. Second, actions are embedded and parameters handled.

The cTLA2PC tool (cf. section 6.3) implements the translation scheme and provides automated translation of cTLA models to PROMELA. Beyond the basic translation, cTLA2PC offers extended translation options. Particularly, options for mapping exist between cTLA level actions and PROMELA statements. This way, we are able to consider the analysis results on the model-level instead of having to work

with the low-level PROMELA representation. Furthermore, various optimizations can be applied automatically during translation.

Optimization Strategies Computer network models typically consist of several nodes buffering, processing, and exchanging packets. This leads to a significant level of complexity even for small models and often to state space explosion effects. Thus, optimizations are nearly always required prior to the successful analysis of a model.

During the course of this thesis, particularly while working on the case studies described in chapters 9 to 11, several optimization ideas were considered and tested. The optimizations which proved to be most useful are described in chapter 8. They are structured according to the modeling stage where they are applied. Furthermore, if they can be applied at the CTLA or PROMELA-level, the optimizations have been integrated into CTLA2PC.

Eclipse Integration We ease the application of our approach by providing an integrated environment for modeling and analysis (cf. section 6.4). The environment is implemented based on the ECLIPSE workbench.

Particularly, the environment integrates CTLA2PC and SPIN with ECLIPSE's core services. Altogether, the environment supports the editing, translation, simulation, debugging, and verification of models. The debugging feature works in simulation mode and behaves similar to common programming environments: breakpoints may be be defined, variable watches added, and the execution can be traced in single steps.

4.3 Workflow

In the following paragraphs, we describe the stages of the workflow for our approach. The depicted workflow (cf. Fig. 4.1) reflects the ideal process; when putting it into practice, however, stages may have to be repeated, e.g., if initial analysis of a model failed due to state space explosion effects.

Real Network The real network is the background of our modeling. Networks are usually linked to other networks, consist of subnetworks, contain many elements which may not be interesting and do not have well-defined borders. Thus, the modeling of complete networks is often not feasible. Instead, we select a scenario, containing a typical subset of the real network.

Scenario Diagram We think about the key nodes, protocols, and networks required. This depends on the setting and the properties we plan to analyze. Gen-

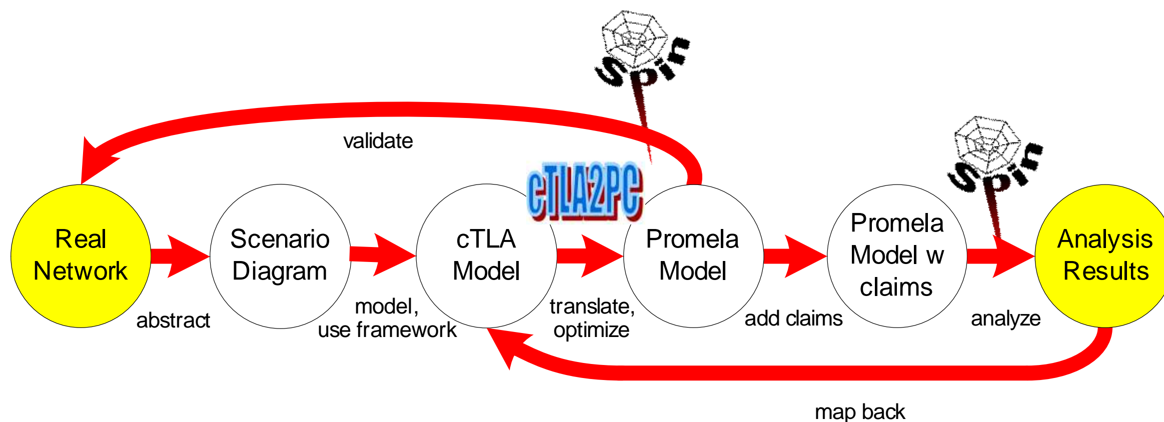


Figure 4.1: Ideal Workflow of Our Approach

erally, a network can be simplified by selecting a few representative nodes. Furthermore, not all layers of the network reference model (cf. section 7.2.1) are required in every case. We consider these facts for devising the scenario diagram.

cTLA Model The cTLA model for the scenario is developed in several steps (following the protocol, node, and network view) as described in section 4.4. The framework provides the generic structure and common elements for the model. Thanks to cTLA 2003's process composition features, specific extensions can be made based on existing elements. This greatly facilitates the development of a specific model.

Promela Model Using cTLA2PC, the PROMELA model is automatically generated from the cTLA model. Furthermore, cTLA2PC supports several optimizations which can be applied during translation. Optimizations not yet included in cTLA2PC can be performed manually. Often, the non-compositional, flat system (cf. section 6.2.1.1), which is optionally produced by cTLA2PC is a good starting point for manual optimizations. Another possibility is to apply low-level optimizations to the PROMELA model directly.

The PROMELA model serves another important function: model validation. We have to make sure that the model relates well to reality; that we are building the right model. SPIN offers basic interactive and guided simulation of PROMELA models. These features are improved with our ECLIPSE integration and cTLA2PC options for simulation verbosity. By checking both selected predetermined and random sequences, we increase our confidence in the model. If unexpected variations occur, we restart the modeling cycle (i.e., adapt our model and validate it again).

Promela Model with Claims As SPIN only supports analysis of PROMELA models, the claims have to be based on the PROMELA model as well. The claims can be expressed using various mechanisms (cf. section 3.1.3), particularly `assert` statements. Depending on the properties we want to analyze and on the modeling of the claims, one or several of such statements may be required. We then add these statements to the PROMELA model (as generated by CTLA2PC from the CTLA model).

Analysis Results The PROMELA model with claims is analyzed using SPIN. SPIN supports different analysis modes and search strategies (cf. section 3.1.4). Generally, however, these options do not make or break the successful analysis of a model. Successful analysis is much more dependent on the model itself (i.e., the abstractions and simplifications chosen with respect to the real network) and the optimizations applied.

If the analysis is successful (i.e., if SPIN neither runs out of memory early nor runs "forever") and a sequence of steps violating a claim is found, SPIN writes a trail file. This file contains an internal encoding of the state sequence leading to the violation. The mapping back to CTLA model-level operations happens in two stages: First, SPIN can play back the trail file in guided simulation mode. This reveals the corresponding PROMELA level sequence. Second, the PROMELA model generated by CTLA2PC contains trace points outputting the CTLA level actions.

Thus, we mapped back the sequence found by SPIN to the CTLA level. Typically, we depict the sequence using a sequence diagram showing the processes involved and actions executed. Finally, the results have to be interpreted and discussed in the context of the real network. As the correspondence between the CTLA model and the real network has already been checked during the validation stage, this should not be too hard.

4.4 Modeling Steps

To design the CTLA model for a scenario, we typically take the steps outlined below (cf. chapter 9 to 11). These steps correspond to the protocol, node, and network views plus a system modeling (integration) step.

Furthermore, after translation, the claims are added to the PROMELA model. This is a PROMELA – not a CTLA – modeling step that may also be considered to be part of analysis.

Protocol Modeling We examine the protocols involved in the scenario. These days, most protocols are based on TCP/IP. Thus, the corresponding reference model (cf. section 7.2.1) is helpful for determining which layers have to be included

in the modeling. As we are, for example, not concerned with the encoding of signals on the physical media, we do not model the physical layer. Furthermore, the required layers can be modeled using different degrees of abstraction.

A scenario involving low-level attacks, for instance, must include hardware addressing mechanisms at the network interface layer. However, if we are interested in attacks on the transport or application level only, this level of detail at the network interface level is counterproductive.

Last but not least, the protocol itself can often be simplified. For example, the protocol may support the sending of several updates in one packet. Typically, this has the same effect as sending several packets with one update each. The latter can often be modeled in an easier way as the packet structure is less complex.

Node Modeling Depending on the scenario, not all nodes may have to be modeled; instead, a set of representative nodes may suffice (cf. section 8.2.1). For these nodes, aspects not directly related to rotocol processing have to be considered during node modeling.

We assign roles to the nodes. Typical roles are e.g., sender, receiver, and attacker. Roles help to minimize the set of actions required for a node (cf. section 8.2.2). An attacker, for example, may have additional actions. By designating a certain node as an attacker, we do not have to include these actions for every node.

Besides the attacker actions, administrative actions may be added to the nodes during the node modeling step as well. Furthermore, the configuration of the node needs to be modeled. The initial configuration, e.g., routing tables after boot-up, is set during node initialization.

Network Modeling The network modeling step deals with representing the network topology. In particular, appropriate zone definitions and the mapping between nodes, interfaces, and zones (e.g., through functions) must be defined. The transmission media has to be modeled. As we do not require sophisticated modeling of transmission media characteristics, the simple media type provided by the framework is usually sufficient.

During system composition, the send and receive system actions have to be defined to follow the network topology. A broadcast receive system action, depending on its modeling, may require that only receive actions of nodes that are in the same broadcast zone are coupled.

System Composition During this step, the system process type (cf. section 5.3.3) is defined. The system process type integrates the instances of all processes that are required for realizing the system, e.g., the node and media processes. To this aim, the system actions have to be provided. Each system action couples the actions of the processes and thus defines the collaboration in the system.

Translation & Claims Modeling In contrast to the previous steps, the claims are not added to the cTLA, but to the PROMELA model. Thus, this step is done after translation to PROMELA. Typically, we model a claim corresponding to a security property using assertions. Assertions can be analyzed most efficiently by SPIN, and they are easy to understand.

5 The cTLA 2003 Modeling Language

For our modeling, we employ cTLA 2003 [RK03], a refined version of cTLA 2000 (cf. section 3.4). After a short overview describing the key differences to cTLA 2000, we explain each part of a cTLA 2003 specification in more detail. If no version is explicitely stated, in the following chapters cTLA refers to cTLA 2003. We conclude this chapter with excerpts from the formal grammar for cTLA 2003. The full grammar is included in Appendix A.

5.1 Comparison to cTLA 2000

cTLA 2003 is designed as an executable subset of cTLA 2000 combined with modeling enhancements. In the following section, we first describe the executable subset, then the key modeling enhancements.

5.1.1 Executability

In contrast to cTLA 2000, cTLA 2003 specifications are *executable*. Models can be efficiently executed using a simulation or analysis tool. This greatly simplifies model validation and automated analysis.

The notion of *executable specifications* stems from the field of software engineering. Instead of having to prove that the implementation meets the specification, the specification *is* the implementation. To put it another way, executable specifications can be seen as a direct implementation of themselves.

Executable Subset of cTLA 2000 To ensure efficient executability of cTLA 2003 specifications, we consider the following subset of cTLA 2000:

E1 all data types, including sets, are *finite*

E2 all actions follow the *standard form*

$$\text{act} ::= g_1 \texttt{ AND } g_2 \texttt{ AND } \ldots \texttt{ AND } g_n \texttt{ AND } e_1 \texttt{ AND } \ldots \texttt{ AND } e_m$$

where the $g_i, i = 1, \ldots, n$ are guards and the $e_j, j = 1, \ldots, m$ are effects

E3 the initialization predicate takes the form $x_1 = c_1$ AND ... AND $x_l = c_l$, where the $x_i, i = 1, \ldots, l$ are the variables occuring in the specification and the $c_i, i = 1, \ldots, l$ are constants

The first property implies that all variables and parameters can only take a finite number of different values. Clearly, as computer systems have only a limited amount of memory, this property is required for any executable specification. The second property is required because in cTLA 2000, arbitrary predicates are allowed, which cannot be efficiently evaluated. The third property makes sure that the initialization predicate can be efficiently evaluated and uniquely determines the initial state.

Structure of Guards *Guards* are negated or non-negated predicates over non-primed variables, optional parameters, and constants (e.g., $v = 3$ or NOT $v > p$).

The OR operator may occur between two guards g_1, g_2. In this case, we replace act by two actions $\text{act}_{g1}, \text{act}_{g2}$. Then, both actions act_{gk}, $k = 1, 2$, are in the standard form: $\text{act}_{gk} ::= g_k$ AND g_3 AND ... AND g_n AND e_1 AND ... AND e_m, $k = 1, 2$.

In cTLA 2003, *quantified guards* are supported, too. Quantified guards are guards over sets of elements. They can be either of the exists or of the forall type. An *exists-type* quantified guard is true if the guard is true for *at least one* of the elements of the set. Symmetrically, an *forall-type* quantified guard is true if the guard is true for *all* elements of the set. As sets are finite (E1), the exists, forall-type quantified guard can be replaced by n simple guards connected by OR, AND, respectively. Thus, we can inductively transform actions involving quantified guards into the standard form.

Structure of Effects *Effects* are assignments, with a primed state variable on the left and an expression built from non-primed variables and constants on the right (e.g., $v' = p + 3$). Effects may not be combined using OR or include NOT.

The logical operator IF−THEN−ELSE may occur in an action where a simple guard or effect occurs and combines a guard with an effect: IF g_1 THEN e_1 ELSE e_2. The operator is a short hand for the logical formula (g_1 AND e_1) OR ($\overline{g_1}$ AND e_2). By replacing the action involving the operator with two actions $\text{act}_{g1}, \text{act}_{\overline{g1}}$ as described above, the OR is removed. Then, by reordering the guards and effects (which are all AND-connected), the standard form is restored.

The IF−THEN−ELSE may also be nested. Then, we can transform the action to the standard form inductively. Consider for example action

act: IF g_1 THEN (IF g_2 THEN $e_{1,1}$ ELSE $e_{1,2}$) ELSE e_2

First, the outer IF, then the inner IF are transformed. Finally, act is replaced by these three actions (in standard form):

$\text{act}_{\overline{g1}}$: $\overline{g_1}$ AND e_2

$\mathsf{act}_{g1,g2}$: g_1 AND g_2 AND $e_{1,1}$
$\mathsf{act}_{g1,\overline{g2}}$: g_1 AND $\overline{g_2}$ AND $e_{1,2}$
Quantified effects are supported in cTLA 2003 with the updateall effect. The *updateall* effect changes the value for all elements of a set that satisfy a certain condition. Similar to quantified guards, this does not adversely affect the standard form.

As the described guards and effects can be evaluated efficiently, so can actions (E2). Furthermore, the initial state can be evaluated efficiently and is uniquely determined (E3). Thus, the efficient executability of cTLA 2003 specifications follows.

Focus on Safety In our models, we aim to analyze security properties. These properties can be expressed (e.g., using assertions or invariants) as pure safety properties. Moreover, automated analysis can be done much more efficiently for safety properties than for fairness properties. Thus, the model checker SPIN, for instance, requires a special parameter (`pan -f`) to enable a quite limited form of fairness – weak fair scheduling on the process level – during analysis.

For these reasons, we focus on safety properties in cTLA 2003. cTLA 2000's action level SF and WF fairness operators are not supported in cTLA 2003. By adding helper variables and guards, however, the behavior of a model can be restricted to meet fairness requirements.

5.1.2 Modeling Enhancements

To facilitate the modeling task, cTLA 2003 supports reuse of process types. Furthermore, specifications are simpler to write as canonical parts are left out.

Reuse of Process Types cTLA 2003 introduces a new process type, the extending process type (cf. section 5.3.2). The extending process type allows to derive new process types from existing process types. This way, a new process type that adds own code (i.e. actions, variables, or initializations) to an existing process type can be defined. Furthermore, actions of the existing process type may be modified by additional guards or effects. Only the new code has to be specified in the new process type; all other code is reused from the existing process type(s). This mechanism to extend existing process types resembles *inheritance* as known from object-oriented programming languages.

Reuse fosters the development of libraries of process types. A set of domain specific process types together with common data types, enumerations, and functions lays the foundations for a modeling framework. We developed a computer network modeling framework for cTLA 2003 (cf. chapter 7) that relies on the extending process type and eases the modeling process greatly.

Implicit Unchanged in Actions In cTLA 2003, process local variables not occuring in the effects of an action remain unchanged by the execution of the action. Thus, unchanged statements (e.g. $v' = v$) are neither required nor supported for cTLA 2003 actions.

The implicit unchanged helps to reduce both the red tape and errors in actions. For example, if a new variable w is added to a process type in cTLA 2000, all actions not modifying w have to be extended with $\wedge w' = w$. This is easily forgotten and leads to the usually unintended effect that the future value of w is random. The implicit unchanged forces all variables to be explicitly set to a new value. This ensures efficient executability of actions.

Internal Actions cTLA 2003 introduces the new concept of *internal actions*. An internal action defines a set of state transitions in exactly the same way as a normal action. The difference between both sorts of actions concerns the composition of systems from processes. When a process instance is employed as a component in a system, the internal actions of the process cannot be coupled with actions of other processes. Each internal action is accompanied by stuttering steps of all other system components.

Typical examples of internal actions are the packet-processing actions of a node (e.g. `rpcs`). An action marked as internal is implicitly added to the system actions as a single action while all other processes stutter. This eases the modeling task, because the system-level coupling only has to be specified manually for the external actions.

Actions that are not internal are called *external*. External actions are typically coupled on the system-level with other actions to model a certain interaction between processes. For example, the send action (`snd`) of a node process is coupled with the `in` action of a media process to represent packet acceptance by a physical transfer medium.

5.2 Specification Structure

A cTLA specification consists of six parts (cf. Listing 5.1).

These parts are constants, types (including enumerations and user data types), functions, predicates, process types, and the system instantiation. In the following paragraphs, we describe the parts one after another, except for the process types and system instantiation, which are described in section 5.3.

5.2.1 Constants

The optional constants part allows for the definition of symbolic constants. Three types of symbolic constants are supported in cTLA: simple constants, enumera-

```
CONST                                  /* 1. constants */
  constName1=value1;
  ...
TYPE                                   /* 2. types */
  enumType1=(val1, ... );                // enumeration
  ...
  userType1=RECORD                       // user data type
    fieldName1: type1;
    ...
  END;
  ...
  FUNCTION func1(x:INT) ::=            /* 3. functions */
    ...
  END;
  ...
  PREDICATE pred1(x:BYTE) ::= ...;     /* 4. predicates */
  ...
PROCESS procType1 ...;                 /* 5. process types */
  ...                                    // actions
END;
...
SYSTEM sysInstance ...;                /* 6. system instantiation */
```

Listing 5.1: cTLA Specification Outline

tions, and compound constants. While simple and compound constants are listed in the `CONST` part, enumerations are listed in the the `TYPE` part.

Simple Constants A simple constant defines a symbolic name for a fixed value. In cTLA, the declaration of simple constants looks like this:

```
CONST
  ...
  simpleConstName1 = value1;
  simpleConstName2 = value2;
  ...
```

After declaration, simple constants are referred to by their name and may be used in all places where a variable or parameter value is read (e.g., on the right side of assignments).

Simple constant declarations are not typed. Our cTLA compiler, cTLA2PC (cf. section 6.3), infers a type during semantic analysis, however. This is used to flag invalid assignments during model translation (e.g., assignment of an `INT`-valued constant to a `BYTE` variable).

Enumerations An enumeration defines a set of symbolic names. Each symbolic name is assigned a unique integer from $0 \ldots n-1$ where n is the cardinality of the set of names for the enumeration. In cTLA, enumerations are declared via the following statements:

```
TYPE
   ...
   enumName1 = (e1_name1, e1_name2, ..., e1_nameN1);
   enumName2 = (e2_name1, e2_name2, ..., e2_nameN2);
   ...
```

These statements have two effects. First, variables of the respective enumeration type can be declared and may take any of the symbolic names as a value. Second, each of the symbolic names defined by the enumerations may be used like a simple constant in the specification.

Compound Constants Compound constants are constants of an array or record type (cf. section 5.2.2). They define an instance of such a data type with a symbolic name and fixed value. Compound constants are declared in cTLA the following way:

```
CONST
   ...
   compConstName1 = arrayTypeName   // array type comp. const.
         { [arr_val1, arr_val2, ..., arr_valN ] };
   compConstName2 = recordTypeName // record type comp. const.
         { {rec_fld1=rec_fld1_val, rec_fld2=rec_fld2_val, ...,
            rec_fldN=rec_fldN_val} };
   ...
```

While `compoundConstName1` declares a compound constant of an array type (`arrayTypeName`), `compoundConstName2` declares a compound constant of a record type (`recordTypeName`). As with simple constants, these compound constants may be used in all places where a variable or parameter value of the corresponding type is read (e.g., on the right side of assignments).

5.2.2 Types

In this section, both the declaration of basic (i.e., built-in) data types and of user data types is described. This specification part is optional as well, i.e., if no user data types are required for a model, this part can be left out.

Basic Data Types cTLA supports five basic data types (cf. Table 5.1).

The `BOOL`, `BIT` data types hold boolean values in their symbolic, numeric representation. Furthermore, different ranges for numeric values are covered by the

Type	Range
BOOL	{ TRUE, FALSE }
BIT	{ 0, 1 }
BYTE	0 ... 255
SHORT	-128 ... 127
INT	-32768 ... 32767

Table 5.1: cTLA Basic Data Types

`BYTE`, `SHORT`, and `INT` types. In contrast to programming languages, types for string and character manipulation are not required.

User Data Types In section 5.2.1, we already explained the declaration of enumerations. Enumerations are a simple example of user-defined types. The cTLA language supports more advanced user-defined types as well. These user data types make use of the `ARRAY` and `RECORD` operators.

Arrays are a fixed size collection of elements $(a_0, \ldots, a_{N-1})$, where N is the cardinality of the collection, of the same data type. In cTLA, an array user data type is declared via:

```
TYPE
  ...
  arrayType = dataType[N];
  ...
```

The ith element of an array is selected using the bracket operator on the array with the parameter i (i.e., `a[i]`).

Records are a fixed size collection of elements, too. In contrast to arrays, however, the elements can have different data types. In cTLA, a record user data type is declared via:

```
TYPE
  ...
  recordType = RECORD
    fieldName1: dataType1;
    fieldName2: dataType2;
    ...
    fieldNameN: dataTypeN;
  END;
  ...
```

Each element of the record type is selected using the dot operator with the field name. For example, to access the field `fieldName2` of the record type, the syntax `recordType.fieldName2` is used.

As both array and record operators can be used recursively, complex types can be built (e.g., a record containing arrays).

5.2.3 Functions

Based on a value table, functions can be declared in cTLA using the `FUNCTION` keyword and the `IF-THEN-ELSE` operator. The generic cTLA template for a function $f(x_1, \ldots, x_m)$ mapping the input values $(v_{i1}, \ldots, v_{im})$, $i = 1 \ldots n$ to the output scalar $y_{v_{i1} \ldots v_{im}}$ is:

```
FUNCTION f(x1:INT, ..., xm:INT) ::=
IF ((x1=v11) AND ... AND (xm=v1m)) THEN y_v11...v1m
  ELSEIF ((x1=v21) AND ... AND (xm=v2m)) THEN y_v21...v2m
  ...
  ELSEIF ((x1=vm1) AND ... AND (xm=vmm)) THEN y_vm1...vmm
  ELSE <undefined value>
END;
```

Typically, to return a value for the full range of the input variables, a special *undefined value* is returned for all input values out of the scope of the function. This is realized using the `ELSE` statement.

The involved `IF-THEN-ELSE` operators have a functional meaning in this context. If the first condition is true, the first value is returned. Otherwise, if one of the `ELSEIF` branches is true, the corresponding value is returned. Finally, if none of the conditions matches, the `ELSE` value is returned.

5.2.4 Predicates

A cTLA predicate is declared as follows:

```
PREDICATE predName( par1:type1; ... parM:typeM; ) ::=
        <formula involving par1, ..., parM>
```

The *formula* may contain parameters, comparison operators, logical operators, and functions. In cTLA, the following *comparison operators* are supported:

```
<,>,=,!=,<=,>=
```

Furthermore, the following *logical operators* may be used:

```
AND, OR, NOT
```

After declaration, a predicate can be used everywhere a guard is allowed, particularly in actions. As an extension, the current version of cTLA2PC allows to define action macros (i.e., including effects) with the `PREDICATE` keyword as well.

5.2.5 Actions

As described in section 5.1.1, CTLA actions are structured into executable guards and effects, and can be transformed to the standard form shown below:

```
actionName(apar1:type1, ..., aparN:typeN) ::=  // head
  guard1          // guards
  AND guard2
  AND ...
  AND guardN
  AND effect1     // effects
  AND ...
  AND effectM;
```

Actions make up the body of a process type definition (cf. section 5.3). Optionally, parameters may be defined in the action head. The parameters are used like symbolic constants inside the actions. For parameterized actions, the definition of enabled (cf. section 3.3.1) is extended: the action is *enabled*, if all guards are satisfied by the actual values of the parameters in combination with the current values of the state variables.

5.3 Process Types

Each CTLA specification describes at least one *process type*, an instance of which corresponds to a state transition system modeling a process of this type. In contrast to PROMELA process types (cf. section 3.2.2), CTLA process types may contain *multiple actions*. Furthermore, the extending and subsystem process types allow for the *composition* of new process types from existing process types.

5.3.1 Simple Process Type

A *simple process type* definition consists of three sections: local declarations, initialization, and actions (cf. Listing 5.2).

In the *local declarations* section, the constants and state variables of the process type are defined. The *initialization* section consists of a predicate determining the initial values of the state variables. As the name suggests, the *actions section* lists the actions (cf. section 5.2.5) of the process type.

Instances of a process type define a state transition system. For the CTLA simple process type, the state space of the state transition system is directly spanned by the local state variables. In the same manner, the transitions are directly given by the local actions. This is in contrast to the composed CTLA process types: CTLA extending process type and CTLA subsystem process type.

```
PROCESS simProcType(ppar1:...);       // simple process type
/* local declarations */
CONST ...;                            // local constants
VAR ...;                              // state variables
/* initialization */
INIT ::= ...;                         // INIT predicate
/* actions */
ACTIONS                               // actions
  act1(apar1:...) ::=                   // action
    ...;
  ...
INTERNAL ACTIONS                      // internal actions
  iAct1(iapar1:...) ::=                 // internal action
    ...;
  ...
END;
```

Listing 5.2: cTLA Simple Process Type

5.3.2 Extending Process Type

An *extending process type* definition is very similar to a simple process type definition (cf. Listing 5.3). A process extension section is added; all other sections remain unchanged.

```
PROCESS ExtProcType(ppar1:...);       // extending process type
/* local declarations */
...
/* process extension */
EXTENDS OtherProcType1, ...
/* initialization */
...
/* actions */
 ...
END;
```

Listing 5.3: cTLA Extending Process Type

In the *process extension* section, the process types which are extended by this process type (i.e., `ExtProcType`) are listed. A process type may extend multiple other process types. This resembles *multiple inheritance* in object-oriented programming, where a class inherits variables and behavior from multiple other classes.

The state transition system of an instance of an extending process type is defined

as follows: Its state space is spanned by the combined state variables of all extended process types plus the state variables defined locally. Similarly, the transitions are given by combining the actions of all extended process types plus the actions defined locally. If multiple definitions exists for the same action, they are merged into a single new action. The new action results from logical conjunction of all guards and effects of the previous definition. This way, existing actions can be constrained and new actions added by an extending process type.

As an example of an extending process type, consider the process type `ActiveHostIpNode` (cf. Listing 5.4).

```
PROCESS ActiveHostIpNode( NID: NodeIdT );
EXTENDS HostIpNode( NID );  // extended process types
ACTIONS
  snd( pkt: PacketT ) ::=
    itf.usd = TRUE
    AND itf.spa.act = SPA_SND
    AND pkt = itf.spa.pkt
    AND itf.spa.act' = SPA_NONE_EMPTY;
END
```

Listing 5.4: cTLA Example Extending Process Type (Adding)

This process type extends the process type `HostIpNode`, which does not define a `snd` action. The action `snd` defined by the extending process type `ActiveHostIpNode` is added to the actions defined by `HostIpNode`. Another example is shown in Listing 5.5.

```
PROCESS NonPromHostIpNode( NID: NodeIdT );
EXTENDS HostIpNode( NID );  // extended process types
ACTIONS
  rcv( pkt : PacketT ) ::=
    pkt.dat_ida = itf.ia;
END
```

Listing 5.5: cTLA Example Extending Process Type (Constraining)

This time, both process type `NonPromHostIpNode` and process type `HostIpNode` provide definitions for action `rcv`. In this case, the existing action `rcv` defined by process type `HostIpNode` is constrained with the additional guard given in action `rcv` from the extending process type `NonPromHostIpNode`. Similarly, an extra effect could be provided by the extending process type `NonPromHostIpNode` and added to action `rcv`.

Initialization of the extending process type is done similar to the merging of guards for multiple definitions of the same action: All initialization predicates of the extended process types and the (optional) local initialization predicate are logically conjugated. As each process type defines its own name space, conflicting initializations are not possible.

5.3.3 Subsystem Process Type

The *subsystem process type* realizes cTLA's process composition concept (cf. section 3.4.2). In comparison with the simple process type definition, a section listing the contained processes is added and the initialization section is removed (cf. Listing 5.6). Furthermore, the local declarations part is restricted to constants. This is not enforced on the grammar level due to parser design considerations, but checked during semantic analysis.

```
PROCESS SysProcType(ppar1:...);     // (sub)system process type
/* local declarations */
CONST ...;
/* process containment */
CONTAINS instName1: OtherProcType1, ...
/* actions */
 ...
END;
```

Listing 5.6: cTLA Subsystem Process Type

The *process containment* section lists the processes the process type (e.g., `SysProcType`) is composed of. A process type may be composed of several processes. This is similar to multiple *aggregation* in object-oriented programming.

Again, the state transition system of an instance of a subsystem process type, is defined indirectly. Its state space is spanned by the vector of the state variables of all contained process instances. In contrast to the extending process type, the transitions are made up of the locally defined actions. As the name suggests, the subsystem process type is used for the system process. Thus, its actions are called system actions. The right side of a *system action* is a conjunction of instances' actions. It lists those actions which have to be performed jointly in order to realize the system action. Each process can contribute to one system action by at most one process action. If a process does not contribute to a system action, it performs a *stuttering step*. To avoid name clashes, the actions of an instance are prefixed with the instance name. In effect, only the system actions exist in the system. The instances' actions are executable only as part of a system action.

The initialization predicate of the subsystem process type is derived by conjugation from the initialization predicates for all contained process instances.

As an example of the subsystem process type, consider process type `SimpleConPT` (cf. Listing 5.7).

```
PROCESS SimpleConPT();
CONTAINS                    // contained processes
  nodeA: Node();
  nodeB: Node();
  physMedia: Media();
ACTIONS                     // system-level actions
  snd_na( pkt: PacketT ) ::=
     nodeA.snd( pkt ) AND physMedia.in( pkt );
  snd_nb( pkt: PacketT ) ::= ...
  rcv_na( pkt: PacketT ) ::=
     nodeA.rcv( pkt ) AND physMedia.out( pkt );
  rcv_nb( pkt: PacketT ) ::= ...
END
```

Listing 5.7: cTLA Example Subsystem Process Type

This process type contains three instances (`nodeA`, `nodeB`, `physMedia`) of two process types (`Node`, `Media`). Furthermore, it defines four system actions (`na_snd`, `nb_snd`, `na_rcv`, `nb_rcv`).

5.3.4 System Instantiation

The previously described parts of a cTLA specification provide for constants, types, functions, predicates and particularly process types. We still have to define which process type is meant to be instantiated as the relevant state transition system for the whole system. This process type is then declared in the system part of the specification as follows:

```
SYSTEM systemName: procTypeName( const1, ... );
```

This statement defines the system model to be an instance of process type `procTypeName`. If the process type is parameterized, appropriate constant values have to be supplied.

Typically, the process type used for the system declaration is a subsystem process type. It contains instances of the other relevant process types for the modeling and couples their actions on the system-level. Consider the following system declaration, instantiating the subsystem process type `SimpleConPT` (cf. Listing 5.7):

```
SYSTEM SimpleSys: SimpleConPT();
```

Figure 5.1 shows the graphical representation of the resulting system SimpleSys.

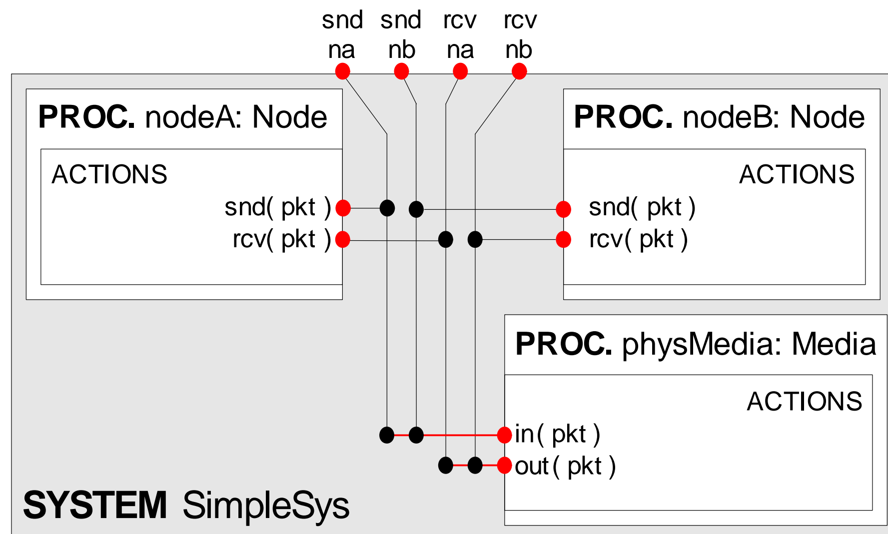


Figure 5.1: Graphical Representation of System SimpleSys

The three process type instances included and four system actions defined by the underlying process type SimpleConPT are clearly evident.

5.4 Grammar

cTLA's grammar is described using the *Extended Backus Naur form (EBNF)* [ISO96]. The main meta symbols occuring are:

- the *quote* symbol " for enclosing terminals,
- the *start-repeat* symbol { and the *end-repeat* symbol },
- the *start-option* symbol [and the *end-option* symbol],
- the *start-group* symbol (and the *end-group* symbol),
- the *repeat-zero-or-more-times* symbol *,
- the *repeat-one-or-more-times* symbol +.

A preliminary version of the grammar was developed in [RK03], then it was improved and extended in [Poh03; Mal05]. The full and final grammar is included in Appendix A. In the following paragraphs, we give a brief overview of selected high-level productions.

Start Symbol The *start symbol* for the cTLA grammar is `specification`. This symbol is defined as follows:

```
specification =
        [const_decl_part]
        [type_decl_part]
        [function_decl_part]
        [predicate_decl_part]
        process_type_decl_part
        system_instantiation_part
;
```

This production shows the structure of cTLA specifications with the optional (constants, types, functions, predicates) and required (process types, system instantiation) parts.

Constant Declararation By going three levels deeper into the constants part (symbol `const_decl_part`), we obtain the following productions:

```
const_decl_part =
        "CONST" { constant_decl ";" }+
;
constant_decl =
        constant_identifier "=" constant
;
constant =
        simple_constant_value | compound_constant
;
```

These rules show that at least one constant has to be declared and multiple may be declared after the `CONST` keyword. The values are assigned to the identifiers with "=". Such values can be either simple or compound constants.

Process Type Declarations The process types (`process_type_decl`) part is required in each cTLA specification. Some of the next-level productions for this part are:

```
process_type_decl_part =
        { "PROCESS" process_decl "END" }+
;
process_decl =
        process_heading
        [const_decl_part]
        [type_decl_part]
        [var_decl_part]
        ( simple_process_body
```

```
        | extending_process_body
        | subsystem_process_body )
;
```

These productions show the structure of each process type declaration, with required (`process_heading`), optional (e.g., `const_decl_heading`), and alternative (`simple_process_body`, `subsystem_process_body`, `extending_process_body`) subparts. From the alternative subpart, we recognize the three process types simple process type, extending process type, and subsystem process type.

Furthermore, we observe that the productions allow for the derivation of a type part for a containing process type. This is in contrast to section 5.3.3, which states that only a constant part should appear in a containing process type. By more complicated production rules, this problem can be avoided. In this case, like in a few other cases, however, we determined that it is better to check this restriction after parsing during semantic analysis. This way, the grammar can be kept simpler and a higher parsing performance is achieved.

6 Translation, cTLA2PC, and Eclipse Integration

This chapter describes the translation of cTLA 2003 specifications to PROMELA. After a short introduction motivation, we present the translation scheme. Then, we report on cTLA2PC, the automated translation tool implementing the scheme. Finally, we briefly outline our ECLIPSE integration.

6.1 Motivation

On the one hand, we want to employ cTLA 2003's compositional and object-oriented modeling features, allowing for easy extension and re-use of modeling elements (cf. chapter 5). These features lay the groundwork for taking the modeling task to another level by providing a custom framework for computer network models (cf. chapter 7). On the other hand, we aim to analyze our models with SPIN, one of the most popular and powerful finite-state model checkers (cf. chapter 3.1). Particularly, SPIN includes state of the art reduction algorithms for producing optimized executable verifiers. Unfortunately, SPIN requires its models to be specified using PROMELA (cf. section 3.2), a low-level, neither compositional nor object-oriented language.

To be able to combine the advantages of cTLA for the modeling phase and the advantages of SPIN for the analysis phase, we have to transform cTLA models to PROMELA. Thus, we engineered a translation scheme for cTLA specifications, which transforms them to PROMELA specification. To provide automated translation of cTLA specifications, we implemented the cTLA2PC tool based on the scheme. Furthermore, to ease the application of our approach, we integrated cTLA2PC together with SPIN into the ECLIPSE workbench.

6.2 Translation Scheme

First, we provide an overview of the translation scheme. Then, we give more details regarding the translation of selected cTLA language elements.

6.2.1 Overall Translation Scheme

Two key phases can be distinguished in the basic translation scheme (cf. Fig. 6.1). First, the expansion phase creates a simplified cTLA system, which consists of a single, non-composed process. Second, the code generation phase translates the simplified cTLA system to PROMELA.

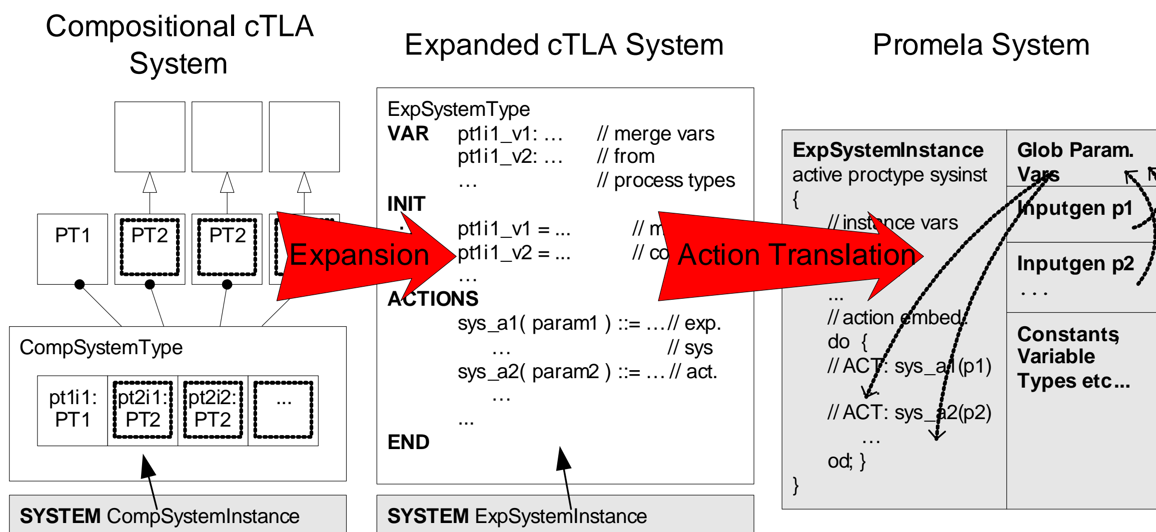


Figure 6.1: Transforming a Compositional cTLA System to Promela

6.2.1.1 Expansion Phase

A compositional cTLA system (`CompSystemInstance`) is an instance of a process type (`CompSystemType`) subsystem process type instances (e.g., pt1i1, pt2i1, . . .). Each process type (PT1, PT2, . . .) may contain or extend further process types. Unfortunately, PROMELA only supports simple process types (cf. section 3.2.2). In contrast to cTLA, these process types may not extend or contain other process types. Thus, during the *expansion* phase, we have to transform the compositional cTLA input system to a simplified cTLA system which contains only simple process types. We call this simplified cTLA system *expanded* or *flat*.

Particularly, the system process type has to be resolved. Typically, it contains process instances for all elements of a scenario and couples their actions to provide the system actions. As an example, consider the expansion of the action `snd_h1` (cf. Listing 6.1) from the IP-ARP model (cf. section 9.2).

The compositional form of the action is given by the coupling of actions from the contained process type instances `h1` (of process type `IpArpNode`) and `med` (`Media`). The expanded form of the action contains no process type instances. Instead, variables from the instances are merged in the flat system type (`ExpSystemType`). Fur-

```
// Original Action as defined in the Compositional System
snd_h1( pkt: PacketT ) ::= h1.snd( pkt ) AND med.in( pkt );

// Action after Expansion (Flat System)
snd_h1( pkt: PacketT ) ::=
  // merged guards
  pValidIf(pkt.sci, NA_MII)
  AND h1_ifs[pkt.sci - 1].usd = TRUE
  AND h1_ifs[pkt.sci - 1].spa.usd = TRUE
  AND pkt = h1_ifs[pkt.sci - 1].spa.pkt
  AND fSrcToZone(pkt.scn, pkt.sci) != UNKNOWN_ZONE
  AND med_buf[fSrcToZone(pkt.scn, pkt.sci) - 1].usd = FALSE
  // merged effects
  AND h1_ifs[pkt.sci - 1].spa.usd' = FALSE
  AND med_buf[fSrcToZone(pkt.scn, pkt.sci) - 1].usd' = TRUE
  AND med_buf[fSrcToZone(pkt.scn, pkt.sci) - 1].pkt' = pkt;
```

Listing 6.1: Compositional and Expanded Form of a cTLA Action

thermore, the guards and effects of the coupled actions are conjugated into new actions. Thus, in the flat system only a single process of a simple process type remains.

6.2.1.2 Action Translation & Promela Code Generation Phase

As the name suggests, this phase deals with handling actions and generating a PROMELA-level representation of the simple CTLA system resulting from the expansion phase. This simple CTLA system is instantiated from a single process not making use of process type composition (e.g., extension or containment). The simple CTLA system may still contain multiple, parameterized actions. In PROMELA, processes have a monolithic body (cf. section 3.2.2). Particularly, multiple actions, far less parameterized actions, are not provided.

Conceptually, we have to translate from the STS view (cf. section 3.3.3) underlying the simple CTLA system to the LFSM view (cf. section 3.2) underlying PROMELA specifications. From the executability property of CTLA 2003 specifications (cf. section 5.1.1) a finite state space and unique initial state follows. The end states required by the LFSM view are only used for liveness properties by SPIN. As we focus on safety properties, we can define an extra unreachable state as the LFSM set of end states. Thus, the main task is to translate the CTLA actions to appropriate PROMELA transition labels.

Action Embedding & Code Translation The set of cTLA actions defined in the simple system are embedded into the body of a PROMELA process type. As actions have to be executable an arbitrary number of times, they are enclosed in a PROMELA non-deterministic `do` selection loop.

The translation of the actions themselves, which are structured into guard and effect statements, can be done quite easily. Most guards can directly be translated into PROMELA guarded statements. Quantified guards (cTLA keywords `FORALL`, `EXISTS`) have to be handled by the introduction of local loop blocks which set corresponding boolean guard variables. Regarding effects, most can be translated into simple PROMELA level assignments. Array effects (`UPDATEALL`) are again translated into local loop blocks. Following the outlined scheme, the action code can be translated to PROMELA. The action parameters still have to be handled, however.

Action Parameters According to cTLA semantics, a parameterized action is executable, if a parameter setting exists so that the action is executable with this parameters (cf. section 5.2.5). We have to reproduce this behavior in PROMELA.

This is done in two steps. First, for each action and each parameter, a global variable is introduced. The parameter is then removed from the parameter list of the action and the occurrences of the parameter inside the action are replaced with the global variable. Second, for all newly introduced global variables, an input generator PROMELA process is created. The input generator for a variable non-deterministically assigns a value (out of the range of values for the parameter type) to the variable. As the input generator runs in its own process, any value can be assigned at any time. Particularly, all value assignments corresponding to parameter settings so that the action is executable, are reachable. As the model checker tries all reachable paths during verification, this reproduces the behavior of the cTLA model.

The input generator processes use the *randomness non-deterministic if-approach* described in [Ruy01]. Different actions may (re-)use the same global variables and input generator processes as long as they have the same parameter types. Thus, the number of additional variables and processes is reduced. We also tried using PROMELA's channel construct instead of global variables. As channels are implemented as FIFO queues, global variables proved to be more efficient.

After the handling of action parameters, the action translation phase is finished. The described basic translation scheme works well, but is relatively costly in terms of possible transitions and – to a lesser extent – state space. Thus we developed optimizations, especially for more efficient handling of parameterized actions. These optimizations are described in section 8.4.

6.2.2 Translation of Selected Language Elements

In this section, we give more details on the translation of the most important cTLA language elements (cf. chapter 5) to PROMELA.

6.2.2.1 Constants

Consider a simple, symbolic cTLA constant `simpleConstName`:

```
CONST simpleConstName = value;
```

In PROMELA, symbolic constants are not supported. As PROMELA adopts the C language preprocessor, however, symbolic constants can be implemented using preprocessor (`cpp`) macros. Thus, the above statement is translated to the macro:

```
#define simpleConstName value
```

This macro tells the preprocessor to replace all occurrences of `constName` in the PROMELA source with `value`.

Enumerations are a slightly more complicated case. Given a cTLA enumeration `enumName`:

```
TYPE enumName = {name1, name2, ..., nameN}
```

In PROMELA this enumeration is realized via:

```
#define enumName byte
#define name1 0
#define name2 1
...
#define valueN N-1
```

Of course, if $N > 256$, then an `int` type instead of `byte` is used for mapping `enumName`.

Compound constants can be splitted into simple constants for all fields, array elements. This way, compound constants are reduced to simple constants.

6.2.2.2 Types

The basic data types of cTLA can be mapped to the basic data types of PROMELA in a straightforward way (`BOOL` $\mapsto$ `bool`, `BYTE` $\mapsto$ `byte`, etc).

Furthermore, both cTLA and PROMELA support array types. A cTLA array declared via

```
arrayName: ARRAY[size] OF type;
```

is translated to the PROMELA array

```
arrayName ptype[size];
```

where `ptype` is the PROMELA data type corresponding to the TLA data type as described above.

Basic types and array types can be used to build composed user-defined data types. In cTLA such a type looks like this:

```
TYPE userTypeName = RECORD
  fieldName1: type1;
  fieldName2: type2;
  ...
  fieldNameN: typeN;
END;
```

By mapping each `type1,...,typeN` as described above, this user type can be realized in PROMELA as follows:

```
typedef userTypeName {
  type1' fieldName1;
  type2' fieldName2;
  ...
  typeN' fieldNameN;
}
```

6.2.2.3 Logical IF-THEN-ELSE Operator

The cTLA logical `IF-THEN-ELSE` operator (cf. section 5.1.1) combines guards and effects, for instance:

```
IF x=4 THEN y'=1
ELSEIF x>2 THEN y'=2
ELSE y'=0
END;
```

PROMELA supports a *selection construct*. It allows to define several guarded options. From the options where the guards are true, one is selected non-deterministically for execution. Using this construct, the above expression is translated to:

```
if
:: (x==4) -> y=1;
:: (x!=4) && (x>2) -> y=2;
:: else -> y=0;
fi;
```

Note the added `(x!=4)` guard in PROMELA. This is to ensure the cTLA `ELSEIF` semantics, i.e., the `ELSEIF` can only be executed if the `IF` cannot be executed.

6.2.2.4 Functions

Consider the following cTLA function:

```
FUNCTION func(x:INT) ::=
   IF ((x=0) OR (x=1)) THEN 1
   ELSEIF (x=2) THEN 3
   ELSEIF (x=3) THEN 4
   ELSE 0
END;
```

PROMELA does not support functions. There exists a *choice construct*, however. The choice construct has the form `c -> v1:v2` and works in the following way: if condition c is true, return value $v1$, else return value $v2$. Combined with preprocessor macros, this allows the above function to be realized in PROMELA via:

```
#define func(x)\
   (((x==0) || (x==1)) -> 1:\
    ((x==2) -> 3:\
     ((x==3) -> 4:0)))
```

Then, an occurrence of `func(a)` in the CTLA input can be translated by putting `func(a)` (i.e., a call of the macro `func(x)`) in the PROMELA code. Finally, the preprocessor will replace `func(a)` with the macro and replace `x` with `a`.

6.2.2.5 Predicates

A CTLA predicate is defined like exemplified by the following example:

```
PREDICATE pred( x:BYTE; y:BYTE ) ::= (0 < x) AND (y <= 2);
```

We transform this predicate into a PROMELA expression using a preprocessor macro:

```
#define pred(x, y)((0 < x) \
        && (y <= 2))
```

Similar to the already described translation of functions, every occurrence of `pred(a,b)` is included in the output and finally replaced by the preprocessor.

6.2.2.6 Guards

Simple guards are boolean expressions combined by the logical operators corresponding to the CTLA keywords `AND`, `OR`, and `NOT`. Consider for example the following simple guard:

```
 (array[i].usd = TRUE)
AND ((a > b) OR NOT (c > d ))
```

Assuming the data types and variables have already been mapped, it can be translated by simply mapping the logical operators between CTLA and PROMELA (`AND` $\mapsto$ &&, `OR` $\mapsto$ `||`, `NOT` $\mapsto$!):

```
(array[i].usd == true)
&& ((a > b) || (!(c > d)))
```

Quantified guards are a more complicated case. With the keywords `EXISTS` and `FORALL`, cTLA supports quantifications in guards. Consider for example the following cTLA guard expression:

```
EXISTS i IN {a..b}: [ guard(i) ] // quantified guard
```

The guard expression is true, if `guard(i)` is true for *any* i between a, b, inclusively. Similarly, the cTLA guard expression

```
FORALL i IN {a..b}: [ guard(i) ] // quantified guard
```

is true, if `guard(i)` is true for *all* i between a, b, inclusively. Quantified guard expressions are realized in PROMELA by the introduction of loop code blocks and temporary variables. For example, the above `EXISTS` guard expression is translated to:

```
hidden byte i_EXISTS_L0;           // temporary loop variable
i_EXISTS_L0=a;                     // initialize loop variable
hidden bool i_EXISTS_L0_R=false; // temporary loop result
do
:: (i_EXISTS_L0 <= b) ->           // loop until upper bound
  if
  :: (guard(i) == true) ->         // if guard(i) true
    i_EXISTS_L0_R=true; break;     // save result, exit
  :: else ->                       // guard(i) not true
    i_EXISTS_L0++;                 // increase loop variable
  fi;
:: else -> break;                  // upper bound exceeded
od;
```

6.2.2.7 Effects

Simple effects are assignment expressions combined by the cTLA keyword `AND`. Consider for example the following simple effects where `arr` is a variable representing an array of records with a field `usd` and `recvar1`, `recvar2` are records with three fields `x`, `y`, `z`:

```
arr[i].usd' = TRUE
AND recvar1' = recvar2
```

Assuming data type and variable mapping are already done, the PROMELA translation looks like this:

```
arr[i].usd=true;
recvar1.x = recvar2.x;
```

```
recvar1.y = recvar2.y;
recvar1.z = recvar2.z;
```

Note that the record variable assignment has to be splitted into its parts, because PROMELA does not directly support multi-field assignments.

Array effects (cTLA keyword UPDATEALL) are a more complicated case. An effect is applied to each element of an array. Consider the example:

```
UPDATEALL i IN {a..b}: [ effect(arr[i]) ]
```

This effect is realized in PROMELA by the introduction of loop code blocks with temporary variables similarly to the quantified guards described above.

```
hidden byte i_UPDATEALL_L0;         // temporary loop variable
i_UPDATEALL_L0=a;                   // initialize loop variable
do
:: (i_UPDATEALL_L0 <= b) ->         // loop until upper bound
  effect(arr[i_UPDATEALL_L0]);      // affect element i
  i_UPDATEALL_L0++;                 // increase loop variable
:: else -> break;                   // upper bound exceeded
od;
```

6.2.2.8 Actions

cTLA actions are structured into guards and effects. We already described how guards and effects can be translated to PROMELA. Furthermore, in section 6.2.1.2 we explained the embedding of actions into a non-deterministic do selection loop and the handling of action parameters via global parameters and input generator processes.

Taken together, these concepts describe the translation of actions. The generated PROMELA outline for the actions looks like this:

```
do                // non-determin. action selection loop
:: d_step {       // ACTION A1: parameters replaced
     ...          //   translated guards
     ...          //   translated effects
   }
...
:: d_step {       // ACTION An: parameters replaced
     ...          //   translated guards
     ...          //   translated effects
   }
...
od;
```

Furthermore, the input generator process generated for a replaced action parameter takes the following form:

```
active proctype param_ParameterName_InputGen()
{ do
  :: param_ParameterName = ...; // first value
  ...
  :: param_ParameterName = ...; // last value
  od;
}
```

6.2.2.9 System Instantiation

After the expansion phase (cf. section 6.2.1.1), only a single, simple system processtype remains. Thus, the system process instantiation can be easily translated by instantiating and running the PROMELA version of that process. As described in section 3.2.2, it suffices to add the PROMELA keyword `active` to the `proctype` declaration corresponding to the system process:

```
// system process
active proctype SysProcType()
{ ... }
```

We conclude this section with the outline of the PROMELA specification (cf. Listing 6.2) that is generated for a typical cTLA model (cf. the cTLA outline in Listing 5.1).

Note the PROMELA input generator processes generated for the `PacketT` data type which is used as a parameter in action `snd_A(pkt:PacketT)`.

6.3 The cTLA2PC Translation Tool

We devised cTLA2PC, a tool for translating cTLA specifications to PROMELA. cTLA2PC implements the translation scheme described in section 6.2. Most of the cTLA2PC's implementation was done as part of the master theses of Andre Pohl [Poh03] and Marc Malik [Mal05]. In the following sections, the architecture, implementation and finally some extended translation options of cTLA2PC are outlined.

6.3.1 Architecture

The compiler is made up of six key components. These components are:

Scanner and Parser The scanner and parser component builds a syntax tree from the cTLA input. In a *syntax tree*, the input file is represented in a tree structure matching the language's grammar. Along the way, the syntax of the input file is checked and errors are flagged. Furthermore, the symbol table is created.

```
#define H1_I1_ID ...      // constants
...
#define H1_ID 1           // enumerations
...
#define fSrcToZone(n,i)\ // functions
...
#define pValidIf(pi,pm)\ // predicates
...
typedef PacketT          // typedefs
{ NodeIdT scn;
  ...    }
...
PacketT param_PacketT;   // global parameter variables
...
// system process
active proctype SysProcType()
{ ... }

// input generator process for first field scn of PacketT
active proctype param_PacketT_scnInputGen()
{ ... }

// input generator process for other fields and parameters
active proctype param_PacketT_...
```

Listing 6.2: Generated Promela Specification Outline

Symbol Table A *symbol table* is a data structure based on an hash table that allows for quick storing and retrieving of symbols (basically name and type pairs). The symbol table component provides both the data structure and the operations for storing and retrieving symbols. This component is called by the parser component to construct a symbol table matching the syntax tree.

Semantic Analysis During semantic analysis particularly the correctness of expressions involving types is checked. This includes return values of CTLA functions and inferring of types for constants. *Implicit type conversions*, e.g., for assigning a `BYTE` type to an `INT` type, are performed as well.

Furthermore, some invalid CTLA expressions can be avoided either by a sophisticated grammar or by simple checks during semantic analysis. In these cases, we opt for the simpler grammar and perform the checks during semantic analysis instead. This design decision brings about better translator performance, as the parser performance is related to the grammar's complexity.

Expansion Based on the syntax tree and the symbol table, this component expands a cTLA system to a simple cTLA system as described in section 6.2.1.1. The result is a new syntax tree and symbol table for the simple cTLA system.

Code Generation The code generation component uses the modified syntax tree and symbol table resulting from the expansion phase to build a PROMELA representation of the input system. For generating the PROMELA representation, the scheme outlined in 6.2.1.2 is applied. Furthermore, this component allows the use of different back-ends for languages other than PROMELA. For example, we integrated a back-in for cTLA output. This is helpful to study the effects of process type composition and optimizations applicable at this level (cf. section 8.4).

Plug-in Interface cTLA2PC is designed with extensibility in mind. Thus, a plug-in interface is provided which allows easy access to the syntax tree and symbol table of the simple cTLA system. By developing a small plug-in, a wide range of additional transformations can be integrated into cTLA2PC's translation process. Particularly, optimizations (cf. section 8.4, 8.5) can be realized as plug-ins.

Altogether, a cTLA2PC translation process works as depicted in Figure 6.2.

First, the input cTLA specification is analyzed by the scanner and parser components. If syntax errors are encountered, cTLA2PC prints an error message and the translation halts right after the parsing phase. After scanning and parsing, the semantic analysis is applied. Semantic analysis adds type checking of action parameters, function return values and assignments. Again, errors are flagged and stop the translation process. Then, the expansion is conducted, followed by optional optimizations. Finally, the translation is completed with the PROMELA (or cTLA) code generation phase.

6.3.2 Implementation

cTLA2PC is implemented using the JAVA language. Overall, cTLA2PC's source code has about 21,000 lines and is structured in 16 packages (including sub-packages). In the following paragraphs we give a very brief overview of the implementation of cTLA2PC's components. Further details implementation are described in [Poh03; Mal05].

Scanner and Parser The scanner and parser components are based on the ANTLR [PQ95] parser construction kit. In section 5.4, we described the EBNF grammar for cTLA. ANTLR accepts an extended variant of EBNF as input grammar. Consider for example the production from the EBNF grammar dealing with the constants declaration part:

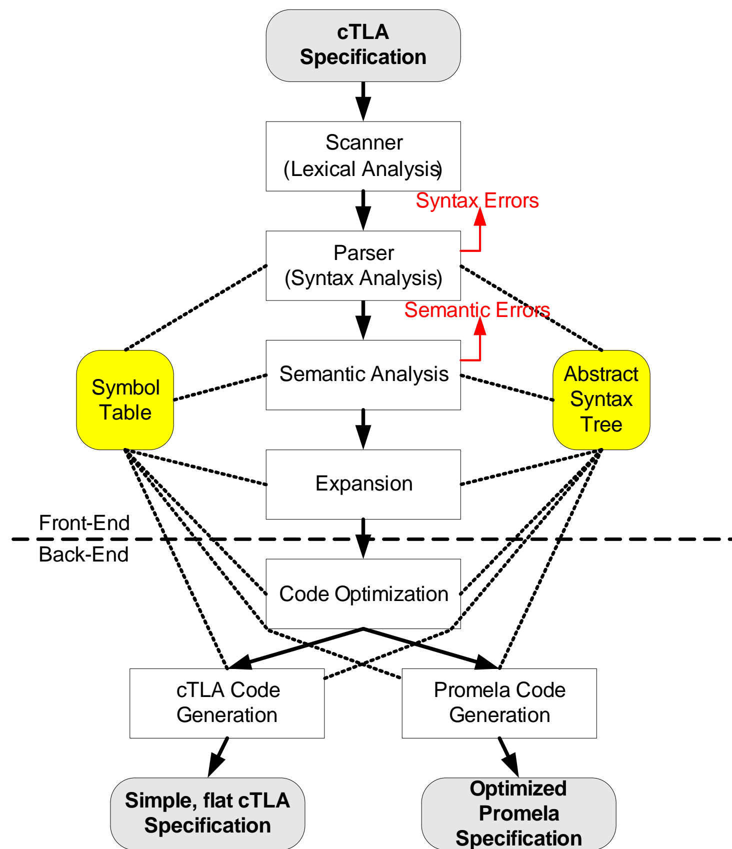


Figure 6.2: cTLA2PC Translation Process

```
const_decl_part =
        "CONST" { constant_decl ";" }+
;
```

Using ANTLR's EBNF variant, this production takes the form:

```
const_decl_part:
        "CONST"! ( constant_decl )+
        (* AST Action *)
        { #const_decl_part =
          #([CONST_DECL_PART, "[const_decl_part]"],
            #const_decl_part);
        }
;
```

Besides minor syntactical differences, ANTLR allows the inclusion of so-called *Abstract Syntax Tree (AST) actions* inside the curly brackets ({, }). AST actions have nothing to do with CTLA actions. Instead, they are used to automatically construct selected subtrees during parsing. For example, the AST action include in

the constant_decl production generates a new subtree for token type CONST_-DECL_PART, names the root node of this subtree [const_decl_part], and returns it.

Furthermore, the grammar has to be of the *LL(k)* type (i.e., it must not include left recursive productions) [ASU06]. Then, ANTLR is able to create JAVA source code for an advanced scanner and parser component for the grammar automatically.

Semantic Analysis For performing the static semantic analysis, the syntax tree and symbol table are traversed and transformed. We make use of the Visitor design pattern [GHJV95] which is especially suited to such tasks. This pattern allows to traverse complex data structures and perform flexible operations on the nodes. Particularly, the pattern largely separates the operations from the data structures. Thus, new operations can be introduced or existing operations be extended without having to modify the nodes.

The semantic analysis is done with the help of the class SemanticAnalysis-Visitor. This class implements a specialized Visitor pattern to traverse the tree. Particularly, a type check for all assignments is done.

Expansion Like with the semantic analysis component, the expansion component makes use of the Visitor pattern. Two Visitor classes for renaming and resolving actions (NodeRenameVisitor, SubtreeReplacementVisitor) are defined.

Action Handling & Code Generation The action handling and PROMELA code generation component is implemented using a specific *Visitor* class as well, this time the PromelaCodeBuilderVisitor. For the optional flat CTLA code generation, essentially the existing current tree has to be output.

Plug-Ins The plug-in interface is implemented by providing methods for registering and unregistering plug-ins. Furthermore, the plug-ins themselves are based on the Visitor pattern again.

6.3.3 Extended Translation Options

Besides implementing the translation scheme (cf. section 6.2) and various optimizations (cf. section 8.4, 8.5), CTLA2PC supports additional translation options. These translation options are used for special cases.

Simulation For example, the --simulation option provides a model better suited to SPIN's simulation mode. This option includes special code that helps to

check specific sequences during simulation. Particularly, this is useful for model validation. The special code includes a control flow generator for actions with an additional guard at the beginning of each action. It allows the execution of the action only if the action has been selected by the control flow generator. This enables scripted testing of execution sequences. Furthermore, we integrate symbolic `mtype` action names in the guards. This makes the selection dialogs in SPIN's interactive simulation mode much more comprehensible.

Mapping Sequences During SPIN verification, sequences violating security properties may be found in a model. Such violating sequences are saved in an internal encoding in a trail file. The trail file can be played back at the PROMELA level using SPIN's guided simulation mode. We are interested in the corresponding cTLA-level sequence, however. Thus, we have to map the PROMELA level statement sequence back to cTLA level actions. This mapping of sequences is greatly simplified by translating the model with the `--trace-points` switch. By supplying this switch code is inserted at the beginning of the PROMELA realization of each cTLA action that outputs the action name and parameters. Thus, by running a SPIN guided simulation from the trail file, the cTLA-level action sequence is output as well.

Debugging Support Further options exist that support the debugging of cTLA specifications from an integrated development environment (IDE). Particularly, the option `--map` provides a line by line mapping between the cTLA and PROMELA version of a model. Based on these options, we engineered an integration of cTLA2PC into the ECLIPSE platform [Kne04]. This integration is described in the following section.

6.4 Eclipse Integration

The ECLIPSE workbench is a well-known, widely adopted *universal tool platform* [OTI03]. We integrate cTLA2PC together with SPIN and extended debugging features into ECLIPSE to provide a comprehensive modeling, translation, and analysis environment for our approach. In the following paragraphs, the architecture and features of the integration are described.

6.4.1 Architecture

A modern plug-in architecture [Bol03] allows extension and customization of ECLIPSE's functionality. ECLIPSE itself is realized as a set of plug-ins which provide services. The basic workbench user interface, for instance, is provided by the

Workbench `.ui` plug-in. Plug-ins collaborate using extension points. An *extension point* provides several slots through which the extended and extending plug-ins can communicate by registering callback objects.

The plug-ins are activated by ECLIPSE as follows: During start-up, the plug-in folder is scanned for `MANIFEST`-files, typically called `plugin.xml`. This file is provided by every plug-in and contains a description of the plug-in indicating which extension points are implemented. This information is saved in a temporary database. Thus, the plugin-code itself will only be loaded if necessary.

Our integration makes use of this plug-in architecture. We provide a set of 8 ECLIPSE plug-ins (cf. Fig. 6.3, depicted as a UML component diagram) which are implemented by 70 JAVA classes, totaling about 12,000 lines of code.

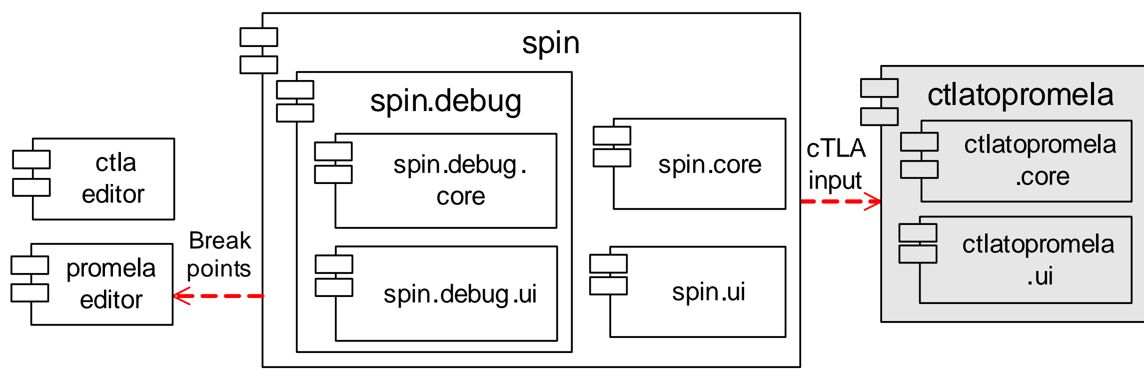


Figure 6.3: Plug-in Architecture of the Eclipse Integration

The `ctlatopromela` plug-in integrates the cTLA2PC translator into ECLIPSE. User interface elements are implemented by the `.ui` component, the corresponding non-graphical functionality is implemented by the `.core` component.

Except for the `promelaeditor` and `ctlaeditor` plug-ins, all plug-ins are separated into a `.ui` and a `.core` component. The underlying architectural pattern of the ECLIPSE framework is that different UI implementations can be used to present the same core functionality. Communication between UI and core components is handled via events. As some dependencies exists, the conceptional separation between different plug-ins cannot always be implemented to the last consequence. As an example, the `spin.debug.ui` plug-in needs to know the name of the `promelaeditor` plug-in in order to register with extensions points for providing the breakpoint functionality and for not interfering with the other plug-ins.

6.4.2 Features

Taken together, the plug-ins in combination with the ECLIPSE workbench provide the following key features:

Editing of Specifications cTLA (and PROMELA) specifications can be edited; and typical capabilities (Search/Replace, Cut/Paste, Open/Save) are provided. Furthermore, *syntax highlighting* is made available. To support the debugging features described in the paragraphs below, breakpoint markers can be set in the editors.

The editing functionality is implemented by two plug-ins (`promelaeditor`, `ctlaeditor`), which both extend the `TextEditor` class. Thus, most of the editing capabilities are inherited from ECLIPSE.

Specification Translation Translation of cTLA (and PROMELA) specifications is supported from within ECLIPSE. The translation options for cTLA2PC can be specified and saved to or restored from a configuration. During translation, each syntax error creates a new entry in ECLIPSE's tasks pane. Double-clicking an entry in the tasks pane scrolls to the corresponding source line in the editor.

These features are implemented by the plug-ins `ctlatopromela.ui`, `ctlatopromela.core` and `spin.ui`, `spin.core`, respectively. The `.ui` plug-ins contain the dialogs, e.g., for configuring translation options. The `.core` plug-ins run the cTLA2PC or SPIN tool in the background and capture the output. This is done with the help of ECLIPSE's *launching architecture* for external tools. For each tool, an environment is derived from the `LaunchConfigurationType` type. This type specifies a method `launch` which executes the tool with a given configuration. A `Configuration` contains a set of parameters as name-value pairs. Actual tool executions with actual parameter values are instances of the `LaunchConfigurationType` type. Default values for source and destination file are derived from the currently selected workspace resource. A `LaunchConfigurationDialog` shows the parameter values and allows their modification prior to the launching of the tool. Furthermore, additional parameters may be given. The captured output is parsed for translation errors which are then transferred to the tasks pane using ECLIPSE's `Markers` mechanism.

Simulation and Debugging Simulation of translated cTLA specifications is supported from within ECLIPSE. In random simulation mode, SPIN's output is simply captured and transferred to ECLIPSE's console window. For interactive simulations, the output is parsed and an interactive selection dialog is displayed for each nondeterministic choice (cf. Fig. 6.4).

Choices marked by SPIN as "unexecutable" are not displayed in the selection dialog. The debugging of translated specifications is supported as well. Breakpoints can be set in the PROMELA editor. If the corresponding line of the specification is hit, the simulation will be stopped. The user can then resume the specification simulation or *single step* through it. Additionaly, variables can be added to the watch window. This means that the current value of such a variable is always displayed by ECLIPSE.

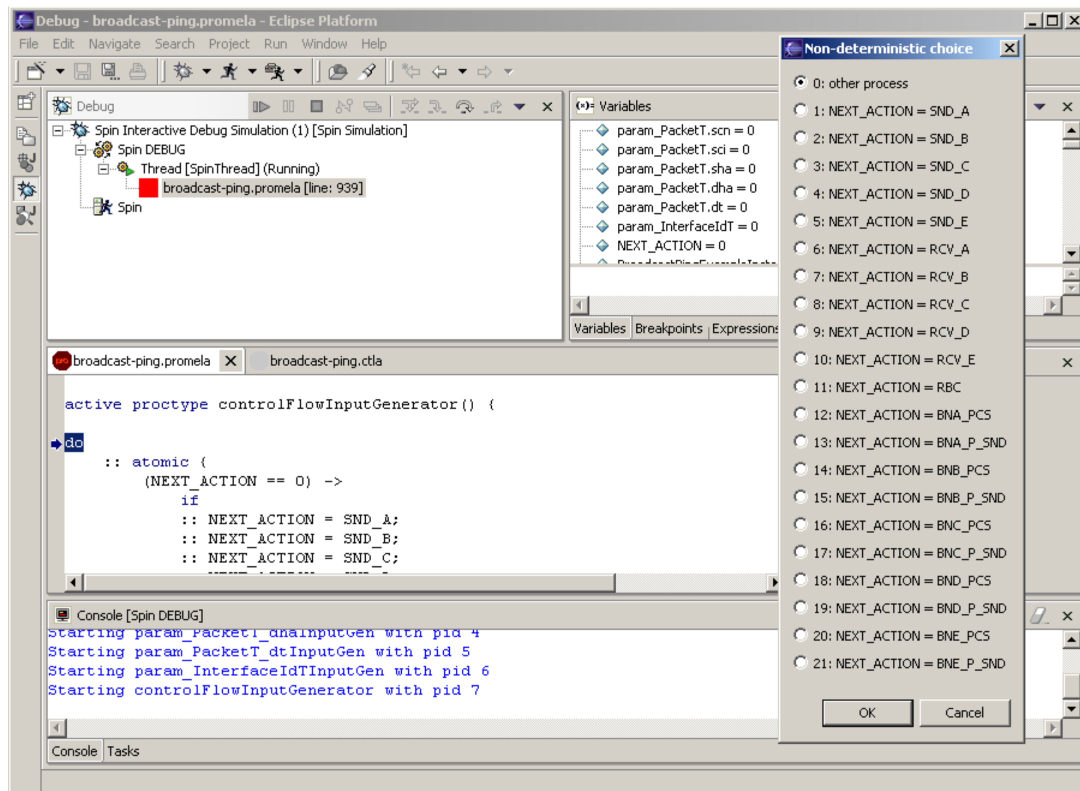


Figure 6.4: Interactive Simulation of a Translated Specification in Eclipse

The plug-in `spin.core` implements the functionality to run the SPIN tool in the background based on the launching architecture as described above. A new `LaunchConfigurationType` is defined for SPIN simulation. The `spin.ui` plug-in contains a dialog for setting additional SPIN options based on ECLIPSE's `LaunchConfigurationDialog` and the selection dialog for interactive simulation. The `spin.debug.core` plug-in parses SPIN's output and detects changes of watched variables, hit breakpoints etc. If breakpoints are defined, a `CodeModifier` is applied to the translated specification prior to starting the simulation. It inserts the following code for each breakpoint:

```
printf("MSC: break?" + nextBreakpoint.getFileName() +
   ":" + nextBreakpoint.getSourceLineNumber()+ "\\n\" );
```

This simple implementation of breakpoints works as follows: The plug-in captures SPIN's output using a buffer of a limited size and scans it for the `MSC:` marker. If the marker is found, a breakpoint has been hit. The breakpoint's file and line number can be extracted from the extra information after the question mark. This implementation of breakpoints is similar to XSPIN [Hol03].

Verification Finally, translated specifications can be verified from within ECLIPSE. Parameters for verifier generation (e.g., `-a`), verifier compilation (e.g., `-DBFS`) and verifier execution (e.g., `-m1000`) can be modified with a dialog by the user. SPIN's verification output is then displayed in ECLIPSE's console window.

The described functionality is implemented through a further `LaunchConfigurationType`, `LaunchConfigurationDialog` in the `spin.core`, `spin.ui` plug-ins, respectively.

Thanks to the core services inherited from ECLIPSE, our integration also covers further aspects, e.g., aggregation of files related to a specification into a project.

7 Computer Network Modeling Framework

In this chapter, we describe the cTLA computer network modeling framework. We begin with a short introduction outlining the purpose and application domain of the framework. Then, we give an overview of key related networking concepts in the second section. In the third section, we describe the framework from a large-scale and a small-scale view. Finally, the chapter concludes with a section detailing the framework's packages and elements.

7.1 Frameworks

Designing models for computer network scenarios integrating different aspects (e.g., protocol, node, and network) is an expensive task. Particularly, the right abstraction level must be chosen. On the one hand, all key aspects of the scenario have to be captured. On the other hand, a very detailed model naturally has a state vector that makes automated analysis very difficult – even after applying advanced optimizations (cf. chapter 8).

One of the key goals of our approach is ease of use (cf. chapter 4.1). Especially, we have to ease the modeling task. Our modeling language, cTLA 2003, provides basic mechanisms like process types, process type extension, and containment. These mechanisms allow for compositional and reusable models. We aim to facilitate the modeling task on a higher level, however. The concepts of patterns and frameworks are well-known from the world of *object-oriented programming*. As defined by Gamma [GHJV95, pp. 26]:

> The *framework* dictates the architecture of your application. It will define the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. [...] The framework captures the design decisions that are common to its application domain.

As the definition shows, frameworks facilitate the modeling task on a higher level. Thus, we decided to carry over the framework concept from object-oriented programming to computer network specifications. While designing computer network specifications for different scenarios, we identified common architectural elements.

These elements form the basis of the cTLA computer network modeling framework. It defines both basic structure, i.e., typical elements like nodes, interfaces and media with their coupling, and basic behavior, i.e., sending and receiving actions, of computer networks. A specific model has to add its own elements (e.g., nodes that include processing for a specific protocol, a lossy transfer medium, additional data types etc), but the overall architecture is given by the framework.

With the rise of the Internet, TCP/IP has become the prevalent networking technology. For this reason, we choose TCP/IP-based computer network attack models as the application domain for our framework. In particular, we aim to model and analyze scenarios involving dynamic routing.

7.2 Networking Concepts

To understand the framework and the case studies (cf. chapters 9, 10, and 11) some background on networking concepts is required. In this section, we give a brief overview of the TCP/IP reference model, the Internet routing architecture, and routing attacks.

7.2.1 TCP/IP Reference Model

The *TCP/IP reference model* structures protocols into five layers (cf. Fig. 7.1): application, transport, internet, network interface, and physical.

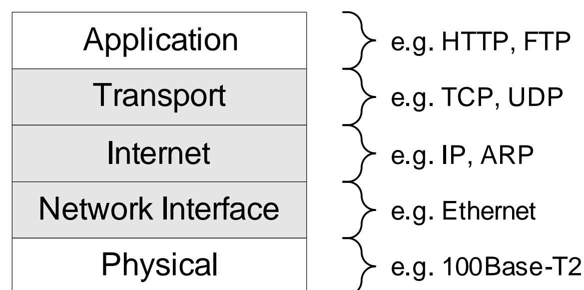


Figure 7.1: Layers of the TCP/IP Reference Model

This contrasts the seven layers of the *ISO model*. The difference is mainly due to the topmost layer, application, of the TCP/IP reference model. The layers presentation and application of the ISO model are both represented by the application layer in the TCP/IP model. Furthermore the layers session and network of the ISO model have no direct correspondence in the TCP/IP model. The TCP/IP model adds the internet layer, however.

Typical examples for protocols in the *application layer* are the well known web protocols HTTP and FTP. The *transport layer* contains protocols like TCP and UDP. On

the *internet layer* the IP protocol with its virtual addressing scheme, the IP addresses, is defined. Thus IP networks are independent of the actual addressing scheme supported by the underlying network hardware and can transfer packets between different network technologies (e.g., ATM and Ethernet). Hardware frames can be sent and received using the *network interface layer*. The actual encoding of the frames into electrical signals depending on the transmission media is handled by the *physical layer*.

During the development of the framework, the TCP/IP reference model provided guidance. In the framework, basic implementations of the network interface to transport layers are provided. On the one hand, concrete scenario models typically have to extend certain layers (e.g., for adding ARP processing). On the other hand, typically only a subset of all layers is required for a concrete scenario.

7.2.2 Internet Routing Architecture

Three levels have to be distinguished for routing in the Internet context (cf. Fig. 7.2). On the lowest level, packets have to be routed between hosts in the same physical network (typically a LAN). This level of routing is provided by the physical addressing scheme of the underlying physical network (e.g., Ethernet or Tokenring). One of the strengths of the TCP/IP protocol suite, however, is the ability to interconnect networks based on different technologies. Thus the *address resolution protocol (ARP)* (cf. chapter 9), which encapsulates different physical addressing schemes, is defined. It allows a unified way of routing *inside* a physical network based on IP addresses.

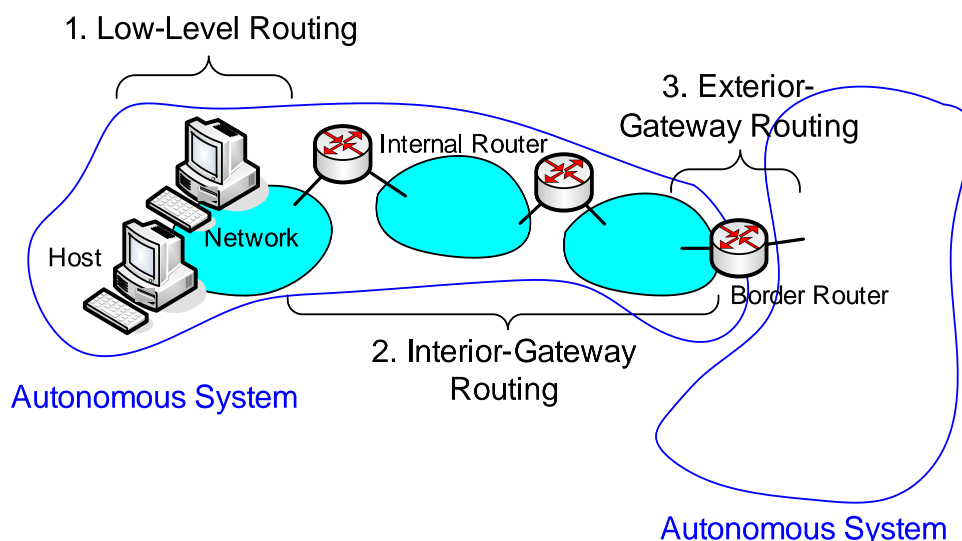


Figure 7.2: Threefold Internet Routing Architecture

The second level deals with routing *between* physical networks. Various so called *interior-gateway* routing protocols are available on this level. The two most popular interior-gateway routing protocols are the *routing information protocol (RIP)* (cf. chapter 10) and the *open shortest path first (OSPF)* (cf. chapter 11) protocol. Sets of routers connected by the same instance of an interior-gateway routing protocol are called *autonomous system (AS)*. A typical example is the set of routers belonging to the same administrative authority (e.g., an organization or company). Routers inside an autonomous system are called *inside routers*.

On the third level, routing between autonomous systems is considered. For this purpose, *exterior-gateway* routing protocols are used. A typical example is the routing between the autonomous system of a company and the autonomous system of its *Internet Service Provider (ISP)*. Nowadays, almost exclusively, the *border gateway protocol (BGP)* is used for exterior-gateway routing. Usually only one or a few routers of an autonomous system connect to other autonomous systems. These routers are called *border routers*.

7.2.3 Routing Attacks

The aim of routing attacks is to violate security properties by consistently injecting false route information into the routing process. If the attacker controls a hop on the route from the source to the destination, this is comparatively easy to achieve. Even when the attacker is not regularly on the route, however, such an injection is often possible. For example, the attacker may send out a special crafted update packets to several hops on the route. Depending on, for instance, the routing protocol, the network topology, the packet propagation, the existing routing tables, and the contents of the update packet, the injection may be successful.

If the update packet is accepted by one of the routers involved in the routing process, it may spread further, again depending on conditions as described above. The targeted router then distributes a modified update packet to neighboring routers (so called *triggered update*). If several such receive-process-modify-distribute cycles occur, the attacker may be able to affect routing in several networks.

The effects of such attacks are for example *black hole routers*, i.e., routers that pull all nearby packets to themselves but never forward them to any other destination or *man-in-the-middle* nodes, i.e., attackers that are able to intercept packets. Even if the packets are securely encrypted, sensitive information may still be leaked (*traffic analysis*). In contrast to a black hole router, the interception is usually not observable for both original sender and intended receiver of the packets.

7.3 Domain View

In this section, we describe the view TCP/IP computer network scenarios taken by the framework. Particulary, the view has to integrate node, network, and protocol aspects. First, we describe the large-scale view, mainly capturing network aspects, then the small scale-view focusing on the individual nodes and their protocol processing.

7.3.1 Large-Scale View

The large-scale view of computer network scenarios is exemplified in Figure 7.3.

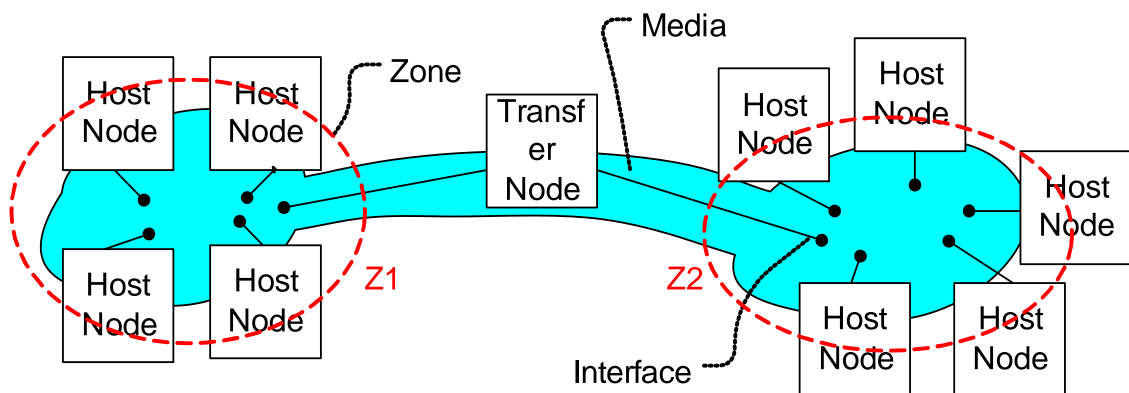


Figure 7.3: Large-Scale Network View

All active network elements are modeled by *nodes*. Nodes are connected by the physical transfer *media*, which is partitioned into *zones*. A zone corresponds to broadcast zones, i.e., the nodes inside a zone can directly communicate with every other node in the same zone. Zones can also be interpreted as network segments or subnets. Nodes communicate using interfaces, which connect to the media. A node is said to belong to a zone if it has an interface in the media's zone. *Interfaces* transmit and receive packets. A node which is connected to multiple zones (i.e., has at least two interfaces) is called a *transfer node*. Transfer nodes (or routers) provide inter zone communication. A node with just one interface is a *host node*.

7.3.2 Small-Scale View

The small-scale view focuses on the nodes and the protocol processing. For the processing actions, we follow techniques from efficient protocol implementation, particularly the activity thread and integrated layers approach (cf. section 8.3.2). Packets are sent and received by the node via actions `snd` and `rcv` (cf. Fig. 7.4).

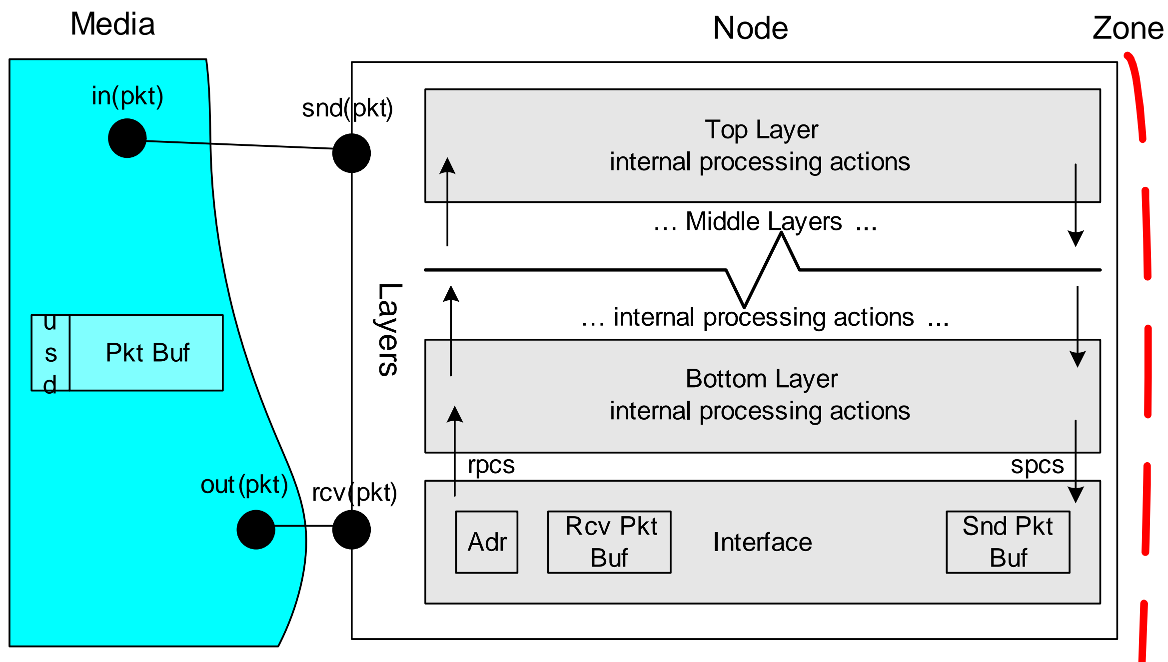


Figure 7.4: Small-Scale Network View

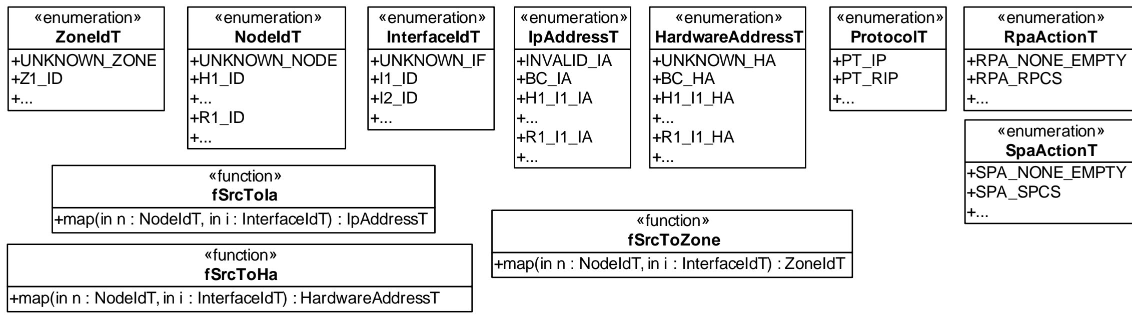
The node's actions `snd`, `rcv` are coupled to media's actions `in`, `out` respectively. Inside the node, the packet processing is structured into layers. A valid packet that is received from media by an interface is stored in the interface's receive buffer and then processed through the layers (action `rpcs`). A packet which shall be sent is processed (action `spcs`) the other way around, down the layers until it has reached the interface level. The exact layers and processing steps required depend on the protocols occuring in a scenario model.

If media does not already contain a packet from this zone, it can be sent to the media. A successful send will move the packet to media's packet buffer for the zone and mark the buffer as used (flag `usd`). The exact layer and address types used in a node vary depending on the specific node and scenario.

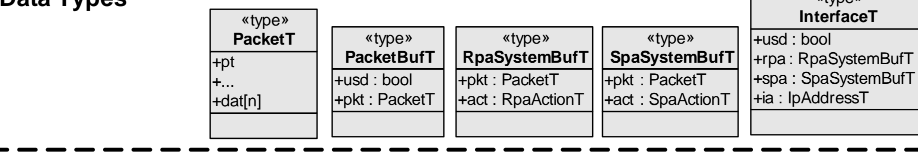
7.4 Packages & Elements

The framework is structured into the three packages Enumerations & Functions, Data Types, and Process Types (cf. Figure 7.5). In this section, we describe these packages and their elements.

Enumerations & Functions



Data Types



Process Types

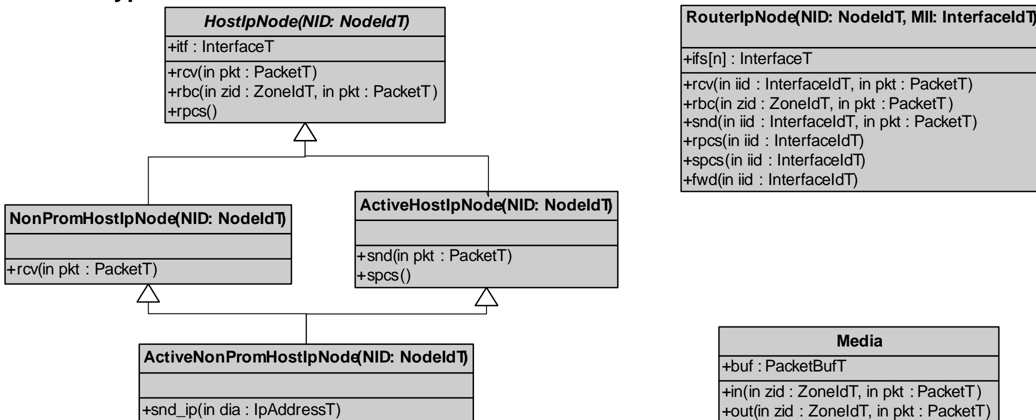


Figure 7.5: Framework overview

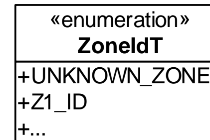
7.4.1 Package Enumerations & Functions

The package *enumerations & functions* is used to define the network topology, initial address assignment and protocols desired for a model. For example, the enumeration `ZoneIdT` contains the model's zones; the function `fSrcToIa` assigns the initial addresses and the enumeration `ProtocolT` lists the required protocols.

In the following paragraphs, we describe each element of the enumerations & functions package in more detail. We begin with the Enumerations.

7.4.1.1 Enumerations

Enumerations define symbolic names. Specific models built on the framework usually extend these enumerations depending on the scenario.

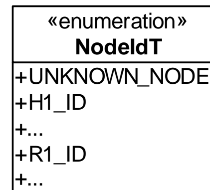


Enumeration ZoneIdT

As described in section 7.3, zones are used to group sets of nodes which can communicate directly with each other. Thus, typically, a physical subnet is equivalent to a zone.

The `ZoneIdT` enumeration provides symbolic names for zones. A special name, `UNKNOWN_ZONE`, is used to denote an invalid or unknown zone. The first regular zone is commonly named `Z1_ID`. Like all enumerations, these symbolic names are internally represented by integers (cf. section 5.2.1).

The `ZoneIdT` enumeration is used throughout the framework. Particularly, the `fSrcToZone` topology function maps to the `ZoneIdT` enumeration.

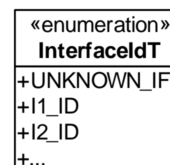


Enumeration NodeIdT

In our framework, all active network elements are nodes (cf. section 7.3).

The `NodeIdT` enumeration is used to assign symbolic names to nodes. An invalid or unknown node is assigned the reserved symbolic name `UNKNOWN_NODE`. As a naming scheme, we suggest to use `Hx_ID`, `Rx_ID` (where x is a unique integer for each node), for host, router nodes, respectively.

Together with the `InterfaceIdT` enumeration, the `NodeIdT` enumeration is used as the source in the `fSrcToHa` and `fSrcToZone` topology functions. A *source* is a pair (n, i) of a node and interface identifier.



Enumeration InterfaceIdT

Nodes transmit and receive packets with their interfaces. Host nodes have one interface; transfer nodes have multiple interfaces.

The `InterfaceIdT` enumeration provides *local* symbolic names for interfaces. These names are unique only in the context of a node identifier (cf. enumeration `NodeIdT`). Unknown or invalid interfaces are represented by the symbolic name `UNKNOWN_IF`. The first regular interface of a node is generally named `I1_ID`.

As described above, elements from the `InterfaceIdT` enumeration are typically used together with node identifiers to represent a source (n, i). The topology function `fSrcToZone` maps sources to zones. Further functions (e.g., `fSrcToHa` and `fSrcToIa`) are used to assign attributes like hardware and IP addresses to interfaces.

«enumeration» **IpAddressT**
+INVALID_IA +BC_IA +H1_I1_IA +... +R1_I1_IA +...

Enumeration IpAddressT

A key element of TCP/IP based computer networks is the logical Internet protocol (IP) address. Logical addresses may span different physical networks with different physical addressing schemes. Each interface is assigned an IP address.

The `IpAddressT` enumeration provides symbolic names for IP addresses assigned to the interfaces used in a model. Two symbolic names are reserved. First, `INVALID_IA` is used to denote an invalid IP address. Second, `BC_IA` is reserved for the broadcast IP address. It is up to a specific model to define the scope of the broadcast address in more detail. Regular addresses follow the `Hi_Ij_IA`, `Ri_Ij_IA` naming scheme for host nodes, router nodes, respectively. For example, the IP address of the second interface ($j = 2$) of the first router ($i = 1$) has the symbolic name `R1_I2_IA`.

The `IpAddressT` enumeration is used, among others, in the `fSrcToIa` interface initialization function which is part of node initialization.

«enumeration» **HardwareAddressT**
+UNKNOWN_HA +BC_HA +H1_I1_HA +... +R1_I1_HA +...

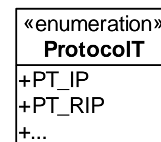
Enumeration HardwareAddressT

The framework provides support for low-level hardware addresses. Hardware addresses depend on the physical addressing scheme. Typically, each network interface has a unique hardware address.

The `HardwareAddressT` enumeration provides symbolic names for hardware addresses. The naming scheme resembles the `IpAddressT` enumeration: Two

names, `UNKNOWN_HA` and `BC_HA`, are reserved for unknown and broadcast addresses, to be defined further by the specific model. Hardware addresses for host and router nodes follow the `Hi_Ij_HA`, `Ri_Ij_HA` naming scheme.

If required by the specific model, the `HardwareAddressT` enumeration is used, among others, in the `fSrcToHa` interface initialization function.

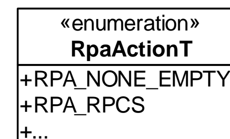


Enumeration ProtocolT

In computer networks, different protocols are employed. These protocols are distinguished on different levels by so-called *frame type identifiers*.

Enumeration `ProtocolT` provides an abstract support for such frame type identifiers. For example, the symbolic name `PT_IP` marks IP packets. For further protocols relevant for a scenario, further symbolic names are added to the `ProtocolT` enumeration (e.g., `PT_RIP` for RIP).

The `ProtocolT` enumeration is particularly used in the nodes' internal processing actions. It determines according to which protocol a packet should be processed.

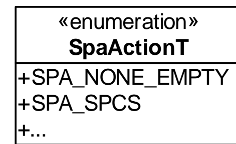


Enumeration RpaActionT

Received packets are processed by the nodes using internal actions. Regularly, the processing is done layer-by-layer. Following efficient protocol implementation techniques, however, the processing is done in one integrated, combined action from the initial processing to the final processing of the packet. Furthermore, copying of packets into new buffers is avoided as far as possible. Thus, it is necessary to keep track of the current processing and buffering state.

The enumeration `RpaActionT` provides symbolic names for this state. Two symbolic names, `RPA_NONE_EMPTY`, `RPA_RPCS`, are used to denote the state no current packet (empty receive buffer), packet ready to be processed, respectively. Depending on the scenario, further states may be added to the `RpaActionT` enumeration.

The `RpaActionT` enumeration is used in the nodes' internal receive processing and buffering. The detailed meaning of the different states depends on the specific model.



Enumeration SpaActionT

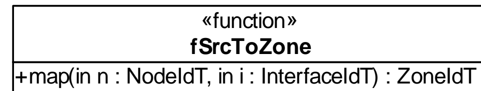
Similarly to received packets, the current processing and buffering state has to be kept, for packets which shall be sent.

The SpaActionT enumeration is used to assign symbolic names to this state. With the symbolic names SPA_NONE_EMPTY, SPA_RPCS, the states no current packet (empty send buffer), packet ready to be processed, respectively, are expressed.

As with the RpaActionT enumeration, SpaActionT is used in the nodes' internal send processing and buffering. Further details depend on the internal modeling of a scenario.

7.4.1.2 Functions

Functions provide mappings which are particularly useful for initializing attributes and defining the network topology. In cTLA, functions are given by value tables (cf. section 5.2.3).

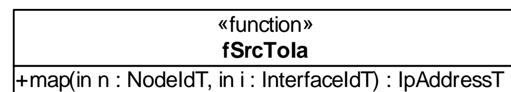


Function fSrcToZone

Interfaces are connected to the media, which is partitioned into zones (cf. section 7.3). Expressed more abstractly, interfaces have to be mapped to zones.

The function fSrcToZone maps a source to its zone, taken from enumeration ZoneIdT. Of course, the definition of the exact mapping depends on the topology of the specific scenario. A special symbol, UNKNOWN_ZONE, is used to denote an unknown (or invalid) zone. This symbol is returned for sources which do not exist in the scenario.

Function fSrcToZone is used throughout the nodes' send, receive actions, to determine the zone of media to which the packet is sent, from which the packed is received, respectively.



Function fSrcToIa

Each interface which is IP enabled has an IP address attribute. This attribute is initialized according to the fSrcToIa function.

The function fSrcToIa maps a source to its IP address, taken from enumeration IpAddressT. Again, the exact mapping depends on the specific scenario. The special symbol, UNKNOWN_ZONE, is returned for sources that do not exist.

Function `fSrcToIa` is mainly used during interface initialization which is part of node initialization.

«function» fSrcToHa
+map(in n : NodeIdT, in i : InterfaceIdT) : HardwareAddressT

Function fSrcToHa

For low-level scenarios, hardware addresses are useful. Hardware addresses are an attribute of network interfaces.

The function `fSrcToHa` maps a source to its hardware address, taken from enumeration `HardwareAddressT`. As before, the exact mapping depends on the specific scenario model. A special symbol, `UNKNOWN_HA`, is returned for unknown (or invalid) sources.

As with function `fSrcToIa`, function `fSrcToHa` is mainly used during interface initialization, part of node initialization.

7.4.2 Package Data Types

The package *data types* contains common data types for interfaces, packets and buffers used throughout the framework. For instance, the type `InterfaceT` combines attributes of an interface; `PacketT` is used to represent a packet and `PacketBufT` defines a buffer for packets.

«type» PacketT
+pt +... +dat[n]

Data Type PacketT

Communication in computer networks is based on packets. Even if higher level protocols are stream-oriented, they are ultimately broken down to packets. Thus, `PacketT` is a key data type. Of course, `PacketT` has to be modified according to the specific requirements of a scenario, particularly the supported protocols.

The data type `PacketT` represents a basic packet. The field `pt` defines the packet's type or the protocol type of the packet. Symbolic names for the `pt` field are defined through the `ProtocolT` enumeration. Essential fields like source and destination addresses can be added in two ways. Either, they are added as their own fields (e.g., `sha`, `dha` for source and destination hardware addresses), or they extend the array `dat` (e.g., `dat[DI_SIA]`, `dat[DI_DIA]`). In the IP-ARP (cf. chapter 9) and IP-OSPF (cf. chapter 11) scenarios, we add extra fields on their own; in the IP-RIP scenario (cf. chapter 10), we map them into an extended `dat` field.

As packets are the basic communication units in computer networks, the data type `PacketT` is used as a parameter in the nodes' send and receive actions.

«type» PacketBufT	«type» RpaSystemBufT	«type» SpaSystemBufT
+usd : bool	+pkt : PacketT	+pkt : PacketT
+pkt : PacketT	+act : RpaActionT	+act : SpaActionT

Data Types PacketBufT, Rpa/SpaSystemBufT

Packets are stored "in the media", by the interface after receive or before send, and during processing in the node. Thus, appropriate packet data types have to be provided.

The data types `PacketBufT`, `RpaSystemBufT`, and `SpaSystemBufT` are packet buffers. All three data types use the field `pkt`, which is of the previously defined data type `PacketT`, to store the packet. They differ in their mechanism for marking the buffer as empty or in use.

For data type `PacketBufT` the field `usd` contains `TRUE` if the buffer is in use and `FALSE` otherwise. The data types `RpaSystemBufT`, `SpaSystemBufT` utilize the field `act` which is of the enumeration `RpaActionT`, `SpaActionT`, respectively.

These packet buffer data types are applied as indicated before: `PacketBufT` stores packet in transit (e.g., for process type `Media`). The data types `RpaSystemBufT`, `SpaSystemBufT` are used by the nodes (i.e., their interfaces) in order to store received, sent packets, respectively.

«type» InterfaceT
+usd : bool
+rpa : RpaSystemBufT
+spa : SpaSystemBufT
+ia : IpAddressT

Data Type InterfaceT

Packets are sent to and received from media via network interfaces. A host node typically has one interface; router nodes usually have at least two interfaces.

The data type `InterfaceT` represents such an interface. The field `usd` contains `TRUE` if the interface is up and `FALSE` otherwise. A received packet is stored in field `rpa` of the previously defined data type `RpaSystemBufT`; a packet to be sent is stored in field `spa` of data type `SpaSystemBufT` (i.e., both directions use their own buffer). The field `ia`, of data type `IpAddressT`, contains the symbolic IP address assigned to this interface.

As described above, all node types contain at least one and possibly multiple `InterfaceT` data types.

7.4.3 Package Process Types

The package *Process Types* contains the core of the framework: process types for nodes and media. For example, process types `HostIpNode`, `RouterIpNode` implement a basic TCP/IP host or router node. Through inheritance, behavior is spe-

cialized. ActiveHostIpNode, for instance, adds behavior for the processing and sending of packets.

In the following paragraphs, we describe the elements of package process types in more detail.

Media
-buf : PacketBufT
+in(in zid : ZoneIdT, in pkt : PacketT) +out(in zid : ZoneIdT, in pkt : PacketT)

Process Type Media

Packets are transfered over a physical media. Typically, the physical media is segmented into zones according to the network's physical structure (cf. section 7.3).

The process type Media represents the transfer media which is used by the nodes to transfer packets. It is partitioned into zones according to the topology function fSrcToZone together with the supporting enumeration ZoneIdT. In each zone, up to one packet can be in transit. Such a packet is stored in the buffer (variable buf[i]) belonging to the zone or *zone buffer* in short.

Predefined actions of process type Media are:

- in Accept a packet pkt and store it inside the zone buffer belonging to zone zid. Only one packet can be stored in the zone buffer at any one time.
- out Retrieve packet pkt from the zone buffer belonging to zone zid.

System-level action coupling between send, receive actions of the nodes and in, out actions of process type Media, respectively, has to make sure that only nodes with an interface in a zone can communicate with that zone. This is easily achieved using the fSrcToZone function.

Process type Media is a key element for the network view of each model. Without the transfer media, no communication between nodes is possible.

HostIpNode(NID: NodeIdT)
+itf : InterfaceT
+rcv(in pkt : PacketT) +rbc(in zid : ZoneIdT, in pkt : PacketT) +rpcs()

Process Type HostIpNode

Host nodes are generally active network elements that receive, process, and send packets (cf. section 7.3). In contrast to router nodes, host nodes have at most one interface.

The process type HostIpNode models a *passive* TCP/IP node, i.e., a node that only receives and processes, but never sends packets. Packets with an address not matching the interface's address are accepted by the receive actions, i.e., the interface is able to work in *promiscuous* mode. The interface is represented by variable itf.

Predefined actions of process type HostIpNode are:

- `rcv` Receive a unicast packet from the zone the interface is connected to.
- `rbc` Receive a broadcast packet from the zone the interface is connected to.
- `rpcs` Basic processing of a received packet. Specific models usually have to add their own processing.

The broadcast receive action only accepts packets with a destination address of `BC_IA` (cf. enumeration `IpAddressT`); the unicast receive action never accepts such packets. Furthermore, to properly model broadcast receive, system-level broadcast receive actions have to be defined coupling the broadcast receive actions of all nodes in a zone.

Process type `HostIpNode` is the key process type for all host nodes in a model.

NonPromHostIpNode(NID: NodeIdT)
+rcv(in pkt : PacketT)

Process Type NonPromHostIpNode

Typical host nodes have their interface configured to only receive packets matching the interface's address (*non-promiscuous*). This "constraint" is added in process type `NonPromHostIpNode`.

Process type `NonPromHostIpNode` extends process type `HostIpNode`. Thus, it inherits all actions of this process type. Furthermore, it specializes the unicast receive action of `HostIpNode`:

- `rcv` Receive a packet if its destination address matches the interface's address.

Depending on the abstraction level of the scenario and the packet modeling, the matching may be either against the hardware or the IP address of the interface.

The process type `NonPromHostIpNode` is typically used to model hosts with a *receiver* role (cf. section 8.2.2).

ActiveHostIpNode(NID: NodeIdT)
+snd(in pkt : PacketT) +spcs()

Process Type ActiveHostIpNode

Most host nodes actively send packets as well. Process type `ActiveHostIpNode` adds this capability to process type `HostIpNode`.

Process type `ActiveHostIpNode` extends process type `HostIpNode` with the ability to send packets.

Added actions of process type `ActiveHostIpNode` are:

- `spcs` Process a packet for sending. Specific models usually have to add their own processing.

- snd Send a packet which has been processed by spcs.

The parameter pkt of action snd is required for appropriate system-level action coupling: Typically, the snd action of a node is coupled with the in action of the transfer media to build the system-level send action. As both node's snd and media's in have the pkt parameter, this ensures that the packet is transfered from the node to the media unchanged.

The process type ActiveHostIpNode is commonly used to derive hosts with an *attacker* role.

ActiveNonPromHostIpNode(NID: NodeIdT)
+snd_ip(in dia : IpAddressT)

Process Type ActiveNonPromHostIpNode

Both non-promiscuous receive and active send behavior are common for regular host nodes. These behaviors are combined in process type ActiveNonPromHostIpNode, which is derived from process type NonPromHostIpNode *and* process type ActiveHostIpNode.

Process type ActiveNonPromHostIpNode extends both NonPromHostIpNode and ActiveHostIpNode. Thus it inherits the actions of these two process types.

Furthermore process type ActiveNonPromHostIpNode adds the following action:

- snd_ip Create a basic IP packet for the specified IP address.

Using actions spcs and snd from process type ActiveHostIpNode, the created IP packet can be processed and finally sent.

The process type ActiveNonPromHostIpNode is typically used to model a host with a *sender* role.

RouterIpNode(NID: NodeIdT, MII: InterfaceIdT)
+ifs[n] : InterfaceT
+rcv(in iid : InterfaceIdT, in pkt : PacketT) +rbc(in zid : ZoneIdT, in pkt : PacketT) +snd(in iid : InterfaceIdT, in pkt : PacketT) +rpcs(in iid : InterfaceIdT) +spcs(in iid : InterfaceIdT) +fwd(in iid : InterfaceIdT)

Process Type RouterIpNode

Router nodes are active network elements like host nodes. However, they have at least two interfaces. These interfaces are connected to different zones. Thus, router nodes can forward packets between zones.

The process type RouterIpNode models a basic IP router node. Its interfaces are contained in the attribute ifs[n]. The actions are roughly comparable to process type ActiveHostIpNode. As there are multiple interfaces, however, parameters specifying the interface or zone have to be added:

- rcv Receive an unicast packet via the interface iid.
- rbc Receive a broadcast packet from the zone zid.
- rpcs Basic processing of a received packet. Specific models usually have to add their own processing.
- fwd Forward a received packet destined for another zone. Specific models have to properly initialize the routing table (also called forward table) or calculate it dynamically.
- spcs Process a packet for sending. Specific models usually have to add their own processing.
- snd Send a packet which has been processed by spcs or has been forwarded (fwd).

As with the host node types, system-level action coupling has to be defined appropriately for send and receive actions.

Process type RouterIpNode is the basic process type for all routers in the framework. More specialized router types, e.g., routers running RIP (cf. chapter 10), can be derived.

7.4.4 Collaboration & Extensions

In this section, we first highlight the cross-package collaboration between the framework's elements. Then, extensions of the framework for specific models are outlined.

7.4.4.1 Collaboration of Framework Elements

Above, we discussed the framework's elements by package. From a functional viewpoint, however, all packages usually collaborate to model a conception. For example, a scenario's network topology is modeled using functions (particularly fSrcToZone) together with enumerations (ZoneIdT, NodeIdT, InterfaceIdT). Additionally, the system-level coupling of the send and receive related actions (e.g., in, out, snd, rcv) of the process instances (e.g., Media, HostIpNode, RouterIpNode) has to be defined appropriately.

Another example is packets and their processing. Packets are sent to and received from Media by nodes using interfaces. Interfaces are represented through the InterfaceIdT data type which includes send and receive buffers, data type PacketBufT. In turn, the PacketBufT data type stores packets using the PacketT data type. A packet's interpretation depends on its protocol type (PacketT.pt), which contains symbolic names from the ProtocolT enumeration.

Packets currently processed at a node are stored using the `SystemBufT` data type. The processing status is determined via the `ActionT` enumeration, usually depending on the protocol type. Similarly, addressing depends on intertwined framework elements as well: functions (e.g., `fSrcToIa`), enumerations (e.g., `IpAddressT`), data types (e.g., `PacketT`) and node process types.

7.4.4.2 Extensions

The framework provides for the general architecture, behavior, and collaboration of derived models. Of course, specific models have to amend and modify the framework according to their needs:

- In the IP-ARP model (cf. section 9.2), the `IpArpNode` process type is added. This process type adds low-level ARP layer support to the basic TCP/IP support provided by the process types derived from `HostIpNode`.
- For modeling the IP-RIP scenario (cf. section 10.2), two new process types, `RipRouterIpNode` and `RipAttackerRouterIpNode`, are implemented. These process types extend the basic `RouterIpNode` process type with support for the routing information protocol. They differ in the attacker action, which is available in the `RipAttackerRouterIpNode` process type only.
- The IP-OSPF model (cf. section 11) includes three new process types. As in the IP-RIP scenario, two process types `OSPFRouter` and `AttackerOSPFRouter` add support for a new protocol, once with and once without attacker action. Furthermore, the `TimedMedia` process type constrains the `Media` process type with regard to LSA aging.

In addition to the examples from these case studies, further directions for extensions which seem particularly rewarding are briefly described in the future work (cf. section 12.2).

8 Optimization Strategies

This chapter describes the key optimizations that proved to be most helpful for the successful analysis of the case studies described in chapters 9 to 11. After a short motivation and overview of the optimization stages, we explain the optimizations at each stage.

8.1 Motivation

Model checking is a challenging task. Typically, models either are highly specialized and have a very narrow scope or they quickly exceed given time and memory constraints. Furthermore, marginal model additions can have a strong adverse impact on the performance of the verification tool and cause verification to fail. This effect is well-known as *state space explosion* and a very serious problem for practical model checking. To prevent or alleviate this problem, it is a top priority to consider the size of the state vector and – to a smaller extent – the number of transitions at different stages during modeling.

Our aim is to be able to analyze dynamic models which integrate protocol, node, and network view. Unfortunately, such models tend to have a prohibitively large state vector. Fortunately, optimizations can significantly reduce the state vector size. In our experience, it is best to follow an iterative approach to develop new optimizations. First, the current cTLA model with the current set of optimizations is translated to PROMELA. Second, SPIN is used to create the C source code for model specific verifier. Third, after the generation of the verifier with GCC, a verifier run with a fixed memory limit (`-DMEMLIM=`*n*) is performed. This way, SPIN will output statistics about the state vector size, the search depth reached, the number of transitions etc. Based on this data, we can then estimate the effects of the current optimizations vs. the previous optimizations. This helps to decide whether particular optimizations are helpful for a particular model.

We consider optimizations at all modeling stages (cf. Fig. 8.1). The stages are *Scenario*, *Model Design*, cTLA *Model*, PROMELA *Model* and SPIN/*Verifier*. They correspond to the stages of the modeling process from the initial consideration of the scenario over the model design, cTLA model and PROMELA translation to the SPIN verifier compile and run-time options. Furthermore, our approach supports optimizations at the various levels by different means. The framework's process types implement the optimizations of the Model level. Our translator, cTLA2PC, can

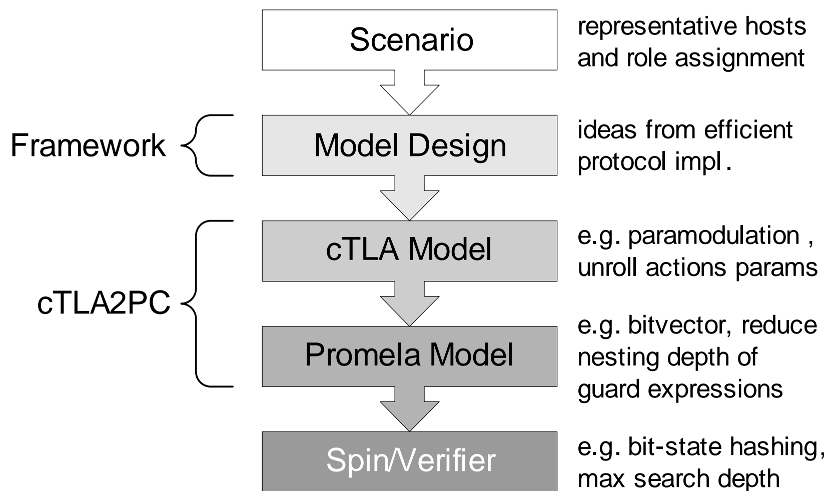


Figure 8.1: Modeling Stages & Optimizations

automatically apply several optimizations from the cTLA and PROMELA levels. In the following sections, we describe each stage in more detail.

8.2 Scenario

At the scenario stage, we devise the scenario diagram. We start from the real network and sketch its main nodes, networks, and connections. Furthermore, we annotate additional key information required for modeling the scenario, like designated routers etc (cf. chapter 11). The scenario diagram is crucial for the following development of a scenario model. Regarding optimizations, early simplifications often have the greatest effect on the complexity of the model developed in the later stages.

8.2.1 Representative Nodes

We minimize the number of nodes which have to be modeled later on by using *representative nodes*. If there is a set of similar nodes in the same network zone, it is often sufficient to include only one of them in the modeling. This node is then called a representative node. We can typically substantiate the use of a representative node by the fact that the nodes in the set are interchangeable with one another. If one of the nodes acts in a specific way, any of the other nodes could act in the same way, too. Thus, we can assume that the acting node is the representative node. The representative node concept allows us to develop a smaller model which is still relevant for the originally intended scenario. Of course, this selection of representative nodes depends on the assigned roles and the circumstances of the scenario as well. In the IP-ARP scenario (cf. chapter 9), for instance, we select three

representative nodes in the same network zone instead of only one. This is due to the fact that each of these nodes has a different role in the scenario.

8.2.2 Role Assignment

Complementary with the representative nodes concept, we can minimize the number of actions which have to be included in the modeling. Actions are combined according to roles. Typical roles are e.g., attacker, sender, and receiver. These roles include basic actions plus specific actions required for the role. By assigning roles to the nodes we know which actions we have to include. This is implemented by instantiating the node from a role-specific process type. For example, in the IP-RIP scenario (cf. chapter 10), the attacking node is instantiated from the role-specific process type `AttackerHostIpNode`. This process type is derived from the basic process type `HostIpNode`; but it adds the attacker actions. The assignment of roles via instantiation from a role-specific process type makes it easy to change roles between nodes. Thus, models which are identical except for different role assignment can be modeled with ease.

Especially when utilized together, the representative nodes and the role assignment concepts help us to cut our scenario down to an essential number of nodes and actions. As the scenario stage is at the beginning of the modeling process, optimizations at the scenario stage have a positive effect on all the other stages as well.

8.3 Model Design

The aim of this stage is to develop a design for a model implementing the scenario. Thus, we take a more detailed look at the elements of the scenario diagram. We particularly consider the involved protocols. The design is still independent of a specific modeling language like cTLA.

8.3.1 Protocol Simplification

The key idea is to either abstract the protocol itself (e.g., by substituting complicated message types with simpler ones) or to restrict the scope of the model (e.g., by stating the conditions). Regarding the first case, RIP (cf. section 10.2.1), for example, allows multiple updates to be contained in a single update packet. This is due to efficiency reasons, because the packet header is required only once. The effect of multiple updates in one packet, however, is typically equivalent to multiple update packets with a single update each. In these cases, without loss of generality, we can assume each update packet contain exactly one update.

Furthermore, we can restrict the scope of our model. Some protocols work in different phases and we may restrict our model to one of these phases. For example, the OSPF protocol (cf. chapter 11) distinguishes between a neighbor acquisition or initialization phase and a following so-called steady phase. By restricting the scope of our model to the steady phase, we do not have to model the message types and processing associated with the initialization phase. Alternatively, for each phase an independent model can be designed.

Beyond this, by making certain assumptions on the environment, we can optimize our protocol modeling as well. OSPF, for example, includes mechanisms to ensure reliable transport. If we assume a reliable media, we do not have to include message types and processing associated with acknowledgment or resubmission of packets.

8.3.2 Efficient Protocol Implementation Techniques

In the course of the upcoming high speed communication protocols, several techniques were invented to implement protocols more efficiently. Particularly, we make use of the activity thread and the integrated layers approaches for the protocol modeling of the framework's node and router types.

Activity Thread Approach The basic idea of the *activity thread approach* [Svo89] is to concatenate all actions which handle the same packet into one uninterruptable execution thread. This encompasses the actions from the initial processing of the packet to the final processing and removal of the packet. Figure 8.2 depicts conventional layered packet processing on the left and layered packet processing after application of the activity thread approach on the right.

Initial processing starts due to a stimulating event (e.g., packet receive at the physical layer). In case of data dependencies, the activity thread approach may have to be implemented in a less strict way allowing for a small number of interruptions. The activity thread approach is most suitable for layered protocol implementations. Its main advantage in comparison to approaches that process packets layer by layer is that partially processed packets do not have to be stored after each layer. Thus, the activity thread approach saves on buffers required for packet processing.

Originally, we implemented the basic node and router types of the framework with layer-by-layer processing using actions of the form `spcs_ln`, `rpcs_ln` for send, receive processing of layer n, respectively. After processing at a layer, the packet was stored in the send or receive buffer of that layer. Later on, we reimplemented the basic node and router types according to the activity thread approach. Packets are processed through all layers with only one action `spcs` (send events) or `rpcs` (receive events). This saves considerably on the size of the state vector because only two (send and receive) instead of n buffers are required for packet

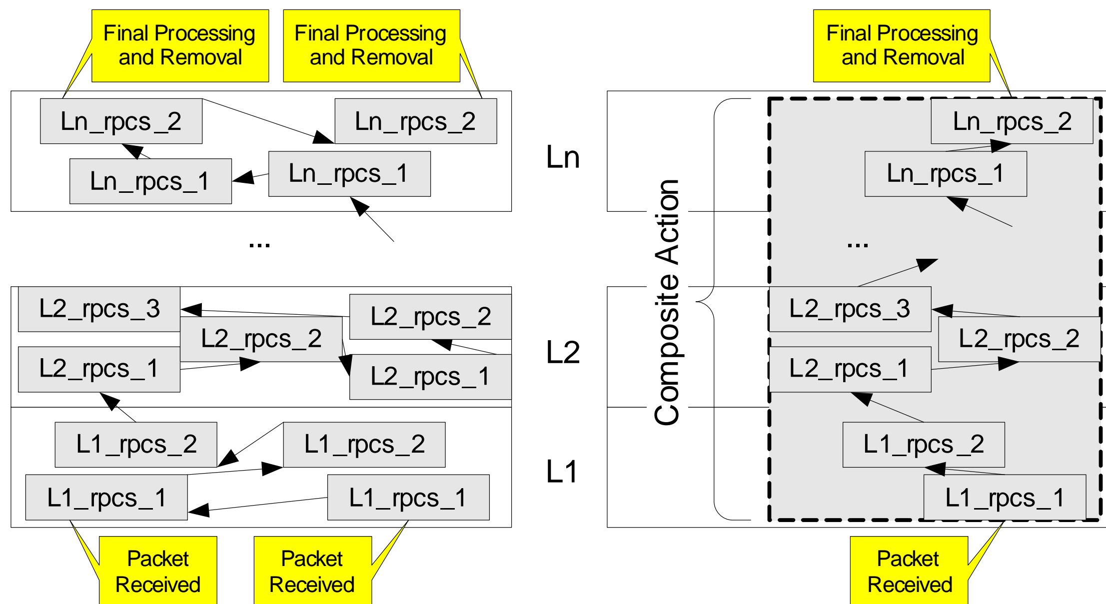


Figure 8.2: Layered Packet Processing and the Activity Thread Approach

processing. In the IP-ARP scenario (cf. chapter 9), which was at first modeled without the activity thread approach, the state vector size is nearly halved (from 480 bytes to 250 bytes) by the application of the activity thread approach.

Integrated Layers Approach The key idea of the *integrated layers approach* [AP93] is to combine several sequential operations (e.g., `read-encrypt-write`, `read-calc_checksum-write`) on a packet into one operation. Particularly, if n operations of the form `read-manipulate_i-write` are required, they are combined into one operation of the form `read-manipulate_1-`...`-manipulate_n-write`. If intermediate results are required by the protocol layers, however, not all operations can be combined.

The main advantage of the integrated layers approach is that multiple read-write cycles are avoided. In many cases, the read-write cycles take more time than the manipulation operations. Thus, the integrated layers approach can improve the performance of a protocol stack by up to 50%.

Performance of the modeled protocol stack is less of a concern for us than model size. In simulation mode, however, during model validation, performance makes a difference. Furthermore, the integrated layers approach goes along particularly well with the activity thread approach. Thus, we followed the integrated layers approach to avoid extra read-write cycles of packet data in framework.

8.4 cTLA Model

The aim of the cTLA stage is to implement the model design in cTLA. Furthermore, we consider transforming selected cTLA constructs to better optimized cTLA constructs. Action parameters are a particularly promising area for optimizations. In the following section, we give the key optimizations that proved useful. Using cTLA2PC (cf. chapter 6), they can be applied automatically.

8.4.1 Paramodulation

cTLA action parameters correspond to existentially quantified value variables (cf. chapter 5). Due to the coupling of actions in system actions, however, value determining equalities often exist. A typical example are send and receive system actions. An input parameter of one coupled action is the output parameter of another coupled action. The input parameter can then be removed from the action head and its occurrences inside the action can be replaced by the output parameter.

We call this action parameter replacement due to equalities *paramodulation*. Paramodulation techniques were orginally introduced in theorem proving for first order logics with equality [RW69].

Basic Paramodulation In the basic case, the equality only includes the parameter on the left side. In that case, the parameter can be fully replaced. We consider an example from the IP-ARP scenario (cf. chapter 9). Listing 8.1 contains the `snd_h1` cTLA system action. The `pkt` parameter refers to the packet to be transfered from the node to the media.

```
snd_h1( pkt: PacketT ) ::= h1.snd( pkt ) AND med.in( pkt );
```

Listing 8.1: snd_h1 cTLA System Action

In the flat system (cf. chapter 6), the process types and their coupling are resolved. Thus, equalities are clearly visible. For example, the action `snd_h1` contains an equality between the parameter `pkt` and the send buffer `h1_ifs[0].spa.pkt` (cf. Listing 8.2).

The parameter `pkt` is only read to write to the media buffer `med_buf[0].pkt`. Furthermore, it does not occur in expressions on the left side. Thus, the parameter `pkt` can be removed from the action head of `snd_h1` and replaced inside the action with `h1_ifs[1-1].spa.pkt`. The resulting action `snd_h1` after paramodulation is depicted in Listing 8.3.

```
snd_h1(pkt: PacketT) ::=
  med_buf[1 - 1].usd = FALSE
  AND h1_ifs[1 - 1].usd = TRUE
  AND h1_ifs[1 - 1].spa.usd = TRUE
  AND h1_ifs[1 - 1].spa.psd = TRUE

  AND pkt = h1_ifs[1 - 1].spa.pkt // <-- Equality

  AND med_buf[1 - 1].usd' = TRUE
  AND med_buf[1 - 1].pkt' = pkt          // <-- Read
  AND h1_ifs[1 - 1].spa.usd' = FALSE;
```

Listing 8.2: Action snd_h1 in the Flat System

```
snd_h1()::= // <-- Parameter removed
  med_buf[1 - 1].usd = FALSE
  AND h1_ifs[1 - 1].usd = TRUE
  AND h1_ifs[1 - 1].spa.usd = TRUE
  AND h1_ifs[1 - 1].spa.psd = TRUE
  AND med_buf[1 - 1].usd' = TRUE
  AND med_buf[1 - 1].pkt' = h1_ifs[1 - 1].spa.pkt // <-- Replaced
  AND h1_ifs[1 - 1].spa.usd' = FALSE;
```

Listing 8.3: Action snd_h1 After Paramodulation

Advanced Paramodulation In more complicated cases, the equality involving the parameter contains parts of the parameter on the right side. Partial paramodulation may still be possible after parameter splitting.

Let action `snd_h1b` be like action `snd_h1` (cf. Listing 8.2), except that the equality is `pkt = h1_ifs[pkt.sci - 1].spa.pkt` instead of `pkt = h1_ifs[0].spa.pkt`. We cannot replace the parameter `pkt` like in the basic case because the replacement still contains `pkt`. The parameter `pkt`, however, has the type `PacketT`, which is a record comprising several fields (`scn, sci, sha, ...`). Accordingly, we can split the equality `pkt = h1_ifs[pkt.sci - 1].spa.pkt` into its parts (cf. Listing 8.4).

This way, we obtain new equalities for all fields of parameter `pkt`. Except for field `sci`, these new equalities are not self-dependent. Similarly, parameter `pkt` is splitted into `pkt_scn`, `pkt_sci`, `pkt_sha`, ... in the head of action `snd_h1`. Thus, we can now apply the basic paramodulation for all new parameters except `pkt_sci`. Only one parameter, `pkt_sci`, remains in `snd_h1` (cf. Listing 8.5).

Both basic paramodulation and advanced paramodulation are implemented in cTLA2PC (cf. chapter 6.3). Using cTLA2PC's option `--optparamod` all system

```
pkt.scn = h1_ifs[pkt.sci - 1].spa.pkt.scn // <-- Equality (1)
pkt.sci = h1_ifs[pkt.sci - 1].spa.pkt.sci // <-- Dependency
pkt.sha = h1_ifs[pkt.sci - 1].spa.pkt.sha // <-- Equality (2)
...
```

Listing 8.4: Equalities After Splitting of Parameter pkt

```
snd_h1(pkt_sci) ::=
  med_buf[1 - 1].usd = FALSE
  AND h1_ifs[1 - 1].usd = TRUE
  AND h1_ifs[1 - 1].spa.usd = TRUE
  AND h1_ifs[1 - 1].spa.psd = TRUE
  AND pkt_sci = h1_ifs[pkt_sci - 1].spa.pkt.sci // <-- Remainder
  AND med_buf[1 - 1].usd' = TRUE

  AND med_buf[1 - 1].pkt.scn = h1_ifs[pkt_sci - 1].spa.pkt.scn
  AND med_buf[1 - 1].pkt.sci = pkt_sci
  AND med_buf[1 - 1].pkt.sha = h1_ifs[pkt_sci - 1].spa.pkt.scn
  ...

  AND h1_ifs[1 - 1].spa.usd' = FALSE;
```

Listing 8.5: Action snd_h1b After Paramodulation

actions are examined and both basic and advanced paramodulation are applied if possible. In the IP-ARP scenario (cf. chapter 9), the paramodulation optimization cuts down the state vector size by about 16% (from 250 Bytes to 210 Bytes).

8.4.2 Unroll Action Parameters

The unroll action parameters technique is another optimization related to parameterized actions. It is especially useful after the paramodulation optimization has been applied. The basic idea is to create fixed value copies of parameterized actions. This has to be done for all possible parameter values and combinations. Then, the parameterized action can be removed and only the fixed value copies remain.

Consider for example action `snd_r1` (cf. Listing 8.6), which is taken from the IP-RIP scenario (cf. chapter 10).

After paramodulation, one parameter, `iid` of type `InterfaceIdT`, remains. In the IP-RIP scenario, because the nodes have at most three interfaces, this type ranges from $0 \ldots 3$. Thus, the parameter `iid` can be unrolled by creating four copies of action `snd_r1` and replacing `iid` with a different fixed value in each (cf. Listing 8.7). Of course, the resulting cTLA action code is much larger after unroll. It

```
snd_r1(iid: InterfaceIdT) ::=  // <-- Parameter iid
  fSrcToZone(R1_ID, iid) != UNKNOWN_ZONE
  AND med_buf[fSrcToZone(R1_ID, iid) - 1].usd = FALSE
  AND pValidIf(iid, 3)
  AND r1_ifs[iid - 1].spa.act = SPA_SND
  AND med_buf[fSrcToZone(R1_ID, iid) - 1].usd' = TRUE
  AND med_buf[fSrcToZone(R1_ID, iid) - 1].pkt' =
    r1_ifs[iid - 1].spa.pkt
  AND r1_ifs[iid - 1].spa.act' = SPA_NONE_EMPTY;
```

Listing 8.6: Action snd_r1

can be handled more efficiently by SPIN after translation to PROMELA, however.

```
snd_r1_0() ::= // <-- Replaced iid with first value, 0
  fSrcToZone(R1_ID, 0) != UNKNOWN_ZONE
  ...
  AND r1_ifs[0 - 1].spa.act' = SPA_NONE_EMPTY;

snd_r1_1() ::=
  fSrcToZone(R1_ID, 1) != UNKNOWN_ZONE
  ...
  AND r1_ifs[1 - 1].spa.act' = SPA_NONE_EMPTY;

snd_r1_2() ::=
  fSrcToZone(R1_ID, 2) != UNKNOWN_ZONE
  ...
  AND r1_ifs[2 - 1].spa.act' = SPA_NONE_EMPTY;

snd_r1_3() ::= // <-- Replaced iid with last value, 3
  fSrcToZone(R1_ID, 3) != UNKNOWN_ZONE
  ...
  AND r1_ifs[3 - 1].spa.act' = SPA_NONE_EMPTY;
```

Listing 8.7: Action snd_r1 After Unroll (Excerpt)

The unroll action parameters optimization is implemented in CTLA2PC. With CTLA2PC's option `--unrollinputgen`, all actions are unrolled. To obtain the best results, this option should be applied together with `--optparamod`.

The unroll action parameters optimization does not reduce the state vector size much. In the IP-RIP scenario (cf. chapter 10), for instance, the reduction is about 5%. Far more significant is a decrease in the search depth required for a specific sequence after translation to PROMELA. This is due to input generator steps (cf.

chapter 6) which are no longer needed after the application of the unroll action parameters optimization.

8.5 Promela Model

Because SPIN's input language is PROMELA, the CTLA model has to be translated to a PROMELA model. The translation is automated using the CTLA2PC tool. Furthermore, we optimize the PROMELA model by taking peculiarities of SPIN into account. The PhD thesis by Ruys [Ruy01] provides a comprehensive summary of low-level PROMELA or SPIN optimization possibilities. Some optimization ideas implemented in CTLA2PC are based on these ideas. Particularly, our bit array mapping is a generalized version of Ruys's bitvector approach.

8.5.1 Bit Array Mapping

A *bit array* is a data structure which stores a set of n variables $\{v_0, \ldots, v_{n-1}\}$. Each v_i occupies a number of bits which can typically not be divided by eight. Thus, using a number of bytes (=8 bit) for every v_i wastes space and increases the state vector. Unfortunately, SPIN handles bit arrays by mapping each v_i to a number of bytes. For example, if each v_i originally requires one bit, it will require one byte with SPIN's bit array mapping, wasting seven bits per byte. Hence, the state vector may increase up to eightfold in comparison to a more intelligent mapping.

Ruys [Ruy01] first observed a related weakness of SPIN regarding bit arrays where each v_i is a *boolean* value (i.e., occupies exactly one bit). Following his *bitvector approach*, accesses to each boolean v_i have to be rewritten using appropriate macros (e.g., `SET_0, SET_1, IS_0, IS_1`). These macros map up to eight booleans in a single byte, ideally wasting no bits at all.

We generalized Ruys' approach by allowing for *arbitrary* element sizes instead of only single bit (i.e., boolean) elements. Basically, we provide macros (cf. Listing 8.8) which map the element at index `idx`, where each element has a bit size of `esz`, into the integer or byte array type `bv`. Furthermore, due to the integration in CTLA2PC, no changes to the CTLA source code are necessary. Particularly, the model designer does not have to work with special macros, because the rewrite of all accesses is done automatically by CTLA2PC. With the switch `--optbitarrays` CTLA2PC appropriately maps all read and write accesses to bit array elements.

Consider for example the packet type `PacketT` (cf. Listing 8.9) from the IP-ARP scenario (cf. chapter 9). In addition to the standard fields, it contains an array `dat` of type `DataT` which contains elements specific to ARP packets. From the `rangedef` file supplied with the CTLA model, each `DataT` element can be restricted to values between 0 and 4. Thus, `dat` can be represented as a bit array with an element size of 3 ($2^3 = 8$).

```
#define BVSET(bv, esz, idx, val)\
  bv = ((bv & (~(((1<<esz)-1) << (idx*esz)))) | (val<<(idx*esz)))
#define BVGET(bv, esz, idx)\
  ((bv>>(idx*esz)) & ((1<<esz)-1))
```

Listing 8.8: Bit Array Mapping BVSET and BVGET Macros

```
PacketT = RECORD
  scn: NodeIdT;
  sci: InterfaceIdT;
  sha: HwAddressT;
  ...
  dat: ARRAY [5] OF DataT;
END;
```

Listing 8.9: PacketT with dat Array in the IP ARP Scenario

At the cTLA level, the action `snd_h1` (cf. Listing 8.5) contains the assignment `med_buf[1 - 1].pkt' = h1_ifs[1 - 1].spa.pkt`. Particularly, the `dat` field of `pkt` has to be read and written. Without the generalized bit vector optimization, this statement is realized in PROMELA by simple assignments for the array elements (cf. Listing 8.10).

```
med_buf[1 - 1].pkt.dat[0] = bnA_ifs[1 - 1].spa.pkt.dat[0];
med_buf[1 - 1].pkt.dat[1] = bnA_ifs[1 - 1].spa.pkt.dat[1];
med_buf[1 - 1].pkt.dat[2] = bnA_ifs[1 - 1].spa.pkt.dat[2];
med_buf[1 - 1].pkt.dat[3] = bnA_ifs[1 - 1].spa.pkt.dat[3];
med_buf[1 - 1].pkt.dat[4] = bnA_ifs[1 - 1].spa.pkt.dat[4];
```

Listing 8.10: Assignment Implementation in Promela Without Bit Array Mapping

Using the bit vector optimization, however, each read access is replaced by the `BVGET` macro and each write access is replaced with the `BVSET` macro (cf. Listing 8.11). Furthermore, the index position of the element to access and its bit size have to be specified in the macro. Fortunately, cTLA2PC inserts the macros with the required parameters automatically.

In both the IP-ARP (cf. chapter 9) and IP-RIP scenario (cf. chapter 10), the bit array mapping optimization helps to cut down the state vector by about 20%.

```
BVSET(med_buf[1 - 1].pkt.dat, 3, 0,
  BVGET(bnA_ifs[1 - 1].spa.pkt.dat, 3, 0));
BVSET(med_buf[1 - 1].pkt.dat, 3, 1,
  BVGET(bnA_ifs[1 - 1].spa.pkt.dat, 3, 1));
BVSET(med_buf[1 - 1].pkt.dat, 3, 2,
  BVGET(bnA_ifs[1 - 1].spa.pkt.dat, 3, 2));
BVSET(med_buf[1 - 1].pkt.dat, 3, 3,
  BVGET(bnA_ifs[1 - 1].spa.pkt.dat, 3, 3));
BVSET(med_buf[1 - 1].pkt.dat, 3, 4,
  BVGET(bnA_ifs[1 - 1].spa.pkt.dat, 3, 4));
```

Listing 8.11: Assignment Implementation in Promela With Bit Array Mapping

8.5.2 Reduce Nesting Depth of Guard Expressions

As described in section 6.2.2.4, cTLA functions are translated to PROMELA via preprocessor macros. During preprocessing, these macros are expanded wherever the function is called. Of course, if multiple function calls are nested, this can lead to very large, nested expressions in the PROMELA model after preprocessing.

Unfortunately, this may lead to the well-known C compiler GCC failing to generate the executable verifier (cf. section 3.1.1) for a model. We experienced this effect in the IP-OSPF scenario (cf. section 11.3.2.4).

Using the reduce nesting depth of guard expressions optimization produces smaller guard expressions. At the expense of additional helper variables, the nesting level of function calls is reduced. Listing 8.12 shows an excerpt of the cTLA guard expression for action `fwd` of the `OSPFRouter` process type. It combines the predicate `pValidIf` with the nested functions `fIaToIna` ∘ `fInaToRtI`.

```
...
AND pValidIf( rt[
  fInaToRtI( fIaToIna( ifs[iid].rpa.pkt.ida ) )
  ].iid, MII )
...
```

Listing 8.12: Excerpt of Action fwd's cTLA Guard Expression

Because of the inline expansion of functions, this causes the PROMELA level guard to be much larger and complicated (cf. Listing 8.13).

Such guard expressions often occur in more than one action and in more than one instance of a process type. In the case of process type `OSPFRouter`, GCC fails. We analyzed the situation and invented the reduce nesting depth of guard expressions optimization. After applying the optimization, the guard expression makes use

```
...
&& r1_rt[
  ((r1_ifs[1].rpa.pkt.ida == 1 -> 12:
  (r1_ifs[1].rpa.pkt.ida == 2 -> 15:
  ...
  (r1_ifs[1].rpa.pkt.ida == 9 -> 14:
  (r1_ifs[1].rpa.pkt.ida == 10 -> 14:
  (r1_ifs[1].rpa.pkt.ida == 11 -> 15:0))))))))))) == 12 -> 0:
  ((r1_ifs[1].rpa.pkt.ida == 1 -> 12:
  (r1_ifs[1].rpa.pkt.ida == 2 -> 15:
  ...
  (r1_ifs[1].rpa.pkt.ida == 10 -> 14:
  (r1_ifs[1].rpa.pkt.ida == 11 -> 15:0))))))))))) == 13 -> 1:
  ((r1_ifs[1].rpa.pkt.ida == 1 -> 12:
  (r1_ifs[1].rpa.pkt.ida == 2 -> 15:
  ...
  (r1_ifs[1].rpa.pkt.ida == 10 -> 14:
  (r1_ifs[1].rpa.pkt.ida == 11 -> 15:0))))))))))) == 14 -> 2:
  ((r1_ifs[1].rpa.pkt.ida == 1 -> 12:
  (r1_ifs[1].rpa.pkt.ida == 2 -> 15:
  ...
  (r1_ifs[1].rpa.pkt.ida == 9 -> 14:
  (r1_ifs[1].rpa.pkt.ida == 10 -> 14:
  (r1_ifs[1].rpa.pkt.ida == 11 -> 15:0))))))))))) == 15 -> 3:
  5))))
].iid < 2
...
```

Listing 8.13: Part of Action fwd's Promela Guard Expression after Macro Expansion

of two temporary variables for the two previously nested functions calls (cf. Listing 8.14). This way, the main guard expression is reduced to the short statement `(r1_rt[FVAL_2].iid < 2)`.

This transformation can be done automatically. We implemented a prototypical PERL script called `t2p`, which is a front-end for our usual translator CTLA2PC. It calls CTLA2PC to produce a PROMELA version of the CTLA model and then transforms the nested functions and predicates.

8.5.3 Further Promela Level Optimizations

Several properties of CTLA allow for further optimizations on the PROMELA level. For example, the execution of CTLA actions can be arbitrarily interleaved; however,

```
        ...
  FVAL_1 = (r1_ifs[1].rpa.pkt.ida == 1 -> 12:
    (r1_ifs[1].rpa.pkt.ida == 2 -> 15:
    (r1_ifs[1].rpa.pkt.ida == 3 -> 14:
    (r1_ifs[1].rpa.pkt.ida == 4 -> 12:
    (r1_ifs[1].rpa.pkt.ida == 5 -> 13:
    (r1_ifs[1].rpa.pkt.ida == 6 -> 13:
    (r1_ifs[1].rpa.pkt.ida == 7 -> 14:
    (r1_ifs[1].rpa.pkt.ida == 8 -> 13:
    (r1_ifs[1].rpa.pkt.ida == 9 -> 14:
    (r1_ifs[1].rpa.pkt.ida == 10 -> 14:
    (r1_ifs[1].rpa.pkt.ida == 11 -> 15:0)))))))))));
  FVAL_2 = (FVAL_1 == 12 -> 0:
    (FVAL_1 == 13 -> 1:
    (FVAL_1 == 14 -> 2:
    (FVAL_1 == 15 -> 3:5))));
 if
 :: (r1_rt[FVAL_2].iid < 2)
   ...
```

Listing 8.14: Part of Action fwd's Promela Guard Expression after Macro Expansion, with Reduce Function Nesting Optimization

each action is atomic and deterministic. SPIN supports the marking of atomic and deterministic code blocks with the PROMELA keyword `d_step`. This helps SPIN to build a more efficient internal model representation for verification.

The PROMELA system generated by CTLA2PC contains only a single instance of a process type (the system process type). Thus, a single PROMELA `active proctype` statement suffices to instantiate the process type and run it. Usually, processes are started from the PROMELA `init` process. In this case, $1 + n$, with n beeing the number of processes in the system, are required.

Temporary variables (e.g., counter variables for loops or partial function results) which are generated inside actions by CTLA2PC are flagged with the PROMELA `hidden` statement. These variables have no relevance outside their local scope and the actions are atomic. Thus, these variables do not have to be included in the state vector by SPIN. Marking them with the `hidden` statement achieves exactly that.

8.6 Verifier Compilation & Run-Time Options

Finally, SPIN supports different options during verifier generation and run-time. Section 3.1.4 contains an overview of these options. The main choice is between

exhaustive and approximative verification modes and between breadth first search and depth first search.

Typically, we use exhaustive verification mode (cf. case studies chapter 9 and 10). In our experience, using graph encoding for the state vector (`-DMA=`n) is the best setting. As we are only interested in safety properties, we give the `-DSAFETY` option. Furthermore, we prefer breadth first search (`-DBFS`) because we are interested in minimal attack sequences. In certain cases, to estimate the effects of an optimization, it makes sense to add the `-DMEMLIM=`n option. This causes the verifier to stop after n MB of memory are exhausted. As described above, SPIN after stopping, gives useful statistics on the number of states explored etc. Verifier run-time options seldomly are advantageous. As with the `-DMEMLIM` switch, however, in some cases `-m`n is useful. This option stops the verification after depth n is reached.

Furthermore, SPIN is under active development. New reductions and verification options are added and bugs are fixed on a frequent basis. While the first models for this PhD thesis were analyzed using SPIN 4.0.7, at the time the final models were analyzed SPIN 4.2.5 was ready. Thus, the exact version of SPIN employed may have an influence on state-vector size as well.

In our experience, however, SPIN versions and options offer only modest improvements relative to each other and the default settings. Thus, most of the time the feasibility of automated analysis does not depend on the right choice of verifier and environment level options, but much more on careful optimizations at the other levels.

9 Case Study: IP-ARP

In this chapter, we apply our integrated formal modeling and automated analysis approach (cf. chapter 4) to the IP-ARP scenario. First, the general setting of the scenario is introduced, followed by the steps taken to build an appropriate scenario model covering protocol, network and node view. Afterwards, the security property, optimizations, and results of the automated analysis are described. Last but not least, we give an outline of the experience gained from applying our approach to the IP-ARP scenario.

9.1 Introduction

In the IP-ARP scenario [RPK04], we consider a switched LAN connecting hosts running a basic TCP/IP stack including the low-level ARP layer. Three of the hosts, H1, H2, HA, are chosen as representatives (cf. Fig. 9.1).

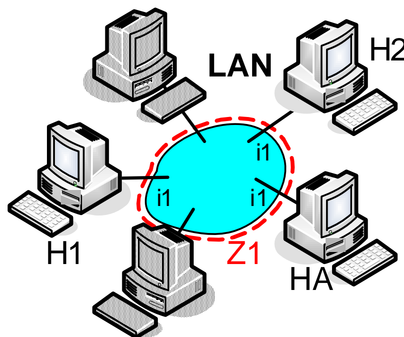


Figure 9.1: IP-ARP Scenario

Classification ARP is used for routing *inside* a single, physical network. In contrast to higher level routing protocols like RIP and OSPF (cf. chapter 10, 11), ARP is much simpler. It is a query and reply protocol to map IP to hardware addresses. This is necessary because the physical networking hardware can only work with its own addresses. There are no distinguished router nodes, each node answers queries for its own IP address.

Attack Ideas & Tools In principle, the generic routing attack ideas described in section 7.2.3 apply. Due to the simplicity of ARP, there is e.g., no calculation of a routing table and no distribution of injected packets via triggered updates. The effects of ARP attacks, however, closely match those in section 7.2.3.

Tools like `nemesis-arp` from the NEMESIS project [NS04] or `arpspoof` from the DSNIFF suite [Son00] provide simple injection of packets with arbitrary source and destination IP address and IP to hardware mapping. More advanced tools like `ettercap` [OV06] support the complete set-up of man-in-the-middle attacks.

To examine the scenario, we first have to compose an appropriate model in cTLA. The following section describes the modeling steps.

9.2 Modeling

The scenario is modeled based on an early version of the current framework (cf. chapter 7). We take different views to describe its modeling. The protocol-oriented view describes the involved protocols and their processing. With the node view we take a look at the local actions of the nodes, including management actions, and the attacker actions. Finally, we describe the network modeling and the composition of the integrated system.

9.2.1 Protocol View

As described in section 7.2.1, our protocol modeling follows the *TCP/IP reference model*. For the IP-ARP scenario, the layers network interface and internet layer are sufficient (cf. Fig. 9.2).

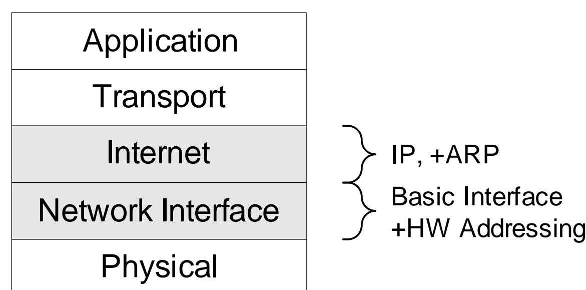


Figure 9.2: Layers and Protocols in the IP-ARP Scenario

Nodes can communicate directly on the internet layer with IP packets. This simplifies our modeling, since we do not have to include a transport layer stub allowing to send UDP or TCP packets. Furthermore, if we are able to mount a man-in-the-middle attack on internet layer packets, we are able to do so for transport layer packets (due to the encapsulation of upper layer in lower layer packets).

The process type HostIpNode from the framework (cf. chapter 7) supplies a node type which already supports both a simplified transport layer and internet layer. Thanks to CTLA's support for compositional modeling, further layers and layer extensions can be easily added. For adding hardware addressing to the network interface layer and extending the internet layer with ARP functionality, we derive a new process type, IpArpNode, from HostIpNode.

9.2.1.1 Internet Layer – Address Resolution Protocol

The process type IpArpNode adds the *address resolution protocol (ARP)* [Plu82] to the internet layer. ARP is used to provided unified routing based on IP addresses inside a physical network (cf. section 7.2.2), no matter what the underlying networking technology is (e.g., Ethernet or Tokenring). Because the network interface layer can only handle hardware addresses, ARP maps IP addresses from the internet layer to hardware addresses. Basically, ARP works as follows: Queries are broadcast to all hosts in the LAN asking for the hardware address belonging to the contained IP address. The host that has been assigned the questioned IP address is supposed to respond with its hardware address. For performance reasons, each host manages an ARP cache. This cache contains previously resolved IP to hardware mappings.

In the IpArpNode, the ARP protocol is implemented as follows: IP packets received from an upper layer are stored in a buffer. The processing action (spcs) on the internet layer checks the IP destination address of the packet. If the hardware address for this IP address is available in the cache, the packet will be submitted to the network interface layer straight away (action spcs_c). Otherwise, an ARP query for the IP address is created (action spcs_nc). After being processed by the network interface layer, the query is sent using a hardware broadcast (action snd). Further processing of the original IP packet is delayed. If a matching ARP reply (or a query by the sought-after node) is received, it is processed and the ARP cache is updated (action rpcs). Processing of the IP packet can then be resumed and the packet can finally be sent (actions spcs, snd).

Additionally, the packet data type (PacketT) is enhanced to be able to represent both IP and ARP packets. ARP packets include a type attribute and two pairs of hardware and protocol addresses. In our case, protocol address is synonymous to IP address. The type can specify either an ARP query (AT_QUERY) or an ARP reply (AT_REPLY). Regardless of the type, the first hardware and protocol address pair always belongs to the sender of the ARP packet and the second one to the receiver.

9.2.1.2 Network Interface Layer – Hardware Addressing

The network interface layer is extended with hardware addressing. Both the interface (data type InterfaceT) and the snd and rcv actions of HostIpNode have to be extended with hardware addresses and associated processing steps.

Received packets (action `rcv`) are stored in the interface's buffer and processed layer by layer as implied by the TCP/IP reference model. The network interface layer processing generally only accepts packets with a hardware destination address matching the interface's hardware address. Hardware broadcasts and promiscuous mode of the interface are the exceptions. Invalid packets are thrown away immediately and do not reach the internet layer. Similarly, the send packet processing (action `spcs`) has been extended to generally set the hardware source address of the packet to the hardware address of the interface.

Furthermore, the packet data type (`PacketT`) from the framework has to be augmented for the network interface layer. Besides hardware source and destination addresses, a frame type attribute is required. The frame type helps to decide which network level protocol is encapsulated in the frames. For example, if the frame type matches constant `L2_IP`, a packet has IP attributes on the internet layer. IP packet attributes include IP source and destination address. Another constant (`L2_ARP`) is used to denote ARP packets.

9.2.2 Node View

All active network elements (hosts, routers etc) in our framework (cf. chapter 7) are called nodes. In the IP-ARP scenario, all hosts are derived from the `IpArpNode` process type. Besides the support for processing the required protocols (see above), we have to consider further aspects related to the nodes.

9.2.2.1 Representative Nodes & Role Assignment

The hosts, NA, NB, NC, are chosen as representatives for the set of hosts in the LAN (cf. Fig. 9.1). We assign them roles like receiver, sender (which includes receiver), and attacker (which includes sender). This way, we only need to consider a subset of hosts (cf. section 8.2).

For this scenario, we map roles to hosts in the following way: HA $\mapsto$ sender, HB $\mapsto$ sender, and HA $\mapsto$ attacker. Accordingly, all of the hosts can initiate communications and receive packets, but only HA is equipped with a special "attacker" action. Because in the IP-ARP example all hosts are modeled using `IpArpNode` instances, the "attacker" action is present in all hosts. Through a guard statement `nid == HA_ID`, however, the "attacker" action can only be activated on host HA.

9.2.2.2 Attacker Action

Typical ARP attacks work by *cache poisoning*. A malicious host, say HB, sends out an ARP reply or query with another host's protocol address, say HC, but its own hardware address. With the right timing, this will cause the wrong protocol to hardware address mapping to be stored in the ARP cache of other hosts, say HA. If HA

later prepares an IP packet to HC, HC's protocol address will be found in the ARP cache. Thus, no further ARP lookup occurs. Instead, the packet is prepended with host HB's hardware address and submitted to the network interface layer. Then, it will be transmitted to host HB instead of HC, the recipient intended by HA.

ARP issues do not have to be caused by a wrong ARP reply sent by a malicious host, however. Administrative oversight or misconfigurations may lead to the same issues. For the scenario model considered here, we focus on such cases. Thus, our modeling includes a local administrative action (`chg_ip`) for changing the IP address of the node with "attacker" role to another IP address.

9.2.2.3 Node Initialization

All attributes of the nodes have to be initialized. The special action `INIT` defined locally for each node handles these initializations. Later on, during system composition, all `INIT` actions of the nodes in the system are linked to build the system initialization. The system initialization is always executed before any other action.

In the IP-ARP scenario, the `INIT` action of the nodes defines send and receive buffers to be empty. No packets are in transit in the media (i.e., empty zone buffers). Furthermore, the ARP cache is set to be empty as well, using a symbolic invalid protocol and hardware address pair. For each node, the network interface is initialized with an IP and hardware address. These address initializations use appropriately defined functions `fSrcToHa` (hardware address initialization) and `fSrcToIa` (IP address initialization) from the framework.

9.2.3 Network View

An important aspect to consider in the IP-ARP scenario is that the LAN is *switched*. In a non-switched LAN, all hosts connected to the same LAN can physically receive communications directed at other hosts. The network interface can be set to the so-called *promiscuous mode*. In this mode, the interface ignores hardware addresses and receives any packet it is physically able to receive.

To model the switched LAN, we use a node type which disallows promiscuous reception. This is realized by a guard in the receive (`rcv`) action stating that either the packet's destination hardware address has to match that of the receiving interface or the packet has to be a broadcast packet.

As all nodes are connected to the same LAN, we only require one explicit zone Z1 for the network view. Through the framework's `ZoneIdT` enumeration a special unknown or undefined zone (constant `UNKNOWN_ZONE`) is always included in models. Furthermore, there are no routers, i.e., nodes with two or more interfaces. Taken together, these two properties make the definition of an appropriate topology function `fSrcToZone` straightforward. For a node *n* and interface *i*, if

$(n, i) \in \{\text{NA}, \text{NB}, \text{NC}\} \times \{\text{I1}\}$ then `fSrcToZone`(n, i) := Z1 and `UNKNOWN_ZONE` otherwise.

The LAN itself is modeled by a `Media` instance. For each zone, the `Media` process type defines a zone buffer. The zone buffer holds a packet, if and only if it is currently in transit in the zone. By means of the already described zone definitions and topology functions, the `Media` instance in the IP-ARP scenario is configured to have only one zone Z1 and thus one zone buffer. This buffer is emptied by the initialization action `INIT` of `Media`.

9.2.4 System Composition

Figure 9.3 shows the structure of the IP-ARP cTLA system process type.

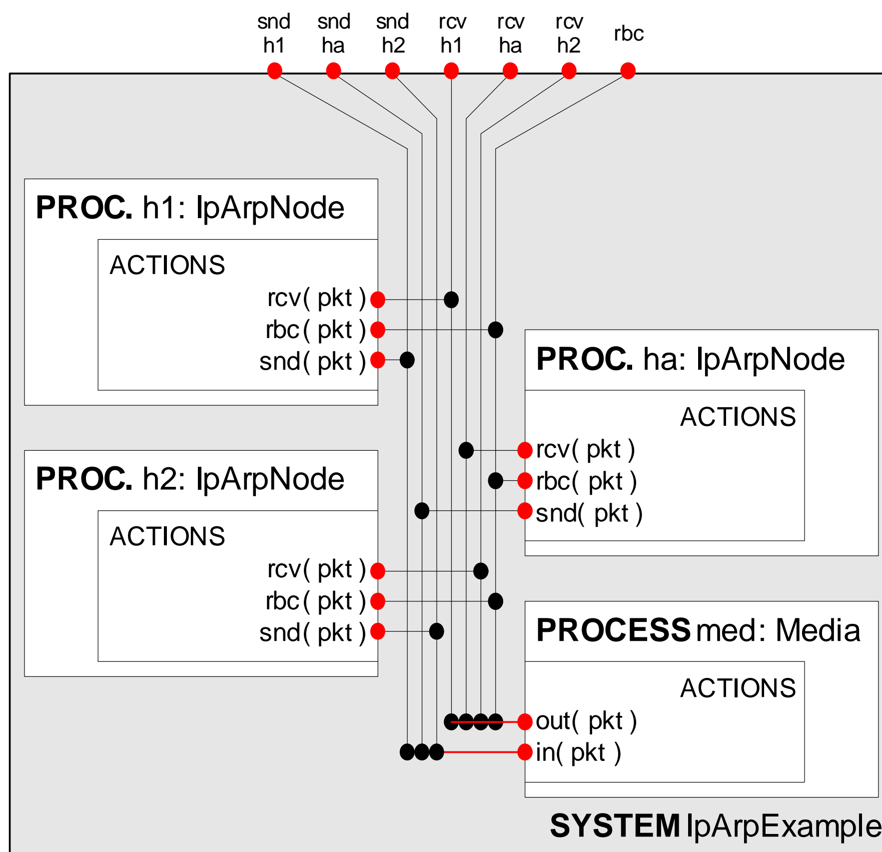


Figure 9.3: Compositional Structure of the IP-ARP Model

Processes The system is instantiated from the process type `IpArpExample`. In turn, process type `IpArpExample` contains three `IpArpNode` process types and

one Media process type. As described above, the process type IpArpNode is used to model the hosts. The LAN is represented by process type Media, with supporting functions and zone definitions corresponding to the network topology.

While the process type Media is taken directly from the framework (cf. Fig. 9.4, depicted as an unfilled box), IpArpNode is a specific process type (depicted as a filled box). It is derived from the framework process type ActiveNonPromHostIpNode, however.

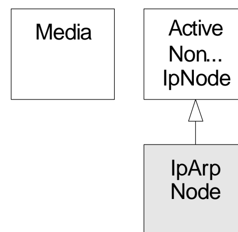


Figure 9.4: Framework and Specific Process Types of the IP-ARP Model

System Actions The collaboration of the process types contained in the system still has to be defined. In CTLA, this is done via system actions. Internal actions (typically modeling internal processing of a node) are implicitly added to the system actions (cf. chapter 5). Thus, two main classes of system actions determine the system composition: send actions (snd_X, where $X \in \{H1, H2, HA\}$) and receive actions.

Regarding the receive actions, we can distinguish the subclasses of unicast receive actions (rcv_X) and broadcast receive (rbc) actions. The send actions couple a node's snd action with Media's in. Thus a packet which is sent by a node, is transmitted into Media's packet buffer. Similarly, the unicast receive actions couple a node's rcv action with Media's out. This models that, in order to be received, a packet has to be taken out of Media's packet buffer. Finally, the broadcast receive system action couples together the rbc action of *all* nodes with Media's out action. Thus, a broadcast packet is received by all nodes and taken out of Media's zone buffer.

9.3 Analysis

In this section we describe the steps taken for automated analysis of the IP-ARP scenario. We begin with the security property the model is checked against. Then, the optimizations taken and their effect for successful analysis is are described. Finally, the results of the automated analysis with SPIN are briefly explained.

9.3.1 Security Property

We aim to analyze our model for attacks on the low-level packet exchange between the hosts. Such attacks typically let hosts receive packets that are directed at other hosts. The corresponding security property is that a node cannot receive non-broadcast IP packets destined for other nodes. To find such attacks using automated analysis, we use assertions. SPIN supports the checking of a PROMELA model for assertion violations. Thus we have to insert the assertions after the translation of the model to PROMELA.

One way to model the security property via assertions is shown in Listing 9.1. In the assertion, $X \in \{\mathrm{h1}, \mathrm{h2}, \mathrm{ha}\}$ stands for the node where the assertion is inserted and Y represents the node which is the intended recipient of the packet. To cover both possible intended recipients $Y_1, Y_2 \in \{\mathrm{h1}, \mathrm{h2}, \mathrm{ha}\} \setminus \{X\}$ for a node X, the assertion has to be inserted twice. The assertions have to be inserted into the send actions (`snd`) of the nodes. Thereby, the assertion will be triggered at the moment a packet violating the property is sent.

```
assert(
        !(bnX_ifs[1 - 1].spa.pkt.l2t==L2_IP &&
          BVGET( bnX_ifs[1 - 1].spa.pkt.dat, 3, DI_IDA) ==
          bnY_ifs[1 - 1].ia &&
          bnX_ifs[1 - 1].spa.pkt.dha != bnY_ifs[1 - 1].ha )
);
```

Listing 9.1: Assertion 1: IP-ARP Example, Send Actions

The security property can also be modeled in another way with an assertion (cf. Listing 9.2) inserted at the end of the non-broadcast receive action (*rcv*) instead of the send actions of each node. In this case, the assertion depends only on the node X where it is inserted into the receive action, and does not have to be inserted multiple times to cover all cases. As the assertion is triggered one step later (receive instead of send action), however, more memory is required for checking the model.

```
assert(
  ! ( bnX_ifs[1-1].rpa.pkt.l2t==L2_IP &&
  BVGET(bnX_ifs[1 - 1].rpa.pkt.dat, 3, DI_IDA) !=
  bnX_ifs[1 - 1].ia )
);
```

Listing 9.2: Assertion 2: IP-ARP Example, Receive Actions

The assertion checks that a received IP packet's destination address equals the IP address belonging to the interface that received the packet. Thus, if a node manages

to receive a packet that is directed at another IP address physically, the assertion will be violated.

Both assertions are triggered by very similar sequences. Due to the design of the assertion, sequences violating assertion 2, are one step longer than sequences violating assertion 1. Furthermore, if $(s_1, \ldots s_{n-1}, s_n)$ violates assertion 2, then $(s_1, \ldots, s_{n-1})$ violates assertion 1. Just as well, if $(s_1, \ldots, s_n)$ violates assertion 1, then $(s_1, \ldots, s_n, \text{rcv_X})$ violates assertion 2. Thus, violating sequences do not depend on the exact modeling of the confidentiality property with assertion 1 or 2 in a considerable way.

9.3.2 Optimizations

In the following section, we describe the key optimizations in the IP-ARP scenario. Table 9.1 gives an overview of the application order and the effects on the state-vector size of these optimizations. The initial state-vector size of the IP-ARP scenario is about 830 bytes.

Optimization	State Vector
(initial version)	$\sim$ 830 Bytes
+ Representative Hosts & Roles	$\sim$ 480 Bytes
+ Activity Thread & Integrated Layers	$\sim$ 250 Bytes
+ Paramodulation	$\sim$ 210 Bytes
+ Bit Array Mapping	$\sim$ 170 Bytes

Table 9.1: Optimization Effects on State-Vector Size in the IP-ARP Example

9.3.2.1 Representative Hosts & Roles

The first optimization taken in the IP-ARP scenario is the selection of representative hosts (cf. section 9.2.2.1). In the initial version of the model there were five hosts; after the optimization there are only three (NA, NB, NC). Accordingly, this nearly halves the state-vector size from about 830 to about 480 bytes.

9.3.2.2 Activity Thread & Integrated Layers Approach

In the initial version of the model, all layers have separate processing actions and buffers. By redesigning process type `IpArpNode` so that it is based on `HostIpNode` from the framework (cf. chapter 7), we “apply” the activity thread and integrated layers approaches (cf. section 8.3.2). Thus, we save on buffers and action steps required for protocol processing. Only one processing action (`rcv`) and

one working buffer for each send processing and receive processing of packets remains. This reduces the state-vector size to about 250 bytes mainly due to saved working buffers.

9.3.2.3 Action Parameter Paramodulation

For the packet related parameters of the `snd` and `rcv` system actions in the IP-ARP scenario, value determining equalities exist in the flat system (cf. section 8.4.1). These system actions couple together `Media` and `Node` process types. Thus, the packet parameter is either an input parameter of `Media`'s action and an output parameter of `Node`'s action or the other way around.

In these cases, therefore the parameter can be replaced by the corresponding symbolic output value and removed from the action's parameter list. Using cTLA2PC's built-in switch `--paramod` for translation of the model, this optimization is applied automatically. The state-vector of the "paramodulated" version of the model is shortened to about 210 bytes.

9.3.2.4 Bit Array Mapping

SPIN has some problems to handle bit arrays in an efficient way. By letting cTLA2PC encode arrays and replace array operations (cf. section 8.5.1), we can reduce the state vector to its final size of about 170 bytes.

9.3.3 Results

Attempts for automated analysis with early models of the IP-ARP scenario fail quickly due to SPIN running out of memory. Using approximative verification modes prevents the memory exhaustion; however, SPIN produces no results even after a long time (about 150 hours).

Fortunately, with the optimization steps described above taken, automated analysis of the IP-ARP scenario with SPIN becomes successful. A series of violating sequences has been found by SPIN. An example SPIN output, with assertion 1 inserted in the model, is shown in Listing 9.3.

The output shows that SPIN found a violation of the security property which, in this case, is modeled using assertion 1. SPIN writes an internal encoding of the steps leading to the violation to a trail file. The trail file can be "played back" using SPIN's guided simulation mode. This results in the corresponding PROMELA level sequence of steps. As cTLA2PC integrates print statements for cTLA action names and parameters in the generated PROMELA code, the mapping of the sequence back to the level of cTLA actions follows easily.

```
pan: assertion violated
  !(((ha_ifs.rpa.pkt.l2t==1) && (((ha_ifs.rpa.pkt.dat>>(1*3)) &&
  ((1<<3)-1))!=ha_ifs.ia))) (at depth 12)
pan: wrote ip-arp-....promela.trail
(Spin Version 4.1.0 -- 6 December 2003)
        Warning: Search not completed
        + Using Breadth-First Search
        + Partial Order Reduction
        ...
State-vector 168 byte, depth reached 12, errors: 1
```

Listing 9.3: Spin Verifier Output in the IP-ARP Example (Assertion 1)

9.3.3.1 Violating Sequence 1

Figure 9.5 shows the attack sequence found by SPIN on the cTLA level. For clarity, instead of including input generator steps, we give the parameters to the actions resulting from these steps.

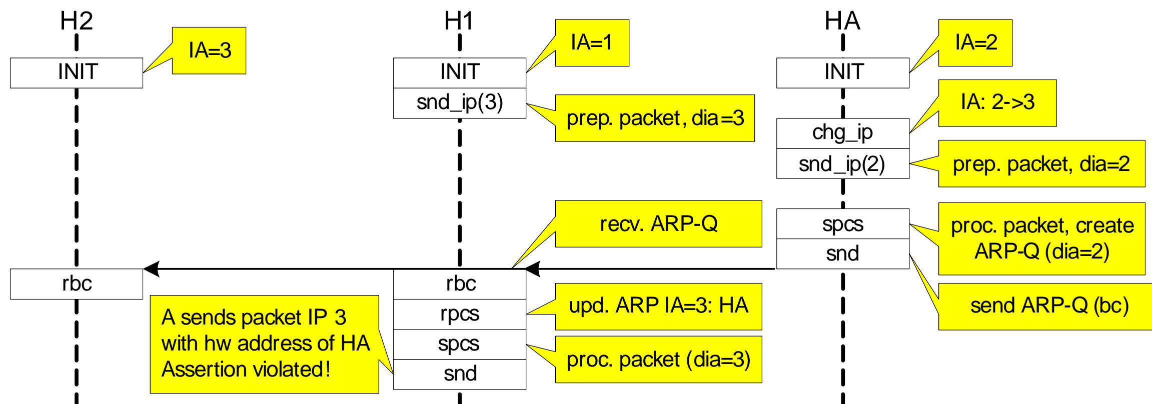


Figure 9.5: Violating Sequence 1 in the IP-ARP model

The sequence starts by Node HA changing its IP address (`chg_ip`) to 3, but the ARP cache of the other nodes is unchanged. Especially, a previously received (`rbc`) ARP query from node HA still contains the old IP address and node H2 updates its ARP cache from this query (`rpcs`) just now. Thus, an already buffered IP packet for the (now unused) IP address 2 can be processed (`spcs`) with the cached IP address to hardware address assignment. The packet is sent to the media with the destination hardware address of node HA. This violates the assertion included in the send action, requiring that IP and hardware destination address match.

Interestingly, this sequence closely resembles cache poisoning ARP attacks. In such an attack, node HA would send an ARP reply with its own hardware address

as answer for an ARP query for node H2's protocol address. The second half of the sequence in Figure 9.5 shows this course of action. The sequence is triggered by an IP change administrator action, however, and not a deliberate ARP attack. Thus, this demonstrates the fact that an IP change may lead to the same effects as an ARP attack. Especially, confidentiality may be violated until all nodes in the subnet have updated their ARP caches correctly. This is particularly relevant to hosts with static ARP configurations, since their settings have to be maintained manually.

9.3.3.2 Violating Sequence 2

Another violating sequence is shown in Figure 9.6.

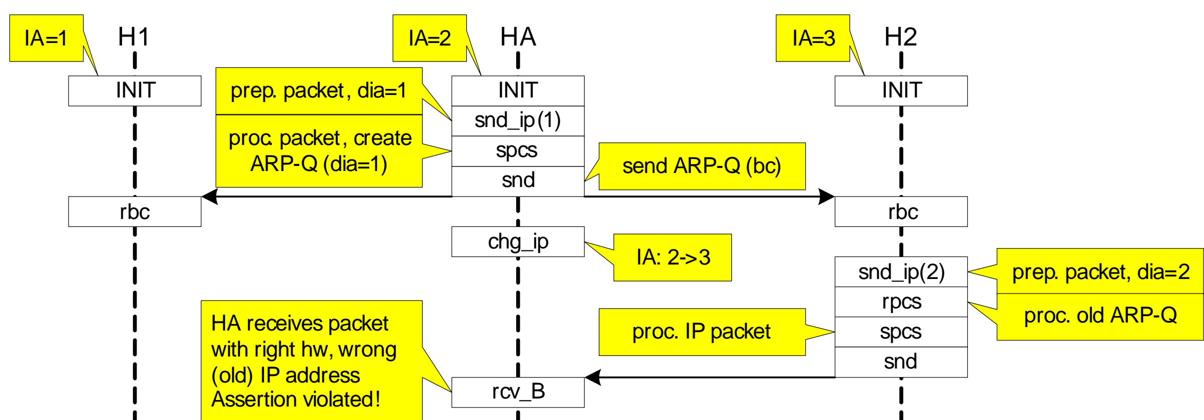


Figure 9.6: Violating Sequence 2 in the IP-ARP model

As this sequence stems from assertion 2, the violation occurs in a receive action instead of a send action. In contrast to sequence 1, this sequence does not resemble an ARP attack. Furthermore, the issue is less confidentiality than *availability*. The packet prepared by node H2 (`snd_ip(2)`) for node HA is indeed transmitted to node HA (`snd`). The packet's and node's IP address don't match, however, probably causing the packet to be thrown away after it has been physically received by node HA (`rcv_B`). Thus, this sequence demonstrates that an IP change may lead to availability problems as well. Again, special care is required for static ARP configurations.

9.4 Discussion

In this section, we summarize the key points learned and offer some discussion points related to the IP-ARP scenario. We begin with the modeling and analysis, move to the attack sequences and finish with some measures for improving security.

Modeling & Analysis

- We successfully modeled a small LAN scenario with three hosts running TCP/IP, including ARP in the network interface layer and administrator actions. The modeling is done in cTLA and based on an early version of the current framework (cf. chapter 7). The ARP-specific behavior is captured in a new process type, `IpArpNode`, derived from `HostIpNode`.
- The size of the cTLA model file is about 15 KB (without comments). After translation with cTLA2PC, the PROMELA model file size is about 35 KB. Automatic analysis with SPIN can be tackled after insertion of the security property and a series of optimizations.
- Optimizations (cf. chapter 8) are particularly effective in the IP-ARP scenario. At the time we began to work on the IP-ARP scenario, our tools and the framework were still emerging. Thus, there was more potential for optimizations than with the current, already heavily optimized versions of the framework and our tools. From the initial version of the IP-ARP scenario with its state-vector of about 830 Bytes to the final version with 170 Bytes, significant savings could be achieved. For current models, achievable savings are typically smaller.

Attack Sequences

- Practical attack sequences can be found with automatic analysis of our model using SPIN. Interestingly, these sequences show that ARP attacks can have very similar effects to administrator actions like IP changes, be they benevolent or malicious.
- Tools for injecting ARP packets like `nemesis-arp` have existed for a long time. These tools require a certain level of knowledge, however. Lately, more advanced tools like `ettercap`, which allow for man-in-the-middle attacks by simply selecting the IP address of a host to listen on, have appeared.

Improving Security

- Careful administration helps to prevent errors caused by duplicate IP assignment etc. Deploying DHCP is useful as well but opens up attack possibilities (e.g., man-in-the-middle attack through *rogue DHCP server*) of its own.
- ARP supports no authentication at all. Some implementations check that they actually have sent a query before they accept a reply (they do not accept so called *gratuitous ARP replies* or *unsolicited responses*). This only provides for partial protection, and may depend on timing in case multiple responses (one

genuine and one spoofed) are received. Furthermore, some applications, e.g., *fail-over* solutions, require gratuitous ARP replies.

- Static ARP mappings provide protection against most attacks. Particularly, they are helpful to protect ARP mappings for especially important destinations like the default gateway or critical servers. Using Linux, ARP can be disabled on a per interface basis using `ifconfig <if> -arp`. Of course, static mappings increase the administrative overhead in case of hardware (e.g., NIC) changes. As already described, some applications might not work anymore.

- Cryptographically secured versions of ARP have been suggested e.g., by Bruschi et al [BOR03]. Their version of ARP depends on PKI based public/private key cryptography. Because of the high complexity of PKI deployment and management, it seems doubtful that these approaches will reach widespread acceptance.

- Advanced switches, namely those from the CISCO CATALYST series, include features to prevent ARP spoofing (*Dynamic ARP inspection*). These features block spoofed ARP packets based on information from DHCP tables. Now rogue DHCP servers have to be prevented. This can be done with further switch configuration which requires to list all ports to which trusted DHCP servers are connected statically. Of course, this somewhat increases administrative overhead as well.

- Deployment of network intrusion detection systems, e.g., SNORT [Sno05] with its experimental preprocessor `arpspoof`, can help to detect ARP spoofing attacks.

10 Case Study: IP-RIP

For the IP-RIP case study, we first introduce the overall background of the scenario. Then, we describe the modeling task. Afterwards, the steps taken to enable automated analysis, particularly security property and optimizations, are presented in detail. Furthermore, the analysis results are explained. Finally, we give a short summary of the lessons learned from the application of our approach to the IP-RIP scenario.

10.1 Introduction

In the IP-RIP scenario [RKK05], we examine multiple LANs and hosts connected by routers as depicted in Figure 10.1. The hosts in the LANs run TCP/IP; the routers additionally run the *routing information protocol (RIP)*.

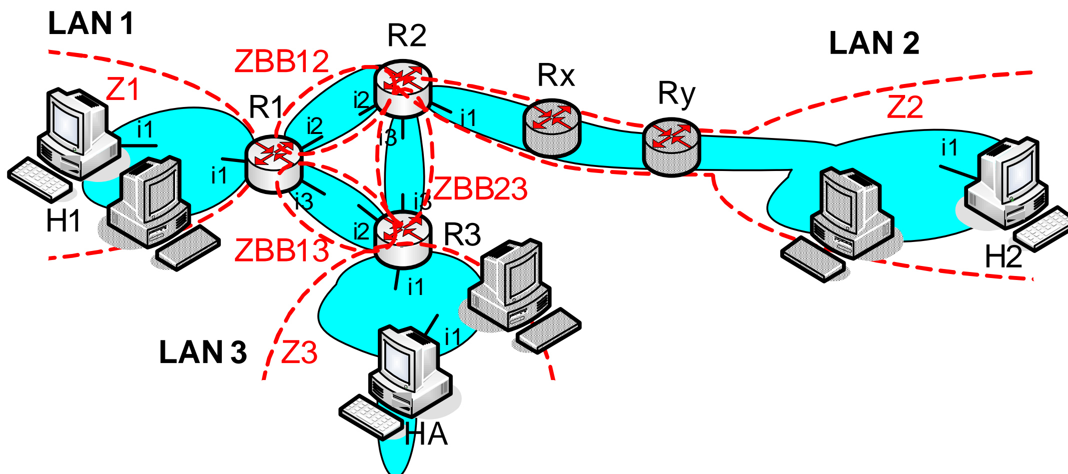


Figure 10.1: IP-RIP Scenario

Classification RIP is the most popular distance-vector, interior-gateway routing protocol (cf. section 7.2.2). In a *distance-vector* protocol, each router maintains a distance value to all other networks it knows about. Each router sends updates of the form “I can get to network N in d hops” to its neighbors. Due to technical

reasons detailed below, RIP is limited to autonomous systems with paths shorter than 16 hops.

Attack Ideas & Tools The attack ideas described in section 7.2.3 are applicable to RIP. Tools like `nemesis-rip` from the Nemesis project [NS04] or `srip` [Hum00] allow to send RIP update packets with an arbitrary source IP, destination and metric to a remote router. As in RIP all routing information is implicitly trusted and the routers have no full topology information, a single packet injection may suffice to effect routing permanently.

10.2 Modeling

The framework (cf. chapter 7) supplies the overall structure of our modeling. Like in the IP-ARP example, we build the IP-RIP cTLA model by taking different views at the scenario. First, we describe the protocol oriented view, then the node and finally the network view. The cTLA 2003 source code of the IP-RIP model is contained in Appendix B.

10.2.1 Protocol View

In the IP-RIP scenario, the layers application, transport, internet, and network interface are required to model the contained protocols (cf. Fig. 10.2).

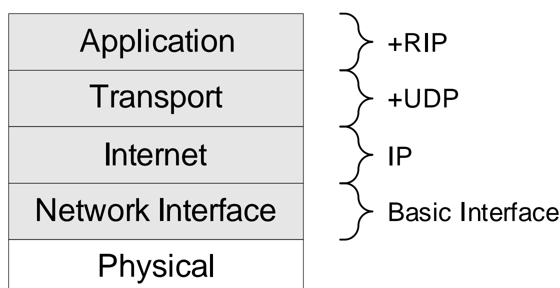


Figure 10.2: Layers and Protocols in the IP-RIP Scenario

The framework's process types `HostIpNode` and `RouterIpNode` already supply adequate internet and network interface layers. Thus, it suffices to implement RIP on the application layer and its encapsulation on the transport layer.

10.2.1.1 Application Layer – Routing Information Protocol (RIP)

In our modeling, RIP is implemented in the process type `RipRouterIpNode` which is based on `RouterIpNode`. We only give a short overview of RIP here,

a more detailed description is contained in [Per99] and the RFC [Hed88].

Each RIP router keeps distance vectors of the form (`dst`, `nho`, `itf`, `met`) in its local *routing information base* (RIB). The field `dst` contains the IP address of the destination and `nho` the IP address of the *next-hop*, i.e., the next router on the way to the destination. If the current router is directly connected to the final destination's network, this field contains a special value (NHO_DIRECT in our modeling). The `itf` field contains the interface connected to the next-hop or the final destination's network. The `met` field is used for storing a *cost metric*, usually the number of hops, from the current router to the final destination. A metric of 1 denotes that the current router is directly connected to the final destination's network. On the other hand, a metric of 16 := MET_INF means infinite cost.

RIP works in two stages: input processing and output processing. *Input processing* handles RIP update packets received from other routers. The critical element of a *RIP update packet* is the pair `dst'`, `met'`. A RIP update packet may contain several such pairs. Since the pairs are processed sequentially, we can assume without loss of generality that all update packets contain only one pair. An update packet with *n* pairs is then replaced by *n* update packets with one pair. The fields describe the best route (in terms of metric) to the destination as known by the router from which the packet originated. If the update packet passes basic sanity checks (e.g., `met'` < `MET_INF`) and – optionally – its IP source address matches a directly connected network, the packet is considered for updating the router's RIB.

Two cases have to be distinguished. If a route for the destination is not yet known by the current router, an entry $(\mathrm{dst}', \mathrm{nho}, \mathrm{itf}, \mathrm{met}' + 1)$ is added, but only if $\mathrm{met}' + 1$ is still less than MET_INF, i.e., the new destination is at most 15 hops away. The field nho is set from the IP source address of the packet, itf is determined through a lookup in the current routing table.

If a route for the destination contained in the update packet already exists, the RIB entry is changed only if $\mathrm{met}' + 1 < \mathrm{met}$, i.e., the new route is better than the existing one. The 'new route is better' condition does not apply, however, if the packet originates from the next-hop of the existing route according to the packet's source address. If a route in the RIB has been changed (no matter which case applied), its change flag is set.

Output processing handles the sending of RIP update packets. Update packets can be sent both periodically (*regular update*) or because of changes (*triggered update*). We only model triggered updates, because we are interested in dynamic behavior resulting from route changes.

All updates are sent observing the *split horizon* principle. That means the updates are sent to the neighboring routers with the exception of the router from which this route was received (i.e., the next-hop nho).

10.2.1.2 Transport Layer – UDP

RIP messages are encapsulated in the transport layer *user datagram protocol (UDP)* and use a reserved port (520). To simplify our modeling, we simulate this encapsulation using IP packets with a reserved packet type value of PT_RIP. This way, we do not have to include a transport layer with UDP.

10.2.2 Node View

In the IP-RIP model, all routers are modeled as instances of the `RipRouterIpNode` process type. The hosts are modeled by different process types based on `HostIpNode` according to their role in the scenario.

10.2.2.1 Representative Nodes & Role Assignment

The selection of representatives and assignment of roles greatly helps with the transition from a loose scenario to a strict analysis model. We chose one host from each LAN (cf. Fig. 10.1) as representative for the set of hosts in the LAN. Host H1 represents the hosts from LAN 1, H2 LAN 2, and HA LAN3.

From the routers, we chose to represent only R1, R2, and R3 directly as process type `RipRouterIpNode` instances in the model. The routers Rx and Ry, which may connect further LANs besides connecting router R2 to LAN 2, are not directly represented. They are implicitly represented in the distance metric of router R2's initial RIB, however.

Furthermore, we assign the roles attacker, active and passive communication partner to the hosts. For the analysis of this scenario, we map the roles to the hosts in the following way: H1 $\mapsto$ sender, H2 $\mapsto$ receiver, HA $\mapsto$ attacker. Accordingly, host H1 is an instance of process type `ActiveNonPromHostIpNode`, and H2 is an instance of `NonPromHostIpNode`. Both process types are taken from the framework (cf. chapter 7).

The attacker host HA is modeled by type `RipAttackerHostIpNode` which extends `ActiveHostIpNode` with a local attacker action.

10.2.2.2 Attacker Action

Typical routing attack tools (e.g., IRPAS [FX01]) allow to inject update packets in the routing process. Thus, we include a local attacker action `snd_ripu` in the process type `RipAttackerHostIpNode`.

This action creates a RIP update packet and puts it in the local send buffer. For example, the packet may announce a new route to a certain destination with a short distance and the local host (i.e., the attacker) as the next-hop.

As process type `RipAttackerHostIpNode` is based on `ActiveHostIpNode`, it inherits the `snd` action. Thus, no additional action for sending the malicious RIP update packet is required; it can be sent using the default action `snd`.

10.2.2.3 Node Initialization

The attributes of each node have to be initialized through the local `INIT` action. As usual, all send and receive buffers of the nodes (i.e., routers and hosts) are set to be empty, as well as media's zone buffers. Furthermore, for all nodes, the network interfaces are set up using the function `fSrcToIa` from the framework. This function maps each interface to a symbolic IP address (e.g., the second interface of router R1 is mapped to the symbolic IP address `R1_I2_IA`).

In the IP-RIP scenario, an additional initialization step remains: setting up the RIBs. For each router, we start with a fully initialized routing table, not an empty table. Thus, the routing tables correspond to a stationary state of the system, not to an empty state immediately after boot-up of a router. As the RIBs depend on the network topology of the scenario, we describe the routing tables in the following section.

10.2.3 Network View

We still have to consider the network view. In particular, the network topology, i.e., the LANs with their connections provided by the interfaces of the nodes and routers has to be modeled. For the IP-RIP scenario, the LANs (cf. Fig. 10.1) are modeled by a `Media` instance with six zones (Z1, Z2, Z3, ZBB12, ZBB13, ZBB23). While the zones Z1, Z2, Z3 correspond to the host LANs, the zones ZBB12, ZBB13, ZBB23 represent the backbone networks between the routers. The topology function `fSrcToZone` is defined appropriately (e.g., the second interface of router R1 is mapped to `ZBB12`).

For each router, the routing table entries have the form (`dest`, `nhop`, `metr`, `itf`) (cf. section 10.2.1.1). The initial routing tables correspond to the scenario's network topology. For example, R1 is directly connected (i.e., metric 1) to zones ZBB12 and ZBB13 over interfaces 2 respectively 3 (cf. Table 10.1).

It has to be noted that the route from R1 to Z2 via next-hop R2 (interface 2) has metric 4 (i.e R1, R2, Rx, Ry). As discussed above, this is due to the indirect representation of routers Rx, Ry in the routing tables only.

10.2.4 System Composition

The IP-RIP cTLA system is composed as depicted in Figure 10.3.

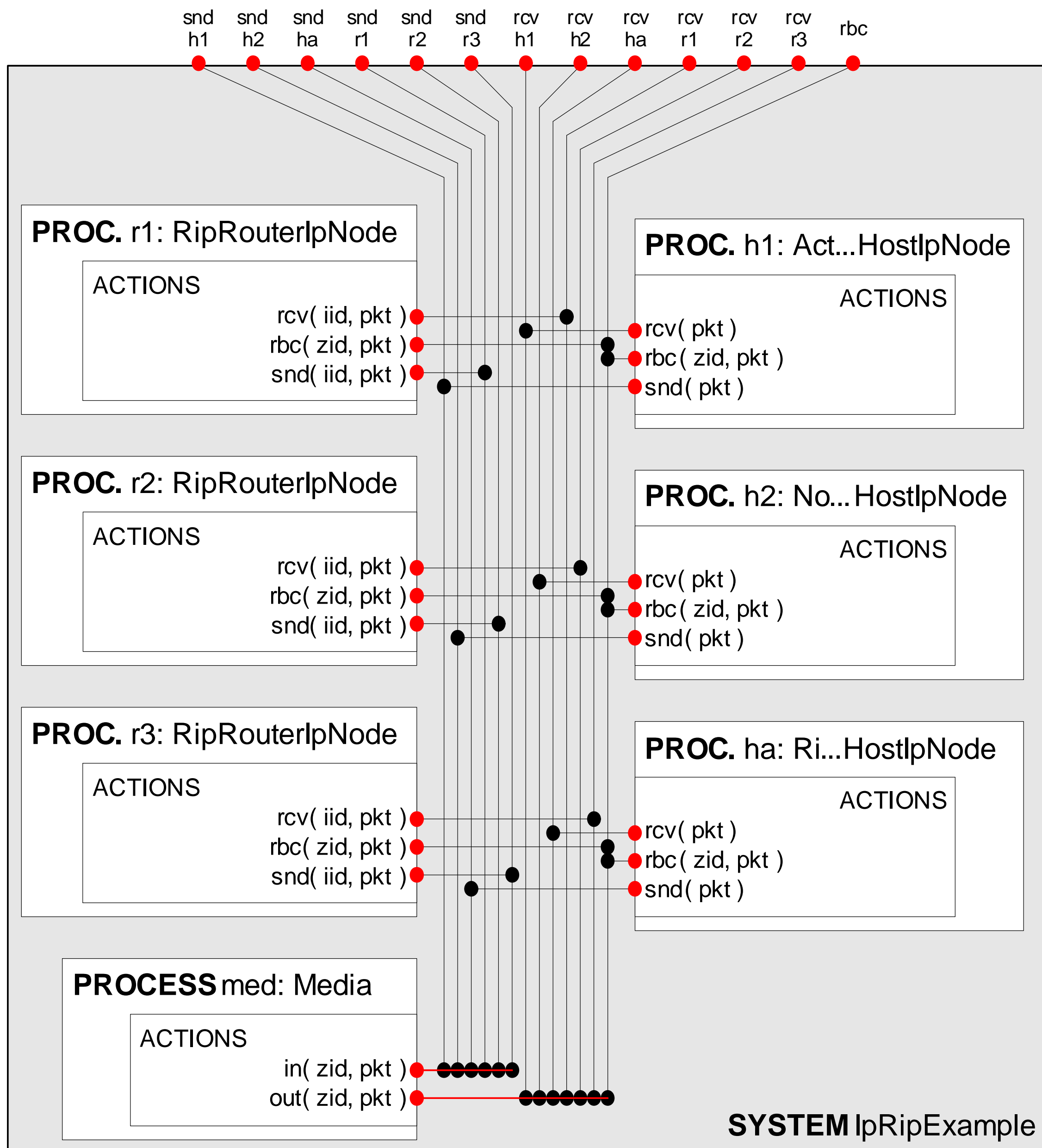


Figure 10.3: Compositional Structure of the IP-RIP Model

Destination	Next-Hop	Metric	Interface
Default	Direct	-	1
ZBB12	Direct	1	2
ZBB13	Direct	1	3
Z1	Direct	1	1
Z2	R2_I2	4	-
Z3	R3_I2	2	-

Table 10.1: Initial routing table of R1

Processes The system is instantiated from the process type `IpRipExample`. This process type contains one `Media` instance representing the LANs, three `RipRouterIpNode` instances for the routers, and three instances derived from `HostIpNode` (`RipAttackerHostIpNode`, `NonPromHostIpNode`, and `ActiveNonPromHostIpNode`) for the hosts.

Of the process types only `RipRouterIpNode` and `RipAttackerHostIpNode` are specific to the IP-RIP model (cf. Fig. 10.4, depicted as a filled box). They are derived from the framework's process types `RouterIpNode` and `ActiveHostIpNode`, however. All other process types are taken directly from the framework (depicted as an unfilled box).

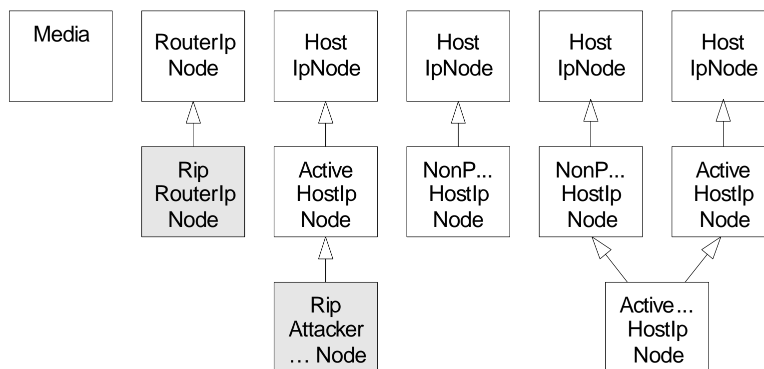


Figure 10.4: Framework and Specific Process Types of the IP-RIP Model

System Actions The collaboration of the processes making up the system is defined via system actions. For the IP-RIP system, the same two classes of system actions as in the IP-ARP system are defined (besides the implicitly added internal actions): send actions (`snd_X`, where *X* is a node or router) and receive actions. The receive actions are structured into subclasses for unicast receive actions (`rcv_X`) and the broadcast receive (`rbc`) action.

As in the IP-ARP system, the send actions couple a node's `snd` action with

Media's in and the unicast receive actions couple a node's rcv action with Media's out action. The broadcast receive system action couples together the rbc action of *all* nodes in *all* zones with Media's out action. Due to the multiple zones in the IP-RIP scenario, however, the rbc system action takes an extra parameter referring to the zone where the broadcast reception shall take place. The nodes' rbc actions are designed to not block even if no broadcast reception is possible in the supplied zone. In this case, the execution of a node's action doesn't change any state variables, it is equivalent to a stuttering step of a node. Thus, although all rbc actions of all nodes are coupled in the rbc system action, it is always executable and will only take a broadcast packet out of the zone buffer of the supplied zone if such a packet exists.

10.3 Analysis

In the following section we describe the steps taken towards automated analysis of the IP-RIP scenario. First, the security property the model is to be checked against is considered. Then, the optimizations are applied and their effects on state-vector size are given in detail. Finally, the results of the automated analysis with SPIN are depicted.

10.3.1 Security Property

We plan to analyze our model for attacks leading to injection of false route information (cf. section 10.2.2.2). In the worst case, such injections lead to packets being routed to the attacker. Thus, we consider the property that an attacker can only receive packets which are destined for its own IP address.

With the assignment of the roles described earlier, the security property can be formalized using the assert statement given in Listing 10.1.

```
assert( (ha_itf.rpa.pkt.dat_ida == HA_I1_IA) );
```

Listing 10.1: Assertion in the IP-RIP Example

This statement requires that host HA (which has the attacker role) can receive a packet only if the packet's destination IP address matches the symbolic IP address of HA's (one and only) interface, H1_I1_IA. After translation of the cTLA model to PROMELA with cTLA2PC, this assertion is inserted into the PROMELA representation of the rcv_ha action (i.e., host HA's non broadcast packet receive action). Because host HA is a host node and *not* a router node, we do not have to worry about unintended violations of the property due to packets received by standard

next-hop routing. Furthermore, broadcast packets can not trigger this assertion either, because they are received by the `rbc` action instead of `rcv_ha`.

10.3.2 Optimizations

In this section, we describe the optimizations applied from the initial version of the IP-RIP model to the final version. Table 10.2 outlines the order of application and the state-vector size resulting from the optimizations applied so far.

Optimization	**State Vector**
(initial version)	∼ 720 Bytes
+ Representative Nodes & Roles	∼ 450 Bytes
+ Paramodulation	∼ 430 Bytes
+ Bit Array Mapping & Low-Level Tweaks	∼ 340 Bytes
+ Unroll action parameters	∼ 320 Bytes

Table 10.2: Optimization Effects on State-Vector Size in the IP-RIP Example

The initial state-vector size of the IP-RIP model is about 720 bytes. Surprisingly, this is smaller than the initial size of the IP-ARP model (830 bytes, cf. chapter 9). This is due to the fact that even in the initial version of the IP-RIP model the nodes are already based on the framework. Thus, they integrate the activity thread and integrated layers (cf. section 8.3.2) approaches and save on actions and buffers for protocol processing right from the start.

10.3.2.1 Representative Nodes & Roles

The initial version of the model consists of five routers and three hosts. By choosing representative nodes and fixed roles, only three routers (R1, R2, R3) and three hosts (H1, H2, HA) with fixed roles remain (cf. section 10.2.2.1). Thus, the state-vector size is decreased by about 38% from 720 to 450 bytes.

10.3.2.2 Action Parameter Paramodulation

As in the IP-ARP scenario (cf. section 9.3.2.3), due to packet related parameters of the `snd` and `rcv` system actions being input parameters of `Media` and output parameters of `Node` or vice versa, paramodulation can be successfully applied in the IP-RIP scenario. Using cTLA2PC's `--paramod` switch during model translation, the state-vector shrinks to about 430 bytes.

10.3.2.3 Bit Array Mapping & Low-Level Tweaks

SPIN maps bit-size variables to bytes. By using a more efficient low-level encoding in the PROMELA model, the state-vector size can be reduced. Furthermore, we decreased the size of the routing tables. In combination, these two optimization save another 20 bytes in the state-vector.

10.3.2.4 Unroll Action Parameters

Still automated analysis efforts of the scenario described failed, however. In test runs, the verifier generated by SPIN exceeded available memory (i.e., 3 GB, the practical x86 per process memory limit) after reaching search depth 23. Thus, we had to consider further optimization possibilities, especially those which help to reduce the depth required for action sequences. By looking at simulation sequences, we found out that input generator steps (i.e., parameter value settings) were contributing about one third to the depth. Therefore, we focused on optimization possibilities related to input generator steps.

We developed the unroll actions approach (cf. section 8.4.2) which replaces parameterized actions by multiple copies, where in each copy the parameters have been replaced by fixed values.

Optimization	State Vector	Stored States	Transitions	Depth	Memory
(previous version)	$\sim$ 340 Bytes	1.19E+06	2.3E+08	14	203 MB
+ Unroll	$\sim$ 320 Bytes	1.99E+04	1.8E+06	11	11 MB

Table 10.3: Effects of the Unroll Actions Optimization on a Benchmark Sequence in the IP-RIP Example

Clearly, this approach has only a slight effect on the state-vector size. In the IP-RIP example, the state-vector only decreases by 5% to about 320 Bytes (cf. Table 10.3). As however the parameter setting input generator steps are no longer required, the search depth required for a specific sequence is greatly reduced. A benchmark sequence's depth decreased by 20% from 14 to 11 after the application of the unroll approach. Consequently, memory usage is vastly reduced.

10.3.3 Results

Early attempts for automated analysis of the IP-RIP scenario failed most often because of SPIN running out of memory. With the development and application of the optimizations described in section 10.3.2, however, automated analysis became possible. We analyzed the described modeling for assertion violations using SPIN on a standard PC system. The corresponding SPIN output is shown in Listing 10.2.

```
pan: assertion violated (ha_itf.rpa.pkt.dat_ida==11) (at depth 21)
pan: wrote ip-RIP-example-veri-flat-para.promela.trail
(SPIN Version 4.2.0 -- 27 June 2004)
...
State-vector 316 byte, depth reached 21, errors: 1
5.5768e+06 states, stored
5.5768e+06 nominal states (stored-atomic)
4.28228e+08 states, matched
4.33804e+08 transitions (= stored+matched)
...
Stats on memory usage (in Megabytes):
1806.883 equivalent memory usage for states
  (stored*(State-vector + overhead))
985.366 actual memory usage for states (compression: 54.53\%)
984.081 total actual memory usage
```

Listing 10.2: Spin Verifier Output in the IP-RIP Example

After about 40min and requiring slightly under 1 GB of RAM, SPIN found an attack sequence (cf. Listing 10.2) of depth 21 violating the specified confidentiality property. Using SPIN's guided simulation feature and mapping the PROMELA level sequence back to cTLA level (with cTLA2PC's `--trace-points`) reveals the attack sequence depicted in Figure 10.5.

The sequence shows host H1 preparing an IP packet (action `snd_ip`) for host H2 and transmitting it to the media (`snd_h1`). As the default gateway of host H1 is router R1, the packet is received by router R1's LAN 1 interface (`rcv_r1`). Next, attacker HA prepares a RIP update packet advertising a new route to zone Z2 with metric 1 from HA (`snd_ripu`) and broadcasts it in zone Z3 (`snd_ha`). The update packet is then received (`rbc`) by router R3, which processes it (`rpcs`). Meanwhile, R1 begins to process the IP packet for H2 received from H1 (`rpcs`). Back to R3, because the advertised new route to Z2 via HA with metric 2 $=$ HA's metric $+ 1$ is better than the existing route via R2 with metric 4 (cf. Table 10.1), R3 updates its routing table (`rip_in_...`). Furthermore, because of its routing change, R3 prepares updates packets for the routers in the other zones (`rip_out`). R3 then broadcasts the triggered update packet in particular to zone ZBB13 (`snd_r3`). R1 receives (`rbc`) and processes (`rpcs`) the triggered update packet. As the advertised new route to Z2 with metric 3 via R3 is still better than the existing route with metric 4 via R2, R1 updates its routing table (`rip_in`). Then, R1 forwards (`fwd`) the packet from H1 according to the just changed table (next-hop R3) and sends it (`snd`). R3 receives the packet (`rcv_r3`), processes it (`rpcs`), and sets the forward next-hop to the attacker, HA (`fwd`). Then, R3 transmits the packet (`snd_r3`) to the media and HA receives it (`rcv_ha`). Thus HA receives a packet from H1 to H2, which violates

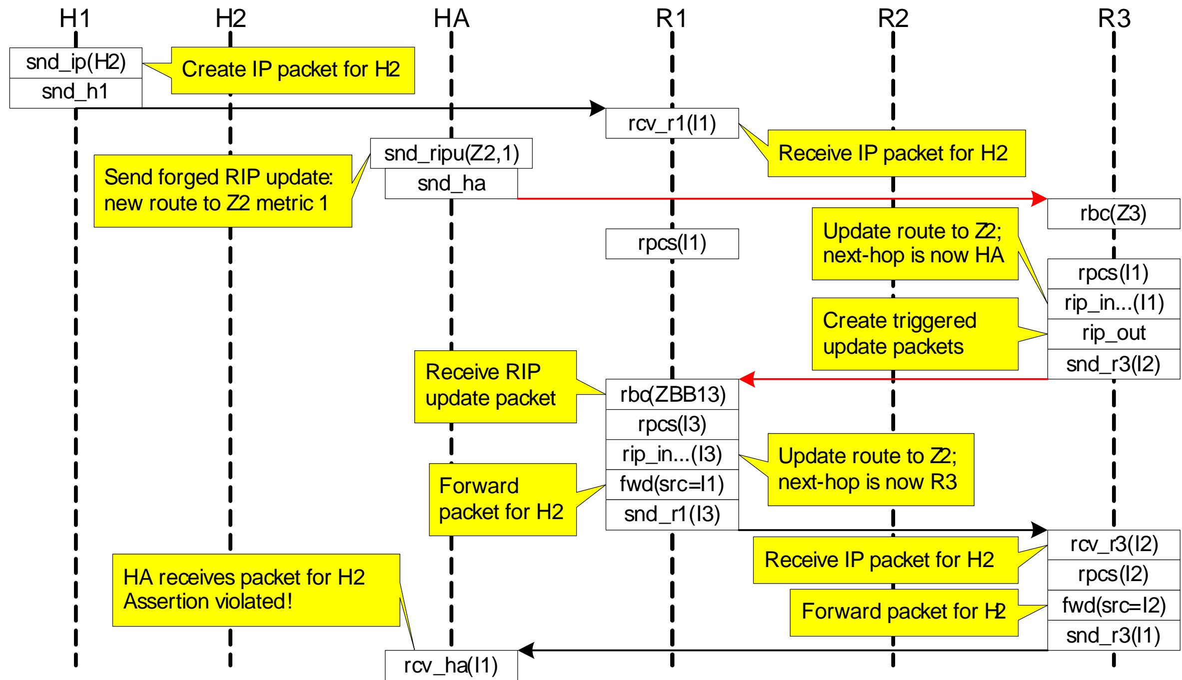


Figure 10.5: Example Attack Sequence in the IP-RIP model

the assertion.

Because of the breadth first search the described sequence is minimal. Thus, only necessary steps are included. Further variants are possible. For example, R3 will usually broadcast the triggered update packet to R2 as well. As this step is not required for the violation of the stated confidentiality property, it is not included in the 21 step sequence.

10.4 Discussion

In this section we briefly recapitulate some of the key points of the IP-RIP scenario.

Modeling & Analysis

- The IP-RIP scenario (cf. Fig. 10.1) involving multiple networks, RIP routers, and TCP/IP hosts can be modeled based on our framework in an integrated and object-oriented way using cTLA. During the modeling phase we enhanced the framework with the `RouterIpNode` process type and derived the specific process type `RipRouterIpNode` for this scenario.
- The size of the cTLA model file is about 30 KB (without comments). After translation to PROMELA with cTLA2PC, the model file has a size of about 60

KB and a SPIN state vector of about 720 bytes. Automatic analysis with SPIN can be tackled after insertion of the security property.

- Optimizations (cf. chapter 8) make a big difference in the IP-RIP scenario, too. The state vector is reduced from its initial size of about 720 Bytes to about 320 bytes. While contributing to the state vector reduction by only a small amount, the Unroll Action Parameters optimization proved particularly helpful (and was invented) during analysis of the IP-RIP scenario. The Unroll Actions optimization significantly reduces the number of transitions and the search depth required for the scenario.

Attack Sequences

- Realistic attack sequences can automatically be found by using SPIN to analyze the scenario model. The attack sequence discovered corresponds nicely to the routing and tunneling protocol attack ideas from [BHE01].

- Attack sequences depend on the attacker position, update propagation, and initial routing tables which in turn depend on low-level network topology aspects.

  For example, an attack sequence like the one shown in Figure 10.5 is not possible in the same way if H2 is connected directly to R2. First, the initial routing tables have different metrics. Second, the metric of the forged RIP update packet from HA is increased by each router on its way to R3. Thus, the new route would not be better than the existing route, and R3's routing table would not be updated. Using forged source IP addresses for RIP update packets to pretend letting the next-hop of a route send the update, circumvents this "topology protection".

- Practical tools to facilitate attacks on RIP via packet injection are widely available, e.g., `nemesis-rip` [NS04]. Some knowledge of RIP and the network topology is still required, however.

Improving Security

- The main security weakness of RIP is the missing authentication so that injected packets can effect the routing easily. Thus, RIP v2 [Mal98] adds password authentication. As the authentication mechanisms are incompletely specified in the RFC, most implementations (e.g., Microsoft [Cor02, p. 78]) for compatibility reasons support *cleartext* authentication only. This leaves the doors for attackers wide open.

- Cryptographically secured variants of RIP (e.g., S-RIP [WKvO04]) have been suggested but are rarely deployed. Their widespread deployment is hindered by the usually required computationally expensive cryptographic operations. Performance is a very important consideration in routing. Furthermore, interoperability is a key requirement in heterogeneous networks, but standardization of new protocols is a lengthy process.
- Deployment of an IPsec VPN between the routers (port 520) helps to protect from injection attacks by *external* nodes. Nodes regularly involved in the routing process, however, are still able to inject packets at will. Furthermore, IPsec is a very complex protocol. Thus, it is both very hard to build a secure IPsec implementation and to manage and configure a running system correctly [FS99].
- Of course, RIP should be blocked at the border routers. It is an *interior* gateway protocol.

11 Case Study: IP-OSPF

This chapter describes the modeling and analysis of the IP-OSPF scenario using our approach (cf. chapter 4). We begin with a short introduction of the scenario followed by the key modeling steps, including the different views taken. Then, we deal with the security property for automated analysis, optimizations taken and analysis results. Due to the complexity of the scenario, we reach the limits of the SPIN and GCC tools. Thus, we take a somewhat different approach regarding optimizations than in the previous scenarios. The chapter concludes with the key points learned from the application of our approach to the IP-OSPF scenario.

11.1 Introduction

In the IP-OSPF scenario [Kon05], we consider four networks N1, N2, N3, N4 connected by routers as depicted in Figure 11.1. The hosts in the networks run TCP/IP, the routers run the *open shortest path first (OSPF)* routing protocol.

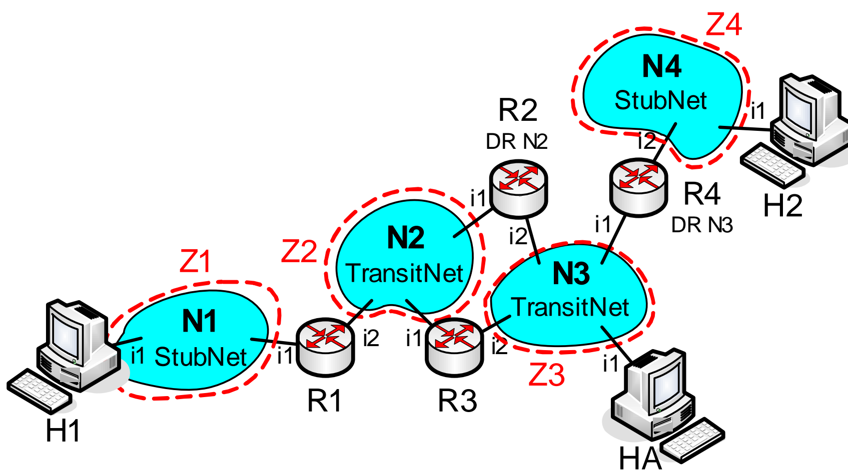


Figure 11.1: IP-OSPF Scenario

Classification OSPF belongs to the interior gateway routing protocols, i.e., it is used for routing inside an autonomous system (cf. section 7.2.2), and is the most

popular protocol based on link-state routing. In a *link-state* routing protocol, each router maintains a *link-state database (LSDB)* with complete topology information (i.e., networks, routers and links) for the autonomous system (or area). More precisely, each router sends updates of the form "I have these links to N1, N2, ...; their cost is c1, c2, ...". This is in contrast to distance-vector routing protocols like RIP (cf. section 10.1), where each router maintains only partial topology information (i.e., next-hop and distance) for each destination.

Attack Ideas & Tools As in the case of RIP (cf. chapter 10), the attack ideas described in section 7.2.3 are applicable. Tools like `nemesis-ospf` from the Nemesis project [NS04] create any OSPF packet, particularly update packets. All values on the OPSF (e.g., sequence number, age, netmask etc) and IP level (e.g., source and destination address) can be chosen arbitrarily. Thus, an attacker may inject packets from, say, a compromised router. As the routers generally have complete topology information and each router calculates its own routing table, it is more difficult to effect OSPF routing than RIP routing (cf. chapter 10). Some specific attack ideas for OSPF, e.g., MaxAge and Seq++, have been described by one of the inventors of OSPF [JlM04] as well as by the authors of the JiNao IDS [JGS+00]. However, these descriptions remain vague and informal. To the best of our knowledge, no formal modeling and analysis of OSPF network scenarios has been published before our work.

11.2 Modeling

We base our modeling on the framework (cf. chapter 7). As in the previous chapters, we take a protocol, node, and network view at the scenario. By integrating the views and composing the model's elements, we then proceed to a system model for the scenario.

11.2.1 Protocol View

In contrast to RIP, OSPF does not use a transport layer protocol to distribute its information. Instead, it encapsulates its packets directly on the internet layer, using its own IP protocol identifier. Thus, inclusion of a transport layer is not required for the IP-OSPF model. As OSPF makes use of the internet layer, it is usually classified as part of the transport layer (cf. Fig. 11.2).

The framework's process types `HostIpNode` and `RouterIpNode` already supply adequate internet and network interface layers. Thus, it suffices to extend them with OSPF on the transport layer.

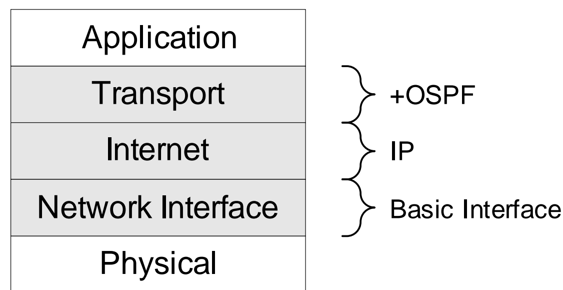


Figure 11.2: Layers and Protocols in the IP-OSPF Scenario

11.2.1.1 Transport Layer – Open Shortest Path First (OSPF)

OSPF is quite a complicated protocol with several different message types and subtypes, options, and additional fields. Thus, we provide a simplified high-level view of OSPF here; for details we refer to RFC 2328 [Moy98] and [Per99].

OSPF message types The OSPF message types are:

- *Hello message*: Hello messages are used after router boot-up, to find out neighboring routers.
- *Database description message*: Database description message are used to initialize the LSDB of a router from neighboring routers.
- *Link-state update message*: Link-state update messages are the key message type in OSPF. They are used to distribute topology information.
- *Link-state request message*: With a link-state request message, specific topology information can be requested from a router.
- *Link-state acknowledgment message*: Link-state acknowledgment messages are used for the reliable transfer of link-state update messages.

For our modeling, we begin with the steady phase with a full neighbor data structure and initialized interfaces, not on the initialization and neighbor discovery phase of the routers. Furthermore, we assume a reliable network where message do not get lost. Thus, it suffices to consider *link-state update* type messages.

Link-State Update (LSU) Message Each link-state update message contains one or more *link-state advertisements (LSA)*. The most important LSA types are:

- *Router-LSA*: LSA describing a router's link.

- *Network-LSA*: LSA describing a broadcast network. It lists the netmask and the routers of the network. This type of LSA is only created if a designated router exists for the network.

- *Summary-LSA*: LSA summarizing an area. This type of LSA is only created if OSPF is used with a hierarchical topology. In this case it is send by an area border router.

Each LSA type has different attributes, however, the `age`, `sequence number`, and `advertising router` attributes are contained in all types. While `age` and `sequence number` attributes are used for aging of LSAs, `advertising router` contains the router that originally created the LSA.

For our modeling, without loss of generality, we allow only one LSA per LSU message. If a LSU message contains multiple LSAs, it is equivalent to multiple LSU messages with one LSA each.

Distribution of LSU Messages LSU Messages are distributed both periodically and due to changes in the topology. Periodic distribution makes use of the `age` and `sequence number` attributes. At the time an LSA is distributed, its `age` is set to zero. After a certain time the LSA has to be redistributed; then, the `sequence number` attribute is increased by one and the `age` attribute is reset to zero.

A reliable *flooding* algorithm is used for distribution of LSU messages. That means that the LSU messages are sent to all neighboring routers until reception is acknowledged (by a link-state acknowledgment message).

Calculation of the Routing Table Each router calculates a shortest path tree (SPT), using Dijkstra's algorithm, from its LSDB. The SPT contains the shortest path from the router to all destinations (networks). A route is defined by the next-hop for a destination. This information follows directly from the SPT.

Designated Routers During OSPF initialization, designated routers are selected for networks with three or more routers (so called *multi-access networks*). A *designated router* for a network provides concise information about the network through network-LSAs, thus reducing the amount of routing information exchanged.

For the scenario considered here (cf. Fig. 11.1), we assume the routers R2 and R4 to be the designated routers for networks N2 respectively N3.

Hierarchical Routing To reduce the amount of topology information exchanged, OSPF allows to divide autonomous systems into areas. With the division of the autonomous system into areas, LSDBs contain information for one area only. Selected routers, called *area border routers*, however, contain LSDBs for multiple areas. Most

LSA types (with the exception of summary-LSAs) are not flooded over area borders. Thus, essentially, each *area* is treated like its own independent *autonomous system*.

For the scenario considered in this chapter, we do not use hierarchical routing (i.e., a single area covers the whole autonomous system).

Process type OSPFRouter We implement OSPF with a new process type `OSPFRouter`, which is based on `RouterIpNode`. Figure 11.3 shows the structure of process type `OSPFRouter`. External actions (e.g., `rcv`) are connected to red dots on the edge of process type `OSPFRouter`, symbolizing interaction with external elements; internal actions (e.g., `calcRT`) are only connected to internal elements.

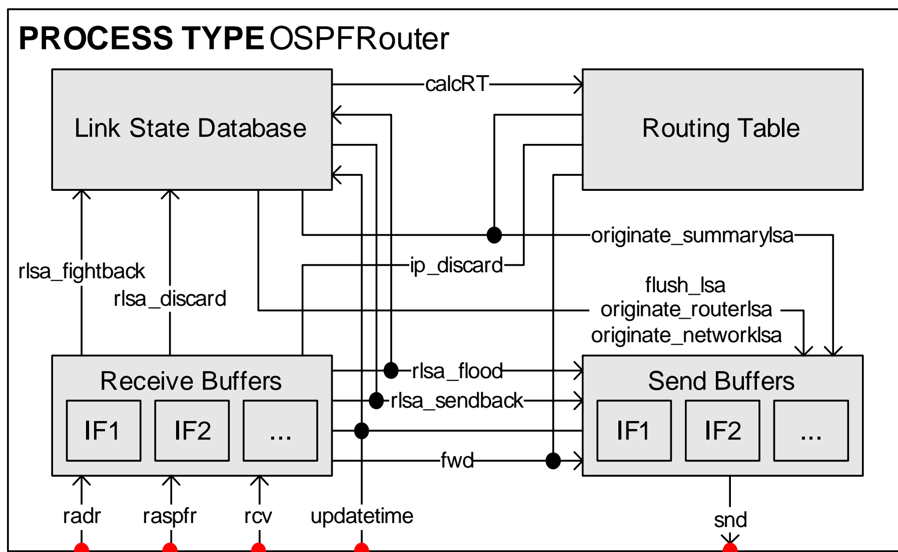


Figure 11.3: Process Type OSPFRouter

Besides send and receive buffers of the interfaces (which are inherited from process type `RouterIpNode`); process type `OSPFRouter` contains data structures for representing the LSDB and the routing table. The action `calcRT` calculates the routing table based on the data contained in the LSDB. Due to efficiency reasons, action `calcRT` is implemented in PROMELA. Only a stub for action `calcRT` is contained in the cTLA model, which is replaced by the PROMELA implementation after translation.

LSAs are received either by the action `radr` – if the router is a designated router – or by the action `raspfr` – for a non-designated router. In any case, they are stored in the interface's buffer and handled by further processing actions. These actions closely follow the different cases described in RFC 2328 [Moy98] for the flooding of received LSAs:

- `rlsa_flood`: Action for flooding of the LSA to all neighboring routers except the advertising router.
- `rlsa_fightback`: The router received an LSA which is marked as newer than an existing LSA and for which it is the advertising router. Thus, this router will "fight-back" by removing the LSA and redistributing its own LSA.
- `rlsa_sendback`: The router received an LSA for which it already has a newer version in its LSDB. Thus, this router sends the new version of the LSA back to its originator.

LSAs created by these actions are put into the send buffers of the respective interfaces and can then be sent using action `snd`. We simplify the flooding algorithm by assuming a reliable transport medium. Thus, we do not have to wait for acknowledgments and do not have to retransmit LSAs.

But LSAs are not only created in reaction to incoming LSAs. After a link-state change or refresh time expiry, new LSAs have to be prepared as well. These cases are handled by the `orginate_*` actions according to the role of the router. Each router sends router LSAs, designated routers and border routers additionally send network and summary LSAs. Aging of LSAs is implemented through the `updatetime` action, which affects all LSAs contained in the local LSDB.

The process type `TimedMedia` is derived from the framework's `Media` process type. It adds a constraint for action `updatetime`, which allows aging only if no LSAs are currently in transit. This is a simplification that helps to cut down on the variations between system runs and is especially helpful during validation.

Validation We validated the OSPF modeling described above by checking both typical and random sequences. For the random sequences, we made use of SPIN's random simulation mode. Beforehand, each action was instrumented with a special debugging code outputting action name, parameters, and values calculated (e.g., routing table entries). The resulting logs greatly helped to fix subtle errors and make sure the behavior of the modeling matches the RFC. For validating the flooding process, sorting system runs by LSA age have proved useful.

11.2.2 Node View

In the IP-OSPF model, all routers are instances of the `OSPFRouter` process type. This process type is capable of both representing designated routers and non-designated routers. The hosts are modeled with instances derived from process type `HostIpNode`.

11.2.2.1 Representative Nodes & Role Assignment

The set of nodes (R1, R2, R3, R4, H1, H2, HA) contained in the OSPF scenario (cf. Fig. 11.1) is already quite minimal. Designated routers occur within multi-access networks. Thus, with two multi-access networks, and two stub networks, four routers is the minimum number for the architecture to make sense. Furthermore, for two communicating hosts and an independent attacker, three host nodes are required. Thus, for this scenario, there is no point in selecting a smaller subset of nodes as representative nodes.

The roles to assign for this scenario are designated router, non-designated router, sender, receiver, passive attacker and active attacker. Active attacker and passive attacker are collaborating. While the former one actively interferes with the routing protocols, the later one is the passive beneficiary of the former one's actions.

Due to the topology dependencies already described, R2 and R4 are designated routers, R1 and R3 are non-designated routers. The host roles are assigned as follows: H1 $\mapsto$ receiver, H2 $\mapsto$ sender, HA $\mapsto$ attacker. Router R2 is assigned the active attacker role. Accordingly, HA is an instance of the framework (cf. chapter 7) process type `HostIpNode`, while router R2 is an instance of the specific process type `AttackerOSPFRouter`.

11.2.2.2 Attacker Action

For the attacker, a specific process type, `AttackerOSPFRouter`, is derived from `OSPFRouter`. It adds two actions, `attacking_rlsa_flood` and `hda_change_fwd`.

The action `attacking_rlsa_flood` interferes the flooding process by invalidating the current LSA. This invalidation can be done in different ways. We consider the attacks suggested by the authors of the JiNao IDS [JGS+00]. For example, the MaxAge attack, which is also described in [JlM04], works by setting the `age` field to a special value which causes the LSA to be flushed from all routers involved in the flooding process. After the MaxAge LSA is recognized by the router owning the LSA, it will "fight back" and issue a new LSA with age 0.

The action `hda_change_fwd` models the collaboration of the attacker router with the attacker host. It forwards a received packet to the attacker host. This way, if the attacker router is able to interfere with the routing process, the attacker host can benefit as well.

11.2.2.3 Node Initialization

As in the previous examples, the send, receive, and zone buffers, are set to empty. The interfaces of all nodes are assigned addresses according to `fSrcToIa` from the framework.

Furthermore, as stated above, our modeling begins with the steady phase and a full neighbor data structure. Thus, the routing table and LSDB have to be initialized for all router nodes. Both routing table and LSDB are initialized via functions. The functions for initializing the routing table have the form `fIniSrcRtIToX(n,r)`, where n is a node identifier and r is the row of routing table from which value X is to be retrieved. For example, `fIniSrcRtIToNextHopIf(R1_ID,0)` gives the initial interface ID of the next hop of the first row in router R1's routing table.

Similarly, the functions for initializing the LSDB are of the form `fIniSrc-LsdbIToY(n,r)`. Thus, function `fIniSrcLsdbIToAge(R1_ID,0)` returns the initial age (=`0`) of the first LSDB entry of router R1. A more detailed example of the contents of a routing table is given in the following section.

11.2.3 Network View

The example scenario consists of four networks (N1, N2, N3, N4) connected by four OSPF routers (R1, R2, R3, R4) (cf. Fig. 11.1). This is modeled by a `TimedMedia` instance with four zones Z1, Z2, Z3, Z4, where Z_i corresponds to network N_i. With the topology function `fSrcToZone`, the interfaces of the nodes are assigned to the appropriate zones (e.g., interface 2 of router 1 is mapped to zone Z2).

For each router, the routing table entries have the form (desttype, area, pathtype, dest, nexthop-iid, nexthop-ia, dist). As this IP-OSPF scenario uses no hierarchical routing (i.e., a single area), and only network links, the area (`0`), pathtype, and desttype (=`NETWORK_DT`) entries are constant. In the steady phase, the entries (dest, nexthop-iid, nexthop-ia, dist) correspond to the network topology of the scenario. For example, router R1 choses router R3 (interface 1) as next-hop for packets destined for network 3, and the distance (cost) of this route is 2 (cf. Table 11.1).

dest	**nexthop-iid**	**nexthop-ia**	**dist**
N1	I1	– (direct)	1
N2	I2	– (direct)	1
N3	I2	R3_I1_IA	2
N4	I2	R3_I1_IA	3

Table 11.1: Initial routing table of R1

In the actual implementation for this scenario, the dest field is for efficiency reasons not a part of the routing table but provided via the function `fInaToRti` which maps from a destination to a corresponding row of the routing table.

11.2.4 System Composition

The cTLA system for the IP-OSPF scenario is composed as depicted in Figure 11.4 (without internal actions).

Processes The system is instantiated from the process type `OspfExample`, which contains a `TimedMedia` instance representing the networks, four instances derived from `OspfRouter` (one of these an `AttackerOspfRouter` instance) for the routers, and three instances of `HostIpNode` (two of these `ActiveHostIpNode` instances) for the nodes.

Of these process types, `OspfRouter`, `OspfAttackerRouter`, and `TimedMedia`, are specific to the IP-OSPF scenario (cf. Fig. 11.5). They are, however, derived from the framework's process types `RouterIpNode` and `Media`.

System Actions The collaboration in the system is determined through the system actions which couple the instances' actions. In addition to the send and receive classes for system actions known from the previous scenarios (cf. chapter 9, 10), there is another class for advancing time made up of the system action `updatetime` (i.e., aging the LSAs). The send system actions are subdivided into system actions for routers (`rXsnd`) and hosts (`hYsnd`) due to an additional parameter specifying the interface for the (multi-homed) routers.

For the receive system actions, we set up different subclasses for reception of generic packets vs. LSA messages. As before, the receive actions for generic packets distinguish between routers (`rXrcv`) and hosts (`hYrcv`). LSA messages are processed only by routers. We distinguish between LSA messages received by the designated routers R2, R4 (`nNradr`, where N specifies the network N2, N3 respectively) and all other routers (`nNraspfr`, where $N \in \{\mathrm{N2}, \mathrm{N3}\}$). Only these actions, which are possible due to the assigned roles (designated router or non-designated router), and the neighbor relationship are included. Furthermore, system actions which are not relevant with respect to the scenario (e.g., broadcast receive, because the routers use multicast) are removed.

11.3 Analysis

Here, we describe the security property to be checked during automated analysis, the optimizations taken, and finally the results of the automated analysis with SPIN. Automated analysis for the IP-OSPF scenario was especially difficult because we reached the limits of the SPIN and GCC tool chain. This problem will be covered in the optimizations section.

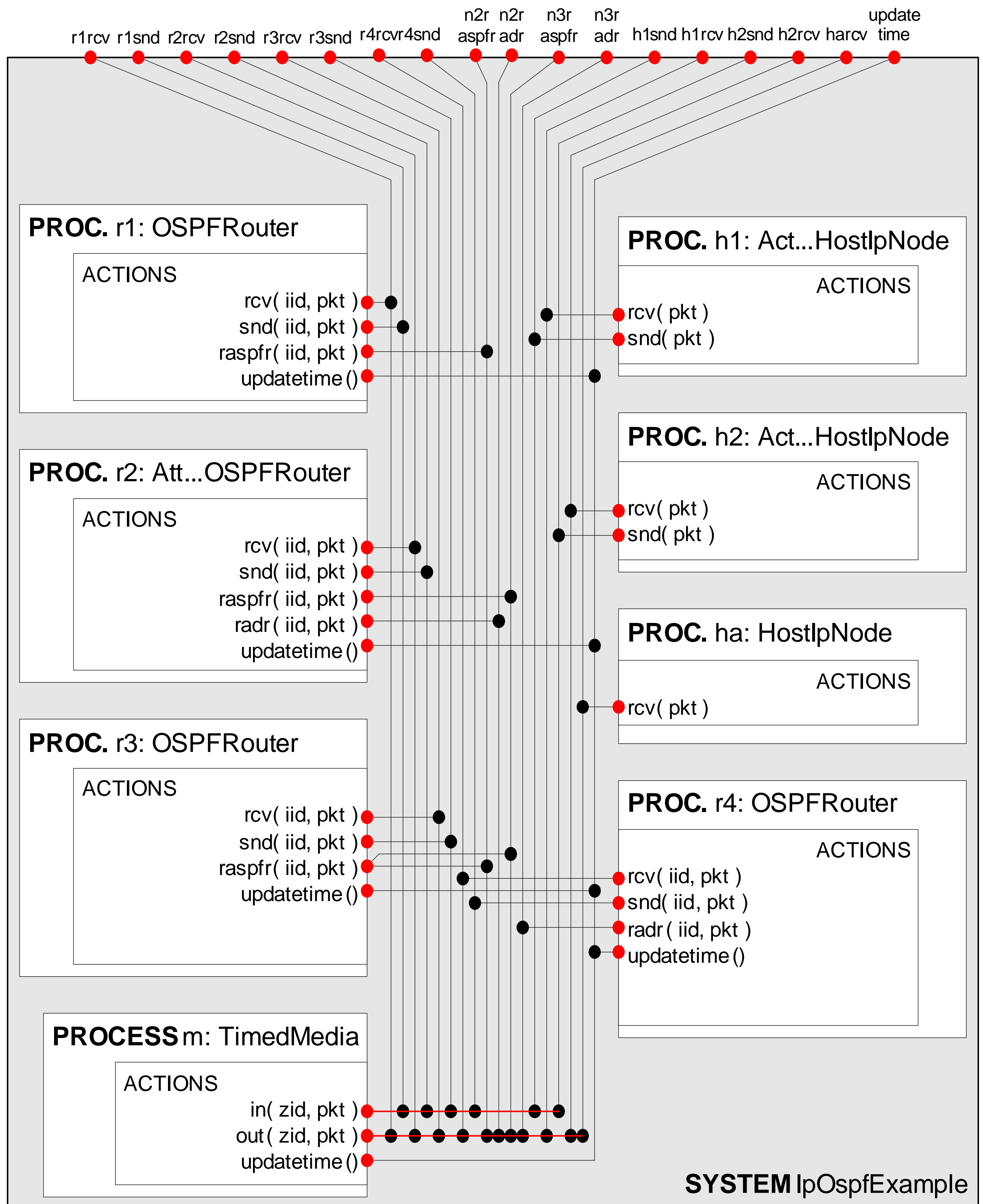


Figure 11.4: Compositional Structure of the IP-OSPF Model

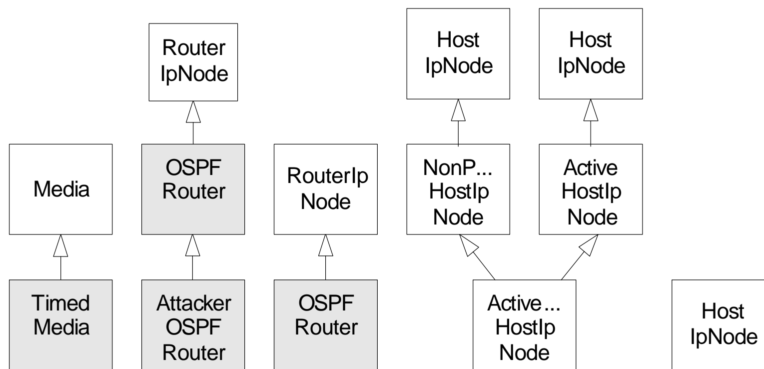


Figure 11.5: Framework and Specific Process Types of the IP-OSPF Model

11.3.1 Security Property

As described in section 11.2.2, our modeling begins in the steady phase. The shortest route from host H1 to H2 uses the routers R1, R3, R4. In particular, packets from host H1 (role sender) to H2 (role receiver) are not transmitted over router R2 (role attacker). Thus, the attacker host HA cannot receive packets from H1 over the collaborating router R2.

We capture this behavior with a security property for host HA. The security property states that, if host HA receives a non-broadcast packet, it has to be destined for HA. In our modeling, this property can be formalized easily using SPIN's assert statement (cf. Listing 11.1). The assert statement requires that a packet which is in host HA's receive buffer has a (final) destination address equal to host HA's address.

```
assert( (ha_itf.rpa.pkt.dat_ida == HA_I1_IA) );
```

Listing 11.1: Assertion in the IP-OSPF Example

As we insert the assertion into the non-broadcast receive action of host HA (after translation of the model to PROMELA), it is not triggered by broadcast packets.

11.3.2 Optimizations

In the following paragraphs we describe the optimizations we applied to the IP-OSPF model in order to be able to conduct automated analysis. Table 11.2 outlines the order of application and the state-vector size resulting from the optimizations applied so far.

Due to its complexity, the IP-OSPF scenario reaches the limits of the GCC tool. We had to spend a lot of effort to transform the model in such a way that the SPIN and

Optimization	State Vector
(initial version)	$\sim$ 1080 Bytes
+ Paramodulation	$\sim$ 990 Bytes
+ Unroll action parameters	$\sim$ 980 Bytes

Table 11.2: Optimization Effects on State-Vector Size in the IP-OSPF Example

GCC tool chain could be used at all. Thus, in contrast to the previous scenarios, the focus of this section lays more on finding ways to work around these limitations than on reducing the state-vector size.

11.3.2.1 Initial Version

For the network architecture to make sense, the nodes contained in the scenario are required (cf. section 11.2.2.1). Thus, we do not select a smaller set of representative nodes for this scenario.

We already assigned roles like attacker, sender, receiver etc. Through this assignment and the intelligent definition of the system actions, we can save on actions executable at each step. For example, as mentioned above, broadcast receive actions are not required for this scenario. The state vector size of this initial version is about 1080 Bytes.

11.3.2.2 Action Parameter Paramodulation

Due to the system action composition (cf. section 11.2.4) in the IP-OSPF scenario, packet related parameters are typically input parameters for one instance's action coupled with another instance's action, where they are output parameters. Thus, paramodulation can be successfully applied in the IP-OSPF scenario, too. By translating our cTLA model with the `--paramod` switch, the state-vector shrinks by about 9% to about 990 Bytes.

11.3.2.3 Unroll Action Parameters

The unroll actions approach already proved useful in the IP-RIP scenario (cf. chapter 10). We apply this approach to the IP-OSPF cTLA model as well. Thus, both the parameter setting input generator and the corresponding parameters are removed which mainly saves on transitions (search depth). The state vector size is slightly reduced to about 980 Bytes. Unfortunately, the PROMELA model file size grows strongly.

The size of the IP-OSPF cTLA model file is about 100 KB (cf. Table 11.3). Without use of the unroll actions approach (variant 1), the PROMELA model file has a size of

about 500 KB after translation. This corresponds to a size blow-up factor of 5 between CTLA and PROMELA. This demonstrates the efficiency of the compositional and object-oriented modeling of CTLA in comparison with PROMELA.

With the application of the unroll actions approach (variant 2), however, the PROMELA model file's size is about 875 KB, i.e., we experience a nearly 9-fold size blow-up. The increased blow-up is due to the copying of actions involved in the unroll approach.

Model	Variant 1	Variant 2	Variant 3
cTLA	∼100 KB (without comments)		
Promela	∼ 500 KB	∼ 875 KB	∼ 780 KB
C (Verifier)	∼ 8,060 KB	∼ 11,320 KB	∼ 3,160 KB
Executable	∼ 3,920 KB	– (failed)	∼ 870 KB

Table 11.3: IP-OSPF Model File Size Comparison

In both cases, SPIN is used to generate a C representation of the model integrated with the procedures required to check the security property, i.e., a verifier. Again, this translation leads to a size blow-up. After translation to C the size of the model files is about 8,060 KB for variant 1 and 11,320 KB for variant 2. Thus, the size blow-up from PROMELA to C is even larger, with a factor of about 16 for variant 1 and 13 for variant 2.

Unfortunately, for variant 2, the GCC compiler required for creating an executable verifier hangs during translation of the verifier C code. Thus, analysis of the model variant with applied unroll actions approach is not possible with the SPIN and GCC tool chain at all.

This forced us to consider new directions for optimizations. Due to the limitations of the SPIN and GCC tool chain, we have to focus on reducing the C model size. Further analysis shows that the `pan.m` file generated by SPIN is the main contributor to the size of the C model files. In this file, model transitions are contained. In particular, their guards involve large expressions due to nested function calls and predicates (e.g., for topology and routing table access).

11.3.2.4 Reduce Nesting Depth of Guard Expressions

Functions in CTLA are defined through value tables (cf. section 5.2.3). As described in section 6.2.2.4, CTLA functions are translated to PROMELA using inline macros containing guard and effect statements. Thus, particularly nested function calls can lead to a PROMELA model with very large guard expressions (cf. section 8.5.2). Such an expression occurs e.g., in the action `fwd` of process type `OSPFRouter` of the IP-OSPF model. Ultimately, the well-known C compiler GCC hangs during translation of the verifier created by SPIN.

To solve this problem, we developed the reduce nesting depth of guard expressions optimization (cf. section 8.5.2). It simplifies nested guard expressions involving functions with the help of temporary variables and possible reordering. This way, a smaller expanded PROMELA and C verifier version of the model (cf. Table 11.3, variant 3) is created. The size blow-up from the PROMELA to the C version of the model is greatly reduced. While the previous blow-up was about factor 13, it is only factor 4 with the function nesting reduction. Accordingly, the verifier source code has a size of about 3,160 KB (instead of 11,320 KB).

11.3.3 Results

After application of the optimizations described above, especially the reduce function nesting optimization, an executable verifier can be built by SPIN and GCC. The state vector size, however, is still about 980 Bytes due to the complexity of the IP-OSPF scenario. Verification attempts in exhaustive mode (i.e., full state space search) run out of memory quickly.

Therefore, in contrast to the previous scenarios, we recompile the verifier with the option for bit-state hashing mode (`-DBITSTATE`). This time, execution of the verifier detected a violation of the specified security property in the IP-OSPF model (cf. Listing 11.2).

The analysis required about 1,1GB of RAM and took 8 minutes using SPIN 4.2.5 on a Linux machine with two 3,1 GHz Xeon CPUs. From the generated trail file (`OspfASExample.promela.trail`) with SPIN's guided simulation mode, the violating sequence can be obtained on the PROMELA level. The mapping of the sequence back to the level of CTLA actions is easy thanks to CTLA2PC's option `--trace-points`. This reveals the attack sequence depicted in Figure 11.6.

First, router R3 creates a new router LSA (`originate_routerlsa`) that is sent to and received by R2, the designated router of network N2 (`r1snd`, `r2radr`). This router is compromised and uses a MaxAge attack to invalidate the LSA (`r2.attacker_rlsa_flood`). This LSA is then sent to R4, the designated router of network N3 (`r2snd`, `r4radr`) and flooded in network N2 (`r2snd`, `r1raspfr`, `r3raspfr`). Router R1 processes the LSA (`r1.rlsa_flood`) and recalculates its routing table (`r1.calcRT`), thereby changing its next-hop for network N2 from R3 to R2. Next, host H1 prepares and sends an IP packet to host H2 (`h1.snd_ip`, `h1snd`). The packet is received by R1 (`r1rcv`) and – due to the recent routing table change – routed to R2 (`r1.fwd`, `r1snd`). Router R2 receives the packet (`r2rcv`) and forwards it to the collaborating attacker host HA (`r2.hda_change_fwd`, `r2snd`). Thus, host HA can receive the packet (`harcv`) from host H1 to H2, violating the security property.

Depending on the scenario model and exact property modeling, various attack sequences have been found. For example, we also successfully examined attacks

```
pan: assertion violated (ha_itf.rpa.pkt.ida==3) (at depth 19)
pan: wrote OspfASExample.promela.trail
(Spin Version 4.2.5 -- 2 April 2005)
...
State-vector 980 byte, depth reached 19, errors: 1
2.23811e+06 states, stored
        2.23811e+06 nominal states (stored-atomic)
7.77519e+07 states, matched
7.999e+07 transitions (= stored+matched)
...
hash factor: 479.754 (best if > 100.)
bits set per state: 3 (-k3)
...
Stats on memory usage (in Megabytes):
2202.301 equivalent memory usage for states
  (stored*(State-vector + overhead))
268.435 memory used for hash array (-w30)
854.979 other (proc and chan stacks)
4.391   memory lost to fragmentation
1127.805        total actual memory usage
```

Listing 11.2: Spin Verifier Output in the IP-OSPF Example

involving multi-area (hierarchical routing) OSPF scenarios.

11.4 Discussion

Here, we give a brief outline of some of the key points encountered in the IP-OSPF scenario.

Modeling & Analysis

- We successfully modeled the IP-OSPF scenario (cf. Fig. 11.1) involving multiple networks, OSPF Routers, and TCP/IP hosts. The framework contributed fundamentally to the modeling. The main work during the modeling phase was the addition of the specific process type `OSPFRouter`, derived from `RouterIpNode`.
- Process type `OSPFRouter` can represent both designated and non-designated routers. An instance acts as either a designated or non-designated router through the system action coupling of the appropriate receive action (`raspfr` or `radr`). Alternatively, two types `DesOSPFRouter` and

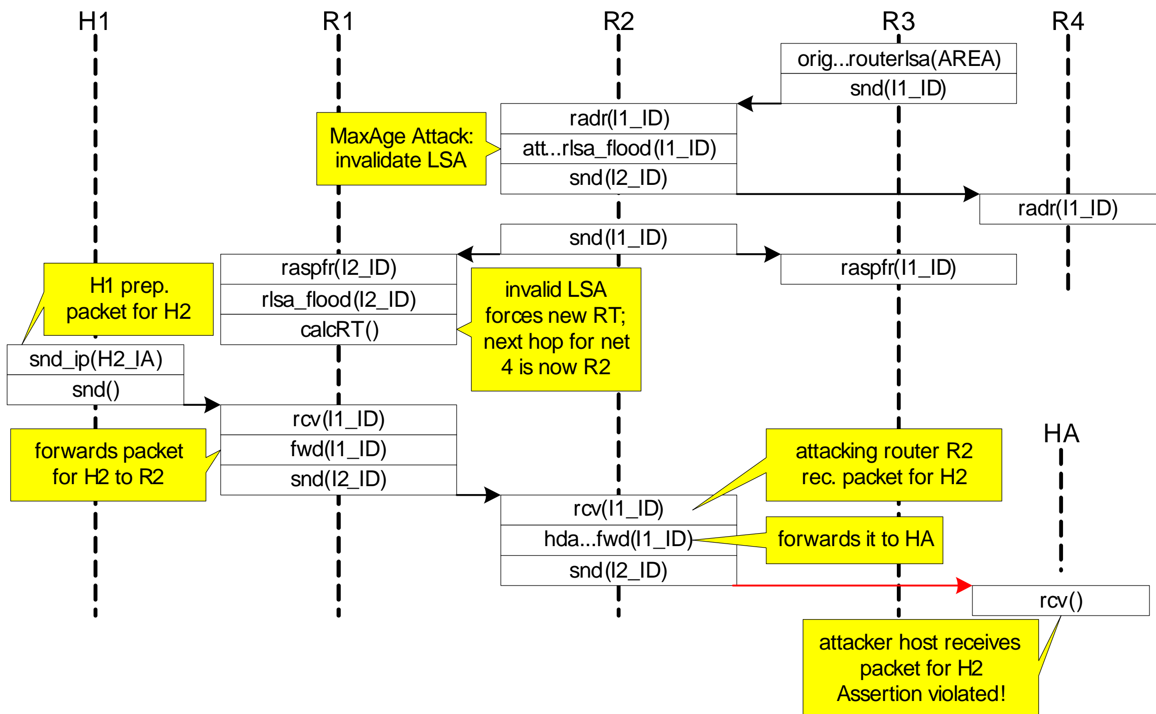


Figure 11.6: Example Attack Sequence in the OSPF model

`NonDesOSPFRouter` with only one receive action each could have been derived from `OSPFRouter`.

- Process type `OSPFRouter` supports hierarchical OSPF with multiple areas as well. Several more scenarios, including multi-area scenarios, have been modeled using `OSPFRouter` [Kon05].

- The size of the IP-OSPF CTLA model file is about 100 KB. After initial translation to PROMELA the size is about 500 KB and the SPIN state vector is about 1080 Bytes. In comparison, the IP-RIP scenario (cf. chapter 10) has a state vector of about 720 Bytes. This shows the much higher complexity of the IP-OSPF scenario.

- Optimization effects are a bit different in the IP-OSPF scenario. Paramodulation is still useful; a new problem is encountered with the Unroll Action Parameters optimization: the SPIN and GCC tool chain fails. It turns out that this is due to nested functions in guard expressions, which are inlined and expanded greatly during SPIN's preprocessing and then cause GCC to hang. After a transformation of such guard expressions, however, the SPIN and GCC tool chain works again.

- Due to a still very large state vector we employed SPIN's approximative verification mode (bitstate hashing) instead of the exhaustive mode for analysis to find attack sequences.

Attack Sequences

- Our work delivers precise attack sequences for the previously only informally described OSPF attack ideas. The attack sequence (cf. Fig. 11.6) detected by SPIN follows the MaxAge attack idea. We have also found attack sequences for other attack ideas, e.g., Seq++. To the best of our knowledge, previous to our work on OSPF, no formal modeling and analysis related to security aspects has been published. Instead, previous work chiefly deals with functional aspects and is based on techniques like simulation and test cases.
- From these sequences we can conclude the special importance of securing designated routers (i.e., routers which provide network LSAs) of a network. A compromised designated router makes it easy to affect the routing of its network.

  Furthermore, topologies with alternative ways with the same cost (like (R1,R2,R4) or (R1,R3,R4) in Figure 11.1) are more suspectible for attacks.
- Practical tools for OSPF packet injection, e.g., `nemesis-ospf` [NS04], exist. Because of the higher complexity of OSPF, these tools are more difficult to use than their RIP counterparts.

Improving Security

- In contrast to RIP, OSPF can authenticate packets based on MD5 in a standardized way. The problems of administrative and computational overhead remain, however. Furthermore, this authentication does not protect against attacks by nodes involved in the routing process (i.e., possess the key). Apart from this, problems have surfaced in the MD5 hash algorithm itself [WFLY04], deteriorating the security of any MD5 based authentication.
- Extensions to sign OSPF LSAs cryptographically have been suggested by Murphy et al [MBW97]. Again, they do not prevent attacks by nodes involved in the routing process [Eti00]. Furthermore, these extensions have not found widespread acceptance.
- OSPF is self-stabilizing because of the periodic flooding of routing information and fight-back by the advertising router. For an attack to be effective for more than a short period of time the attacker has to inject bad packets continuously. Of course, continuous attacks can be detected much better. Thus, OSPF is inherently more secure than RIP.

- With JiNao an experimental IDS for preventing OSPF attacks exists.
- Like RIP, OSPF can be protected by attacks from nodes external to the routing process using an IPsec VPN. Again OSPF should be blocked at the border routers.

12 Conclusion

In this concluding chapter, the main contributions of the thesis are summarized and directions for further work are suggested.

12.1 Summary of Contributions

Existing approaches related to computer network security are typically focused on one of the protocol, node, or network views. This thesis introduced the first approach that integrates these three views of computer network attack scenarios on a medium level of detail, allowing for consistent formal modeling and automated analysis of rich, dynamic, multi-view computer network attack scenarios. In more detail, the approach delivers:

- a workflow and modeling steps for applying the approach
- cTLA 2003, an expressive, executable, object-oriented, formal modeling language based on cTLA
- a modeling framework for the domain of integrated, dynamic computer network attack scenarios
- a scheme for translating cTLA 2003 specifications to PROMELA
- model optimization strategies at different levels to prevent or ease state-space explosion effects and enable successful analysis
- cTLA2PC, a compiler tool implementing the scheme and optimizations at the cTLA and SPIN-level providing for automated translation and optimization of cTLA 2003 specifications
- automated analysis of translated models with the SPIN model checker using a rich set of security properties (all mechanisms supported by SPIN) and the mapping of the discovered attack sequences back to the level of cTLA actions

The feasibility of the approach has been demonstrated by application to several case studies:

- IP-ARP: a single network scenario involving administrative actions, several host nodes and low-level IP and ARP protocols [RPK04]
- IP-RIP: a multiple network dynamic distance-vector routing scenario involving multiple host and router nodes [RKK05]
- IP-OSPF: a multiple network and router types, dynamic routing scenario involving IP and the complex link-state routing protocol OSPF

In these scenarios, precise attack sequences were found automatically. Formerly, there were only informal attack suggestions. To the best of our knowledge, for OSPF in particular, only partial modeling and analysis related to *functional* aspects and work based on *simulation* and *test cases* has been done before. This may be due to the high complexity of OSPF; we encountered limitations of the powerful SPIN and GCC tools during the work on the IP-OSPF case study.

12.2 Future Work

While this thesis has provided new answers for formal modeling and analysis of computer network attack scenarios, rewarding areas for future work still exist. In the following, we outline these areas briefly.

Implementation By extending CTLA2PC and providing new tools, both the analysis and modeling can be further facilitated.

- Add a back-end for another model checker, e.g., TLC or SMV, to CTLA2PC. (Re-)analyze the scenario models from the case studies and evaluate the analysis possibilities (e.g., time and memory required) vs SPIN.
- Implement the reduce function nesting optimization (cf. section 8.5.2) as a switch for our regular CTLA2PC translator instead of the PERL script in order to keep a single translation and optimization tool.
- Build a set of translators for the output of network discovery and scanner tools and devise a way to automatically produce a basic CTLA model. Particularly, the model has to reflect the network topology and instantiate matching process types to cover a basic network, node, and protocol view.
- Devise a policy resolver tool to derive security properties from policies automatically. Add these properties to the analysis model to provide automated analysis without having to specify the properties.
- Implement a visualization component to directly output CTLA-level attack sequence diagrams.

Modeling Augmenting the framework with node types supporting further popular protocols allows to build models for additional scenarios quickly.

- Devise router process types supporting exterior-gateway routing protocols like the *border gateway protocol (BGP)*. This would top off the existing framework by providing process types for all three levels of the Internet routing architecture (cf. section 7.2.2).
- *Domain name system (DNS)* poisoning and denial of service attacks have gained new attention after incidents in 2005 [SAN05]. Add node process types for DNS clients and servers to be able to quickly analyze such scenarios.

New Directions Besides computer network attack models, other application domains offer interesting prospects for our approach. As the framework is specific for computer network models, a new or adapted framework has to be worked out for new application domains.

- *Mobile ad-hoc networks (MANETs)* are an emerging area based on wireless, peer-to-peer, mobile networking. They pose a set of new challenges particularly in the areas of routing and security. As our approach is expressive enough to handle highly dynamic node behavior, it is well-suited to MANETs. The framework has to be reworked significantly to allow for the wireless media, node mobility etc, however.
- *Web services* are components that collaborate and communicate over the Internet using open, XML based standards. One of the most challenging problems of web services is to ensure confidentiality of data exchanged between and processed by multiple components. As our approach is based on a full modeling language and translation to a powerful model checker, it seems quite possible to apply it to web services. For that purpose, however, a completely new framework with notions for web service components and interfaces has to be developed.

12.3 Looking Ahead

With the advancement of integrated approaches supported by powerful modeling and analysis tools, the frontiers with respect to scope and size of the considered scenarios will always be pushed further, and further. Ultimately, the vision of automated, "push button" modeling and analysis of integrated, large-scale computer network attack models comes within reach. The approach presented in this thesis together with its underlying concepts and techniques forms a foundation for future research to ultimately realize this vision.

Bibliography

[ABB$^+$05] Armando, Alessandro; Basin, David; Boichut, Yohan; Chevalier, Yannick; Compagna, Luca; Cuellar, Jorge; Hankes Drielsma, Paul; Heám, Pierre-Cyrille; Mantovani, Jacopo; Mödersheim, Sebastian; von Oheimb, David; Rusinowitch, Michaël; Santiago, Judson; Turuani, Mathieu; Viganò, Luca; Vigneron, Laurent: The AVISPA tool for the automated validation of Internet security protocols and applications. In: Etessami, Kousha; Rajamani, Sriram K., editors, *Proceedings of the 17th International Conference on Computer-Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pp. 281–285. Springer, 2005. ISBN 3-540-27231-3.

[AP93] Abbott, Mark B.; Peterson, Larry L.: Increasing network throughput by integrating protocol layers. In: *ACM Transactions on Networking*, volume 1(5):pp. 600–610, 1993. ISSN 1-063-669-2.

[AR00] Ammann, Paul; Ritchey, Ronald: Using model checking to analyze network vulnerabilities. In: *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pp. 156–165. 2000.

[AS85] Alpern, Bowen; Schneider, Fred B.: Defining liveness. In: *Information Processing Letters*, volume 21(4):pp. 181–185, 1985.

[ASU06] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006. ISBN 0-321-428-9.

[AWK02] Ammann, Paul; Wijesekera, Duminda; Kaushik, Saket: Scalable, graph-based network vulnerability analysis. In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 217–224. 2002.

[BAN90] Burrows, Michael; Abadi, Martin; Needham, Roger: A logic of authentication. In: *ACM Transactions on Computer Systems*, volume 8(1):pp. 18–36, 1990.

[BHE01] Blackhat Europe Conference: *Routing and Tunneling Protocol Attacks*, 2001. URL: `http://www.blackhat.com/html/bh-europe-01/bh-europe-01-speakers.html#FX`.

[Bol03] Bolour, Azad: *Notes on the Eclipse Plug-in architecture*, 2003. URL: `http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html`.

[BOR03] Bruschi, Danilo; Ornaghi, Alberto; Rosti, Emilia: S-ARP: a secure address resolution protocol. In: *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC'03)*, pp. 66–75. IEEE Computer Society, 2003. ISBN 0-7695-2041-3.

[CMU01] Carnegie Mellon University Model Checking Group: *The SMV System*, 2001. URL: `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[Cor02] Corporation, Microsoft: *Windows 2000 Server Internetworking Guide*. Microsoft Press, 2002. ISBN 0-7356-1797-X.

[DY83] Dolev, D.; Yao, A.: On the security of public key protocols. In: *IEEE Transactions on Information Theory*, volume 30(2):pp. 198–208, 1983.

[eEy06] eEye Digital Security: *Blink Endpoint Vulnerability Protection*, 2006. Online Document, URL: `http://www.eeye.com/html/products/Blink/features.html`.

[Eti00] Etienne, Jerome: *OSPF with digital signatures against an insider*, 2000. Online Document, URL: `http://off.net/~jme/rfc2154_rem.pdf`.

[FG05] Flanagan, Cormac; Godefroid, Patrice: Dynamic partial-order reduction for model checking software. In: *SIGPLAN Not.*, volume 40(1):pp. 110–121, 2005. ISSN 0362-1340.

[FS99] Ferguson, Niels; Schneier, Bruce: *A Cryptographic Evaluation of IPsec*. Counterpane Internet Security Inc., 1999. Online Document, URL: `http://www.schneier.com/paper-ipsec.pdf`.

[FX01] FX of Phenoelit: *Internet Routing Protocol Attack Suite (IRPAS)*, 2001. URL: `http://www.phenoelit.de/irpas`.

[GA06] Govindavajhala, Sudhakar; Appel, Andrew W.: TR-744-06: Windows access control demystified. Technical report, Princeton University, 2006.

[GCH03] Garg, Ashish; Curtis, Jeffrey; Halper, Hilary: Quantifying the financial impact of IT security breaches. In: *Information Management and Computer Security*, volume 11(2):pp. 74–83, 2003. ISSN 0968-5227.

[GHJV95] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[HC98] Harkins, D.; Carrel, D.: RFC 2409: The Internet key exchange (IKE). Technical report, Network Working Group, 1998. URL: `http://rfc.net/rfc2409.html`.

[Hed88] Hedrick, C.: RFC 1058: Routing information protocol. Technical report, Network Working Group, 1988. URL: `http://rfc.net/rfc1058.html`.

[Her98] Herrmann, Peter: *Problemnaher korrektheitssichernder Entwurf von Hochleistungsprotokollen*. Ph.D. thesis, Universität Dortmund, 1998.

[HK00] Herrmann, Peter; Krumm, Heiko: A framework for modeling transfer protocols. In: *Computer Networks*, volume 34(2):pp. 317–337, 2000.

[Hol03] Holzmann, Gerard J.: *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 1st edition, 2003.

[HP99] Holzmann, Gerard J.; Puri, Anuj: A minimized automaton representation of reachable states. In: *International Journal on Software Tools for Technology Transfer (STTT)*, volume 2(3):pp. 270–278, 1999.

[Hum00] Humble (pseudonym): *Spoofing RIP (Routing Information Protocol)*, 2000. Online Document and Source Code. URL: `http://packetstorm.linuxsecurity.com/groups/horizon/ripar.txt`.

[ISO96] International Standards Organization: *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF*, 1996.

[JGS+00] Jou, Y.; Gong, F.; Sargor, C.; Wu, X.; Wu, S.; Chang, H.; Wang, F.: Design and implementation of a scalable intrusion detection system for the protection of network infrastructure. In: *Proceedings of DARPA Information Survivability Conference and Exposition*, volume 2, pp. 69–83. 2000.

[JlM04] Jones, Emanuele; le Moigne, Olivier: *OSPF Security Vulnerability Analysis*, 2004. Expired IETF Internet Draft. URL: `http://www3.ietf.org/Proceedings/05mar/IDs/draft-ietf-rpsec-ospf-vuln-01.txt`.

[JNO03] Jajodia, Sushil; Noel, Steven; O'Berry, Brian: Topological analysis of network attack vulnerability. In: *Managing Cyber Threats: Issues, Approaches and Challenges*. Kluwer Academic Publisher, 2003.

[Kin94] Kindler, Ekkart: Safety and liveness properties: A survey. In: *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, volume 53:pp. 268–272, 1994.

[KKN+03] Kawauchi, Kiyoto; Kitazawa, Shigeki; Nakano, Hatsumi; Ohkoshi, Takehiro; Fujii, Seiji; Kawaki, Motokazu: A vulnerability assessment tool using first-order predicate logic. In: *IPSJ SIGNotes Computer SECurity*, volume 19, 2003.

[Kne04] Kneiphoff, Tobias: *Prozessunterstützung für die Sicherheits-Analyse vernetzter Systeme in der integrierten Entwicklungsumgebung Eclipse*. Master's thesis, Universität Dortmund, 2004.

[Kon05] Konetzka, Helge: *Sicherheitsanalyse komplexer Routingprozesse: Ein Spin- und Frameworkbasierter Ansatz*. Master's thesis, Universität Dortmund, 2005.

[Lam77] Lamport, Leslie: Proving the correctness of multiprocess programs. In: *IEEE Transactions on Software Engineering*, volume 3(2):pp. 125–143, 1977.

[Lam94] Lamport, Leslie: The temporal logic of actions. In: *ACM Transactions on Programming Languages and Systems*, volume 16(3):pp. 872–923, 1994.

[LBN04] Lawrence Berkeley National Laboratory: *Bro Intrusion Detection System*, 2004. URL: `http://www.bro-ids.org/`.

[Mal98] Malkin, G.: RFC 2453: RIP version 2. Technical report, Network Working Group, 1998. URL: `http://rfc.net/rfc2453.html`.

[Mal05] Malik, Marc: *Optimierte Transformation von cTLA Modellen zur Spin basierten Sicherheitsanalyse vernetzter IT Systeme*. Master's thesis, Universität Dortmund, 2005.

[MBW97] Murphy, S.; Badger, M.; Wellington, B.: RFC 2154: OSPF with digital signatures. Technical report, Network Working Group, 1997. Experimental RFC. URL: `http://rfc.net/rfc1246.html`.

[Mea96] Meadows, Catherine: The NRL protocol analyzer: An overview. In: *Journal of Logic Programming*, volume 26(2):pp. 113–131, 1996.

[Mea99] Meadows, Catherine: Analysis of the Internet key exchange protocol using the NRL protocol analyzer. In: *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pp. 216–231. 1999.

[Met06] Metasplot.com: *The Metasploit Project*, 2006. URL: `http://www.metasploit.com/projects/Framework`.

[Moy98] Moy, J.: RFC 2328: OSPF version 2. Technical report, Network Working Group, 1998. URL: http://rfc.net/rfc2328.html.

[Nes06] Nessus.org: *Nessus Frequently Asked Questions (FAQ)*, 2006. URL: http://www.nessus.org/plugins/index.php?view=faq.

[NS04] Nathan, Jeff; Stone, Rob J.: *The Nemesis Project, version 1.4.*, 2004. URL: http://sourceforge.net/projects/nemesis.

[OGA05] Ou, Xinming; Govindavajhala, Sudhakar; Appel, Andrew W.: MulVAL: A logic-based network security analyzer. In: *Proceedings of the 14th USENIX Security Symposium*. 2005.

[OTI03] Object Technology International Inc.: *Eclipse Platform Technical Overview Whitepaper*, 2003. URL: http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[OV06] Ornaghi, Alberto; Valleri, Marco: *ettercap*, 2006. URL: http://ettercap.sourceforge.net/.

[Per99] Perlman, Radia: *Interconnections: Bridges and Routers*. Addison-Wesley, 2nd edition, 1999. ISBN 0-20163448-1.

[Plu82] Plummer, David C.: RFC 826: An Ethernet Address Resolution Protocol. Technical report, Network Working Group, 1982. URL: http://rfc.net/rfc826.html.

[Poh03] Pohl, Andre: *Rechnergestützte Analyse von Sicherheitsproblemen verteilter Systeme mit cTLA und Spin*. Master's thesis, Universität Dortmund, 2003.

[PQ95] Parr, T. J.; Quong, R. W.: ANTLR: A predicated-ll(k) parser generator. In: *Software – Practice and Experience*, volume 25(7):pp. 789–810, 1995.

[PYB+04] Pang, Ruoming; Yegneswaran, Vinod; Barford, Paul; Paxson, Vern; Peterson, Larry: Characteristics of Internet background radiation. In: *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, pp. 27–40. 2004.

[RK03] Rothmaier, Gerrit; Krumm, Heiko: cTLA 2003 Description. Technical report, LS4, FB Informatik, Universität Dortmund, 2003. URL: http://ls4-www.cs.uni-dortmund.de/RVS/MA/hk/cTLA2003description.pdf.

[RK05a] Rothmaier, Gerrit; Krumm, Heiko: Formale Modellierung und Analyse protokollbasierter Angriffe in TCP/IP Netzwerken am Beispiel von ARP und RIP. In: Federrath, Hannes, editor, *Sicherheit 2005 - 2. Jahrestagung des*

Fachbereiches Sicherheit der Gesellschaft für Informatik e.V. (GI), volume 62 of *Lecture Notes in Informatics*, pp. 77–88. Springer, 2005. ISBN 3-88579-391-1.

[RK05b] Rothmaier, Gerrit; Krumm, Heiko: A framework based approach for formal modeling and analysis of multi-level attacks in computer networks. In: Wang, Farn, editor, *Formal Techniques for Networked and Distributed Systems - FORTE 2005*, volume 3731 of *Lecture Notes in Computer Science*, pp. 247–260. Springer, 2005. ISBN 3-540-29189-X.

[RKK05] Rothmaier, Gerrit; Kneiphoff, Tobias; Krumm, Heiko: Using Spin and Eclipse for optimized high-level modeling and analysis of computer network attack models. In: Godefroid, Patrice, editor, *Model Checking Software*, volume 3639 of *Lecture Notes in Computer Science*, pp. 236–250. Springer, 2005. ISBN 3-540-28195-9.

[RON02] Ritchey, Ronald; O'Berry, Brian; Noel, Steven: Representing TCP/IP connectivity for topological analysis of network security. In: *Proceedings of the 18th Annual Computer Security Applications Conference*, pp. 25–31. IEEE Computer Society, 2002.

[Rot04] Rothmaier, Gerrit: *cTLA Computer Network Specification Framework*, 2004. URL: `http://ls4-www.cs.uni-dortmund.de/RVS/MA/hk/framework.html`.

[RPK04] Rothmaier, Gerrit; Pohl, Andre; Krumm, Heiko: Analyzing network management effects with Spin and cTLA. In: Cuppens, Frederic; Deswarte, Yves; Jajodia, Sushil; Wang, Lingyu, editors, *Security and Protection in Information Processing Systems, Proceedings of the IFIP 18th World Computer Congress (WCC), TC11 19th International Information Security Conference*, pp. 65–81. Kluwer Academic Publishers, 2004. ISBN 1-4020-8142-1.

[RRCQ03] Romano, Paolo; Romero, Milton; Ciciani, Bruno; Quaglia, Francesco: Validation of the sessionless mode of the HTTPR protocol. In: König, Hartmut; Heiner, Monika; Wolisz, Adam, editors, *Proceedings of the IFIP 23rd International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'03)*, volume 2767 of *Lecture Notes in Computer Science*, pp. 62–78. Springer, 2003. ISBN 3-540-20175-0.

[RS98] Ramakrishnan, C. R.; Sekar, R. C.: Model-based vulnerability analysis of computer systems. In: *Proceedings of the 2nd International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'98)*. 1998.

[RS02] Ramakrishnan, C. R.; Sekar, R. C.: Model-based analysis of configuration vulnerabilities. In: *Journal of Computer Security*, volume 10(1):pp. 189–209, 2002.

[Ruy01] Ruys, T. C.: *Towards Effective Model Checking*. Ph.D. thesis, University of Twente, 2001.

[RW69] Robinson, G.; Wos, L.: Paramodulation and theorem-proving in first-order theories with equality. In: *Proceedings of the 4th Annual Machine Intelligence Workshop*, pp. 135–150. Edinburg University Press, 1969.

[SAN05] SANS Internet Storm Center: *March 2005 DNS Poisoning Summary*, 2005. Online Document, URL: `http://isc.sans.org/presentations/dnspoisoning.php`.

[Sch00] Schneier, Bruce: *Crypto-Gram: Software Complexity and Security*, 2000. URL: `http://www.schneier.com/crypto-gram-0003.html#8`.

[SHJ+02] Sheyner, Oleg; Haines, Joshua; Jha, Somesh; Lippmann, Richard; Wing, Jeannette M.: Automated generation and analysis of attack graphs. In: *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pp. 273–284. IEEE Computer Society, 2002.

[Sno05] The Snort Project: *Snort Users Manual*, 2005. Online Document, URL: `http://www.snort.org/docs/snort_htmanuals/htmanual_2.4/rc1/`.

[Son00] Song, Dug: *dsniff Frequently Asked Questions*, 2000. URL: `http://www.monkey.org/~dugsong/dsniff/faq.html`.

[SSW+05] Sagonas, Konstantinos; Swift, Terrance; Warren, David S.; Freire, Juliana; Rao, Prasad; Cui, Baoqiu; Johnson, Ernie; de Castro, Luis; Dawson, Steve; Kifer, Michael: *The XSB System Version 2.7.1 Volume 1: Programmers Manual*, 2005. Online Document, URL: `http://xsb.sourceforge.net/manual1/manual1.pdf`.

[Svo89] Svobodova, L.: Implementing OSI systems. In: *IEEE Journal on Selected Areas in Communications*, volume 7(7):pp. 1115–1130, 1989.

[WFLY04] Wang, Xiaoyun; Feng, Dengguo; Lai, Xuejia; Yu, Hongbo: Collisions for hash functions. MD4, MD5, HAVAL-128 and RIPEMD. Technical report, Cryptology ePrint Archive 2004/199, 2004. URL: `http://eprint.iacr.org/2004/199.pdf`.

[WKvO04] Wan, T.; Kranakis, E.; van Oorschot, P. C.: S-RIP: A secure distance vector routing protocol. In: Jakobsson, Markus; Yung, Moti; Zhou, Jianying, editors, *Proceedings of Applied Cryptography and Network Security (ACNS'04)*, volume 3089 of *Lecture Notes in Computer Science*, pp. 103–119. Springer, 2004.

A cTLA 2003 Grammar

This appendix contains the EBNF grammar for cTLA 2003. For a description of the meta symbols cf. section 5.4. The start symbol for the grammar is `specification`.

```
(* Top-Level Productions *)
specification =
        [const_decl_part]
        [type_decl_part]
        [function_decl_part]
        [predicate_decl_part]
        process_type_decl_part
        system_instantiation_part
;

const_decl_part =
        "CONST" { constant_decl ";" }+
;

type_decl_part =
        "TYPE" { type_decl ";" }+
;

function_decl_part =
        { "FUNCTION" function_decl ";" }+
;

predicate_decl_part =
        { "PREDICATE" predicate_decl ";" }+
;

process_type_decl_part =
        { "PROCESS" process_decl "END" }+
;

system_instantiation_part =
        "SYSTEM" process_instantiation ";"
;
```

```
type_decl =
        type_identifier "=" type
;

function_decl =
        function_heading "::=" expression
;

predicate_decl =
        predicate_heading "::=" general_expression
;

function_heading =
        function_identifier "(" [formal_parameter_list] ")"
;

predicate_heading =
        predicate_identifier "(" [formal_parameter_list] ")"
;

(* Productions for Process Types *)

process_decl =
        process_heading
        [const_decl_part]
        [type_decl_part]
        [var_decl_part]
        ( simple_process_body
        | extending_process_body
        | subsystem_process_body )
;

simple_process_body =
        process_init ";"
        { process_action_part }+
        [ process_internal_action_part ]
;

subsystem_process_body =
        "CONTAINS"
        { process_instantiation ";" }+
        { subsystem_process_action_part }+
;
```

```
extending_process_body =
        "EXTENDS"
        process_extension
        [ process_init ";" ]
        [ process_action_part ]
        [ process_internal_action_part ]
        process_extension_part = { process_extension }+
;

process_extension =
        process_type_identifier "(" [actual_parameter_list] ")" ";"
;

process_heading =
        process_type_identifier "(" [formal_parameter_list] ")" ";"
;

var_decl_part =
        "VAR"   { var_decl ";" }+
;

var_decl =
        identifier_list ":" type
;

process_init =
        "INIT" "::=" general_expression
;

process_action_part =
        "ACTIONS" { process_action ";" }+
;

subsystem_process_action_part =
        "ACTIONS" { subsystem_process_action ";" }+
;

process_internal_action_part =
        "INTERNAL" "ACTIONS" { process_action ";" }+
;

process_action =
        action_identifier "(" [formal_parameter_list] ")"
        "::=" general_expression
```

```
;

subsystem_process_action =
        action_identifier "(" [formal_parameter_list] ")"
        "::=" action_call { "AND" action_call }
;

action_call =
        process_identifier "."
        action_identifier "(" [actual_parameter_list] ")"
;

process_instantiation =
        process_identifier ":"
        process_type_identifier "(" [actual_parameter_list] ")"
;

(* Productions for Expressions (Guards and Effects) *)

general_expression =
        m_expression { "OR" m_expression }
;

m_expression =
        l_expression { "AND" l_expression }
;

l_expression =
        relational_expression
        | quantification
        | "NOT" l_expression
;

quantification =
        universal_quantification
        | existential_quantification
;

universal_quantification =
        "FORALL"
        inner_quantification
;

existential_quantification =
```

```
        "EXISTS"
        inner_quantification
;

inner_quantification =
        bound_variable_decl { ";" bound_variable_decl }
        ( ( ":" "[" general_expression "]" )
        | quantification )
;

bound_variable_decl =
        bound_variable { "," bound_variable } bound
;

bound =
        simple_bound | range_set_bound
;

simple_bound =
        ":" scalar_type
;

range_set_bound =
        "IN" ( expression | linear_range )
;

linear_range =
        "{" expression ".." expression "}"
;

relational_expression =
        actual_value [ relational_operator actual_value ]
;

conditional_expression =
        "IF" general_expression "THEN" general_expression
        { "ELSEIF" general_expression "THEN" general_expression }
        [ "ELSE" general_expression ] "END"
;

expression =
        simple_expression
        | conditional_expression
        | updateall_expression
```

```
;

simple_expression =
        term_set_expression
        { addition_operator term_set_expression }
;

term_set_expression =
        atom{ (multiplication_operator | set_operator) atom }
;

atom =
        actual_variable
        | simple_constant_value
        | "(" general_expression ")"
        | function_predicate_call
;

function_predicate_call =
        identifier "(" [actual_parameter_list] ")"
;

actual_parameter_list =
        actual_value { "," actual_value }
;

actual_value =
        expression
        | compound_constant
;

updateall_expression =
        "UPDATEALL"
        bound_variable_decl
        { ";" bound_variable_decl } ":" "[" general_expression "]"
;

formal_parameter_list =
        formal_parameter {";" formal_parameter}
;

formal_parameter =
        identifier_list ":" simple_type
;
```

```
(* Productions for Variables *)

actual_variable =
        variable
;

variable =
        common_variable ["'"]
;

common_variable =
        var_identifier | component_variable
;

component_variable =
        var_identifier
        ( array_index [ field_selector ] | field_selector )
;

array_index =
        "[" expression "]"
;

field_selector =
        "." common_variable
;

(* Productions for Constants *)

constant_decl =
        constant_identifier "=" constant
;

constant =
        simple_constant_value | compound_constant
;

simple_constant_value =
        number | hex_number | simple_boolean
;

compound_constant =
        constant_type_identifier
```

```
        "{" inner_compound_constant "}"
;

inner_compound_constant =
        array_set_constant | structured_constant
;

array_set_constant =
        "[" unlabeled_constant_field
        {"," unlabeled_constant_field} "]"
;

structured_constant =
        "{" labeled_constant_field
        {"," labeled_constant_field} "}"
;

unlabeled_constant_field =
        unlabeled_field | inner_compound_constant
;

labeled_constant_field =
        labeled_field | inner_compound_constant
;

labeled_field =
        field_identifier "="
        ( unlabeled_field | inner_compound_constant )
;

unlabeled_field =
        simple_constant_value | identifier
;

(* Productions for Types *)

type =
        scalar_type
        | record_type
        | enumeration_type
        | array_type
        | set_type
        | set_enumeration_type
        | type_identifier
```

```
;

simple_type =
        scalar_type | type_identifier
;

record_type =
        "RECORD" { record_field ";" }+ "END"
;

record_field =
        identifier_list ":" type
;

enumeration_type =
        "(" identifier_list ")"
;

array_type =
        "ARRAY" "[" (unsigned_number | constant_identifier) "]"
        "OF" simple_type
;

(* Low-Level Productions *)

bound_variable =
        var_identifier
;

identifier_list =
        identifier { "," identifier }
;

function_identifier =
        identifier
;

predicate_identifier =
        identifier
;

system_identifier =
        identifier
;
```

```
process_type_identifier =
        identifier
;

prefix_identifier =
        identifier
;

action_identifier =
        identifier
;

process_identifier =
        identifier
;

var_identifier =
        identifier
;

type_identifier =
        identifier
;

field_identifier =
        identifier
;

constant_identifier =
        identifier
;

constant_type_identifier =
        identifier
;

variable_identifier =
        identifier
;

set_identifier =
        identifier
;
```

```
identifier =
        letter { letter | digit | underscore }
;

number =
        signed_number | unsigned number
;

signed_number =
        [sign] {digit}+
;

unsigned_number =
        {digit}+
;

hex_number =
        "0x" { digit | letter }+
;

scalar_type =
        "BOOL" | "BYTE" | "INT" | "SHORT" | "BIT"
;

addition_operator =
        "+" | "-"
;

multiplication_operator =
        "*" | "/"
;

relational_operator =
        "<" | ">" | "=" | "!=" | "<=" | ">="
;

simple_boolean =
        "TRUE" | "FALSE"
;

sign =
        "+" | "-"
;
```

```
underscore =
        "_"
;

letter =
        "A"  |  "B"  |  "C"  |  "D"  |  "E"  |  "F"  |
        "G"  |  "H"  |  "I"  |  "J"  |  "K"  |  "L"  |
        "M"  |  "N"  |  "O"  |  "P"  |  "Q"  |  "R"  |
        "S"  |  "T"  |  "U"  |  "V"  |  "W"  |  "X"  |
        "Y"  |  "Z"  |  "a"  |  "b"  |  "c"  |  "d"  |
        "e"  |  "f"  |  "g"  |  "h"  |  "i"  |  "j"  |
        "k"  |  "l"  |  "m"  |  "n"  |  "o"  |  "p"  |
        "q"  |  "r"  |  "s"  |  "t"  |  "u"  |  "v"  |
        "w"  |  "x"  |  "y"  |  "z"
;

digit =
        "0"  |  "1"  |  "2"  |  "3"  |  "4"  |  "5"  |
        "6"  |  "7"  |  "8"  |  "9"
;
```

B IP-RIP cTLA 2003 Model

This appendix contains the cTLA source code for the IP-RIP model (cf. chapter 10). For consistency, the (slightly reformatted) original source code, as developed for the earlier v1.x versions of the cTLA2PC tool, is given. Thus, the syntax differs slightly from the final cTLA 2003 syntax as described in Appendix A. Furthermore, with the earlier cTLA2PC version, the paramodulation optimization (cf. section 8.4.1) had to be done manually based on the generated flat cTLA system (cf. section 6.2.1.1). The source code shown here is taken after paramodulation, immediately before translation to PROMELA.

Using cTLA2PC version 1.x, the source is translated to PROMELA using the following command (which should be typed without the line breaks):

```
> ctla2pc.jar
 --unrollinputgen --unrollloops --trace-points
 --noifguard -r --map --comment --typecheck
 --rangedef ip-routing-w-RIP-example-veri-flat-para.rangedef
 --outfile ip-routing-w-RIP-example-veri-flat-para.promela
 ip-routing-w-RIP-example-veri-flat-para.ctla
```

To prepare the resulting PROMELA model for automatic analysis with SPIN, assertions must be inserted as described in section 10.3.1. Then, the following commands have to be entered to create the executable verifier and perform the automatic analysis:

```
> spin -a ip-routing-w-RIP-example-veri-flat-para.promela
> gcc -DBFS -DSAFETY -DNOFAIR -DMA=312 -o pan pan.c
> pan -E
```

The results of the automatic analysis are explained in section 10.3.3.

```
// ip-routing-w-RIP-example-veri-flat-para
// Flat cTLA 2003 source code for the IP-RIP model,
// after paramodulation
//
// translate with cTLAtoPC v1

// framework's and model specific constants
CONST
  MAXZONES = 6;
  MAXRTE = 6;
```

```
  MAXIFS = 3;

  RT_DEF = 0;
  DST_DEF = 0;
  NHO_DIR = 0;
  UNKNOWN_IF = 0;

  R1_I1_ID = 1;
  R1_I2_ID = 2;
  R1_I3_ID = 3;
  R2_I1_ID = 1;
  R2_I2_ID = 2;
  R2_I3_ID = 3;
  R3_I1_ID = 1;
  R3_I2_ID = 2;
  R3_I3_ID = 3;
  H1_I1_ID = 1;
  H2_I1_ID = 1;
  HA_I1_ID = 1;

  DI_ISA = 0;
  DI_IDA = 1;
  DI_RDE = 2;
  DI_RME = 3;

  MAXDTA = 4;
  ME_INF = 16;

// framework's and model specific types
/*
 Adaptions to framework's data types for this model:
 - InterfaceIdT: field usd removed
   (not required, as usd initialized to true and no action for
   deactivating an interface )
 - PacketT: fields sha,dha removed
   (HW addressing not required for this model)
*/

TYPE
  SYS_NIDS = (UNKNOWN_NODE, H1_ID, R1_ID, R2_ID, H2_ID,
    R3_ID, HA_ID);
  SYS_IAS = (INVALID_IA, BC_IA, R1_I1_IA, R1_I2_IA,
    R1_I3_IA, R2_I1_IA, R2_I2_IA, R2_I3_IA, R3_I1_IA,
    R3_I2_IA, R3_I3_IA, HA_I1_IA, H1_I1_IA, H2_I1_IA);
```

```
SYS_HAS = (UNKNOWN_HA, BC_HA);
SYS_ZNS = (UNKNOWN_ZONE, Z1_ID, Z2_ID, Z3_ID, ZBB12_ID,
  ZBB13_ID, ZBB23_ID);

NodeIdT = BYTE;
InterfaceIdT = BYTE;
IpAddressT = BYTE;
DataT = BYTE;
ZoneIdT = BYTE;
DstT = BYTE;
MetricT = BYTE;
PrT = (PT_IP, PT_RIP);
PacketT = RECORD
  pt: PrT;
  dat_isa: BYTE;
  dat_ida: BYTE;
  dat_rde: BYTE;
  dat_rme: BYTE;
END;
PacketBufT = RECORD
  usd: BOOL;
  pkt: PacketT;
END;
RpaActionT = (RPA_NONE_EMPTY, RPA_RPCS, RPA_RIPIN, RPA_FWD);
RpaSystemBufT = RECORD
  act: RpaActionT;
  pkt: PacketT;
END;
SpaActionT = (SPA_NONE_EMPTY, SPA_SPCS, SPA_SND);
SpaSystemBufT = RECORD
  act: SpaActionT;
  pkt: PacketT;
END;
InterfaceT = RECORD
  rpa: RpaSystemBufT;
  spa: SpaSystemBufT;
  ia: IpAddressT;
END;
DtyT = (DT_ZONE, DT_HOST);
RipRouEntryT = RECORD
  dty: DtyT;
  dst: DstT;
  nho: IpAddressT;
  met: MetricT;
```

```
    rto: InterfaceIdT;
    rcf: BOOL;
  END;
  RouTableT = RECORD
    num: BYTE;
    tab: ARRAY [MAXRTE] OF RipRouEntryT;
  END;

// (topology) functions
  FUNCTION fSrcToIa(n: NodeIdT, i: InterfaceIdT) ::=
  IF ((n = R1_ID)
    AND (i = R1_I1_ID))
  THEN
    R1_I1_IA
  ELSEIF
    ((n = R1_ID)
    AND (i = R1_I2_ID))
  THEN
    R1_I2_IA
  ELSEIF
    ((n = R1_ID)
    AND (i = R1_I3_ID))
  THEN
    R1_I3_IA
  ELSEIF
    ((n = R2_ID)
    AND (i = R2_I1_ID))
  THEN
    R2_I1_IA
  ELSEIF
    ((n = R2_ID)
    AND (i = R2_I2_ID))
  THEN
    R2_I2_IA
  ELSEIF
    ((n = R2_ID)
    AND (i = R2_I3_ID))
  THEN
    R2_I3_IA
  ELSEIF
    ((n = R3_ID)
    AND (i = R3_I1_ID))
  THEN
    R3_I1_IA
```

```
ELSEIF
  ((n = R3_ID)
  AND (i = R3_I2_ID))
THEN
  R3_I2_IA
ELSEIF
  ((n = R3_ID)
  AND (i = R3_I3_ID))
THEN
  R3_I3_IA
ELSEIF
  ((n = H1_ID)
  AND (i = H1_I1_ID))
THEN
  H1_I1_IA
ELSEIF
  ((n = H2_ID)
  AND (i = H2_I1_ID))
THEN
  H2_I1_IA
ELSEIF
  ((n = HA_ID)
  AND (i = HA_I1_ID))
THEN
  HA_I1_IA
ELSE
  INVALID_IA
END;

FUNCTION fSrcToZone(n: NodeIdT, i: InterfaceIdT) ::=
IF ((n = R1_ID)
  AND (i = R1_I1_ID))
THEN
  Z1_ID
ELSEIF
  ((n = R1_ID)
  AND (i = R1_I2_ID))
THEN
  ZBB12_ID
ELSEIF
  ((n = R1_ID)
  AND (i = R1_I3_ID))
THEN
  ZBB13_ID
```

```
ELSEIF
  ((n = R2_ID)
  AND (i = R2_I1_ID))
THEN
  Z2_ID
ELSEIF
  ((n = R2_ID)
  AND (i = R2_I2_ID))
THEN
  ZBB12_ID
ELSEIF
  ((n = R2_ID)
  AND (i = R2_I3_ID))
THEN
  ZBB23_ID
ELSEIF
  ((n = R3_ID)
  AND (i = R3_I1_ID))
THEN
  Z3_ID
ELSEIF
  ((n = R3_ID)
  AND (i = R3_I2_ID))
THEN
  ZBB13_ID
ELSEIF
  ((n = R3_ID)
  AND (i = R3_I3_ID))
THEN
  ZBB23_ID
ELSEIF
  ((n = H1_ID)
  AND (i = H1_I1_ID))
THEN
  Z1_ID
ELSEIF
  ((n = H2_ID)
  AND (i = H2_I1_ID))
THEN
  Z2_ID
ELSEIF
  ((n = HA_ID)
  AND (i = HA_I1_ID))
THEN
```

```
  Z3_ID
ELSE
  UNKNOWN_ZONE
END;

FUNCTION fIaToZone(ia: SYS_IAS) ::=
IF (ia = R1_I1_IA)
THEN
  Z1_ID
ELSEIF
   (ia = R1_I2_IA)
THEN
  ZBB12_ID
ELSEIF
   (ia = R1_I3_IA)
THEN
  ZBB13_ID
ELSEIF
   (ia = R2_I1_IA)
THEN
  Z2_ID
ELSEIF
   (ia = R2_I2_IA)
THEN
  ZBB12_ID
ELSEIF
   (ia = R2_I3_IA)
THEN
  ZBB23_ID
ELSEIF
   (ia = R3_I1_IA)
THEN
  Z3_ID
ELSEIF
   (ia = R3_I2_IA)
THEN
  ZBB13_ID
ELSEIF
   (ia = R3_I3_IA)
THEN
  ZBB23_ID
ELSEIF
   (ia = H1_I1_IA)
THEN
```

```
    Z1_ID
  ELSEIF
    (ia = H2_I1_IA)
  THEN
    Z2_ID
  ELSEIF
    (ia = HA_I1_IA)
  THEN
    Z3_ID
  ELSE
    UNKNOWN_ZONE
  END;

// predicates
  PREDICATE pValidIf(piid: InterfaceIdT, pmii: BYTE) ::=
    (UNKNOWN_IF < piid)
    AND (piid <= pmii);

  PREDICATE pRipPacketIn(b: RpaSystemBufT) ::=
    b.act = RPA_RIPIN
    AND b.pkt.pt = PT_RIP;

  PREDICATE pRipPacketInValid
  (b: RpaSystemBufT, n: NodeIdT, i: InterfaceIdT) ::=
    b.pkt.dat_rme <= ME_INF
    AND fIaToZone(b.pkt.dat_isa) = fSrcToZone(n, i)
    AND b.pkt.dat_rde != INVALID_IA;

  PREDICATE pRipPacketInMatchesRte
  (b: RpaSystemBufT, rt: RouTableT, i: BYTE) ::=
    i < rt.num
    AND rt.tab[i].dty = DT_ZONE
    AND rt.tab[i].dst = b.pkt.dat_rde;

  PREDICATE pRipPacketInMsl
  (b: RpaSystemBufT, rt: RouTableT) ::=
    b.pkt.dat_rme < ME_INF
    AND rt.num < MAXRTE;

  // pseudo predicate (macro)
  PREDICATE pRipPacketClear(b: RpaSystemBufT) ::=
    b.act != RPA_NONE_EMPTY
    AND b.act' = RPA_NONE_EMPTY;
```

```
// system process (flat version, i.e. after expansion!)
/*
 Adaptions to framework's process types before flat system
 expansion:
 - ActiveHostIpNode: spcs action removed
   (replaced by direct send, in derived types as well,
   saves one processing step)

 Adaptions due to assigned roles & topology:
 - ActiveNonPromHostIpNode (H1): snd_ip action parameters adapted
 - AttackerHostIpNode (HA): snd_ripu action parameters adapted
*/
PROCESS IpRipRoutingExample ();
CONST
  // initial routing tables
  R1_RT : RouTableT = {num:6, tab:[
    {dty:DT_ZONE, dst:DST_DEF, nho:NHO_DIR, met:0,
      rto:1, rcf:false},
    {dty:DT_ZONE, dst:ZBB12_ID, nho:NHO_DIR, met:1,
      rto:2, rcf:false},
    {dty:DT_ZONE, dst:ZBB13_ID, nho:NHO_DIR, met:1,
      rto:3, rcf:false},
    {dty:DT_ZONE, dst:Z1_ID, nho:NHO_DIR, met:1,
      rto:1, rcf:false},
    {dty:DT_ZONE, dst:Z2_ID, nho:R2_I2_IA, met:4,
      rto:UNKNOWN_IF, rcf:false},
    {dty:DT_ZONE, dst:Z3_ID, nho:R3_I2_IA, met:2,
      rto:UNKNOWN_IF, rcf:false}]};

  R2_RT : RouTableT = {num:6, tab:[
    {dty:DT_ZONE, dst:DST_DEF, nho:NHO_DIR, met:0,
      rto:1, rcf:false},
    {dty:DT_ZONE, dst:ZBB12_ID, nho:NHO_DIR, met:1,
      rto:2, rcf:false},
    {dty:DT_ZONE, dst:ZBB23_ID, nho:NHO_DIR, met:1,
      rto:3, rcf:false},
    {dty:DT_ZONE, dst:Z1_ID, nho:R1_I2_IA, met:2,
      rto:UNKNOWN_IF, rcf:false},
    {dty:DT_ZONE, dst:Z2_ID, nho:NHO_DIR, met:3,
      rto:1, rcf:false},
    {dty:DT_ZONE, dst:Z3_ID, nho:R3_I3_IA,met:2,
      rto:UNKNOWN_IF, rcf:false}]};

  R3_RT : RouTableT = {num:6, tab:[
```

```
    {dty:DT_ZONE, dst:DST_DEF, nho:NHO_DIR, met:0,
      rto:1, rcf:false},
    {dty:DT_ZONE, dst:ZBB13_ID, nho:NHO_DIR, met:1,
      rto:2, rcf:false},
    {dty:DT_ZONE, dst:ZBB23_ID, nho:NHO_DIR, met:1,
      rto:3, rcf:false},
    {dty:DT_ZONE, dst:Z1_ID, nho:R1_I3_IA, met:2,
      rto:UNKNOWN_IF, rcf:false},
    {dty:DT_ZONE, dst:Z2_ID, nho:R2_I3_IA, met:4,
      rto:UNKNOWN_IF, rcf:false},
    {dty:DT_ZONE, dst:Z3_ID, nho:NHO_DIR, met:1,
      rto:1, rcf:false}]};

  VAR
    med_buf: ARRAY [MAXZONES] OF PacketBufT;
    h1_itf: InterfaceT;
    h2_itf: InterfaceT;
    ha_itf: InterfaceT;
    r1_rt: RouTableT;
    r1_ifs: ARRAY [3] OF InterfaceT;
    r1_fwd_iid: InterfaceIdT;
    r1_fwd_rte: IpAddressT;
    r2_rt: RouTableT;
    r2_ifs: ARRAY [3] OF InterfaceT;
    r2_fwd_iid: InterfaceIdT;
    r2_fwd_rte: IpAddressT;
    r3_rt: RouTableT;
    r3_ifs: ARRAY [3] OF InterfaceT;
    r3_fwd_iid: InterfaceIdT;
    r3_fwd_rte: IpAddressT;

  INIT ::=
    FORALL med_i:MAXZONES :[med_buf[med_i].usd = FALSE]
    AND h1_itf.rpa.act = RPA_NONE_EMPTY
    AND h1_itf.spa.act = SPA_NONE_EMPTY
    AND h1_itf.ia = fSrcToIa(H1_ID, 1)
    AND h1_itf.rpa.act = RPA_NONE_EMPTY
    AND h1_itf.spa.act = SPA_NONE_EMPTY
    AND h1_itf.ia = fSrcToIa(H1_ID, 1)
    AND h2_itf.rpa.act = RPA_NONE_EMPTY
    AND h2_itf.spa.act = SPA_NONE_EMPTY
    AND h2_itf.ia = fSrcToIa(H2_ID, 1)
    AND ha_itf.rpa.act = RPA_NONE_EMPTY
    AND ha_itf.spa.act = SPA_NONE_EMPTY
```

```
    AND ha_itf.ia = fSrcToIa(HA_ID, 1)
    AND r1_rt = R1_RT
    AND FORALL r1_j:3 :[r1_ifs[r1_j].rpa.act = RPA_NONE_EMPTY
    AND r1_ifs[r1_j].spa.act = SPA_NONE_EMPTY
    AND r1_ifs[r1_j].ia = fSrcToIa(R1_ID, r1_j + 1)]
    AND r1_fwd_iid = UNKNOWN_IF
    AND r1_fwd_rte = RT_DEF
    AND r2_rt = R2_RT
    AND FORALL r2_j:3 :[r2_ifs[r2_j].rpa.act = RPA_NONE_EMPTY
    AND r2_ifs[r2_j].spa.act = SPA_NONE_EMPTY
    AND r2_ifs[r2_j].ia = fSrcToIa(R2_ID, r2_j + 1)]
    AND r2_fwd_iid = UNKNOWN_IF
    AND r2_fwd_rte = RT_DEF
    AND r3_rt = R3_RT
    AND FORALL r3_j:3 :[r3_ifs[r3_j].rpa.act = RPA_NONE_EMPTY
    AND r3_ifs[r3_j].spa.act = SPA_NONE_EMPTY
    AND r3_ifs[r3_j].ia = fSrcToIa(R3_ID, r3_j + 1)]
    AND r3_fwd_iid = UNKNOWN_IF
    AND r3_fwd_rte = RT_DEF;

ACTIONS // external actions for all nodes & routers

  // paramod: med_buf[fSrcToZone(H1_ID, 1) - 1].pkt = pkt
  rcv_h1() ::=
    fSrcToZone(H1_ID, 1) != UNKNOWN_ZONE
    AND med_buf[fSrcToZone(H1_ID, 1) - 1].usd = TRUE
    AND med_buf[fSrcToZone(H1_ID, 1) - 1].pkt.dat_ida = h1_itf.ia
    AND med_buf[fSrcToZone(H1_ID, 1) - 1].pkt.dat_ida != BC_IA
    AND med_buf[fSrcToZone(H1_ID, 1) - 1].usd' = FALSE
    AND h1_itf.rpa.pkt' = med_buf[fSrcToZone(H1_ID, 1) - 1].pkt
    AND h1_itf.rpa.act' = RPA_RPCS;

  // pkt = h1_itf.spa.pkt
  snd_h1() ::=
    fSrcToZone(H1_ID, 1) != UNKNOWN_ZONE
    AND med_buf[fSrcToZone(H1_ID, 1) - 1].usd = FALSE
    AND h1_itf.spa.act = SPA_SND
    AND med_buf[fSrcToZone(H1_ID, 1) - 1].usd' = TRUE
    AND med_buf[fSrcToZone(H1_ID, 1) - 1].pkt' = h1_itf.spa.pkt
    AND h1_itf.spa.act' = SPA_NONE_EMPTY;

  // med_buf[fSrcToZone(H2_ID, 1) - 1].pkt = pkt
  rcv_h2() ::=
    fSrcToZone(H2_ID, 1) != UNKNOWN_ZONE
```

```
    AND med_buf[fSrcToZone(H2_ID, 1) - 1].usd = TRUE
    AND med_buf[fSrcToZone(H2_ID, 1) - 1].pkt.dat_ida = h2_itf.ia
    AND med_buf[fSrcToZone(H2_ID, 1) - 1].pkt.dat_ida != BC_IA
    AND med_buf[fSrcToZone(H2_ID, 1) - 1].usd' = FALSE
    AND h2_itf.rpa.pkt' = med_buf[fSrcToZone(H2_ID, 1) - 1].pkt
    AND h2_itf.rpa.act' = RPA_RPCS;

  // pkt = ha_itf.spa.pkt
  snd_ha() ::=
    fSrcToZone(HA_ID, 1) != UNKNOWN_ZONE
    AND med_buf[fSrcToZone(HA_ID, 1) - 1].usd = FALSE
    AND ha_itf.spa.act = SPA_SND
    AND med_buf[fSrcToZone(HA_ID, 1) - 1].usd' = TRUE
    AND med_buf[fSrcToZone(HA_ID, 1) - 1].pkt' = ha_itf.spa.pkt
    AND ha_itf.spa.act' = SPA_NONE_EMPTY;

  // med_buf[fSrcToZone(HA_ID, 1) - 1].pkt = pkt
  rcv_ha() ::=
    fSrcToZone(HA_ID, 1) != UNKNOWN_ZONE
    AND med_buf[fSrcToZone(HA_ID, 1) - 1].usd = TRUE
    AND med_buf[fSrcToZone(HA_ID, 1) - 1].pkt.dat_ida != BC_IA
    AND med_buf[fSrcToZone(HA_ID, 1) - 1].usd' = FALSE
    AND ha_itf.rpa.pkt' = med_buf[fSrcToZone(HA_ID, 1) - 1].pkt
    AND ha_itf.rpa.act' = RPA_RPCS;

  // pkt = r1_ifs[iid - 1].spa.pkt
  snd_r1(iid: InterfaceIdT) ::=
    fSrcToZone(R1_ID, iid) != UNKNOWN_ZONE
    AND med_buf[fSrcToZone(R1_ID, iid) - 1].usd = FALSE
    AND pValidIf(iid, 3)
    AND r1_ifs[iid - 1].spa.act = SPA_SND
    AND med_buf[fSrcToZone(R1_ID, iid) - 1].usd' = TRUE
    AND med_buf[fSrcToZone(R1_ID, iid) - 1].pkt' =
      r1_ifs[iid - 1].spa.pkt
    AND r1_ifs[iid - 1].spa.act' = SPA_NONE_EMPTY;

  // med_buf[fSrcToZone(R1_ID, iid) - 1].pkt = pkt
  rcv_r1(iid: InterfaceIdT) ::=
    fSrcToZone(R1_ID, iid) != UNKNOWN_ZONE
    AND med_buf[fSrcToZone(R1_ID, iid) - 1].usd = TRUE
    AND pValidIf(iid, 3)
    AND r1_ifs[iid - 1].rpa.act = RPA_NONE_EMPTY
    AND med_buf[fSrcToZone(R1_ID, iid) - 1].pkt.dat_ida != BC_IA
    AND med_buf[fSrcToZone(R1_ID, iid) - 1].usd' = FALSE
```

```
  AND r1_ifs[iid - 1].rpa.pkt' =
    med_buf[fSrcToZone(R1_ID, iid) - 1].pkt
  AND r1_ifs[iid - 1].rpa.act' = RPA_RPCS;

// pkt = r2_ifs[iid - 1].spa.pkt
snd_r2(iid: InterfaceIdT) ::=
  fSrcToZone(R2_ID, iid) != UNKNOWN_ZONE
  AND med_buf[fSrcToZone(R2_ID, iid) - 1].usd = FALSE
  AND pValidIf(iid, 3)
  AND r2_ifs[iid - 1].spa.act = SPA_SND
  AND med_buf[fSrcToZone(R2_ID, iid) - 1].usd' = TRUE
  AND med_buf[fSrcToZone(R2_ID, iid) - 1].pkt' =
    r2_ifs[iid - 1].spa.pkt
  AND r2_ifs[iid - 1].spa.act' = SPA_NONE_EMPTY;

// med_buf[fSrcToZone(R2_ID, iid) - 1].pkt = pkt
rcv_r2(iid: InterfaceIdT) ::=
  fSrcToZone(R2_ID, iid) != UNKNOWN_ZONE
  AND med_buf[fSrcToZone(R2_ID, iid) - 1].usd = TRUE
  AND pValidIf(iid, 3)
  AND r2_ifs[iid - 1].rpa.act = RPA_NONE_EMPTY
  AND med_buf[fSrcToZone(R2_ID, iid) - 1].pkt.dat_ida != BC_IA
  AND med_buf[fSrcToZone(R2_ID, iid) - 1].usd' = FALSE
  AND r2_ifs[iid - 1].rpa.pkt' =
    med_buf[fSrcToZone(R2_ID, iid) - 1].pkt
  AND r2_ifs[iid - 1].rpa.act' = RPA_RPCS;

// pkt = r3_ifs[iid - 1].spa.pkt
snd_r3(iid: InterfaceIdT) ::=
  fSrcToZone(R3_ID, iid) != UNKNOWN_ZONE
  AND med_buf[fSrcToZone(R3_ID, iid) - 1].usd = FALSE
  AND pValidIf(iid, 3)
  AND r3_ifs[iid - 1].spa.act = SPA_SND
  AND med_buf[fSrcToZone(R3_ID, iid) - 1].usd' = TRUE
  AND med_buf[fSrcToZone(R3_ID, iid) - 1].pkt' =
    r3_ifs[iid - 1].spa.pkt
  AND r3_ifs[iid - 1].spa.act' = SPA_NONE_EMPTY;

// med_buf[fSrcToZone(R3_ID, iid) - 1].pkt = pkt
rcv_r3(iid: InterfaceIdT) ::=
  fSrcToZone(R3_ID, iid) != UNKNOWN_ZONE
  AND med_buf[fSrcToZone(R3_ID, iid) - 1].usd = TRUE
  AND pValidIf(iid, 3)
  AND r3_ifs[iid - 1].rpa.act = RPA_NONE_EMPTY
```

```
  AND med_buf[fSrcToZone(R3_ID, iid) - 1].pkt.dat_ida != BC_IA
  AND med_buf[fSrcToZone(R3_ID, iid) - 1].usd' = FALSE
  AND r3_ifs[iid - 1].rpa.pkt' =
    med_buf[fSrcToZone(R3_ID, iid) - 1].pkt
  AND r3_ifs[iid - 1].rpa.act' = RPA_RPCS;

// med_buf[zid - 1].pkt = pkt
rbc(zid: ZoneIdT) ::=
  zid != UNKNOWN_ZONE
  AND med_buf[zid - 1].usd = TRUE
  AND med_buf[zid - 1].pkt.dat_ida = BC_IA
  AND med_buf[zid - 1].usd' = FALSE
  AND
  IF (h1_itf.rpa.act = RPA_NONE_EMPTY)
    AND (fSrcToZone(H1_ID, 1) = zid)
    AND (med_buf[zid - 1].pkt.dat_isa != h1_itf.ia)
  THEN
    h1_itf.rpa.pkt' = med_buf[zid - 1].pkt
    AND h1_itf.rpa.act' = RPA_RPCS
  END
  AND
  IF (h2_itf.rpa.act = RPA_NONE_EMPTY)
    AND (fSrcToZone(H2_ID, 1) = zid)
    AND (med_buf[zid - 1].pkt.dat_isa != h2_itf.ia)
  THEN
    h2_itf.rpa.pkt' = med_buf[zid - 1].pkt
    AND h2_itf.rpa.act' = RPA_RPCS
  END
  AND
  IF (ha_itf.rpa.act = RPA_NONE_EMPTY)
    AND (fSrcToZone(HA_ID, 1) = zid)
    AND (med_buf[zid - 1].pkt.dat_isa != ha_itf.ia)
  THEN
    ha_itf.rpa.pkt' = med_buf[zid - 1].pkt
    AND ha_itf.rpa.act' = RPA_RPCS
  END
  AND
  IF EXISTS r1_rbc_i:3 :[
    r1_ifs[(r1_rbc_i + 1) - 1].rpa.act = RPA_NONE_EMPTY
    AND r1_ifs[(r1_rbc_i + 1) - 1].ia !=
      med_buf[zid - 1].pkt.dat_isa
    AND fSrcToZone(R1_ID, (r1_rbc_i + 1)) = zid]
  THEN
    r1_ifs[(r1_rbc_i + 1) - 1].rpa.pkt' = med_buf[zid - 1].pkt
```

```
    AND r1_ifs[(r1_rbc_i + 1) - 1].rpa.act' = RPA_RPCS
  END
  AND
  IF EXISTS r2_rbc_i:3 :[
    r2_ifs[(r2_rbc_i + 1) - 1].rpa.act = RPA_NONE_EMPTY
    AND r2_ifs[(r2_rbc_i + 1) - 1].ia !=
      med_buf[zid - 1].pkt.dat_isa
    AND fSrcToZone(R2_ID, (r2_rbc_i + 1)) = zid]
  THEN
    r2_ifs[(r2_rbc_i + 1) - 1].rpa.pkt' = med_buf[zid - 1].pkt
    AND r2_ifs[(r2_rbc_i + 1) - 1].rpa.act' = RPA_RPCS
  END
  AND
  IF EXISTS r3_rbc_i:3 :[
    r3_ifs[(r3_rbc_i + 1) - 1].rpa.act = RPA_NONE_EMPTY
    AND r3_ifs[(r3_rbc_i + 1) - 1].ia !=
      med_buf[zid - 1].pkt.dat_isa
    AND fSrcToZone(R3_ID, (r3_rbc_i + 1)) = zid]
  THEN
    r3_ifs[(r3_rbc_i + 1) - 1].rpa.pkt' = med_buf[zid - 1].pkt
    AND r3_ifs[(r3_rbc_i + 1) - 1].rpa.act' = RPA_RPCS
  END;

INTERNAL ACTIONS // processing actions for all nodes & routers
 h1_snd_ip() ::=
  h1_itf.spa.act = SPA_NONE_EMPTY

  AND h1_itf.spa.pkt.pt' = PT_IP
  AND h1_itf.spa.pkt.dat_isa' = h1_itf.ia
  AND h1_itf.spa.pkt.dat_ida' = H2_I1_IA
  AND h1_itf.spa.pkt.dat_rde' = 0
  AND h1_itf.spa.pkt.dat_rme' = 0
  AND h1_itf.spa.act' = SPA_SND;

 h1_rpcs() ::=
  h1_itf.rpa.act = RPA_RPCS
  AND h1_itf.rpa.act' = RPA_NONE_EMPTY;

 h2_rpcs() ::=
  h2_itf.rpa.act = RPA_RPCS
  AND h2_itf.rpa.act' = RPA_NONE_EMPTY;

 ha_snd_ripu() ::=
  ha_itf.spa.act = SPA_NONE_EMPTY
```

```
    AND ha_itf.spa.pkt.pt' = PT_RIP
    AND ha_itf.spa.pkt.dat_isa' = ha_itf.ia
    AND ha_itf.spa.pkt.dat_ida' = BC_IA
    AND ha_itf.spa.pkt.dat_rde' = Z2_ID
    AND ha_itf.spa.pkt.dat_rme' = 1
    AND ha_itf.spa.act' = SPA_SND;

  ha_rpcs() ::=
    ha_itf.rpa.act = RPA_RPCS
    AND ha_itf.rpa.act' = RPA_NONE_EMPTY;

  r1_rip_in_inv(iid: InterfaceIdT) ::=
    pValidIf(iid, 3)
    AND pRipPacketIn(r1_ifs[iid - 1].rpa)
    AND NOT pRipPacketInValid(r1_ifs[iid - 1].rpa, R1_ID, iid)
    AND pRipPacketClear(r1_ifs[iid - 1].rpa);

  r1_rip_in_v_ree_nsc_nmb(iid: InterfaceIdT, i: BYTE) ::=
    pValidIf(iid, 3)
    AND i < r1_rt.num
    AND pRipPacketIn(r1_ifs[iid - 1].rpa)
    AND pRipPacketInValid(r1_ifs[iid - 1].rpa, R1_ID, iid)
    AND pRipPacketInMatchesRte(r1_ifs[iid - 1].rpa, r1_rt, i)
    AND r1_rt.tab[i].nho != r1_ifs[iid - 1].rpa.pkt.dat_isa
    AND r1_ifs[iid - 1].rpa.pkt.dat_rme < r1_rt.tab[i].met - 1
    AND EXISTS j:3 :[i < r1_rt.num
    AND fSrcToZone(R1_ID, j + 1) = fIaToZone(r1_rt.tab[i].nho)]
    AND r1_rt.tab[i].rto' = (j + 1)
    AND r1_rt.tab[i].met' = r1_ifs[iid - 1].rpa.pkt.dat_rme + 1
    AND r1_rt.tab[i].nho' = r1_ifs[iid - 1].rpa.pkt.dat_isa
    AND r1_rt.tab[i].rcf' = TRUE
    AND pRipPacketClear(r1_ifs[iid - 1].rpa);

  r1_rip_in_v_ree_nsc_nmw(iid: InterfaceIdT, i: BYTE) ::=
    pValidIf(iid, 3)
    AND i < r1_rt.num
    AND pRipPacketIn(r1_ifs[iid - 1].rpa)
    AND pRipPacketInValid(r1_ifs[iid - 1].rpa, R1_ID, iid)
    AND pRipPacketInMatchesRte(r1_ifs[iid - 1].rpa, r1_rt, i)
    AND r1_rt.tab[i].nho != r1_ifs[iid - 1].rpa.pkt.dat_isa
    AND r1_ifs[iid - 1].rpa.pkt.dat_rme >= r1_rt.tab[i].met - 1
    AND pRipPacketClear(r1_ifs[iid - 1].rpa);
```

```
r1_rip_in_v_ree_fsc(iid: InterfaceIdT, i: BYTE) ::=
  pValidIf(iid, 3)
  AND i < r1_rt.num
  AND pRipPacketIn(r1_ifs[iid - 1].rpa)
  AND pRipPacketInValid(r1_ifs[iid - 1].rpa, R1_ID, iid)
  AND pRipPacketInMatchesRte(r1_ifs[iid - 1].rpa, r1_rt, i)
  AND r1_rt.tab[i].nho = r1_ifs[iid - 1].rpa.pkt.dat_isa
  AND r1_rt.tab[i].met' = r1_ifs[iid - 1].rpa.pkt.dat_rme
  AND r1_rt.tab[i].rcf' = TRUE
  AND pRipPacketClear(r1_ifs[iid - 1].rpa);

r1_rip_in_v_nre_msl(iid: InterfaceIdT) ::=
  pValidIf(iid, 3)
  AND pRipPacketIn(r1_ifs[iid - 1].rpa)
  AND pRipPacketInValid(r1_ifs[iid - 1].rpa, R1_ID, iid)
  AND FORALL i:r1_rt.num :[pValidIf(iid, 3)
  AND NOT pRipPacketInMatchesRte(r1_ifs[iid - 1].rpa, r1_rt, i)]
  AND pRipPacketInMsl(r1_ifs[iid - 1].rpa, r1_rt)
  AND r1_rt.tab[r1_rt.num].dty' = DT_ZONE
  AND r1_rt.tab[r1_rt.num].dst' =
    r1_ifs[iid - 1].rpa.pkt.dat_rde
  AND r1_rt.tab[r1_rt.num].nho' =
    r1_ifs[iid - 1].rpa.pkt.dat_isa
  AND r1_rt.tab[r1_rt.num].met' =
    r1_ifs[iid - 1].rpa.pkt.dat_rme
  AND r1_rt.tab[r1_rt.num].rto' = INVALID_IA
  AND r1_rt.tab[r1_rt.num].rcf' = TRUE
  AND r1_rt.num' = r1_rt.num + 1
  AND pRipPacketClear(r1_ifs[iid - 1].rpa);

r1_rip_in_v_nre_mns(iid: InterfaceIdT) ::=
  pValidIf(iid, 3)
  AND pRipPacketIn(r1_ifs[iid - 1].rpa)
  AND pRipPacketInValid(r1_ifs[iid - 1].rpa, R1_ID, iid)
  AND FORALL i:r1_rt.num :[pValidIf(iid, 3)
  AND NOT pRipPacketInMatchesRte(r1_ifs[iid - 1].rpa,
    r1_rt, i)]
  AND NOT pRipPacketInMsl(r1_ifs[iid - 1].rpa, r1_rt)
  AND pRipPacketClear(r1_ifs[iid - 1].rpa);

r1_rip_out(i: BYTE) ::=
  i < r1_rt.num
  AND r1_rt.tab[i].rcf = TRUE
  AND FORALL j:MAXIFS :[
```

```
      (fIaToZone(r1_ifs[j].ia) != fIaToZone(r1_rt.tab[i].nho))
      OR (r1_ifs[j].spa.act = SPA_NONE_EMPTY)]
    AND UPDATEALL k:MAXIFS :[
      (fIaToZone(r1_ifs[k].ia) != fIaToZone(r1_rt.tab[i].nho))
      AND r1_ifs[k].spa.act' = SPA_SND
      AND r1_ifs[k].spa.pkt.pt' = PT_RIP
      AND r1_ifs[k].spa.pkt.dat_isa' = r1_ifs[k].ia
      AND r1_ifs[k].spa.pkt.dat_ida' = BC_IA
      AND r1_ifs[k].spa.pkt.dat_rde' = r1_rt.tab[i].dst
      AND r1_ifs[k].spa.pkt.dat_rme' = r1_rt.tab[i].met]
    AND r1_rt.tab[i].rcf' = FALSE;

  r1_rpcs(iid: InterfaceIdT) ::=
    pValidIf(iid, 3)
    AND r1_ifs[iid - 1].rpa.act = RPA_RPCS
    AND
    IF ((r1_ifs[iid - 1].rpa.pkt.dat_ida != r1_ifs[iid - 1].ia)
      AND (r1_ifs[iid - 1].rpa.pkt.dat_ida != BC_IA))
    THEN
      r1_ifs[iid - 1].rpa.act' = RPA_FWD
    ELSE
      IF (r1_ifs[iid - 1].rpa.pkt.pt = PT_RIP)
      THEN
        r1_ifs[iid - 1].rpa.act' = RPA_RIPIN
      ELSE
        r1_ifs[iid - 1].rpa.act' = RPA_NONE_EMPTY
      END     END;

  r1_fwd(iid: InterfaceIdT) ::=
    pValidIf(iid, 3)
    AND r1_ifs[iid - 1].rpa.act = RPA_FWD
    AND r1_rt.num > 0
    AND
    IF EXISTS i:r1_rt.num :[r1_rt.tab[i].dty = DT_HOST
      AND r1_rt.tab[i].dst = r1_ifs[iid - 1].rpa.pkt.dat_ida]
    THEN
      r1_fwd_rte' = i
    ELSIF
       EXISTS j:r1_rt.num :[
         r1_rt.tab[j].dty = DT_ZONE
         AND r1_rt.tab[j].dst =
           fIaToZone(r1_ifs[iid - 1].rpa.pkt.dat_ida)]
    THEN
      r1_fwd_rte' = j
```

```
  ELSE
    r1_fwd_rte' = RT_DEF
  END
  AND
  IF (r1_rt.tab[r1_fwd_rte].nho = NHO_DIR)
  THEN
    r1_fwd_iid' = r1_rt.tab[r1_fwd_rte].rto
  ELSE
    IF EXISTS k:r1_rt.num :[r1_rt.tab[k].dty = DT_HOST
      AND r1_rt.tab[k].dst = r1_rt.tab[r1_fwd_rte].nho]
    THEN
      r1_fwd_rte' = k
    ELSIF
       EXISTS l:r1_rt.num :[
         r1_rt.tab[l].dty = DT_ZONE
         AND r1_rt.tab[l].dst =
           fIaToZone(r1_rt.tab[r1_fwd_rte].nho)]
    THEN
      r1_fwd_rte' = l
    ELSE
      r1_fwd_rte' = RT_DEF
    END
    AND
    IF (r1_rt.tab[r1_fwd_rte].nho = NHO_DIR)
    THEN
      r1_fwd_iid' = r1_rt.tab[r1_fwd_rte].rto
    END     END
  AND
  IF (r1_fwd_iid != UNKNOWN_IF)
    AND (pValidIf(r1_fwd_iid, 3))
    AND (r1_ifs[r1_fwd_iid - 1].spa.act = SPA_NONE_EMPTY)
  THEN
    r1_ifs[r1_fwd_iid - 1].spa.act' = SPA_SND
    AND r1_ifs[r1_fwd_iid - 1].spa.pkt' =
      r1_ifs[iid - 1].rpa.pkt
    AND r1_ifs[iid - 1].rpa.act' = RPA_NONE_EMPTY
  END
  AND r1_fwd_iid' = UNKNOWN_IF
  AND r1_fwd_rte' = RT_DEF;

r2_rip_in_inv(iid: InterfaceIdT) ::=
  pValidIf(iid, 3)
  AND pRipPacketIn(r2_ifs[iid - 1].rpa)
  AND NOT pRipPacketInValid(r2_ifs[iid - 1].rpa, R2_ID, iid)
```

```
    AND pRipPacketClear(r2_ifs[iid - 1].rpa);

  r2_rip_in_v_ree_nsc_nmb(iid: InterfaceIdT, i: BYTE) ::=
    pValidIf(iid, 3)
    AND i < r2_rt.num
    AND pRipPacketIn(r2_ifs[iid - 1].rpa)
    AND pRipPacketInValid(r2_ifs[iid - 1].rpa, R2_ID, iid)
    AND pRipPacketInMatchesRte(r2_ifs[iid - 1].rpa, r2_rt, i)
    AND r2_rt.tab[i].nho != r2_ifs[iid - 1].rpa.pkt.dat_isa
    AND r2_ifs[iid - 1].rpa.pkt.dat_rme < r2_rt.tab[i].met - 1
    AND EXISTS j:3 :[i < r2_rt.num
    AND fSrcToZone(R2_ID, j + 1) = fIaToZone(r2_rt.tab[i].nho)]
    AND r2_rt.tab[i].rto' = (j + 1)
    AND r2_rt.tab[i].met' = r2_ifs[iid - 1].rpa.pkt.dat_rme + 1
    AND r2_rt.tab[i].nho' = r2_ifs[iid - 1].rpa.pkt.dat_isa
    AND r2_rt.tab[i].rcf' = TRUE
    AND pRipPacketClear(r2_ifs[iid - 1].rpa);

  r2_rip_in_v_ree_nsc_nmw(iid: InterfaceIdT, i: BYTE) ::=
    pValidIf(iid, 3)
    AND i < r2_rt.num
    AND pRipPacketIn(r2_ifs[iid - 1].rpa)
    AND pRipPacketInValid(r2_ifs[iid - 1].rpa, R2_ID, iid)
    AND pRipPacketInMatchesRte(r2_ifs[iid - 1].rpa, r2_rt, i)
    AND r2_rt.tab[i].nho != r2_ifs[iid - 1].rpa.pkt.dat_isa
    AND r2_ifs[iid - 1].rpa.pkt.dat_rme >= r2_rt.tab[i].met - 1
    AND pRipPacketClear(r2_ifs[iid - 1].rpa);

  r2_rip_in_v_ree_fsc(iid: InterfaceIdT, i: BYTE) ::=
    pValidIf(iid, 3)
    AND i < r2_rt.num
    AND pRipPacketIn(r2_ifs[iid - 1].rpa)
    AND pRipPacketInValid(r2_ifs[iid - 1].rpa, R2_ID, iid)
    AND pRipPacketInMatchesRte(r2_ifs[iid - 1].rpa, r2_rt, i)
    AND r2_rt.tab[i].nho = r2_ifs[iid - 1].rpa.pkt.dat_isa
    AND r2_rt.tab[i].met' = r2_ifs[iid - 1].rpa.pkt.dat_rme
    AND r2_rt.tab[i].rcf' = TRUE
    AND pRipPacketClear(r2_ifs[iid - 1].rpa);

  r2_rip_in_v_nre_msl(iid: InterfaceIdT) ::=
    pValidIf(iid, 3)
    AND pRipPacketIn(r2_ifs[iid - 1].rpa)
    AND pRipPacketInValid(r2_ifs[iid - 1].rpa, R2_ID, iid)
    AND FORALL i:r2_rt.num :[pValidIf(iid, 3)
```

```
  AND NOT pRipPacketInMatchesRte(r2_ifs[iid - 1].rpa,
    r2_rt, i)]
  AND pRipPacketInMsl(r2_ifs[iid - 1].rpa, r2_rt)
  AND r2_rt.tab[r2_rt.num].dty' = DT_ZONE
  AND r2_rt.tab[r2_rt.num].dst' =
    r2_ifs[iid - 1].rpa.pkt.dat_rde
  AND r2_rt.tab[r2_rt.num].nho' =
    r2_ifs[iid - 1].rpa.pkt.dat_isa
  AND r2_rt.tab[r2_rt.num].met' =
    r2_ifs[iid - 1].rpa.pkt.dat_rme
  AND r2_rt.tab[r2_rt.num].rto' = INVALID_IA
  AND r2_rt.tab[r2_rt.num].rcf' = TRUE
  AND r2_rt.num' = r2_rt.num + 1
  AND pRipPacketClear(r2_ifs[iid - 1].rpa);

r2_rip_in_v_nre_mns(iid: InterfaceIdT) ::=
  pValidIf(iid, 3)
  AND pRipPacketIn(r2_ifs[iid - 1].rpa)
  AND pRipPacketInValid(r2_ifs[iid - 1].rpa, R2_ID, iid)
  AND FORALL i:r2_rt.num :[
    pValidIf(iid, 3)
    AND NOT pRipPacketInMatchesRte(r2_ifs[iid - 1].rpa,
      r2_rt, i)]
  AND NOT pRipPacketInMsl(r2_ifs[iid - 1].rpa, r2_rt)
  AND pRipPacketClear(r2_ifs[iid - 1].rpa);

r2_rip_out(i: BYTE) ::=
  i < r2_rt.num
  AND r2_rt.tab[i].rcf = TRUE
  AND FORALL j:MAXIFS :[
    (fIaToZone(r2_ifs[j].ia) != fIaToZone(r2_rt.tab[i].nho))
    OR (r2_ifs[j].spa.act = SPA_NONE_EMPTY)]
  AND UPDATEALL k:MAXIFS :[
    (fIaToZone(r2_ifs[k].ia) != fIaToZone(r2_rt.tab[i].nho))
    AND r2_ifs[k].spa.act' = SPA_SND
    AND r2_ifs[k].spa.pkt.pt' = PT_RIP
    AND r2_ifs[k].spa.pkt.dat_isa' = r2_ifs[k].ia
    AND r2_ifs[k].spa.pkt.dat_ida' = BC_IA
    AND r2_ifs[k].spa.pkt.dat_rde' = r2_rt.tab[i].dst
    AND r2_ifs[k].spa.pkt.dat_rme' = r2_rt.tab[i].met]
  AND r2_rt.tab[i].rcf' = FALSE;

r2_rpcs(iid: InterfaceIdT) ::=
  pValidIf(iid, 3)
```

```
    AND r2_ifs[iid - 1].rpa.act = RPA_RPCS
    AND
    IF ((r2_ifs[iid - 1].rpa.pkt.dat_ida != r2_ifs[iid - 1].ia)
      AND (r2_ifs[iid - 1].rpa.pkt.dat_ida != BC_IA))
    THEN
      r2_ifs[iid - 1].rpa.act' = RPA_FWD
    ELSE
      IF (r2_ifs[iid - 1].rpa.pkt.pt = PT_RIP)
      THEN
        r2_ifs[iid - 1].rpa.act' = RPA_RIPIN
      ELSE
        r2_ifs[iid - 1].rpa.act' = RPA_NONE_EMPTY
      END      END;

  r2_fwd(iid: InterfaceIdT) ::=
    pValidIf(iid, 3)
    AND r2_ifs[iid - 1].rpa.act = RPA_FWD
    AND r2_rt.num > 0
    AND
    IF EXISTS i:r2_rt.num :[r2_rt.tab[i].dty = DT_HOST
      AND r2_rt.tab[i].dst = r2_ifs[iid - 1].rpa.pkt.dat_ida]
    THEN
      r2_fwd_rte' = i
    ELSIF
       EXISTS j:r2_rt.num :[
         r2_rt.tab[j].dty = DT_ZONE
         AND r2_rt.tab[j].dst =
           fIaToZone(r2_ifs[iid - 1].rpa.pkt.dat_ida)]
    THEN
      r2_fwd_rte' = j
    ELSE
      r2_fwd_rte' = RT_DEF
    END
    AND
    IF (r2_rt.tab[r2_fwd_rte].nho = NHO_DIR)
    THEN
      r2_fwd_iid' = r2_rt.tab[r2_fwd_rte].rto
    ELSE
      IF EXISTS k:r2_rt.num :[r2_rt.tab[k].dty = DT_HOST
        AND r2_rt.tab[k].dst = r2_rt.tab[r2_fwd_rte].nho]
      THEN
        r2_fwd_rte' = k
      ELSIF
         EXISTS l:r2_rt.num :[
```

```
        r2_rt.tab[l].dty = DT_ZONE
        AND r2_rt.tab[l].dst =
          fIaToZone(r2_rt.tab[r2_fwd_rte].nho)]
    THEN
      r2_fwd_rte' = l
    ELSE
      r2_fwd_rte' = RT_DEF
    END
    AND
    IF (r2_rt.tab[r2_fwd_rte].nho = NHO_DIR)
    THEN
      r2_fwd_iid' = r2_rt.tab[r2_fwd_rte].rto
    END     END
  AND
  IF (r2_fwd_iid != UNKNOWN_IF)
    AND (pValidIf(r2_fwd_iid, 3))
    AND (r2_ifs[r2_fwd_iid - 1].spa.act = SPA_NONE_EMPTY)
  THEN
    r2_ifs[r2_fwd_iid - 1].spa.act' = SPA_SND
    AND r2_ifs[r2_fwd_iid - 1].spa.pkt' =
      r2_ifs[iid - 1].rpa.pkt
    AND r2_ifs[iid - 1].rpa.act' = RPA_NONE_EMPTY
  END
  AND r2_fwd_iid' = UNKNOWN_IF
  AND r2_fwd_rte' = RT_DEF;

r3_rip_in_inv(iid: InterfaceIdT) ::=
  pValidIf(iid, 3)
  AND pRipPacketIn(r3_ifs[iid - 1].rpa)
  AND NOT pRipPacketInValid(r3_ifs[iid - 1].rpa, R3_ID, iid)
  AND pRipPacketClear(r3_ifs[iid - 1].rpa);

r3_rip_in_v_ree_nsc_nmb(iid: InterfaceIdT, i: BYTE) ::=
  pValidIf(iid, 3)
  AND i < r3_rt.num
  AND pRipPacketIn(r3_ifs[iid - 1].rpa)
  AND pRipPacketInValid(r3_ifs[iid - 1].rpa, R3_ID, iid)
  AND pRipPacketInMatchesRte(r3_ifs[iid - 1].rpa, r3_rt, i)
  AND r3_rt.tab[i].nho != r3_ifs[iid - 1].rpa.pkt.dat_isa
  AND r3_ifs[iid - 1].rpa.pkt.dat_rme < r3_rt.tab[i].met - 1
  AND EXISTS j:3 :[i < r3_rt.num
  AND fSrcToZone(R3_ID, j + 1) = fIaToZone(r3_rt.tab[i].nho)]
  AND r3_rt.tab[i].rto' = (j + 1)
  AND r3_rt.tab[i].met' = r3_ifs[iid - 1].rpa.pkt.dat_rme + 1
```

```
    AND r3_rt.tab[i].nho' = r3_ifs[iid - 1].rpa.pkt.dat_isa
    AND r3_rt.tab[i].rcf' = TRUE
    AND pRipPacketClear(r3_ifs[iid - 1].rpa);

  r3_rip_in_v_ree_nsc_nmw(iid: InterfaceIdT, i: BYTE) ::=
    pValidIf(iid, 3)
    AND i < r3_rt.num
    AND pRipPacketIn(r3_ifs[iid - 1].rpa)
    AND pRipPacketInValid(r3_ifs[iid - 1].rpa, R3_ID, iid)
    AND pRipPacketInMatchesRte(r3_ifs[iid - 1].rpa, r3_rt, i)
    AND r3_rt.tab[i].nho != r3_ifs[iid - 1].rpa.pkt.dat_isa
    AND r3_ifs[iid - 1].rpa.pkt.dat_rme >= r3_rt.tab[i].met - 1
    AND pRipPacketClear(r3_ifs[iid - 1].rpa);

  r3_rip_in_v_ree_fsc(iid: InterfaceIdT, i: BYTE) ::=
    pValidIf(iid, 3)
    AND i < r3_rt.num
    AND pRipPacketIn(r3_ifs[iid - 1].rpa)
    AND pRipPacketInValid(r3_ifs[iid - 1].rpa, R3_ID, iid)
    AND pRipPacketInMatchesRte(r3_ifs[iid - 1].rpa, r3_rt, i)
    AND r3_rt.tab[i].nho = r3_ifs[iid - 1].rpa.pkt.dat_isa
    AND r3_rt.tab[i].met' = r3_ifs[iid - 1].rpa.pkt.dat_rme
    AND r3_rt.tab[i].rcf' = TRUE
    AND pRipPacketClear(r3_ifs[iid - 1].rpa);

  r3_rip_in_v_nre_msl(iid: InterfaceIdT) ::=
    pValidIf(iid, 3)
    AND pRipPacketIn(r3_ifs[iid - 1].rpa)
    AND pRipPacketInValid(r3_ifs[iid - 1].rpa, R3_ID, iid)
    AND FORALL i:r3_rt.num :[pValidIf(iid, 3)
    AND NOT pRipPacketInMatchesRte(r3_ifs[iid - 1].rpa,
      r3_rt, i)]
    AND pRipPacketInMsl(r3_ifs[iid - 1].rpa, r3_rt)
    AND r3_rt.tab[r3_rt.num].dty' = DT_ZONE
    AND r3_rt.tab[r3_rt.num].dst' =
      r3_ifs[iid - 1].rpa.pkt.dat_rde
    AND r3_rt.tab[r3_rt.num].nho' =
      r3_ifs[iid - 1].rpa.pkt.dat_isa
    AND r3_rt.tab[r3_rt.num].met' =
      r3_ifs[iid - 1].rpa.pkt.dat_rme
    AND r3_rt.tab[r3_rt.num].rto' = INVALID_IA
    AND r3_rt.tab[r3_rt.num].rcf' = TRUE
    AND r3_rt.num' = r3_rt.num + 1
    AND pRipPacketClear(r3_ifs[iid - 1].rpa);
```

```
r3_rip_in_v_nre_mns(iid: InterfaceIdT) ::=
  pValidIf(iid, 3)
  AND pRipPacketIn(r3_ifs[iid - 1].rpa)
  AND pRipPacketInValid(r3_ifs[iid - 1].rpa, R3_ID, iid)
  AND FORALL i:r3_rt.num :[pValidIf(iid, 3)
  AND NOT pRipPacketInMatchesRte(r3_ifs[iid - 1].rpa,
    r3_rt, i)]
  AND NOT pRipPacketInMsl(r3_ifs[iid - 1].rpa, r3_rt)
  AND pRipPacketClear(r3_ifs[iid - 1].rpa);

r3_rip_out(i: BYTE) ::=
  i < r3_rt.num
  AND r3_rt.tab[i].rcf = TRUE
  AND FORALL j:MAXIFS :[
    (fIaToZone(r3_ifs[j].ia) != fIaToZone(r3_rt.tab[i].nho))
    OR (r3_ifs[j].spa.act = SPA_NONE_EMPTY)]
  AND UPDATEALL k:MAXIFS :[
    (fIaToZone(r3_ifs[k].ia) != fIaToZone(r3_rt.tab[i].nho))
    AND r3_ifs[k].spa.act' = SPA_SND
    AND r3_ifs[k].spa.pkt.pt' = PT_RIP
    AND r3_ifs[k].spa.pkt.dat_isa' = r3_ifs[k].ia
    AND r3_ifs[k].spa.pkt.dat_ida' = BC_IA
    AND r3_ifs[k].spa.pkt.dat_rde' = r3_rt.tab[i].dst
    AND r3_ifs[k].spa.pkt.dat_rme' = r3_rt.tab[i].met]
  AND r3_rt.tab[i].rcf' = FALSE;

r3_rpcs(iid: InterfaceIdT) ::=
  pValidIf(iid, 3)
  AND r3_ifs[iid - 1].rpa.act = RPA_RPCS
  AND
  IF ((r3_ifs[iid - 1].rpa.pkt.dat_ida != r3_ifs[iid - 1].ia)
    AND (r3_ifs[iid - 1].rpa.pkt.dat_ida != BC_IA))
  THEN
    r3_ifs[iid - 1].rpa.act' = RPA_FWD
  ELSE
    IF (r3_ifs[iid - 1].rpa.pkt.pt = PT_RIP)
    THEN
      r3_ifs[iid - 1].rpa.act' = RPA_RIPIN
    ELSE
      r3_ifs[iid - 1].rpa.act' = RPA_NONE_EMPTY
    END     END;

r3_fwd(iid: InterfaceIdT) ::=
```

```
pValidIf(iid, 3)
AND r3_ifs[iid - 1].rpa.act = RPA_FWD
AND r3_rt.num > 0
AND
IF EXISTS i:r3_rt.num :[r3_rt.tab[i].dty = DT_HOST
  AND r3_rt.tab[i].dst = r3_ifs[iid - 1].rpa.pkt.dat_ida]
THEN
  r3_fwd_rte' = i
ELSIF
   EXISTS j:r3_rt.num :[
     r3_rt.tab[j].dty = DT_ZONE
     AND r3_rt.tab[j].dst =
       fIaToZone(r3_ifs[iid - 1].rpa.pkt.dat_ida)]
THEN
  r3_fwd_rte' = j
ELSE
  r3_fwd_rte' = RT_DEF
END
AND
IF (r3_rt.tab[r3_fwd_rte].nho = NHO_DIR)
THEN
  r3_fwd_iid' = r3_rt.tab[r3_fwd_rte].rto
ELSE
  IF EXISTS k:r3_rt.num :[r3_rt.tab[k].dty = DT_HOST
    AND r3_rt.tab[k].dst = r3_rt.tab[r3_fwd_rte].nho]
  THEN
    r3_fwd_rte' = k
  ELSIF
     EXISTS l:r3_rt.num :[
       r3_rt.tab[l].dty = DT_ZONE
       AND r3_rt.tab[l].dst =
         fIaToZone(r3_rt.tab[r3_fwd_rte].nho)]
  THEN
    r3_fwd_rte' = l
  ELSE
    r3_fwd_rte' = RT_DEF
  END
  AND
  IF (r3_rt.tab[r3_fwd_rte].nho = NHO_DIR)
  THEN
    r3_fwd_iid' = r3_rt.tab[r3_fwd_rte].rto
  END     END
AND
IF (r3_fwd_iid != UNKNOWN_IF)
```

```
      AND (pValidIf(r3_fwd_iid, 3))
      AND (r3_ifs[r3_fwd_iid - 1].spa.act = SPA_NONE_EMPTY)
    THEN
      r3_ifs[r3_fwd_iid - 1].spa.act' = SPA_SND
      AND r3_ifs[r3_fwd_iid - 1].spa.pkt' =
        r3_ifs[iid - 1].rpa.pkt
      AND r3_ifs[iid - 1].rpa.act' = RPA_NONE_EMPTY
    END
    AND r3_fwd_iid' = UNKNOWN_IF
    AND r3_fwd_rte' = RT_DEF;
END

// system instantiation
SYSTEM IpRipRoutingExampleInstance: IpRipRoutingExample();
```