An SCL-based Constraint Representation Language for Intrusion Detection

by

Ranjan Kumar Rahul

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
November 2017

Abstract

In this thesis, we have extended the SCL (Structured and Context Language) network protocol description language to describe the complex constraints for the network engineer. Previous SCL developed with the focus of penetration testing and not sufficient for constraint scenarios. The constraint scenarios include multiple-packet with order and environmental information. To address the current limitation of the SCL, we have proposed syntaxes which are declarative in nature. We have studied three different styles of syntaxes to handle constraint scenarios of an IDS (Intrusion detection system). The three syntaxes are based on Java expressions, QUEL and Prolog. We have represented three constraints for command and control systems such as ATC (Air Traffic Control) network using our syntaxes. The same constraints have been previously used by a constraint engine to demonstrate the capability of the IDS. We evaluate each of the syntax based on the four design guidelines for the domain specific language (DSL). The Java-based syntax shows better capability to represent constraints based on four DSL design guidelines. Finally, we show the mapping of the constraints represented in our syntaxes with the low-level DSL (Domain Specific Language) of the constraint engine. The mapping shows our syntaxes has all relevant information to translate into the low-level DSL.

Acknowledgments

With the utmost respect, I would like to thank my supervisor, Dr. Thomas R. Dean for his continuous support and guidance throughout my research at Queens University. My supervisor's incredible patience in explaining my queries during my research is commendable. I would also like to thank him for giving me enough time and showing patience during the start of the project. Finally, his availability for his students even on busiest times, shows his concern for his students. I will always cherish the learning and experience that I have gained working with my supervisor.

I would also like to thank my lab mates who were involved with the department of defense project along with me. Being the newest member of this project, they always showed there availability to help me with my queries.

My sincere thanks to Dr. James R. Cordy of the school of computing, the director of the ULSS program. He happens to be one the most humble person I have met in Canada till yet. His dedication to help and support students will always inspire me for the rest of my life to do the same in my career. His continuous support thoughtout the program with any queries regarding the program is incredible. I would also like to thank him for introducing to my supervisor and to this project.

At last, I would like to dedicate my work to my family and friends. Special thanks to Suhas for helping me at various stages of my work. I would like to thank Ruwanthi

and Eric for making my stay in Kingston a pleasant stay. I would also like to thank my parents, uncle and aunt for showing support in difficult times.

Statement of Originality

I, Ranjan Kumar Rahul, hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Even the best-protected networks can be vulnerable to the cyber-attacks. The ramification of these attacks can often lead to severe damages to the associated systems. Depending on the types of the attacks and the real-time dependencies of a network, different measures can be taken to protect the network. Continuous and periodical scanning of the network is often needed in order to identify potential attacks. After scanning, looking further into the behaviors of the networks might lead to interruptions and possible shutdown of the systems. Therefore, protecting a network is a challenging task for the network engineers.

Attacks on private networks can have catastrophic consequences. For instance, the Center for Internet Security (CIS), a government-endorsed non-profit organization, reports that in 2013 alone, 75 airport networks across the United States have been compromised by intrusion attacks [1]. Out of these 75 attacks 4 were identified as advanced persistent attacks on the ATC (Air Traffic Control) system. An advanced persistent attack is a kind of attack when an attacker gets unauthorized access to the network and remains undetected for a long period of time [52, 10, 8]. In an advanced persistent attack, an attacker may have the ability to steal sensitive data

or gain complete access to the network [10]. Another critical attack on the ATC is a jamming attack where the attackers may introduce a ghost flight, altering the position and projection of the flight on the radar [6]. These attacks can compromise the accuracy of flight related information on the flight management system on-board. A further example of an attack on private networks is an attack on a nuclear reactor using a computer worm [12]. For example, in Iran a nuclear facility was attacked using the Stuxnet worm which made the centrifuge to spin in a manner which caused permanent damage [12]. Networks such as the ATC network are also vulnerable to various other types of attacks such as the DDoS (Denial of service) [33]. These attacks on such highly secured facilities demonstrate that the risk of network attacks can be far more than just system shutdown or data loss [3, 40].

For better security of private networks, intrusion detection is required along with other security measures such as installing the anti-virus and firewall [26]. The goal of an intrusion detection system (IDS) is to protect these networks from outside attacks as well as monitor the internal behaviors of these networks. Post-deployment maintenance of an IDS system, is a tedious and costly affair. The maintenance activities include accommodating new network protocols and adding new defense strategies to handle new intrusion scenarios. Our IDS is designed to protect the private network such as the command and control system with limited network protocols. For the private networks with well-defined functionality and a limited number of protocols for communication, we can define the behavior of network traffic which does not change over time. Considering this, anomaly based IDS is a popular choice for private networks [7]. Our research group has developed an IDS for command and control system similar to the ATC system. Our IDS scan each of the network packets for potential

intrusion in the network. It is a constraint based IDS where the healthy traffic on the network is defined in terms of constraints and deviation from those constraints would be considered as an intrusion. Currently our IDS has constraints written in the C language for all the protocols. These constraints are based on a tree data structure whose leaf nodes are values from the packets. The constraints are checked with logical operations on these leaf values. In order to auto-generate the constraint engine code, Hassan [15] has developed a low-level DSL (Domain Specific Language) for constraint representation. The low-level DSL shows code-repetition and redundancy for the constraint representation. The primary purpose of the low-level DSL is to auto-generate the C code and was not developed with consideration for user readability and hiding the implementation complexity. The low-level DSL requires the user to know the tree data-structure and it contains code-repetition and redundancy for the simplicity of code generation. The low-level DSL design makes it more challenging for the user to learn and write code using the DSL. It is also harder to maintain code in terms of adding changes in low-level DSL code. The low-level DSL requires user to be aware of the under-lying tree data-structure to write constraints. In order to mitigate these issues, we have proposed high-level DSL representations on top of the low-level DSL. The low-level DSL can be inferred from the high-level DSL without knowing the design complexity of the constraint engine.

The goal of our representation is to provide a interface to write, modify, and manage the constraints for the network engineers. The network engineers should be able to write a constraints in abstract form using their familiar notation such as ASN.1. To do this, we use the motivation from domain specific modeling (DSM).

The domain specific modeling is often used when abstraction is needed for auto-generation of code. The DSL approach often increases the productivity of developers and reduces the effort of writing code [53, 54]. Using the DSM also reduces the defects in the code by removing any chance of human induced errors [55]. We used various DSL design guidelines such as reusing existing language definition wherever possible, keep it simple, avoiding unnecessary generality, reducing the number of language elements and avoid redundancy in representation. We chose the SCL (Structured and Context Language) for our representation as it is already been used for auto-generation of the parser code for our IDS. The SCL provides a platform where the protocol specification can be defined in terms of the ASN.1 notation [28]. The ASN.1 notation is already popular and well known by the network engineers. We used SCL for constraint representation along with the ASN.1 notation. The SCL provides extra capability of deeper packet inspection with its marker named as size, transfer and constraint [28, 60]. The proposed syntaxes of this thesis aim to provide following design goals for our constraint representation.

1. *Constraint scenarios*: It should cover constraint scenarios relevant to our IDS system. These scenarios include constraints with multiple protocols and using environmental information.

2. *Hide Implementation*: Constraint syntax should hide low-level implementation of the constraint engine which includes the complex tree data-structure.

3. *Simplicity*: The syntax should be simple to understand and follow some DSL design guidelines.

4. *Scope of Language*: The scope of the syntax is to represent the constraint scenarios limited to true and false output. This means our constraints are only used for conditional scenario verification and not for calculating an absolute result such as number output.

Considering these goals, we have proposed a high-level DSL representation for our constraint engine. Using a high-level syntax gives the users capability of quick prototyping of constraints [39]. A systematic auto-generation of code from the high-level representation also proves to show higher code quality [2].

1.1 Contributions

This thesis extends the current SCL to provide an interface to write constraints in simple and easy to understand format for the network engineers. The syntaxes proposed in this thesis provide the capability to the SCL to handle scenarios such constraints requiring multiple packets, verifying constraints by the order of packets in network traffic and to handle the environment related information of the network. We have looked into the declarative language paradigm to build new constraint syntax for our language. Declarative language has advantage over imperative while representing non-sequential information [11]. According to Llyod [24] "declarative programming involves stating what is to be computed, but not necessarily how it is to be computed". The goal of our constraint representation is to hide the low-level implementation details. Therefore, we propose three different syntaxes based on the declarative programming language. We used QUEL and Prolog which are declarative in nature and Java arithmetic expression which give us a representation similar to infix notations. We have presented three different constraints for each of these

syntaxes. The constraints used to demonstrate the three syntaxes are the same as mentioned in Hassan [15].

We also show the qualitative comparison of the three syntaxes based on the representation of three constraints. The criteria for our syntax comparison are the following: re-usability of information, redundant information, the number of elements, and repetition of code to represent constraint in each of the syntaxes [20]. Finally, we show the mappings between our Java-based expression and the low-level DSL representation of our constraint engine. The mapping shows that our representation contains all the relevant information to translate into the low-level DSL developed by Hassan [15] and, further for complete auto-generation of the C code for the constraints.

1.2 Organization of Thesis

1. *Chapter 2: Background*

   In this chapter, we discuss the current SCL and how it is related to ASN.1 notation. We discuss the feature and limitations of the current SCL in this chapter. We also mention some of the work which inspired our research as related work in the last section of this chapter.

2. *Chapter 3: Overview*

   In this chapter, we discuss the high-level design of the intrusion detection system and how our SCL based constraint syntaxes will fit with the system. We also discuss common design questions raised during the syntax design.

3. *Chapter 4 & 5 & 6: Each of the three syntax proposed for SCL constraint*

   In these three chapters, we have discussed the motivation and the design for the syntaxes, we have proposed for our SCL constraint. Each chapter is dedicated

to one type of syntax format to represent three different constraints. Chapter 4 is on the Java-based syntax, chapter 5 for QUEL-based, and, Chapter 6 is about the Prolog based syntax.

4. *Chapter 7: Evaluation and Mapping of Java-based syntax with low-level DSL*

   In this chapter, we have compared the three syntaxes bases on the four DSL design guidelines. Comparison shows the strengths and weaknesses of each syntax. Finally, we show the mapping between the Java-based syntax in SCL with the low-level DSL for the constraint engine by Hasan [15].

5. *Chapter 8: Summary and Future Work*

   In this chapter, we have summarized our thesis and mentioned the current limitations of the proposed syntaxes which might be useful during the next stage of the SCL development.

Chapter 2

Background

In this chapter, we discuss domain specific terminology and network protocols used for the constraints. We also mention components of current SCL and how to write protocol specification in the SCL.

2.1 Terminology and Concepts

1. *Binary protocols*: These protocols are often used for machine to machine communication, as it is much faster and efficient compared to the text based protocols. The same reason which makes it faster and efficient, also makes it harder for humans to understand these protocols. Protocol translators such as wireshark are required to understand these protocols. The binary protocols are more compact compared to a text based protocols. In contrast, the text based protocols such as HTTP, SMTP can be directly readable to the human from the data stream. For binary protocols the sequence of the bits not organized in same order as the sequence of the characters in case of transferring characters over the network, whereas in the text protocol, the bits sequence is arranged as the sequence of characters. In our study, we used the binary protocols for our

constraint such as IGMP and RTPS [13, 14, 45].

2. *Protocol Data Units*: The Protocol Data Unit (PDU) is the smallest unit of the network protocol to deliver information across the network. For each layer of the network, called the OSI (Open Systems Interconnection) model, these PDUs are different such as for physical layer, bit would be the smallest unit for transferring the information. For network layer, PDUs are the network packet for the protocols. Our IDS works at network layer so our smallest unit is the packets from two protocols named IGMP and RTPS.

2.2 SCL:Structure and Context-Sensitive Language

Before going further, it is important to understand the entities on which SCL is based on. SCL has taken a lot of ideas from ASN.1 (Abstract Syntax Notation One) and extended it to add more features [36, 28].

1. *What is ASN.1 (Abstract Syntax Notation one)?*

   ASN.1 is developed to provide a platform which is language independent and an internationally-standardized representation (i.e data structures at the high level of abstraction) for the network protocols. The ASN.1 notation is widely used to describe the notation for encoding and decoding of the PDUs, which is the smallest entity of the network protocols traveling in the network to share informations. The presentation layer of the OSI layer is where ASN.1 is referenced to specify the data structure of the PDUs exchanges during the network communication. Although ASN.1 is the notation for the purpose of machine communication, its simplicity provides a clean human readable

structure of the protocols. Industrial networks such as in the telecommunication and security domain have extensively used the ASN.1 notation for their communication network and other purposes. The ASN.1 is used to specify protocols such as SNMP and OSPF [5]. ASN.1 being platform independent used to handle scenarios where SNMP manager using Java data types where as SNMP agent listening written in C language.

These behaviors and features of the ASN.1 largely motivated the initial SCL development. SCL has gone through several phases of improvement to accommodate various scenarios of the network communication to build a protocol tester. The SCL was originally used to test OSPF, BGP, PKS (Public Key Server) and other similar protocols [34, 27]. Later, the SCL was extended to non ASN.1 protocols such as CIFS, AppleShare and DRDA [23, 35, 32]. To understand the description of the ASN.1 representation, we show an example of ASN.1 for a hypothetical protocol below [28]. As the concepts of classes and types are common in programming languages to represent complex structures, ASN.1 is no different and extensively uses these concepts. We can define our custom types like in other programming languages to refer it multiple times and for the code clarity. Example:

```
1 PROTOCOL DEFINITIONS ::= BEGIN
2
3 PDU ::= (net-info | room-loc)
4 header ::= SEQUENCE {
5   length  INTEGER
6   sppType INTEGER
```

```
 7 }
 8 net-info ::= SEQUENCE {
 9   headerNfo header
10   numOfSubH INTEGER
11   subheader SET OF IP-Address
12 }
13 IP-Address ::= SEQUENCE {
14   floor INTEGER
15   ipaddress INTEGER
16 }
17 room-loc ::= SEQUENCE {
18   headerNfo header
19   subheader SET OF room-info
20 }
21 room-info ::= SEQUENCE {
22   floor INTEGER
23   officeNumber  INTEGER
24   extension VISIBLESTRING
25 }
26 END
```

The ASN.1 features and keywords which are relevant for the SCL.

(a) **IMPORT**: Within an application, ASN.1 have multiple modules defined for different protocols. Keeping these modules separate makes the ASN.1 codes simple and easy to access. But to avoid

redundancy and repetition of code, ASN.1 provides a concept to import modules from different protocol specifications. This is analogous to the importing package concepts in the object oriented language. The IMPORT statement in ASN.1 provides the ability to use a specific module from a given set of the module definitions. We can see it in our example below where we import module B from from PDU description PDUB to the new PDU description PDUA.

```
1 PDUA DEFINITIONS ::= BEGIN
2 IMPORTS B From PDUB;
3 A ::= B
4 END
5
6 PDUB DEFINITIONS ::= BEGIN
7 B ::= INTEGER
8 END
```

As we can see from above example for the IMPORT in ASN.1, we can't import the whole PDU. We have to specifically name the module from other PDU, we want to import. Import statement should always be before any type or value statements.

(b) **EXPORT** : EXPORT is another important keyword which is used similar to IMPORT. It is important to know that, we import modules from different PDU specification only if modules has been defined with EXPORT before, Which means modules need to be exported before importing. We can say that EXPORT makes the

modules available for importing it later similar to the public in the object oriented design. Though it is important to know if there is no EXPORT defined inside the PDU definition which means all the modules are available for import. This is done because export is added later in the ASN.1 so to make import and export work from the previously declared definition. Thus, by default everything is exported if no explicit export is used in the PDU definition. Similar to import, Export is defined before type and value definition to be exported. Here is an example of the EXPORT statement.

```
1  PDU1 DEFINITIONS ::=
2  BEGIN
3  EXPORTS  Outlet, Address;
4  OutletType ::=  SEQUENCE{
5  A INTEGER
6  B OCTETSTRING
7  }
8  Address ::= SEQUENCE{
9  X INTEGER
10 Y OCTETSTRING
11 PIN OCTETSTRING
12 }
13 END
```

Another important point is that the *not* keyword makes all the modules inaccessible. We have to simply define all EXPORT

statements with *not* parameter to make those modules inaccessible. An export without any parameter means nothing is exported from that PDU specification. For example:

```
1 EXPORT ;
```

(c) **Type**: Types is a part of a module definition, where we define the attributes and modules based on the value we are expecting for a given protocol specification. There are a few IMPLICIT types which are INTEGER, BOOLEAN, OCTET STRING, REAL, SET OF, and the user-defined type named SEQUENCE. We have only mentioned the types from ASN.1 which are relevant to SCL.

- **INTGER** is primitive type referring to the numeric value. It should always be defined inside a module definition.
- **BOOLEAN** is another primitive type with two state value 0 or 1.
- **OCTET STRING** is used to define non-numeric in protocol description.
- **REAL** is used to define floating values in protocol description. The precision of floating point is defined based on the number of bytes defined in the module description.
- **SET OF** is a keyword used as the prefix before attributes which has the list of same primitive or user-defined types.
- **SEQUENCE** is a keyword used before any module definition in the ASN.1. It also makes it possible to use modules as the

user-define type and can be referred anywhere in the application, depending on exported or not.

```
1   PDU1 DEFINITIONS ::=
2   BEGIN
3   header ::=  SEQUENCE {
4     A INTEGER
5     B OCTETSTRING
6     flag  BOOLEAN
7     C SET OF DATAS
8   }
9   DATAS ::=( DATA1 | DATA2 )
10  DATA1 ::= SEQUENCE {
11    X INTEGER
12    Y OCTETSTRING
13  }
14  DATA2 ::= SEQUENCE {
15    X1  INTEGER
16    X2  OCTETSTRING
17  }
18  END
```

2. *What is SCL?*

SCL is motivated with the goal of making network protocol more reliable in terms of security [28]. It is designed to validate PDUs of network protocol

through their specification and data structure for network communication. It was developed as specification language for the testing of protocol packets for evaluating the network behavior. SCL incorporated the ASN.1 grammar in its design which let us use ASN.1 notation with the additional functionality of the SCL. To expand its capability beyond ASN.1 notation, SCL introduces XML markers along with the ASN.1 notation. XML markers are size, transfer, and constraint has been combined with the ASN.1 notation. Initially, SCL was capable of describing the design and semantic of a single packet of the network protocols through its marker. SCL has gone through improvement to increase the capabilities of SCL to handle certain scenarios of multiple packets [60].

First SCL can handle semantics of a single packet, which means it can validate packets based on static information such as fixed version or fixed value of field and relation of two or more field of the same packet. Later versions of SCL, have introduced the capability of building the relationships between multiple packets which is a packet field can be validated against another packet field in the series of packet moving across our network.

For example, lets see the example of sample hypothetical protocol and their representation in SCL for each of these markers. Example.

```
1 PDU ::= (net-info | room-loc)
2 header ::= SEQUENCE {
3   length  INTEGER <SIZE-DEFINED>
4   sppType INTEGER
5 }
```

```
<size>
  length is 2 bytes
</size>
net-info ::= SEQUENCE {
  headerNfo header
  numOfSubH INTEGER
  subheader SET OF IP-Address
}
<size>
  headerInfo is SELFDEFINED
  numOfSubH is 1 bytes
  subheader is CONSTRAINED
<\size>
<transfer>
  MATCHES(headerNfo.sppType == 1)
  CARDINALITY(subheader) = numOfSubH
</transfer>
IP-Address ::= SEQUENCE {
  floor INTEGER
  ipaddress INTEGER
}
room-loc ::= SEQUENCE {
  headerNfo header
  subheader SET OF room-info
```

```
30 }
31 room-info ::= SEQUENCE {
32   floor INTEGER
33   officeNumber  INTEGER
34   extension VISIBLESTRING
35 }
36 <constraint>
37   value(floor) = 1|2|3
38 </constraint>
```

2.3 Network Protocols

For our analysis, we have studied two protocols used by our constraint engine [15], IGMP [13] and RTPS(Real Time Publisher and Subscriber) [45] protocol. We have a complete SCL notation for IGMP in Appendix A.

To study these two protocols based on their implementation, Hassan [15] and ElShakankiry [9] used an air traffic control (ATC) system simulation, that used IGMP and RTPS protocols for the communication. Hassan used Michaud's [31] work on finding vulnerabilities in the publisher and subscriber specification. Michaud shows 69 vulnerability in his work which includes environment, configuration and constraint based vulnerability. Michaud uses same RTPS implementation for his analysis which is used by the ATC simulation for our IDS. Using Michaud's analysis and the ATC simulation data, Hassan [15] presents constraints for RTPS and IGMP to shows constraint engine capability. We have used the same constraint as motivation and evaluation of our syntax analysis.

2.3.1 IGMP

IGMP (The Internet Group Management Protocol), is used to manage the host and member system that is listening to host packet through multicast addresses. The routing devices in the network use the IGMP to learn where to send packets in the network. It keeps track of which physical subnet has members listening to the multicast address and passes packets to those subnets. The IP host continuously uses IGMP to report their membership to any near multi-cast routing devices.

At the junction of the network divergence, multicast packets are replicated to be sent to different subnets. These multicast protocols are very effective in reducing the traffic congestion by delivering source traffic to multiple receivers without one to one communication. It is useful in various real-time scenarios like video-streaming where a host sends video packets to multiple viewers at a time, and maintaining the command and control system where multiple systems are connected to one command center.

IGMP has various types of packets for various purpose. The type of the IGMP packet is determined by the *type* field in the packet. The code and meaning of each type field are mentioned in the Table2.1.

Keywords

Type Code	Packet Type
0x11	IGMP Membership Query for all the versions
0x12	IGMP V1 Membership Report
0x13	Distance Vector Multicast Routing Protocol (DVMRP)
0x14	PIM version 1
0x16	IGMPv2 Membership Report
0x17	IGMPv2 Leave Group
0x1e	Multicast Traceroute Response
0x1f	Multicast Traceroute
0x22	IGMPv3 Membership Report

Table 2.1: IGMP packet Type

As IGMP has been part of continuous improvement since its existence. It has three versions, IGMP V1, V2 and V3. For our study, we have packets for two version, V2 and V3. Once multicast groups are set-up for the network. The following process continues keep track of all the systems connected in this network.

1. To join IGMP multicast group, a system has to send a membership information through IGMP join packet.

2. Multicast router sends a membership query at periodic interval to the multicast group.

3. Once the system is joined the multicast group, it has to respond to membership query request with the membership report to that multicast group.

4. If a member doesn't respond to two continuous membership query request. The router assumes member left that multicast group and remove it from the multicast listening group.

5. Member of the multicast group can also leave the group by sending a Leave packet to the multicast group.

The above steps of IGMP protocol is based on the our ATC simulation, as protocol implementation varies from its standard specification instruction. For instance, in the IGMP protocol a membership query should have response from one system listening to that multicast group. In actual implementation all the system response with membership report. This is done to keep the IGMP implementation simple. Our constraint are based on the actual behaviors of these protocols in the network not on the standard protocol specification.

2.3.2 RTPS(Real Time Publisher Subscriber)

RTPS is based on the DDS (Data Distribution Service) standards proposed by Object Management Group (OMG) for machine to machine communication. It is developed with the aim of providing scalable, real-time, high-performance data exchanges using the publish-subscriber pattern. It has various applications in the transportation system, air traffic control system where various machine units need to communicate with each other for proper functioning.

The RTPS is designed to run over a transport protocol like UDP/IP which is unreliable but faster. The specific goals of RTPS development can be as follows:

1. Scalability of the network system, adding a new system should be as easy as plug and play connection and which enables the automatic discovery of the

new system. A system can leave or join these networks without any changes or disruption in the normal operation of the network.

2. Fault tolerance by making system proof of single point of failure.

3. It provides the ability to maintain the quality of service(QoS) to enable reliable real-time communication.

To establish a publish-subscriber communication system, RTPS runs the list of domain participants. A domain participant packet will always be the first packet from new system joining in the DDS system. Domain participant shares their endpoint information through which they receive or send RTPS packets later on. These end points can either be as the publisher (writer) or subscriber (reader).

Once the participant joins the publish-subscriber service using the domain participant, then they can behave as the publisher or the subscriber of a topic. To become publisher of a topic, a system has to send Writer packet (Data(w)) and to become a subscriber of a topic, they have to send reader packet (Data(r)) with the topic information, QoS etc. Between publishing and subscribing they are other logical messages exchanged among the systems for synchronization and reliability of the communication.

Three important packets based on the sub-messages of the RTPS packets for publisher and subscriber service to work:

- **Data(r)**: The data reader packets are those packets which have sub-message with reader type, which means this packet is generated by subscriber system. A valid Data(r) packets must have topic and key information to join DDS service as the subscriber.

- **Data(w)**: The data writer packets are those packets which have sub-message with writer type, which means this packet is only generated by the publisher system. Similar to Data(r) packets, Data(w) packets must have topic and key information for which they would be publishing.

- **Data(p)**: The data participant packets are required to be sent by all the systems before joining as publisher or subscriber. A single participant can be publisher and subscriber at the same time given that they have different publishing and subscribing topics.

The RTPS protocol uses five logical messages for maintaining the reliability of communication:

- **ISSUE**: Contains the application's user data. ISSUE packets are sent by publisher to one or more subscribers.

- **VAR**: Contains information about the attributes of an entity, which is part of a composite state. VARs are sent by CSTWriters to CSTReaders [45].

- **HEARTBEAT**: Describes the information that is available in a Writer. HEARTBEATs are sent by a Writer (Publication or CSTWriter) to one or more Readers (Subscription or CSTReader).

- **GAP**: Describes the information that is no longer relevant to Readers.

- **ACK**: Provides information on the state of a Reader to a Writer.

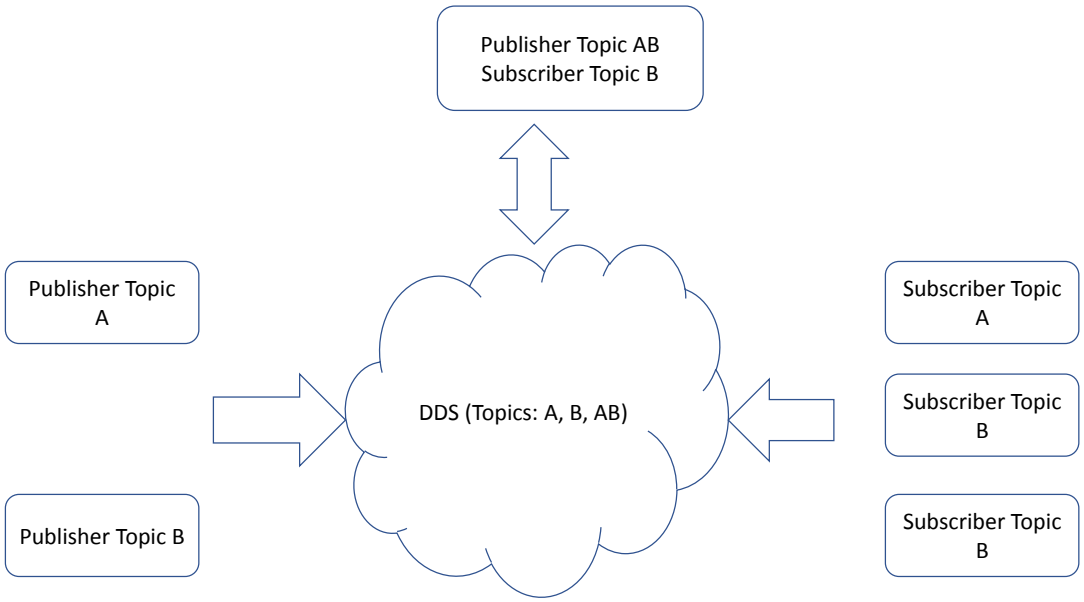

Figure 2.1: Publisher subscriber system based on RTPS communication

In the Figure 2.1, we have the total of six systems as publisher, subscriber or both. We have 3 publishers and 4 subscribers, though we have only 6 participant packets. As we can see in the Figure 2.1, we have one participant system as both publisher and subscriber but they are linked to different topic thus they are the valid participant in this publisher and subscriber pattern communication.

2.4 Related Work

Syntax based validation of the protocol for security vulnerability is not a novel idea. Previous studies have used syntax based technique for communication fault detection, protocol related security issues. Our related work section is divided into two sub-section.

2.4.1 Model driven programming

Modeling has been part of the study of formal methods and model driven engineering community. Modeling is a concept of presenting your system functioning into simple abstract form. Model driven is aimed to hide the implementation of low-level implementation and exposing components as reusable modules[44]. Some popular model engineering language and tools are UML by OMG [22, 56, 21] and SysML by OMG, an another variant of UML. Model driven programming has been popularly seen as the graphical based system where modules are connected through GUI based system. It also termed as boxology (i.e, boxes and arrows). But completely relying on graphical notation made these model based language difficult to debug and analyze. Bringing extensive simplicity comes with lack of flexibility in design. Above graphical notation based platform drawbacks can be solved to a certain extent by using text based notation, where boxes can be represented by a small abstract piece of code, parameters of modules are easy to maintain in text based notation as adding or removing parameter in text based notation require less design changes in model driven platform. Language such as Scala programming language shows an abstract representation for including object oriented and functional programming concepts together at one platform [38, 37].

Fernandez [14] presents a thorough theoretical study on developing a model based framework for application protocol robustness testing. In their study, they propose two types of test-cases which return the verdicts (similar to alert generation) pass or fail. Pass test case in this framework is generated to good packets, whereas fail test cases are written based on failure scenarios, where conditions for bad packets are checked against each packet. On success of fail test cases they generate fail verdict

alerting for bad packets, whereas on success of pass test case no verdict is generated. Pass test-cases generate verdict on their fail to execute the scenarios.

Similar work on model driven language developed by Havelund [16] to represent textual notation for SysML through a language name K. The K language support SysML feature such as classes, multiple inheritances and set theory. Further in their implementation they translate K syntax into SMT-LIB[4, 41], which is a lower level representation.

2.4.2 Syntax based representation of protocols

The PROTOS project by Kaksonen [19] uses a syntax based technique for generating a large variant of PDUs for assessing the security of a protocol implementations. Generating all possible variant of PDUs packets for a given network protocol is not cost effective and efficient. PROTOS provide a platform to generated those packets in cost effective and efficient way. Similarly, Jing's work [18], uses protocol representation in transitional state based verification of protocol communications. The main purpose of their work is to generate test-cases for network robustness and reliability. Jing considers the multiple flows of packets in the network to verify the correctness of those packets. Jing's work includes fault detection for application layer protocols such as OSPFv2, HTTP, which handle human user level inputs. Their study also considers the communication of single protocol at a time.

Saad [43] presents a framework for detecting the fault in TCP related protocols. In their paper, to describe the test scenarios they have used Specification and description language(SDL). In Saad's paper, they have mentioned three types of test cases which are Pass, Fail and Inconc. Three types of test cases represent different

scenarios through which fault can be detected. First scenario, invalid input: packet with having faults such as error in any field, wrong sequence number and wrong destination addresses. Second scenario, the communication system having inopportune inputs: a valid packet is generated at an unexpected time such as Acknowledgement packet generated after establishing the successful connection. And third scenario, possible reaction of the system after receiving inopportune inputs which might lead to unwanted behavior of the system. These test has only been done on a single TCP protocol to evaluate their system, a scenarios where multiple protocol working together to perform a task has not been addressed in their work. Similar work has been done at Cisco by Xiam [58] for multicast protocols. They have used BNF(Backus-Naur form) notation for representing their test scenarios for fault detection. They proposed a framework which works on matching the field value with the set of an option provided. Using this framework, representing test-case based on multiple protocols are not possible. This framework designed to handle scenarios for only single multicast protocols.

2.5 Background Summary

This chapter introduces important concepts in this field of study and discusses the basic of SCL, a platform for network security testing. The network protocols IGMP and RTPS used for the constraint representation. In this chapter, we also mentioned few works in field of domain specific modeling language design, which is relevant and close to our study. These related work has been part of the motivation for our work in this thesis.

Chapter 3

Overview

To understand the role of SCL in the IDS, it is important to know the two modules of the IDS system. The modules are the parser [9] and the constraint engine [15]. A general idea of each of these IDS modules will give us better idea how these modules are connected to SCL.

1. *Parser and SCL relation*

   A parser is the first part of the IDS system which takes network packets for analyzing. The parser intercepts the network traffic to do initial verification of the network traffic. It inspects the packets to verify the structure of the packets based on the protocol implementation. The parser is written in C language. For the network engineer, the parser has to be maintainable with continuous change in network design such as addition of new protocols. ElShakankiry [9] auto generates the parser code from the protocol specification written in SCL. The parser code generator uses three parts from SCL, namely the ASN.1 notation, size and transfer markups. The SCL markup enables the parser to inspect a wider ranged packets compared to the

simple ASN.1 notation. The Parser verifies not just the structure and the type of each field in the packets but also the size of each field through SCL size markup. The transfer markup enables the parser to do some verification for the fixed information such as version and flag values. In addition to these properties, the simplicity of SCL to represent packet structure makes it a reliable choice for auto generation.

2. *Constraint engine and building relation with SCL*

   The Constraint engine is the module where intra-packet and intra-protocol constraints are inducted to ensure no intrusion in the network. This is the part of IDS system where complex scenarios of network behavior are verified to ensure intrusion free communication. Like the parser the constraint engine is written in C, which will be auto generated. Hassan [15] has proposed a low-level DSL (Domain specific language), which is written for the purpose of auto generation of the constraint engine code. The low-level DSL is designed for and has redundancy which makes it hard to understand. It is not human comprehensible and designed with purpose to auto generation. For example, we can see the Hassan's DSL representation of sample constraints, which shows repetition of information, requiring knowledge of tree structure and knowing the type information of each field of the protocol specification.

   Therefore, in our new syntax through SCL, we provided a simpler syntax for representing constraints. Incorporating new constraint syntaxes with SCL provides a single interface for our IDS system do all the changes. Hence, it makes our IDS scalable and easy to manage.

```
1  INSTATNTIATE
2  IGMP V2member or V3Member
3  if V2member then
4  SrcIPJ = Packet.SrcIP,
5  groupIPJ = Packet.groupaddress
6  end
7  if V3Member then
8  loop Packet.groupRecordInfo SrcIPJ = Packet.SrcIP,
9  groupIPJ = Packet.groupRecord
10 endif
11 Key = Packet.SrcIP, Packet.groupaddress
12
13 BIND
14 RTPS Packet.SUBMSG contains DATAPSUB
15 SEARCH Packet.SrcIP, Packet.DstIP
16 SrcIPP = Packet.SrcIP,
17 DstIPJ = Packet.DestIP
18 Key = Packet.SrcIP, Packet.groupaddress
19
20 EVALUATE
21 RTPS Packet.SUBMSG contains DATARSUB or Packet.SUBMSG contains
      DATAPSUB
22 SEARCH Packet.SrcIP, Packet.DstIP
23 EVAL Packet.SrcIP, Packet.DstIP
```

```
24
25 DESTROY
26 IGMP V2Leave or V3Leave
27 if V2Leave then
28 SEARCH Packet.SrcIP, Packet.groupaddress
29 endif
30 if V3Leave then
31 loop Packet.groupRecordInfo
32 SEARCH Packet.SrcIP, Packet.groupaddress
33 endif
```

An overview Figure 3.1 shows how SCL constraints are connected with constraint engine through low level domain specific language for constraint engine. The overview diagram shows the connection of SCL and its entities relation with other components of the intrusion detection system. The constraint engine primarily uses constraint markup along with other ASN.1 notations in SCL for inferring information for the low-level DSL, whereas the parser uses ASN.1 notation along with size and transfer markup for its auto generation of the parser code. Our IDS system operates in two different environment which is called as Learning and Checking mode.

3.1 Two modes for Intrusion Detection System Operation

1. **Learning Mode**: is simulation of real-time environment with real-time data. During the learning mode, constraint engine is run through previous network

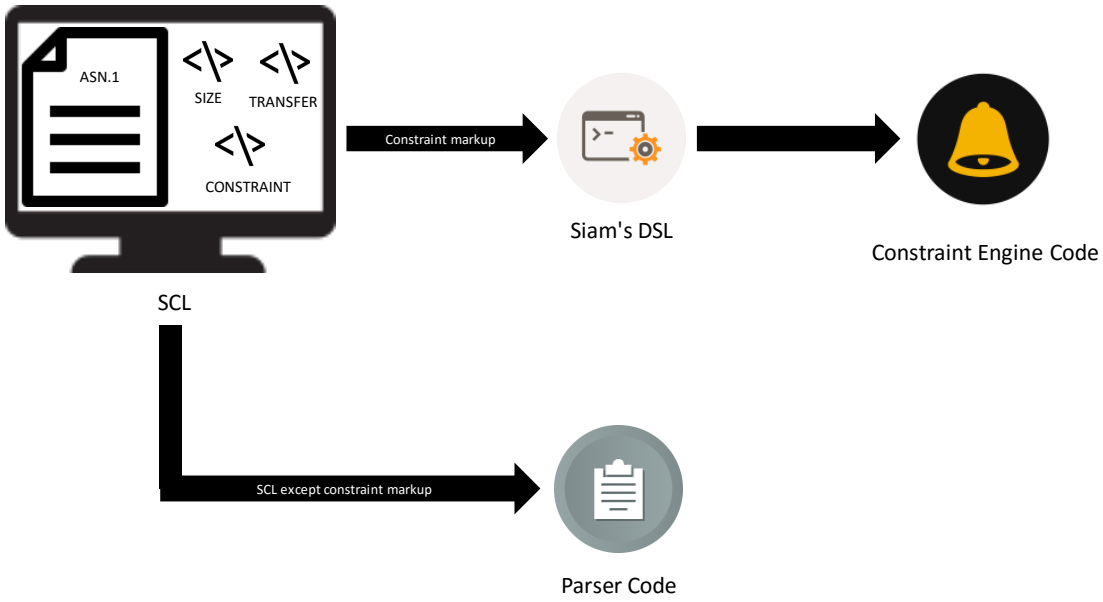

Figure 3.1: IDS system overview with SCL interface

data in real time environment. Learning mode is a clean and an attack free environment which help us to define healthy behavior of these networks. During this process, we store static information related to the system which will be used later for constraint checking in real environment. For example, source and destination IP, Topic information for RTPS protocols are some of the attributes which can be considered as static information, as this information doesn't change often for network such as ATC system.

2. **Checking Mode**: is the actual working environment where real time data from actual system is fed into the constraint engine. Checking mode use the static information collected during learning mode for constraint verfication. Storing static informaiton during checking mode simplifies the constraint as well as

reduce the time for running a constraint for the relevant packets.

To make constraints even simpler, we use the same constraint representation for both learning and checking mode.

3.2 Various Scenarios of Constraints

There are various constraint scenarios. To build the constraint representation, we present the three types of scenarios which cover single and multi packet constraint representations. These scenarios are as follows:

3.2.1 Single packet Single protocol

These constraints are quite simple. It can be easily handled by our already existing SCL constraint representation [60, 59]. Single packet constraints are based on a packet attributes such as field values and size. All comparison of packets attribute are kept within one packet boundary. SCL size and transfer markup handle lot of constraint around single packet and single protocol such as endianess of RTPS packets. Though, our IDS system include single packet constraint, we do not need to change anything in our SCL constraint representation.

3.2.2 Multi packet Single protocol

As the name suggests, these constraints are based on multiple packets of a single protocol. In this type of constraint, more than one packets is required. A protocol can have different types of packets. For example IGMP has packets such as Join report, Leave report, membership query. A Multi packet constraint can use different types of packets at different interval in the stream of packets. This mean that a

constraint can be between two Join report packets at different interval in the packet stream. Our current SCL constraint representation partially handles multi-packet scenarios. At Present SCL can refer the previous packet in the stream of packets but cannot handle scenarios of more than two packets where the order of packets is relevant. To handle this scenario, we have considered the order of all the previous packets from a given reference packet. With our new SCL constraint representation, we can write constraint with n-number of packets with keeping track of their order.

3.2.3 Multi packet Multi protocol

These constraints comprise of complex scenarios where multiple protocols are running on a single network for communication. A individual protocol is different and does not have any dependencies with other protocols. When multiple protocols work together in a network, their combined behavior must be verified. In these constraints, we take packets from multiple protocols to build a single constraint. While doing so, certain parameters such as source or destination for a packet can depend on packets from different protocols. Our current SCL is not built to handle such scenarios. To represent constraints with multiple packets from different protocols, we have introduced module referencing system in our constraint representation, which lets us access any modules from different protocols seamlessly. We use the ASN.1 EXPORT and IMPORT statement to make relevant modules available for constraints. Once imported these modules can be referred to in our syntaxes.

To make our constraint representation simpler, we introduce the concept of variables to the SCL.

3.2.4 Environmental Variables

For a private network such as an air traffic control system, there are various systems that have fixed environmental information such as the source IP of radar data and other components. The static information is tightly bound with these systems and rarely changes. At the same time, these networks might have some application level information which changes when the system restarts and should be handled differently than static information. To handle these scenarios in our constraints, we propose an efficient way to track this information. We introduce the concepts of static and dynamic variables in the SCL constraint markup.

1. **Static Variable**: To handle the information which rarely changes over time for the network data, we have introduced static variables. Static variables stores values such as source and destination and topic information for a component such as radio system. Once these informations are stored as a static variable it is easily accessible during the constraint building and further simplifies our constraints. In our scenarios, we have two modes for running our constraint engine, learning mode and checking mode. Static variables are handles differently during each mode. During the learning mode, we simulate a network scenario of healthy behavior, where we store all the static information in variables. We use these stored variables during checking mode without recreating them again.

2. **Dynamic Variable**: There are informations in many protocols which is auto generated every time system start itself. Once these informations generated, it does not change until next reboot of the system. The process of the reboot of private network such air traffic control system is not frequent. The dynamic variables are behaves same independent of mode of operation. This means

that, dynamic variables are created and destroyed every time system restart. Therefore, named dynamic variables.

To further simply the constraint representation, we have introduced a annotation based concept which determine the types of constraints.

3.3 Types of Constraints

We categorize constraints based on their ability to represent the healthy or unhealthy scenarios of the network. It is called a pass or fail annotation of a constraint.

1. **Pass Constraint** Pass constraints represent a healthy scenarios for network traffic. A success of pass constraint without any error represent packets parsed by IDS system is valid, and no alarm should be raised for intrusion. On other hand, failing of these constraints signifies malicious packets in the network and alarm should be raised. These are written to identify scenario for the healthy packets in the network. A success of single pass constraint for packets does not imply that packet is intrusion free. This means a packet might be valid for a constraint but fail with another constraint. Therefore, Pass constraints are written to cover all possible healthy network traffic.

2. **Fail Constraint** The fail constraint are those which are written to identify the malicious packets in our network. Passing of the fail constraints signifies that the packet is malicious and alarm should be raised. The main reason to introduce the fail constraints is not to identify the invalid packets as it can be

done by pass annotation as well. Fail annotation is to make constraint representation easy for constraint writer where writing pass constraint become complex. For example, scenarios where the pass constraint might require more packets compared to the fail constraints. Another example, to cover the healthy traffic we might require multiple pass constraints, whereas one fail constraint might do the job for us.

3.4 Backward Constraint and Current packet

Our syntax is based on looking backward from the current packet. This means that for a given constraint it can be assumed that all the packets required for the building of the constraint has already been received. To make it work as backward looking constraints, we have introduced the implicit concept of the *current* packet. The current packet is implicitly defined in our design which removes redundant code for explicitly defining and maintaining the current packet. This is also the reference packets for all the previous packets in a constraint.

3.4.1 Concept of Current packet

In backward constraint logic, a current packet is one which is most recent in the sequence of packet required for the constraint. Therefore, network engineer writing constraint works with the assumption that all other packets will always precede current packet in a constraint. To build and incorporate this concept in the SCL design, a constraint markup for given constraint is located in the ASN.1 notation of the module representing the current packet for the constraint.

From Figure 3.2, in which a constraint required three packets from the stream of

packets to build its constraint. In the time series, the current packet is one which arrived latest and other two packet P1 and P2 precedes current packet. To build this constraint, SCL constraint markup with our syntax representing constraint would be put along with ASN.1 notation representing the current packet.

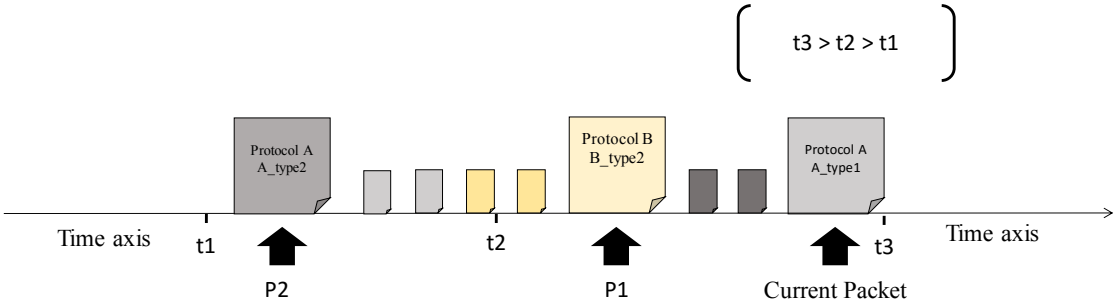

Figure 3.2: Backward constraint and Current packet design

3.5 Representation and Evaluation

Three syntaxes are proposed in this thesis are checked against three constraints. These three constraints based on RTPS and IGMP protocol implementation. The ATC system simulation has been used to generated the RTPS and IGMP packets. These three constraint cover all the scenarios discussed in this chapter.

The representation of three constraints using three proposed syntaxes are used for

evaluation. In evaluation, we compare three syntaxes on the basis of four DSL design guidelines that is redundancy, number of elements, code repetition and re-usability. We show which of three syntaxes is able to adhere most to design guidelines while representing all three constraints. Finally, we pick the best syntax to show mapping of proposed syntax with low-level DSL.

3.6 Overview Summary

This chapter laid the outline for the design of syntax and the requirements and limitation of current SCL which we are addressing through our research. We show high-level diagram how SCL is connected to other modules of IDS system.

We have defined network scenarios covered through the syntax representations presented in this thesis. To address constraint scenarios and keep syntax simple, we have introduced few general concepts for syntaxes. All three syntaxes uses annotation and they are based on backward constraint. Concepts of environmental variables is introdu ced for all three syntaxes.

Chapter 4

Java-Based Syntax

SCL is based on the ASN.1 notation. Considering that SCL derives its motivation and concepts from ASN.1 such as modular representation for the protocol specification. The SCL modules can be seen as object-oriented design, where each module can be treated as an object. These modules comprised of implicit attributes or references to other modules. The implicit attributes are INTEGER, STRING OCTECT same as ASN.1 data types. Similar to object-oriented design, modules from a given protocol can be imported in another protocol definition. Once imported, these modules can be used across the protocol description. These object-oriented similarities shown in SCL design, therefore, we choose syntax based on Java expression. Using the Java-based syntax allow us to use modules and packet types as object. We can refer module attributes in our syntax directly with their reference name. Using Java-based syntax, we can form logical relation among different modules in our syntax.

In our syntax, we have introduced a concept of implicit attributes which can be accessed by any modules in the SCL. These implicit attributes can be seen as the inheritance of object class in Java. These attributes are physical layer information of the packet such as source IP, destination IP and other physical layer information.

These informations are associated with all the packets for a given protocol description. These attributes are not required to be mentioned explicitly in the protocol description.

Previously, Java-based expressions have been used with UML based language to introduced security authorization. Lodderstedt et.al[25] presents UML based language for security named SecureUML. Their goal is to provide an interface for role based authorization using UML design. They have also proposed auto generation of backend infrastructure for this authorization constraint using UML block informations. To present their authorization constraints they used similar Java-based expression using Object Constrained Language (OCL).

Our Java-based syntax derives its motivation from Java logical expressions. In this syntax, we have assumed modules defined for each packets as Java objects, and their attributes are either variables or object references. Using the Java-based syntax simplifies our representation to a great extent. The SCL modules when seen as objects, makes it easy to use the SCL attributes directly in Java expression. Another reason of using the Java-based expressions is that we already have packet definition in our current SCL definition and doing any thing to represent this information in any other format will be redundancy in our representation and may confuse the user. The Java-based syntax give us access to write our complete constraint in one single expression. We describe the Java-based syntax in next section using an example.

4.1 Abstract Syntax and Grammar

The four major parts of the Java-based syntax are:

1. **Name** {*ConstraintName*}: The first part is naming the constraints which can

be used for various operation such as ordering of constraints. The constraint name is also used as reference for reusing this constraint for other packets.

2. **Annotation** {*annotation*}: Annotation in the constraint definition tells whether the constraint is a pass or a fail constraint. Constraint engine use this annotation to determine weather to treat the constraint as a healthy scenario or a malicious scenario.

3. **Packet Declaration** {*Declaration*}: The declaration is first of three parts of constraint definition. It assign variable with packet and sub-packet reference for writing a constraint logic. It also declared the packets which are with *or* condition.

4. **Logical Expression** {*ConstraintLogic*}: The second part of constraint definition is constraint logic which is a arithmetic expression between packet attributes. We access packet attributes similar to object accessing their fields in object-oriented concept.

5. **Packet Order** {*Ordering*}: The third part of constraint definition is the packet orders. This is similar to declaration, where we assign value to the implicit attribute *order* of all the packet. This determines how previous packet precede from the current packets.

The structure of our constraints as follows:

```
1 <ConstraintName>
2 <annotation>
3 <begin><constraintDefinition><end>
```

The constraint definition in Java-based syntax for representing packets and relationship among them. The Constraint definition grammar is as follows:

```
1  <annotation>  := pass/fail
2  <begin>   := '('
3  <end>   := ');
4  <ConstraintDefinition>  := <Declaration> ';'
5        <ConstraintLogic> ';' <Ordering>;
6
7  <Declration>  := <ListOfassignment>
8  <listofAssignment>  := <assignment>','<assignment>
9  <assignment>  := <variable> '='<protocolModule>
10 <protcolModule> := V2Member
11                 | V3Member
12                 | V2Report
13                 | V3Report
14                 | V2Join
15                 | V3Join
16                 | V2Leave
17                 | V3Leave
18                 | DATARSUB
19                 | DATASUB
20                 | DATAWSUB
21
22 <ConstraintLogic> := <expression>
```

```
23 <expression>  := <expression><airthmeticOperator><expression>
24     | <variable><airthmeticOperator><variable>
25 <airthmeticOperator>  := '>'
26                         | '<'
27                         | '=='
28                         | '||'
29                         | '&&'
30
31 <Ordering>  := <varaible> '==' <digit>
32 <value>   := [0-9]+
```

The syntax details is discussed through a hypothetical example. In our hypothetical constraint, we have a protocol of type A and it has 3 types of packets packet1, packet2, packet3. Each packet represents a module in SCL. For a given constraint, a packet1 packet should always arrive before packet2 in the packet stream. The packet1 packet source IP should be equal to the packet2 packet source IP. In this case, we have two packets for a constraint, of which packet2 would be our current packet where constraint would be placed.

```
1 PDU ::= (packet1 | packet2)
2
3 header ::= SEQUENCE {
4   length    INTEGER
5   srcIP    STRING
6 }
7 packet1 ::= SEQUENCE{
```

```
8   headerInfo    header,
9   data    OCTECTSTRING
10 }
11 packet2 ::= SEQUENCE{
12   headerInfo    header,
13   data    OCTECTSTRING
14 }
15 <constraint>
16   C1:
17   @pass
18   (p1 = packet1; p1.headerInfo.srcIP == srcIP;
19   p1.order == 1;)
20 <\constraints>
```

In the example, we have created a reference for the packet1 and packet2 which represent packets for protocolA. In the example the current packet is packet2, which would be the latest packet in the packet stream for our constraint. Thus, we write our constraint right below the module represting packet packet2 in SCL. The current packet reference is implicit and all the attributes of the current packet can be directly accessed with its attributes reference name. Our expression has four parts as seen in the example above, First, we decide name C1 and annotation which is **@pass**. Then, we define all the reference of modules for all the packets from different protocols, after references we build conditional statement, where comparison among those packet fields is done. At last, we put the order of the packet in which we expect packet to arrive for the constraint to be true using **order** keyword. Order value one means

that packet1 is first previous packet from current packet. In other words, packet1 is ,most recent previous packet in series of previous packet required for our constraint. In above example, we have only one previous packet, therefore, we have order for only one packet. We have introduced some modification in the actual Java expression to handle our scenarios. Those modification are in terms of new keyword and syntax changes introduced in regular Java expression.

4.1.1 Keywords and Concepts

We have introduced some modification in regular Java expression syntax for our syntax. These modification help us cover scenarios relevant for our IDS system. We have introduced one syntax modification and one keyword for the syntax. The syntax modification is called *or* condition and keyword introduced is called *order*.

1. *OR*: In the complex scenario, where the order of packet is not predetermined and the same constraint might be true with multiple types of packets, we provided an *or* condition at declaration part of the syntax to declare multiple previous packet with *or* condition. This *or* condition require to declare all the possible packets in the reference and subsequently in ordering part of syntax.

2. *Order*: *Order* is a keyword which is used in association with the module definition for a given protocol in the SCL. It defines the order of the previous packet for a given constraint to satisfy. It can be used as a packet attributes or a SCL module attributes associated with that packet. It is assigned as fixed positive integer value. The larger the value of *order* the further that packet is from current packet.

Example for the *or* and the *order*, we have a hypothetical example, for a current

packet of type V2Join, the constraint should check for the previous packet of type either a V2Member or V3Member. Here, we identify the previous packet which can be either a V2Member or V3Member. To represent this scenario in our syntax, we first define packets with the *or* condition in the declaration part of the syntax. The logical expressions are written considering the *or* condition of the previous packets. Using the *or* condition in our constraint reduces the effort of rewriting the constraint for each individual packet and also reduces false alarms. Finally, we define an *order* for these packets, we define order of packets using *or* statement similar to declaration part of the syntax.

```
1
2  V2Join ::= SEQUENCE {
3    ...
4  } (ENCODED BY CUSTOM)
5  <constraint>
6    constraint1:
7    @pass
8    (v2Mem = V2Member | v3Mem = V3Member;
9    (v2Mem.groupAddress == DstIP
10   | v3Mem.groupAddress == DstIP);
11   v2Mem | v3Mem == 1;)
12 </constraint>
```

Keywords for The Environmental Variables

To initialize and access the environmental variables, we have introduced two functions namely static and dynamic. The static and dynamic functions are defined based on the concepts of the object oriented programming. A variable in our Java-based expression syntax is declared and initialized with its first use in any constraints. A variable declared in a constraint are available to be used across the application by the other constraints. Static and dynamic variables are explained below with their respective functions.

1. ***Static Variable***: Static variables works differently in the two modes of the constraint engine which are learning and checking mode. First, during the learning mode, static variables are populated with static environmental information about the network components. Once the static variables are populated, its values are saved and used in next stage, which is checking mode. During checking mode, static variable information does not change and is used for checking the constraints. Checking mode is where these variables does not change and are used for verification. To use the static variable, we have to use it with a function named *static*. This parameterized *static* function simplifies the code by keeping the same code for learning and checking mode.

   Our static variable is a parameter for the implicit function *static*. In example below, we want to store all the source and destination IP for the devices in the network as static variables. To achieve this goal for the IDS system, we have two modes for constraint engine to work. Static variables are defined as an object with multiple fields, which means we define a static variable as an object which store static values associated with the name. A single static variable can store

multiple fields such as source IP as srcIP, destination IP as destIP and can be associated with the simple static variable which can also be seen as the static object. To access static information from this object, static function takes this *object.field* as a parameter. Explicit declaration of static variable structure and its field are not required. Structure of static variable is defined based on the use and field associated with in the constraint declaration.

```
1 <constraint>
2   C1:
3   @pass
4   (packet1 = p1;
5   packet1.headerInfo.srcIP == srcIP & static(dataFrame.srcIP) ==
     srcIP;
6   packet1.order = 1;)
7 <\constraints>
```

2. ***Dynamic Variable***: In contrast to the static variable, dynamic variable works same for both learning and checking mode. Similar to the static variable, dynamic variable access and declaration is done using an explicit function named *dynamic*. The dynamic variable keeps track of the information which is partially static properties which mean, these values once generated during the start of the application doesn't change during the operation of the network until all the system stopped completely and restarted again. These dynamic variables are more complex to handle as it stores the information at first appearance and uses later for constraint verification. Similar to static, our variable and its fields is a parameter for implicit function *dynamic*.

A dynamic variable is different from static in its declaration syntax. First, the dynamic variables are represented in terms of key-value pair. The dynamic values are associated with a key which means corresponding to key there might be multiple dynamic fields. For instance, in our example, the key is source IP which doesn't change but entityId is something which is dynamic for every time system starts. The reason for having a key value system is to ensure that we instantiate our dynamic variable only once and there is no repetition of values.

```
1 <constraint>
2   C2:
3   @pass
4   (packet1 = p1;
5   packet1.headerInfo.srcIP == srcIP,
6   dynamic(dynamicFrame(srcIP).entityId) == entityId;
7   packet1.order = 1;)
8 <\constraints>
```

4.2 Constraint for RTPS and IGMP

Packets for RTPS and IGMP protocols are generated by the ATC simulation. Constraints are written based on these simulation packets. We mention the steps for analyzing the constraints to write in our Java-based syntax. We should three examples of the constraints covering multi-packet and environmental variables scenarios using these two protocols. To represent the constraint in our Java-based expression, we recommend following points for the network engineers to build their own constraints and represent in our syntax. These points are not a strict rule for representation but

helpful steps for beginners.

Steps for representing a constraint in the Java-based expression syntax are as follows:

1. The constraint might be associated with one or more protocols. Identify each protocol and the packets required for validation.

    - Identify the SCL modules based on the packets associated with the constraint.
    - Based on a constraint, identify the order of packets for the constraint to be true.
    - From all the selected modules, identify the current module. The current modules are one which is associated with the most recent packet in the constraint.

2. Once the SCL modules are identified, we can focus on the relationship between these modules. These are called conditional logic among constraint packets.

    - A relationship among module attributes can be expressed as a conditional statements which uses logical operation such as equality and comparison. The relation among attributes are joined together with orand relations.

3. Finally, we decide the order of packets for our constraints.

4.2.1 Constraints Representation

Using above steps, we build following representation for our ATC simulation data collected for the IGMP and the RTPS protocols. We have same three constraint used by Hassan for analysis of the constraint engine [15].

1. *All the publishers and subscribers must be valid members of at least one IGMP multicast group. These participants should send their membership reports to specific group addresses before showing their interests in a topic.*

   The above constraint has current packet as a publisher or a subscriber, therefore we have two current packets. We will define our constraint at one of these packets and use the naming reference at second place. As we have declared constraint for the DATAWSUB (subscriber), then reused the same constraint for the DATARSUB(publisher). For this constraint, membership report is sent using IGMP protocol version 2 or version3, but only one of it can be true at a time therefore, we have given a previous packet with *or* condition for the V2Member and the V3Member packet types.

   Constraint structure.

   - **Current Packets(2)**: Publisher, Subscriber
   - **Previous Packets(2)**: RTPS Participant, Membership report
   - **OR for packets**: V2Member or V3Member

   Steps for writing constraint

(a) Naming and annotation which is C1, @pass respectively, as shown on Line 14 and 15.

(b) Declaration for our two previous packets and OR packet V2Member and V3Member in the declaration, as shown on Line 16.

(c) Write the logical relation among the previous and current packet DATAWSUB, shown on Line 17,18. Do this for V2Member and V3 member separately and then OR the two expression, as shown on Line 19.

(d) Put the order statement considering the OR packets as shown on Line 20.

(e) Use this constraint for referring for other current packet DATARSUB. As shown on Line 39, we use the constraint C1 defined for DATAWSUB on Line 14, again for DATARSUB. DATARSUB has same attributes as DATAWSUB therefore, our constraint can be reused without rewriting it.

```
1  DATAWSUB ::= SEQUENCE {
2    kind       INTEGER (SIZE 1 BYTES),
3    flags      INTEGER (SIZE 1 BYTES),
4    nextHeader INTEGER (SIZE 2 BYTES),
5    extraFlags INTEGER (SIZE 2 BYTES),
6    qosOffset  INTEGER (SIZE 2 BYTES),
7    readerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
8    writerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
9    writerSEQ  INTEGER (SIZE 8 BYTES),
10   inlineQos  QOSPARM (SIZE DEFINED) OPTIONAL,
```

```
11   serializedData TOPICS (SIZE DEFINED) OPTIONAL
12 } (ENCODED BY CUSTOM)
13 <constraint>
14   C1:
15   @pass
16   (dataP = DATAPSUB, v2Mem = V2Member | v3Mem = V3Member;
17   dataP.SrcIP == SrcIP &
18   dataP.DstIP == DstIP &
19   (v2Mem.groupAddress == DstIP | v3Mem.groupAddress == DstIP);
20   dataP.Order = 1, v2Mem | v3Mem = 2;);
21   C2:
22   @pass
23   ...
24 </constraint>
25
26 DATARSUB ::= SEQUENCE {
27   kind       INTEGER (SIZE 1 BYTES),
28   flags      INTEGER (SIZE 1 BYTES),
29   nextHeader INTEGER (SIZE 2 BYTES),
30   extraFlags INTEGER (SIZE 2 BYTES),
31   qosOffset  INTEGER (SIZE 2 BYTES),
32   readerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
33   writerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
34   writerSEQ  INTEGER (SIZE 8 BYTES),
```

```
35    inlineQos  QOSPARM (SIZE DEFINED) OPTIONAL,
36    serializedData TOPICS (SIZE DEFINED) OPTIONAL
37  } (ENCODED BY CUSTOM)
38  <constraint>
39    C1;
40  </constraint>
```

2. *A topic name of a particular domain is only published from a specific set of publishers with a fixed quality of service*

   This constraint uses to static environmental information to validate the conditions for the constraint. As these constraints do not require any previous packet to check its conditions, packet declaration part would be left empty, with no previous packet required in this constraints, order defining part of constraint also be left empty with a semicolon to indicate start and end of empty order statement. TOPICSPARAM is set in our RTPS ASN.1 declaration of SCL, so it should be left empty.

   - **Current Packets(1)**: Publisher
   - **Previous packets(0)**: None
   - **Static Variable**: Yes

   Steps for writing the constraint:

   (a) Naming and annotation which is C1, @pass respectively, as shown on Line 14.

(b) Declaration should be left empty as there is no previous packets, as shown on Line 16.

(c) Write the logical relation among the packet attributes and static variable for verification, as shown on Line 16-18.

(d) No ordering required as there is no previous packets, so leave it empty, as shown on Line 18.

```
1  DATAWSUB ::= SEQUENCE {
2    kind       INTEGER (SIZE 1 BYTES),
3    flags      INTEGER (SIZE 1 BYTES),
4    nextHeader INTEGER (SIZE 2 BYTES),
5    extraFlags INTEGER (SIZE 2 BYTES),
6    qosOffset  INTEGER (SIZE 2 BYTES),
7    readerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
8    writerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
9    writerSEQ  INTEGER (SIZE 8 BYTES),
10   inlineQos  QOSPARM (SIZE DEFINED) OPTIONAL,
11   serializedData PARTICIPANTS (SIZE DEFINED) OPTIONAL
12 } (ENCODED BY CUSTOM)
13 <constraint>
14   C1:
15   @pass
16   ( ;static(FactData.SrcIP) == SrcIP &
17   static(FactData.TopicName) == TOPICS.TOPICSPARAM<
       PIDTOPICNAME>
```

```
18    & static(FactData.QoS == TOPICS);;)
19  <constraint>
```

3. Data of a topic is considered valid if it is produced from a valid publisher and consumed by valid subscriber.

   This constraint is for the data packet, which require publisher and subscriber packet for validation. Therefore, current packet would be of type DATASUB. Order of publisher and subscriber is not fixed. If first previous packet is publisher then second previous packet should be subscriber or vice-versa.

   - **Current Packets(1)**: Data
   - **Previous packets(2)**: Publisher, Subscriber
   - **OR for Packets**: Publisher—Subscriber, Subscriber—Publisher

   Steps for writing the constraint:

   (a) Naming and annotation which is C1, @pass respectively, as shown on Line 4-5.

   (b) Declaration for our two previous packets and OR packet. Both the first and second previous packet have OR condition, as shown on Line 6.

   (c) Write the condition statement, to compare all the scenarios of previous packet with current packet. We maintain the OR of packet we declared in our logical statement everywhere, as shown on Line 7-12.

   (d) Put the order statement considering the OR packets as shown Line 13.

```
1  DATASUB ::= SEQUENCE {
2    ...
3  }
4  <constraint>
5    C1:
6    @pass
7    (dw1 = DATAWSUB | dr1 = DATARSUB, dr2 = DATARSUB | dw2 =
      DATAWSUB;
8    ((dw1.DstIP == dr2.SrcIP & dr2.SrcIP == DstIP)
9    | (dr1.DstIP == dw2.SrcIP & dw2.SrcIP == SrcIP))
10   & ((dw1.TopicParams.TopicName == dr2.TopicParams.TopicName)
11   | (dr1.TopicParams.Topic == dw2.TopicParams.TopicName))
12   &(dw1.entityId == writerEnt )
13   &( dw2.entityId == writerEnt );
14   dw1.order | dr1.order = 1, dr2.order | dw2.order = 2;
15   )
16 </constraint>
```

The same constraint can be written using only static and dynamic variable with no previous packets. The constraint for current packet of sub-message type DATASUB, we can write constraint as follows. As we do not have any previous packets, syntax declaration and ordering part is left empty.

- Current Packets(1): DATA
- Previous Packets(0): None

- Static variable: Yes
- Dynamic variable: Yes

```
1  DATASUB ::= SEQUENCE {
2    ...
3  }
4  <constraint>
5    C1:
6    @pass
7    (;
8    ((static(staticInfo.SrcIP) == SrcIP | static(staticInfo.
       DstIP) == SrcIP)
9    &(static(staticInfo.SrcIP) == DstIP | static(staticInfo.
       DstIP) == DstIP)
10   &(dynamic(dynamicInfo(SrcIP).entityId) == writerId));
11   ;)
12 </constraint>
```

4.3 Java-based Syntax Summary

We propose the Java expressions like syntax, where syntax design is based on hypothesizing module as object and attribute the field of each object. We have used the data structure present for object oriented design for handling our environmental values.

We have modified regular Java expression to handle of scenario relevant for our constraint. Our modification is in declaration and order part of syntax. Logical

expression part has no modification and follows actual Java expression syntax. At last, we have represented three constraints from ATC simulation network. We have also showed how using environmental variables can simplify our last constraint.

Chapter 5

QUEL-based Syntax

QUEL is a structured query language [46, 47]. QUEL was not initially popular, but it gained its name by its implementation as POSTQUEL to support POSTGRES[49, 50, 51, 42]. To use the QUEL like syntax we represent network data in structured form. QUEL requires data to be stored in tables with columns and rows. To make the best of a QUERY based syntax such as QUEL, it is best to represent data in tabular form.

QUEL in comparison to SQL is normalized [48]. This means that SQL statements for various operation have different format, whereas in QUEL a single syntax pattern is used for all the statement [48]. The QUEL statements use a tuple based representation which gives it uniformity. This means statements for selection, deletion, update and insertion of a row from the table will have a similar syntax format. A simple example of QUEL syntax based on employee table.

Example:

```
1 range of E is EMPLOYEE
2 retrieve into W
3   (COMP = E.Salary / (E.Age - 18))
```

```
4    where E.Name = "Jones"
```

In above example, E represents a tuple referencing of type Employee where all tuples of type Employee satisfy the following condition E.Name = Jones. Following that the COMP value is calculated and assigned to the new tuple variable W. The retrieve statement has similar tuple structure as which is type of operation, reference and value calculation. We Each statement in QUEL is tuple based [48].

We present the example from SQL language for the same scenarios as the QUEL example. It is a single select query to do the operations. In the SQL example, we cannot break this statement into smaller units. The employee *e* reference in SQL query is tightly bound with this select query and no other query can use it. In QUEL once the reference of table is defined, it can be used any number of times in multiple retrieve statement.

```
1 select (e.salary / (e.age - 18)) as comp
2 from employee as e
3 where e.name = "Jones"
```

For our analysis of constraint based on RTPS and IGMP has smallest unit are packets. The network communication exchange different types of packet. A protocol can have packets with different attributes and structures. To express our constraint in QUEL like syntax. We have to represent our information in the structured form in terms of row and columns. To do so, we looked into the structure of the packet for each protocol. We realized not only attributes changes for a packet but same attributes can have different values for different packets. To present packets information in the table, we categorized IGMP packets on the basis packet type and RTPS packets on the basis of sub-message.

For IGMP, all the packets have the same attributes but different values. We can create a single table for IGMP where table column are the attributes name from SCL modules and each row points to a packet and their values. To make our representation simpler for network engineer, we sub divide tables for each packet type of IGMP packets. Each tables in our intuition are considered as table with infinity rows and fixed column. Rows in our intuition represents a packet from network traffic. This make our syntax simple by removing the effort to query the packet type from one single table. Network engineers do not have to check the types of packets while writing constraints. For RTPS protocol, we filter packets on the basis of sub-message type in the packet. The Sub-message types determine the role of each packet in the RTPS protocol. Therefore, we have table for RTPS based on their sub-message. Similar to IGMP, it reduces the effort of query sub-message information in our constraint. Example of table representation for IGMP packet of type join shown in Table 5.1. We also show the RTPS packet of sub-message type participant. RTPS packets has large number of attributes, that why we only shows partial structure of participant table. Examples, Table 5.1 for packet type join and Table 5.2 for RTPS packet of sub-message type participant.

V2Join

Column	Type
Type	INTEGER
Reserved1	INTEGER
Headerchecksum	INTEGER
Num_group_records	INTEGER
Group_record.type	INTEGER
Group_record.Aux_data_len	INTEGER
Group_record.Num_src	INTEGER

Table 5.1: Tabular representation of IGMP join packet

RTPS Participant Packet

Column	Type
kind	INTEGER
flags	INTEGER
nextHeader	INTEGER
extraFlags	INTEGER
qosOffset	INTEGER
readerEnt.Key	INTEGER
readerEnt.Kind	INTEGER

Table 5.2: Tabular representation of RTPS participant packet

5.1 Abstract Syntax and Grammar

Naming and annotation of constraints are very similar to the Java-based syntax. After annotation, we have declaration and retrieval part, which is similar to logical condition in Java-based syntax. The Syntax keywords motivated from QUEL to represent the certain action which is discussed in keyword sub-section of abstract syntax. We have added our own keyword as well to address the scenarios relevant to our problem. To understand this syntax, we will see an example of constraint representation in QUEL based syntax. The two important part of QUEL-based syntax are:

1. **Declaration** {*rangeDeclaration*}: In this part of syntax, we declare all the packet types and sub-types for the constraint. One range statement point to single packet type or sub-types. Similarly for static and dynamic variable where one range statement points to one static variable.

2. **Packet logic definition** {*retrieveStatement*}: In this part of syntax, we use the packet reference declared in in range statement for building our retrieve statement. For each range statement we define one retrieve statement. Retrieve statement define the conditional statement using where clause to define all the logical expression for packet to be valid. Retrieve statement has attribute named recent used to define the order of packet in a packet stream.

The grammar for the constraint representation in QUEL-based syntax are as follows:

```
1 <constraintDefinition>  := <rangeDeclaration> <retrieveStatement>
2 <rangeDeclaration>  := 'range of' <prefix> <variable> of <
     tableReference>
```

```
3  <prefix>  := static/dynamic
4              | <empty>
5  <retrieveStatement>   := 'retrieve into' <variable> <whereclause>
6  <whereclause>   := 'where'<expression>
7  <expression>  := <expression><airthmeticOperator><expression>
8                  | <variable><airthmeticOperator><variable>
9                  | <variable><airthmeticOperator><fixedValue>
10 <airthmeticOperator>  := '>'
11                          | '<'
12                          | 'or'
13                          | 'and'
14                          | '='
15 <fixedValue>    := [0-9]+
16 <empty>     := NULL
```

Consider a hypothetical constraint, where we have a protocol of type A and it has 3 types of packets packet1, packet2, packet3. Each packet type can be considered as table in our syntax. For a given constraint, where the packet1 packet should always come before packet2, and packet1 packet source IP and should be equal to packet2 packet source IP. In this case, the packet2 packet is our current module where constraint would be placed.

```
1 <constraint>
2   @pass
3   range of p1 is packet1
4   retrieve into (p1) where p1.headerInfo_srcIP = srcIP
```

```
5   and recent = 1;
6 </constraint>
```

5.2 Keywords

All keywords used for QUEL-based syntax for constraints is shown in Table 5.3.

Keywords

Keyword	Definition
RANGE OF	This keyword allow user to declare the types of table references.
RETRIEVE INTO	It is used to row or column calculated value into a variable.
WHERE	Statements followed by where are condition(s) to filter the required rows which follow the conditions.
AND	To join condition in where caluse.
OR	To define any one true condition in where clause.
RECENT	It is a keyword introduced by us, which is implicit column attached to all the tables. It is to find the previous packet position in the packet stream.

Table 5.3: Keywords and their definition in QUEL-based syntax

5.2.1 Multiple Packet Using The OR Condition

In this scenario, previous packet types is uncertain, OR condition need to done for all the possible packets that can be valid for constraint. In QUEL based syntax it is handled differently than Java-based syntax, without any modification in QUEL

syntax. In QUEL based syntax, we declare all the packet type that could be previous packets and write retrieve statement for all those packets with the same recent value. Constraints are only said to be valid if at least one and only one filter condition for packets with same recent value turns to be satisfied. It can be seen in the example below, constraints state that at least and only one retrieve condition with recent value 2 will be valid for the constraint to be validated, shown in line. In below example, both v2mem and V3mem has recent value 2, constraint will be valid if second previous packet is either V2Member or V3Member type.

```
1  <constraint>
2    C1:
3    @pass
4    (range of datap is DATAPSUB,
5    range of v2mem is V2Member,
6    range of V3mem is V3Member,
7    retrieve into (datap) where datap.SrcIP = SrcIP
8    and datap.DstIP = DstIP and recent = 1;
9
10   retrieve into (v2mem) where v2mem.SrcIP = SrcIP
11   and v2Mem.groupAddress = DstIP and recent = 2;
12
13   retrieve into (v3mem) where v3mem.SrcIP = SrcIP
14   and v3Mem.groupAddress = DstIP recent =2;
15   )
16 </constraint>
```

5.2.2 Keywords for Environmental Variables

Static and dynamic variables need to be represented in tabular format to use QUEL syntax. We use tables to represent static and dynamic variables. An environment variable is declared as a table. For static variables, we introduce the keyword *static* that should be used in declaration statement of our syntax. Similarly, dynamic variables are declared using a keyword *dynamic* before in declaration statement. Once declared these static and dynamic variables can be use as any other packet type tables. However, there is one key difference between dynamic and static variable use. A dynamic variable retrieve statement should always include one attribute which is static in nature. An example below, can be seen on how to use using static and dynamic environmental variables in QUEL based syntax. In this example, we have static table named table1 with reference t1 and table2 as dynamic variable with reference t2. Once declared it can be used same as other table reference of packets, as shown on Line 5.

```
1  </constraint>
2    @pass
3    range of static t1 is table1
4    range of dynamic t2 is table2
5    retrieve into (t1) where
6    t1.headerInfo_srcIP = srcIP ;
7    retrieve into (t2) where
8    t2.headerInfo_srcIP = srcIP
9    and t2.entityId = entityId ;
10 </constraint>
```

The major difference between using static and dynamic variables is using prefix *static* and *dynamic* is how dynamic variables are retrieved. Dynamic variables are always associated with one static attribute such as source information (srcTP) which is static for both learning and checking mode. In above example, dynamic information *entityId* along with static attribute source IP.

5.3 Constraint for RTPS and IGMP

Steps for representing a constraint in the QUEL based syntax are as follows:

1. Identify the tables for the packets required for the constraints.

2. Identify the current table in the list of tables and identify the respective module for it in SCL.

3. Build a relationship among these tables and find the order in which we are expecting to retrieve values from each table. Order of retrieving not relevant for the tables storing environmental variable values.

5.3.1 Constraints Representation

Using our syntax, we build following representation for our IGMP and RTPS protocols through ATC simulation. We use the same three constraint used for Java-based syntax. First step is recognizing the main entity of each protocols, for IGMP we have *packet type* which we use to differentiate packets, for RTPS protocols we use *sub-message* type for choosing packets interested for constraints. All of the tables for IGMP would be base on IGMP packet type and for RTPS based on sub-message

type. Then for each constraint, we should follow steps mentioned above. The constraint structure will remain same as Java-based syntax, which means current packets, previous packet and environmental variables does not change.

1. **All the publishers and subscribers must be valid members of at least one IGMP multicast group. These participants should send their membership reports to specific group addresses before showing their interests in a topic.**

   In this constraint, the current packet is a publisher(DATAWSUB) and subscriber(DATARSUB), where the membership report can be valid for either version 2 IGMP or version 3 IGMP protocol. To handle either version of IGMP protocol as our previous packet, we will include both version packets in our constraints as two distinct packets. To make it as OR condition, which makes sure only one packet is considered as the previous packet, we use same recent value for retrieve condition for both version of membership report.

   Steps for writing the constraint:

   - Naming C1 and annotation pass same as Java-based syntax, shown on Line 14-15.
   - we define the table using range statement for previous packet type (IGMP) or sub-message type (RTPS), shown on Line 16-18.
   - we write our conditions using retrieve statement, shown on Line 19-27. Here all the retrieve statement are using values from current packet,

which means no two previous packet have any conditional statement among them. Order of packet stream is determined in retrieve statement using RECENT column.

- Constraint reuse for publisher packet using name reference C1, as shown on Line 42. It is same as using constraint for another current packet as in Java-based syntax.

```
1  DATAWSUB ::= SEQUENCE {
2    kind       INTEGER (SIZE 1 BYTES),
3    flags      INTEGER (SIZE 1 BYTES),
4    nextHeader INTEGER (SIZE 2 BYTES),
5    extraFlags INTEGER (SIZE 2 BYTES),
6    qosOffset  INTEGER (SIZE 2 BYTES),
7    readerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
8    writerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
9    writerSEQ  INTEGER (SIZE 8 BYTES),
10   inlineQos  QOSPARM (SIZE DEFINED) OPTIONAL,
11   serializedData TOPICS (SIZE DEFINED) OPTIONAL
12 } (ENCODED BY CUSTOM)
13 <constraint>
14   C1:
15   @pass
16   (range of datap is DATAPSUB,
17   range of v2mem is V2Member,
18   range of V3mem is V3Member,
```

```
19   retrieve into (datap) where datap.SrcIP = SrcIP
20   and datap.DstIP = DstIP and recent = 1;
21
22   retrieve into (v2mem) where v2mem.SrcIP = SrcIP
23   and v2Mem.groupAddress = DstIP and recent = 2;
24
25   retrieve into (v3mem) where v3mem.SrcIP = SrcIP
26   and v3Mem.groupAddress = DstIP recent =2;
27   )
28 </constraint>
29 DATARSUB ::= SEQUENCE {
30   kind       INTEGER (SIZE 1 BYTES),
31   flags      INTEGER (SIZE 1 BYTES),
32   nextHeader INTEGER (SIZE 2 BYTES),
33   extraFlags INTEGER (SIZE 2 BYTES),
34   qosOffset  INTEGER (SIZE 2 BYTES),
35   readerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
36   writerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
37   writerSEQ  INTEGER (SIZE 8 BYTES),
38   inlineQos  QOSPARM (SIZE DEFINED) OPTIONAL,
39   serializedData TOPICS (SIZE DEFINED) OPTIONAL
40 } (ENCODED BY CUSTOM)
41 <constraint>
42   C1;
```

```
43 </constraint>
```

2. *A topic name on a particular domain is only published from a specific set of publishers with fixed quality of service*

   This constraint uses static environmental information to validate the conditions for the constraint to be true. As these constraints do not require any previous packet to check the conditions, the only declaration we require is for *static table*.

   Steps for writing the constraint:

   - Naming C1 and annotation pass same as Java-based syntax, shown on Line 14-15.
   - we define a static table using range and static keyword, as shown on Line 16.
   - We write retrieve statement for static table like any other previous packet constraint with no RECENT column value, as shown on Line 17-18.

   We write a retrieve query for the static table with no order information as the static table is not a part of the stream of packets. The static table doesn't have any recent column associated with them, as shown on Line 17-19.

```
1 DATAWSUB ::= SEQUENCE {
2   kind       INTEGER (SIZE 1 BYTES),
```

```
3    flags      INTEGER (SIZE 1 BYTES),
4    nextHeader INTEGER (SIZE 2 BYTES),
5    extraFlags INTEGER (SIZE 2 BYTES),
6    qosOffset  INTEGER (SIZE 2 BYTES),
7    readerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
8    writerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
9    writerSEQ  INTEGER (SIZE 8 BYTES),
10   inlineQos  QOSPARM (SIZE DEFINED) OPTIONAL,
11   serializedData TOPICS (SIZE DEFINED) OPTIONAL
12 } (ENCODED BY CUSTOM)
13 <constraint>
14   C2:
15   @pass
16   (range of static factTable is FACTDATA;
17   retrieve into (factTable) where factTable.SrcIP = SrcIP
18   and factTable.TopicName = TOPICS.TOPICSPARAM.PIDTOPICNAME
19   and factTable.QoS = inlineQoS;
20   )
21 </constraint>
```

3. Data of a topic is considered valid if it is produced from a valid publisher and consumed by valid subscriber.

   This constraint has current packet pointing to RTPS data packet. This constraint has two previous packets. Each previous packet has two option to

be true. Therefore, we will write total of four packet declaration statement, shown on Line 6-9. Following that, we have retrieve statement for conditions for the constraint. In this constraint, we have conditional statement build using two previous packets. For example, retrieve statement for *dr1* uses *dw2* attributes to builds it expression, shown on line 10. We have two review statement with recent value 1 and two packets with recent value 2 to handle the OR condition of previous packets.

Steps for writing the constraint:

- Naming C1 and annotation pass same as Java-based syntax, shown on Line 4-5.
- we define four tables pointing to 4 packet based on sub-messages type, shown on Line 6-9.
- we write our conditions using retrieve statement, shown on Line 10-29. We write retrieve condition for all four packets. Two packet will have RECENT column value 1 and two will have 2. This makes the OR condition for the packets with same RECENT value.

```
1 DATASUB ::= SEQUENCE {
2   ...
3 }
4 <constraint>
5   C1:
6   @pass
7   range of dw1 is DATAWSUB;
```

```
8   range of dr1 is DATARSUB;
9   range of dr2 is DATARSUB;
10  range of dw2 is DATAWSUB;
11  retrieve into dr1 where
12  and dr1.TopicParams.TopicName = dw2.TopicParams.TopicName
13  and dr1.DstIP = dw2.SrcIP and recent = 1;
14  retrieve into dw1 where
15  and dw1.TopicParams.Topic = dr2.TopicParams.Topic
16  and dw1.DstIP = dr2.SrcIP
17  and dw1.entityId = writerEnt
18  and recent = 1;
19  retrieve into dr2 where
20  dr2.SrcIP = DstIP
21  and dr2.TopicParams.Topic = dr1.TopicParams.Topic
22  and dr2.SrcIP = dw1.DstIP
23  and recent = 2;
24  retrieve into dw2 where
25  dw2.SrcIP = SrcIP
26  and dw2.TopicParams.Topic = dr1.TopicParams.Topic
27  and dw2.entityId = writerEnt
28  and dw2.SrcIP = dr1.DstIP
29  and recent = 2;
30 </constraint>
```

Alternative representation of the constraint using dynamic and static variables, using environmental variables reduces the redundancy in the code.

```
1  <constraint>
2    C1:
3    @pass
4    range of static factTable is FACTDATA;
5    range of dynamic dynTable is DYNAMICTABLE;
6    retrieve into factTable where
7    factTable.SrcIP = SrcIP or factTable.DstIP = SrcIP
8    and factTable.SrcIP = DstIP or factTable.DstIP = DstIP;
9    retrieve into dynTable where
10   dynTable.SrcIP = SrcIP or dynTable.SrcIP = DstIP
11   and factTable.entityId = writerEnt or factTable.entityId =
      readerEnt;
12   SrcIP != DstIP;
13 </constraint>
```

5.4 Quel-based Syntax Summary

In this chapter, we have proposed another type of syntax, based on the QUEL programming language which is similar to SQL (Structured Query Language). This syntax gives new perspective for representing network constraint with the hypothesis of packets information as a row in the tables.

In our last constraint, we require to build conditional statements using two pevious packet information. This is a modification from actual QUEL syntax, where building

a conditional statement without retrieving a information is not possible. We have added three keywords to the QUEL-based syntax for managing the order of the packet stream and environmental variable. Finally, we also show the capability of using dynamic variable for last constraint, which simplifies and reduces the redundancy in the syntax.

Chapter 6

Prolog-based Syntax

The Prolog-based syntax (PBS) as the name suggests uses a syntax based on Prolog logic. Prolog-based syntax requires the users to know the basics of Prolog. More in-depth knowledge of Prolog is required to understand our Prolog-based syntax for writing constraints.

Prolog has been popular as a translation framework. McCord's work [30, 29] on prolog based machine learning translation is a good example of the use of Prolog to translate English to German. In their approach, they use a logic grammar to understand the syntax and lexicon of English. They have written their transformation logic in Prolog based pattern matching. They use a prolog based representation for their 5 steps transformation of English to German language. Similar inspiration has been taken from Wilks's[57] work where they investigate syntax transformation related to semantic relation using Prolog like syntax.

Prolog is a declarative programming language, that is popular in logic programming. Compared to procedural languages, the user doesn't have to define the complete steps for achieving the final goal. Prolog is based on defining the relationship among the objects and using these relationships as a fact base for solving problems. The two

most important aspects of Prolog are facts and rules which need to be understood to know Prolog better.

Facts are the representation of information that is used by Prolog. Facts can be properties of objects or relationships between multiple objects. We see an example of facts for office information, where office facts with employee name and id has been saved, shown on Lines 1-3. Similarly, other facts such as animal and their features which is *animal* and *has_feather* as fact, as shown on Lines 5-7.

```
1 office(X,Y) :- office has X as employee name and Y as employee id.
2   office(alison, s134).
3   office(ryam, r213).
4
5   animal(lion).
6   animal(sparrow).
7   has_feathers(sparrow).
```

Rules are logic that queries the fact information for validating and retrieval of any information. It is also used for building relationships among the facts.

We create rules to validate weather a animal is bird or not. We use the fact base of animal and has_feather to build the rule for bird.

```
1 bird(X) :-
2   animal(X),
3   has_feathers(X).
```

In this above example, we have used two facts named animal and has_feature to build our rule bird. This rule takes X as input and returns true, if all the facts in the rule are satisfied, else it is false. For the given X of lion the bird rule will return false

whereas for sparrow it will return true. This might be the simplest possible example for Prolog, more complex scenarios can be seen where facts have multiple parameters.

To build a Prolog like syntax, we want to represent our constraints similar to the rules. To do so our packet information should be considered as a fact base. To represent our packets as a fact base, We investigate protocols and their packets, after closely looking into our scenarios of two protocols IGMP and RTPS. The names of fact attribute are same as ASN.1 notation. This allow name matching technique to work, further discussed in this chapter how name matching technique works. We have decided as follows.

For IGMP, we should consider the fact base on the basis of packet type structure which means number of parameter and their order same for same packet type. For example, IGMP version 2 leave packet will have same fact structure. Similarly, for other packet types such join and membership report. On the other hand, RTPS fact base is categorized on the basis of sub-message type because RTPS protocol packets don't have any particular type for individual packets rather each packet is the group of sub-messages, and also all our constraints are based on sub-message as smallest unit for RTPS packet querying. Example of fact schema base for IMGP query packet. The side by side representation shows ASN.1 notation for IGMP query packet and its equivalent fact schema.

Listing 6.1: ASN.1 notation

```
1 IGMP DEFINITIONS ::= BEGIN
2 PDU ::= (Query)
3 Query ::= SEQUENCE {
```

Listing 6.2: Fact Schema

```
1 Packet(Query,
2   context(time,SrcIP, DstIP),
3   type,
```

```
4     type INTEGER (SIZE 1 BYTES),
5     maxRespTime INTEGER (SIZE 1
      BYTES),
6     checksum INTEGER (SIZE 2 BYTES
      ),
7     groupAddr INTEGER (SIZE 4
      BYTES),
8     v3Add V3Addition (SIZE DEFINED
      ) OPTIONAL
9  }
10
11 V3Addition ::= SEQUENCE {
12    resvSQRV INTEGER (SIZE 1 BYTES
      ),
13    QQIC INTEGER (SIZE 1 BYTES),
14    numSources INTEGER (SIZE 2
      BYTES),
15    srcAddrs SET OF SOURCEADDRESS
      (SIZE CONSTRAINED)
16 } (ENCODED BY CUSTOM)
17
18 SOURCEADDRESS ::= SEQUENCE {
19   srcAddr INTEGER (SIZE 4 BYTES)
20 }
```

```
4    maxRespTime,
5    checksum,
6    groupAddr,
7    v3Add (resvSQRV,
8      QQIC,
9      numSources,
10     srcAddrs( srcAddr)
11   )
12 )
```

```
21 END
```

Similarly RTPS packets would be represented, with their first parameter as a sub-message type and second parameter as the all the physical layer related information named as context information in our syntax. These context informations are source, destination and time of packet arrival. Following context information, we will have rest of the attributes of a packet. RTPS has comparatively large number of attributes so, we have only shown the partial fact structure of the RTPS publisher packets whose type is represented with DATAW. In below example, Line 12 is to depict, we have more attributes in this fact structure.

```
1  packet(DATAW,
2    context(time, src, dest),
3    kind,
4    flags,
5    nextHeader,
6    extraFlags,
7    qosOffset,
8    readerEnt,
9    writerEnt,
10   writerSEQ,
11   inlineQos,
12   .....,....,
13 )
```

Packet facts representation might change with new attributes in packets for a given protocols. The packet facts representation is created based on the packets attributes

for IGMP and RTPS data collected using ATC simulation. The fact attribute names are same as ASN.1 notation equivalent, as shown on above example of ASN.1 notation and equivalent fact representation.

There are two important parameters for the packet fact representation. The first parameter of fact base always represents the type of packet, sub-message or for any other type of protocols category. The second parameter is context, which has its own parameters. The context information of the packet are information which are independent of packet types. It has the parameter such as packet's source and destination information, time of packet origin relative to the first packet if required or any other general information which can be retrieved from the physical layer. The fact representation is the base of our Prolog-based syntax.

6.1 Abstract Syntax and Grammar

This syntax is similar to writing rules in Prolog language. Constraints are either pass or fail constraints, which is defined using annotation on the first line of our syntax. Following that is the constraint name as a functor(rule) which takes parameters. These parameters are the attributes of the current packet. Current packet attribute are used to query previous packets for the constraint. To remove the redundancy and simplify our syntax, a modification in Prolog syntax has been introduced. The modification is called triple dot technique, which is inspired from Java triple dot function parameter. Triple dot technique is extension of underscore in Prolog with better capability for the constraint representation. An example below shows the advantage of triple dot over underscore technique. For example, a constraint requires last two attributes of hypothetical packet with fact schema as follows:

```
Packet(type1,
  relativeTime,
  recordId,
  aliveTime,
  srcIP,
  dstIP
)
```

Listing 6.3: Using underscore

```
<constraint>
  @pass
  constUnderScore(srcIP,dstIP) :-
          Packet(_,
          _,
          _,
          _,
          srcIP,
          dstIP
      )
</constraint>
```

Listing 6.4: Using triple dot

```
<constraint>
  @pass
  constTripleDot(srcIP,dstIP) :-
          Packet( ...,
          srcIP,
          dstIP
      )
</constraint>
```

From above example, a constraint using underscore and our triple dot technique shown side by side. In underscore representation, we have four underscore for parameters not relevant to the constraint, whereas using triple dot technique, we replace this four underscore with single triple dot. Therefore, triple simplifies our syntax representation

and remove redundant underscore from constraint representation.

6.1.1 Keywords

These features are discussed and how to use it in the syntax are as follows:

1. *Triple dots* (...): Three dots in our syntax is used to remove unnecessary information of facts and only focusing on information relevant packet information for our constraints. It is based on name matching where it look for attribute name between two point of packet structure. It represents relative location of an attribute compared to other attributes, Which means attributes lie as first, middle or as the last parameter of fact representation. Using triple dots, removes the requirement of defining the complete packet fact structure which is required in actual Prolog syntax.

2. *Mostrecent*: Mostrecent is a implicit functor, which provide a capability to represent the previous packet and their order in the packet stream. It has two parameters which is packet fact and order number.

6.1.2 Name and Position Matching

The constraint uses implicit functor named **mostrecent** to access previous packets. The mostrecent functor has two parameters, first parameter is packet fact base using current packet attributes to query it. The second parameter is the order of previous packet from current packet. The constraint parameter used inside fact base as known information to filter facts. Previous packet querying is two step process:

1. *Position matching*: Previous packet fact should include packet type, then following the fact structure, constraint parameter should be place at the attribute position against which it is compared. For example below, for first previous packet, constraint parameter p1 will match against two parameters of type1 packet, second and third parameter, as shown on Lines 3.

```
1 <constraint>
2   @pass
3   ConstraintName(p1):-
4     mostrecent(packet(type1, p1,p1), 1).
5 <constraint>
```

2. *Name matching*: In this scenario, when the packet fact structure is large and hard to represent complete structure, we represent partial fact structure using triple dot technique. In this representation, using the relative position the packet attribute are matched with the constraint attribute name.

   An example using name matching, consider a hypothetical fact base which already exists for all the packets associated with constraints. Example has a protocol having two types of packets *pack1*, *pack2*. Both the packets should have same source and destination. In our Prolog like syntax, this constraint would represented as follows. In below example, Line 2 have annotation for our constraint. Following that Line 3, rule is defined as with parameter required from current packet to be used to find the previous packet for constraint. Line 4, has *mostrecent* functor with parameter packet fact and second parameter as 1 which means first previous packet. For a packet type pack1, complete fact

structure is not clear, so we use triple dot notation to locate the relative position of attributes, as shown on Line 5. Using relative position our constraint will match packet attribute of SrcIP name with SrcIP name of current packet, similarly for DstIP, as shown on Line 5.

```
1 <constraint>
2   @pass
3   ConstraintName( SrcIP, DstIP) :-
4     mostrecent(packet(pack1
5     ,context( ..., SrcIP, DstIP), ...)
6     ,1)
7 </constraint>
```

The grammar for constraint representation in Prolog-based syntax are as follows:

```
1
2 <constraintDefinition>  := <constraintRule> ':-' <listOfFacts>.
3 <constraintRule>  := <constraintName>( <listOfpacketAttributes> )
4 <listOfpacketAttributes>:= <packetAttribute>,<listOfpacketAttributes>
5                          | <packetAttribute>
6                          | ''
7 <packetAttributes>  := <identifier>
8
9 <listOfFacts>   := <factStructure> <operator> <listOfFacts>
10                    | <factStructure> <operator> <packetAttribute>
11 <operator>    := ','
12                | ';'
```

```
13
14 <factStructure>   := <functor>
15                     | <factSchema>
16
17 <factSchema>    := <factName> '(' <requiredAttributes> ')'
18 <requiredAttributes>  := <requiredAttributes><tripledot><
      requiredAttributes>
19                         | <listOfpacketAttributes>
20                         | <factSchema>
21 <functor>     := <functorName> '(' <functorAttributeList> ')'
22 <functorAttributeList>    := <functorAttribute>, <functorAttributeList
      >
23 <functorAttribute>    := <factSchema>
24                         | <functor>
25                         | <varaible>
26                         | <fixedValue>
27 <fixedValue>    := [0-9]+
28 <varaible>    := String
29
30 <functorName>   := String
31 <factName>    := String
32 <tripledot>   := '...'
```

The constraint definition of prolog-based syntax has few important elements which is explained below.

1. **ConstraintRule**: A Constraint rule is a prolog rule like structure with rule name equivalent to our constraint name. The constraint rule also takes packet attribute as parameter.

2. **ListOfFacts**: This is part of the constraint definition, which form the body of the constraint rule. It is list comprise of either functor or packet fact schema.

3. **Functor**: Functor is implicitly defined which takes fixed set of parameters as their input. Each input parameter type is predefined for the functor.

4. **Factschema**: Fact schema is a packet fact schema as a part of constraint rule. It is based on packet structure and uses packet attributes for verification of constraint. It use triple dot technique to simplify its structure.

6.1.3 Keywords for Environmental Variables

To handle environmental variables for both learning and checking mode, we have introduced a design which is based on using functor and creating a fact base of these environmental informations

We have introduced a functor to access static and dynamic fact based which stores static and dynamic environmental information respectively. The functor is named as accessVar to access both static and dynamic fact base. In Prolog, the number of parameters is also called as *arity*. Prolog also treats a functor with the same name but taking a different number of parameters as different functors. To access static fact base, functor *accessVar* takes one parameter where as for dynamic fact base access it takes two parameters. We can say we have two functors as *accessVar/1* and *accessVar/2* of arity one and arity two respectively.

Syntax for access of static and dynamic fact base has been shown below, in Lines 8-9. As we can see in the example below that *staticData* represent the fact base storing the static environmental values and *accessVar* functor takes only one parameter which is a fact for static information. To represent the dynamic information, we have *accessVar* with two parameters, the first parameter is a static value from packet and the second parameter is fact base representing dynamic values. Both static and dynamic fact base can have multiple parameters, which is declared with their first use.

```
1 <constraint>
2   @pass
3   ConstraintName( SrcIP, DstIP) :-
4     mostrecent(packet(pack1
5       ,context( ..., SrcIP, DstIP), ...)
6       ,1),
7     accessVar(staticData(SrcIP, ...)),
8     accessVar(SrcIP, dynamicData(entityId, ...)).
9 </constraint>
```

6.1.4 The OR Condition

In Prolog syntax, semicolon ';' between two functors consider an *or* condition. Thus to handle *or* condition in Prolog we use the semicolon. An example is shown in the RTPS protocol constraint examples. Functors by default separated by comma are treated an *and* operation.

An *or* condition handles the scenarios similar to that which we have discussed for

the other syntaxes as well, where we are not sure about the type of packet we will be receiving as a previous packet. To handle such situations, we include all the possible packets with the *or* (;) condition. The *mostrecent* functor with same value of its last parameter which is order the number of previous packets. Same value of order signifies, if one of the packet with same order is valid, the constraint should proceed satisfactorily. For example, a constraint with current packet P1, where the constraint is true if the previous packet from the same system would be of either packet type P2 or P3. In the example below, we can see two *mostrecent* functors have a same order number and separated by a semicolon to depict the or condition.

Reusing constraints doesn't require any extra naming assignment of constraints as each constraint functor name is unique in itself. To reuse the constraint, we require to call the rule again its parameters.

```
1  <constraint>
2    @pass
3    ConstraintName(SrcIP)
4      mostrecent(packet(P2
5        ,context( ..., SrcIP, ...), ...)
6        ,1);
7      mostrecent(packet(P2
8        ,context( ..., SrcIP, ...), ...)
9        ,1)
10 </constraint>
```

6.2 Constraint For RTPS and IGMP

We use the same constraints that we use for the QUEL-based syntax and Java-based syntax. We recommend the following steps for writing constraints in this representation. It is a two step process, where first steps are common for all the three syntaxes, we have proposed. Common steps include identifying pass or fail constraint, then identifying the current. The second steps different from the other two syntaxes and they are as follows:

- Define the name of constraint relevant to the constraints logic.
- Identify the previous packets required and relevant fact representation for it.
- Identify the information from the current packet that required to make logical connection for querying previous packets. It is also important because our previous packet filtering is based on attribute name matching.
- Look for the order in which we are expecting our packets for validating our constraints.

Here, we have described three constraints.

1. *All the publisher and subscribers must be valid members of at least one IGMP multi-cast group. These participants should send their membership reports to a specific group addresses before showing their interests in a topic.*

   This constraint has two current packets and IGMP member can join a multicast group using two versions IGMP protocol version2 and version3. Thus

the previous packet either from IGMP version 2 or version 3 is valid, which is represented in our constraints as V2Member and V3Member. To handle both versions of IGMP conditions, two mostrecent functors with order value 2 has been presented. Mostrecent functor with order value 2 is separated by a semicolon.

Steps for writing a constraint:

- Annotation for the constraint.
- Identify the current packets publisher (DATAWSUB) and subscriber (DATARSUB). Pick one current packet to write the constraints. In below example, current packet is publisher packet. Naming of constraint with attributes from current packet as parameter to it, as shown on Lines 15.
- To represent the previous packets, mostrecent functor with three packet representation, of which two of are in OR condition for second previous packets, as shown on Lines 16-23. After that, first previous packet shown on Lines 24.
- Finally, the constraint name is referred for another current packet subscriber (DATARSUB). As shown on Lines 41, we call this rule name with parameter which is same for both the current packets.

```
1 DATAWSUB ::= SEQUENCE {
2   kind       INTEGER (SIZE 1 BYTES),
3   flags      INTEGER (SIZE 1 BYTES),
4   nextHeader INTEGER (SIZE 2 BYTES),
```

```
5    extraFlags INTEGER (SIZE 2 BYTES),
6    qosOffset  INTEGER (SIZE 2 BYTES),
7    readerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
8    writerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
9    writerSEQ  INTEGER (SIZE 8 BYTES),
10   inlineQos  QOSPARM (SIZE DEFINED) OPTIONAL,
11   serializedData TOPICS (SIZE DEFINED) OPTIONAL
12 } (ENCODED BY CUSTOM)
13 <constraint>
14   @pass
15   datapIsIGMPMember(SrcIP, DstIP):-
16     mostrecent(packet(V2Member,
17       context( ..., SrcIP, DstIP, ...),
18       ..., DstIP, ...),
19       2);
20     mostrecent(packet(V3Member,
21       context( ..., SrcIP, DstIP, ...),
22       ..., DstIP, ...),
23       2),
24     mostrecent(packet(DATAPSUB,
25       context(.... SrcIP, DstIP, ...),
26       1)).
27 </constraint>
28 DATARSUB ::= SEQUENCE {
```

```
29   kind       INTEGER (SIZE 1 BYTES),
30   flags      INTEGER (SIZE 1 BYTES),
31   nextHeader INTEGER (SIZE 2 BYTES),
32   extraFlags INTEGER (SIZE 2 BYTES),
33   qosOffset  INTEGER (SIZE 2 BYTES),
34   readerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
35   writerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
36   writerSEQ  INTEGER (SIZE 8 BYTES),
37   inlineQos  QOSPARM (SIZE DEFINED) OPTIONAL,
38   serializedData TOPICS (SIZE DEFINED) OPTIONAL
39 } (ENCODED BY CUSTOM)
40 <constraint>
41   datapIsIGMPMember(SrcIP, DstIP)
42 </constraint>
```

2. *A topic name on a particular domain is only published from a specific set of publishers with fixed quality of service*

   In this constraint, no previous packets are required. This constraint is only required to access the static environmental information to be completed.

   Steps for writing the constraint:

   - Annotation for the constraint.
   - Identify the current packets publisher (DATAWSUB). Naming of constraint and assign parameters from current packet, as shown on Lines

14.

- Unlike the QUEL based syntax, environmental variables in the Prolog-based syntax cannot be treated as another previous packet in representation. The functor accessVar/1 with static fact base as parameter. As shown on Lines 16, the fact base of static information named staticInfo, which takes parameter as static attribute to match with current packet attributes.

```
1  DATAWSUB ::= SEQUENCE {
2    kind       INTEGER (SIZE 1 BYTES),
3    flags      INTEGER (SIZE 1 BYTES),
4    nextHeader INTEGER (SIZE 2 BYTES),
5    extraFlags INTEGER (SIZE 2 BYTES),
6    qosOffset  INTEGER (SIZE 2 BYTES),
7    readerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
8    writerEnt  ENTITYID (SIZE DEFINED) BIGENDIAN,
9    writerSEQ  INTEGER (SIZE 8 BYTES),
10   inlineQos  QOSPARM (SIZE DEFINED) OPTIONAL,
11   serializedData PARTICIPANTS (SIZE DEFINED) OPTIONAL
12 } (ENCODED BY CUSTOM)
13 <constraint>
14   @pass
15   topicStatic(SrcIP, DstIP, PIDTOPICNAME,inlineQoS):-
16     accessVar(staticInfo(SrcIP, ..., PIDTOPICNAME, ...,
     inlineQoS))
```

```
17 </constraint>
```

3. *Data of certain topic is considered valid if it is produced from a valid publisher and consumed by valid subscriber.*

   The triple dot format of syntax will not work to match attributes of different names. This means triple dot format cannot match SrcIP of current packet with DstIP of any other packet. As a result, this constraint requires complete representation of packet fact to be used on constraint representation. This will bring excessive redundancy and complexity in our syntax. Therefore, we show use of environmental variables to represent this constraint.

   Steps for writing the constraint:

   - Annotation for the constraint.
   - Identify the current packets RTPS data packets.
   - We define all the static and dynamic variables to represent this constraints, as shown on Lines 6-10.

```
1 DATASUB ::= SEQUENCE {
2   ...
3 }
4 <constraint>
5   @pass
6   dataTopicConstraint(SrcIP, DstIP, PIDTOPICNAME, readerEnt,
     writerEnt):-
```

```
7      accessVar(staticInfo(SrcIP, ..., PIDTOPICNAME));
8      accessVar(staticInfo( ..., DstIP, ..., PIDTOPICNAME))
9      accessVar(dynamicInfo(SrcIP, readEnt,));
10     accessVar(dynamicInfo(SrcIP,writerEnt))
11 </constraint>
```

There is another alternate representation presented for this constraint. In this representation, we are trying to shows how without name matching constraint representation becomes complex and non comprehensible for the user. This syntax, assign a naming for each packet attributes in code below with source information of packets as SrcIP1, SrcIP2, etc. Logical relationship with previous packets and current packet are represented in the flowing statement after packet description, as shown on Lines 8. After all the packet description, we write relationship among previous packet, as shown on Lines 11-14.

```
1  DATASUB ::= SEQUENCE {
2    ...
3  }
4  <constraint>
5    @pass
6    dataTopicConstraint(SrcIP, DstIP, writerEnt):-
7      mostrecent(packet(DATARSUB, context(SrcIP1, DstIP2, ...,
       serializationData( ...,topicData(..., topicName1, ...)))))),
        1);
```

```
 8      mostrecent(packet(DATAWSUB, context(SrcIP2, DstIP2, ...,
       serializationData(entityId2,topicData(..., topicName2, ...)
       ))), 1),
 9      mostrecent(packet(DATARSUB, context(SrcIP3, DstIP3, ...,
       serializationData(entityId3,topicData(..., topicName3, ...)
       ))), 2),
10        SrcIP3 = DstIP;
11      mostrecent(packet(DATAWSUB, context(SrcIP4, DstIP4, ...,
       serializationData(entityId4,topicData(..., topicName4, ...)
       ))), 2),
12        SrcIP4 =SrcIP,
13      DstIP1 =SrcIP4;
14      DstIP2 = SrcIP3,
15      topicName1 = topicName4;
16      topicName2 = topicName3.
17  </constraint>
```

6.3 Prolog-based Syntax Summary

In this chapter, we propose the third and last syntax for representing the constraints for the private network. Prolog brings a totally different perspective, being a functional language. The major challenge is understanding the Prolog concepts and then creating a hypothesis for representing our network packets in Prolog facts. We have defined few implicit functor to address various scenarios of the stream of packets in

the network. We introduce a triple dot concepts, which hugely simplify the fact representation in constraint. It removes redundancy and only requires relevant part of fact base to be presented for constraint. We introduce mostrecent functor, to access the packets in proper order for the constraints. We introduce another functor with name accessVar with one and two parameter to access static and dynamic variable respectively.

The Prolog-based constraint representation, based on matching attribute name from current packet if used with triple dot technique. To use in complex scenarios of matching attribute with different name become difficult with this syntax. On the other hand removing triple dot completely, bring huge redundancy in representation by remembering the exact structure of fact base.

Chapter 7

Evaluation and Mapping

7.1 Evaluation

All the three syntaxes are designed to address the current limitation of the SCL constraint representation. We covered scenarios such as multi-packet representation with packet order. We introduced the concept of environmental variables for all the three syntaxes. Our three syntaxes based on Java, QUEL and Prolog are complete in their own programming paradigm. To incorporate scenarios relevant for our constraint, we have introduced changes in each of these syntaxes. Though each syntax as there are based on different programming language, we have came up with comparison parameter to evaluate these syntax. We use four parameters based on the domain specific language design guidelines proposed to compare our syntaxes [20]. These four parameters are measured on three values which is low, medium and high. The inference of benchmark is defined for each parameters. The four parameters are:

1. *Redundancy*: In this parameter, we check syntax for information which is not relevant for the constraint. Redundancy can be come from our modification and from actual syntax handling our constraint. To compare the three syntaxes, we

rank the constraint representation on the basis of their redundancy. The syntax with least redundancy are best in rank and termed as *low*. The syntax with most redundancy are ranked wost and termed as *high*. The rank between least redundant and most redundant are termed as *medium*.

2. *Number of elements (NOE)*: In this parameter, we check how many keywords constraint writer need to be aware to write any constraint in our syntaxes. A low number of elements makes it easier for learning about the syntax. A lower the NOE is better for DSL. A low NOE is not more than 3 elements. A medium NOE is NOE greater than 3 but no other information. A high NOE means the number of elements is more than 3 and other packet structural information is required. The NOE is not dependent of constraint, it is a property of each syntax. Therefore, a syntax will have same NOE for all constraint in our evaluation.

3. *Re-usability of constraint*: This parameter determine the ability to reuse the syntax component without re-writing complete representation. A *low* re-usability signifies that user cannot re-use the already existing constraint without additional information. A *high* re-usability for our constraint would mean, we can re-use existing constraint as part of other constraint. A *medium* re-usability means the constraint are re-usable constraint can be re-use for current packet with same parameters.

4. *Code-repetition (CE)*: This parameter evaluates code repetition. This is based on the fact that how much code repetition is happening in for a syntax to represent a constraint. A high repetition of code add redundancy and extra

effort in the representation. It also how much syntaxes are following software engineering principle 'don't repeat yourself' popularly know as DRY principle [17]. A low CE value means that syntax is best in constraint representation compared to other two syntaxes. Three syntaxes are ranked, best one given *low* CE and worst rank syntax means high *CE*.

For the purpose of evaluation, we have compared these parameters based on three constraint representation. Three constraint are referred as C1, C2 and C3 in the same order as discussed in syntax chapters. We see, how each syntax measure on these parameter for syntax representation for three constraints.

Syntax Comparison

Criteria	JAVA			QUEL			PROLOG		
	C1	C2	C3	C1	C2	C3	C1	C2	C3
Redundancy	low	low	low	low	low	low	low	low	high
NOE	low	low	low	med	med	med	high	high	high
Re-usability	med	med	med	med	med	med	med	med	med
CE	low	low	low	low	low	med	low	low	low

med : Medium

Table 7.1: Comparison of syntaxes based on 4 parameters

We have observed the Prolog-based syntax has high redundancy for the constraint C3 as it cannot be represented using triple dot format. Another shortcoming of using the Prolog-based syntax is that it only matches previous packet based on current packet attributes in name matching. The Java and QUEL based syntax has low

redundancy in all the constraints. Java and QUEL based syntax doesn't require redundant information in terms of packet structure represent the constraint, as shown in Table 7.1.

Constraints represented in all the three syntax can be easily reusable for same constraint scenario with any other current packets. The number of elements for Prolog-based syntax is highest as user has to know the packet fact for all the protocols required for the syntax. The user require to know the order in which attributes are present in the packet facts to use it in the constraint. Similarly, user are required to know the table format for all the packet type to use it in the constraints. Using table structure doesn't require user to know the order of columns to write constraints. For Java-based syntax, user can refer the SCL modules to use as object representation to write constraint.

The QUEL based syntax has highest repetition of code for the constraint C3. The where condition have same logical statements for multiple retrieve statement for the constraint C3. From Table 7.1, we have observed our Java-based syntax are best compared to other two syntax proposed in this thesis.

7.2 Mapping

In this chapter, we have shown the mapping with our SCL constraint syntax with lower level Domain Specific language developed by Hassan [15] for auto-generation. Mapping aims to show information for automatic conversion of SCL constraint to constraint engine DSL. For generating constraint engine DSL, we use SCL modules defined with SCL constraints. Generated constraint engine DSL should be independent of SCL definition and will be further used for automatic generation of constraint

engine code.

In our studied, we have shown mapping of Java based expression syntax 4 to constraint engine DSL. The constraint engine DSL is four step process, which is Instantiate, Bind, Evaluate and Destroy.

1. ***Instantiate***: Instantiate is the stage where the first packet for constraint validation is received, in this stage, we extract the information from the packet for the further capture of packets that might be required for constraints. Packet information required for filtering future packet in the stream of packets are sent as search parameter to next stage.

2. ***Bind***: Bind stage is where packet information passed as search parameter from Instantiate stage are used as filter parameter for packets. These parameter only use to search a given packet type mentioned at the start of bind stage. Once the packets are found, new search parameter is set either same as Instantiate or new parameter for evaluation stage.

3. ***Evaluation***: It is a final stage of validation of constraints, at this stage search parameter from the previous stage is used to find a packet for evaluation stage, if a packet is found with given search parameter, then the constraint is passed else failed.

4. ***Destroy***: At this stage constraints tree which has been formed at C language code are deleted if the condition of destroy stage is met. Destroying is not currently part of our syntax shown in our thesis. Destroy require more information than just packet information. It is also not the part of constraint validation. Therefore, currently our representation doesn't show mapping for destroy stage.

7.2.1 Mapping Example

For a constraint from ATC simulation network, low-level DSL is represented as follows:

Constraints: All the subscribers must be valid members of at least one IGMP multicast group. These participants should send their membership reports to specific group addresses before showing their interests in a topic.

The goal of syntax representation, is to make constraint representation easy and simple to write. Using SCL makes the parser and the constraint engine with a single interface. We required an interface which is human readable and simple for use. A parser is already using SCL, so we extended our SCL for constraint engine integration with a goal to keep it simple for the user, remove redundancy from representation. Lower level DSL does not shows these properties, it is mainly written for generating C language code for constraint engine. It has redundancy and repetition of information and representation is not easily human comprehensible.

Here, we present the SCL representation for a constraint using Java-based syntax and equivalent low level DSL to which we will show mapping.

Our SCL constraint representation for the above-mentioned constraint.

```
1 <constraint>
2   C1:
3   @pass
4   (dataP = DATAPSUB, v2Mem = V2Member | v3Mem = V3Member;
5   dataP.SrcIP = SrcIP &
6   dataP.DstIP = DstIP &
7   (v2Mem.groupAddress = DstIP | v3Mem.groupAddress = DstIP);
8   dataP.order = 1, v2Mem | v3Mem = 2;);
```

```
9  </constraint>
10
```

Low-level DSL representation for the same constraint, as shown by Hassan [15].

```
1  INSTATNTIATE
2  IGMP V2member or V3Member
3  if V2member then
4  SrcIPJ = Packet.SrcIP as Integer,
5  groupIPJ = Packet.groupaddress as Integer
6  end
7  if V3Member then
8  loop Packet.groupRecordInfo SrcIPJ = Packet.SrcIP,
9  groupIPJ = Packet.groupRecord as Integer
10 endif
11 Key = Packet.SrcIP as Integer, Packet.groupaddress as Integer
12
13 BIND
14 RTPS Packet.SUBMSG contains DATAPSUB
15 SEARCH Packet.SrcIP, Packet.DstIP
16 SrcIPP = Packet.SrcIP as Integer,
17 DstIPJ = Packet.DestIP as Integer
18 Key = Packet.SrcIP as Integer, Packet.groupaddress as Integer
19
20 EVALUATE
```

```
21 RTPS Packet.SUBMSG contains DATARSUB or Packet.SUBMSG contains
       DATAPSUB
22 SEARCH Packet.SrcIP, Packet.DstIP
23 EVAL Packet.SrcIP as Integer, Packet.DstIP as Integer
```

In this chapter, we are showing the mapping between the various entries from syntax with constraint engine DSL. Side by side mapping for each of the Instantiate, Bind and Evaluate stage has been shown. Our current SCL constraints have only considered the first three stage.

Mapping for Instantiate from our Java-based expression. It is important to know that, we should have all the information.

7.2.2 Instantiate Mapping

Instantiate stage can be further divided into three parts. First, the part where we declare a packet, we required for the instantiate stage. Instantiate can handle one packet at a time, but we can give multiple choice of packets of which one can be used at a time. Mapping to this part of instantiate with Java-based syntax coding can be seen in the diagram 7.1, where we have shown a mapping what java-syntax will be used to translate syntax for the first part of instantiate. We use a simple version of SCL constraint for demonstrating our mapping.

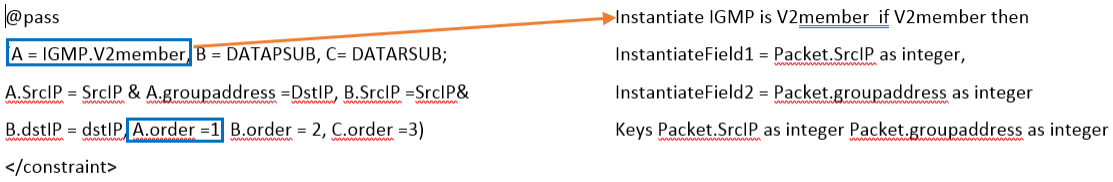

Figure 7.1: Instantiate declaring packets

In the second part, we extract information from the packet which we require to further to build the constraint. We assign extracted value to a variable with the type of each field. Once all the variables is defined, in the third part, we make these variables available for next stage Bind to filter the right packet required for this stage. From the diagram 7.2, we can see that the mapping from Java-based syntax to second part of instantiate requires the type information from ASN.1 notation for translation.

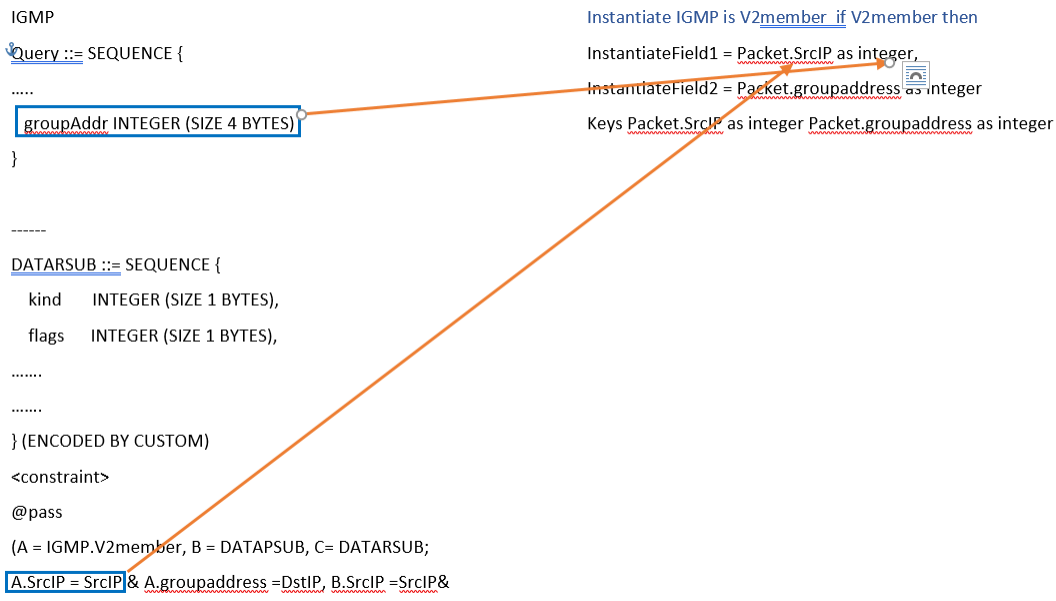

Figure 7.2: Instantiate packet information to variables

In the third part, we make the information available for next stage which is the

bind stage, in this, low-level DSL pick the values from instantiate packet to be used to filter bind packet as the key. Our mapping in diagram 7.3, shows that our syntax has enough information to generate all the three parts of instantiate stage. The major difference between the second and third part is that second part is to validate the instantiate packet, the third part is for next stage to filter packet from the packet stream.

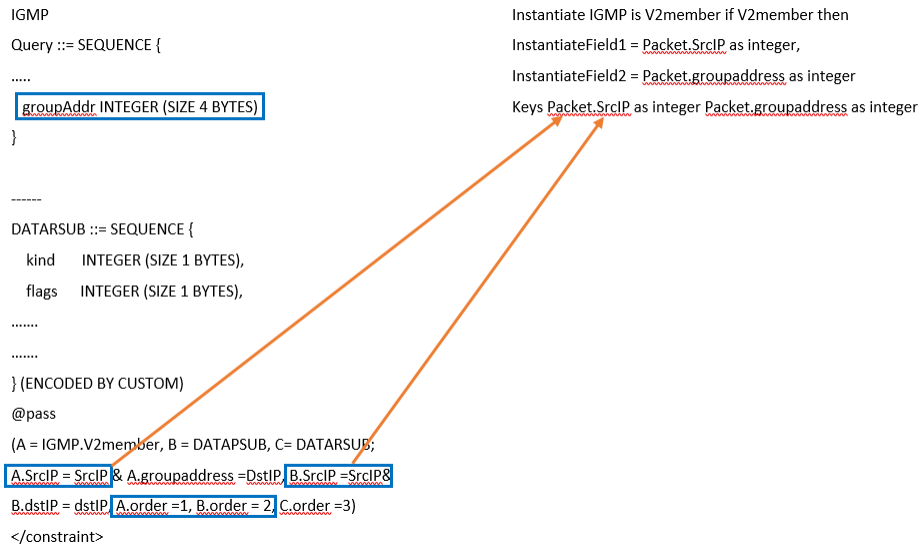

Figure 7.3: Instantiate packet information for next stage

7.2.3 Bind Mapping

Similar to instantiate stage, bind stage also filters packet on their type and as the first part of their syntax. Unlike instantiate which only uses packet type or sub-message (for RTPS protocols) type to filter packet, bind stage use information from instantiate stage as well to filter the packets. Bind stage can be divided into four parts for the purpose of mapping: Declaration of packet type, search value from the

previous stage, information assignment from current stage, information extraction for next stage which is an evaluation.

Declarations part is very similar to instantiate declaration part except for the prefix of BIND in syntax.

Search part is where bind stage uses the information from the previous stage for further filtering the packet which is required for our constraints. It starts with prefix *Search* followed by the previous packet attributes which have been set as the key.

The third part of the binding is similar to one with instantiate state, extraction of information from the packet and assigning to the variables.

The fourth part is also similar to third part of instantiate stage which is assigning the information for the next stage. This part of bind start with prefix *Key* and followed by the attributes of the packet to be used for the next stage of searching. Similar mapping can be found for bind stage shown in diagram 7.4

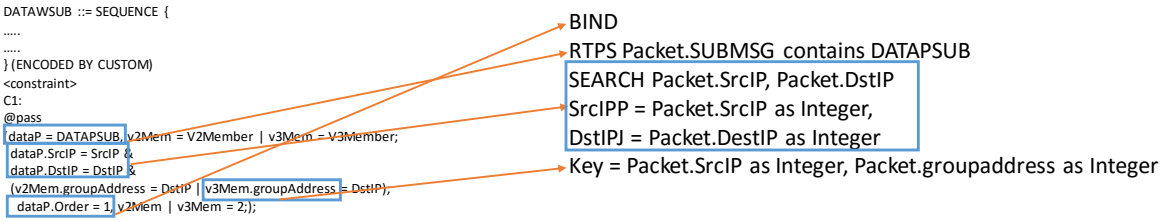

Figure 7.4: Bind stage

7.2.4 Evaluate Mapping

Evaluate stage is a final stage of validation of constraint. At this stage, it doesn't transfer any information from current packet to next stage, we don't have *key* part to extract information for next stage. This stage has three parts which are declaration of packet similar to instantiate and bind stage, search stage where we further filter packets based on information from the previous stage which is similar to bind stage *search* and finally we have evaluation part, where we evaluate we evaluate finally this packet is valid or not, validity of this packet and evaluation stage also finalizes the complete validation of constraints. Evaluate stage doesn't transfer any information to next stage for filtering and further evaluation of any time though we have one more stage called destroy, which is only here to make sure destroy internal tree created for constraint validation should be cleaned for fresh packets and constraints. In this thesis, we have not provided a system to destroy stage yet. Mapping for evaluate stage can be found here in diagram 7.5

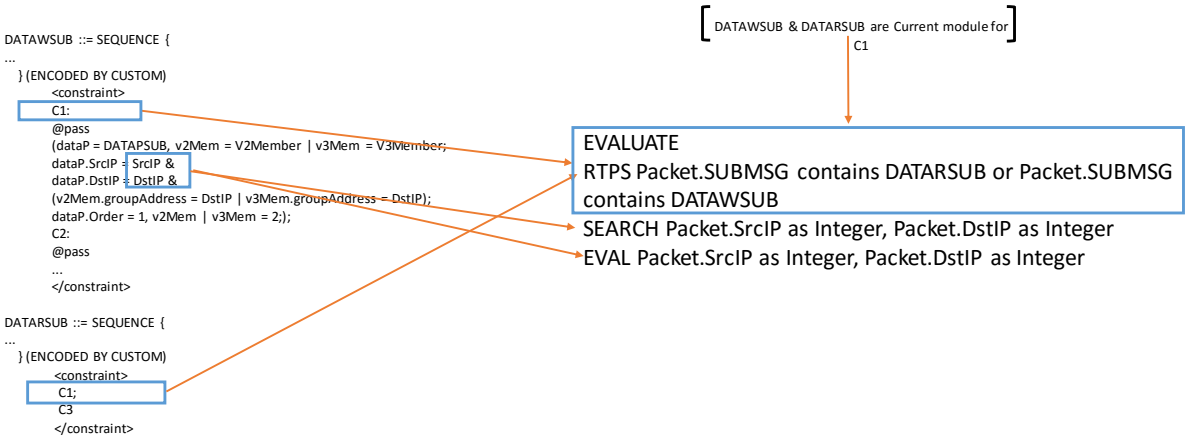

Figure 7.5: Evaluate stage

The below example shows side by side view of other two constraint low-level DSL and Java-based representation in side by side view.

Constraint: A topic name of a particular domain is only published from a specific set of publishers with fixed quality of service

Listing 7.1: Java-based syntax

```
1 DATAWSUB ::= SEQUENCE {
2 }
3 <constraint>
4   C1:
```

Listing 7.2: Low-level DSL

```
1 INSTANTIATE load
2 SrcIP = FactC8.SrcIP,
3 Topic = HashKey(Fact8.TopicName),
4 Entity = Fact8.Entity,
```

```
5   @pass
6   ( ;static(FactData.SrcIP) ==
     SrcIP &
7   static(FactData.TopicName) ==
     TOPICS.TOPICSPARAM<
     PIDTOPICNAME>
8   & static(FactData.QoS == TOPICS
     );;)
9  <constraint>
```

```
5  QoS = struct(Fact8.QoS)
6  Key = Fact8.SrcIP,
7  HashKey(Fact8.TopicName), Fact8.
       Entity
8
9  BIND
10
11 EVALUATE
12 RTPS Packet.SUBMSG contains
       DATAPSUB
13 EVAL
14 Packet.SrcIP, HashKey(Packet.
       TOPICPARMS.TOPICNAME),
15 Packet.Entity, struct(Packet.QoS)
```

Constraint: Data of certain topic is considered valid if it is produced from a valid publisher and consumed by valid subscriber.

Listing 7.3: Java-based syntax

```
1 DATASUB ::= SEQUENCE {
2 }
3 <constraint>
4   C1:
5   @pass
```

Listing 7.4: Low-level DSL

```
1 Instantiate
2 RTPS Packet.SUBMSG contains
       DATAWSUB Or
3 RTPS Packet.SUBMSG contains
       DATARSUB
```

```
6    (dw1 = DATAWSUB | dr1 =
      DATARSUB, dr2 = DATARSUB |
      dw2 = DATAWSUB;
7    ((dw1.DstIP == dr2.SrcIP & dr2.
      SrcIP == DstIP)
8    | (dr1.DstIP == dw2.SrcIP & dw2
      .SrcIP == SrcIP))
9    & ((dw1.TopicParams.TopicName
      == dr2.TopicParams.TopicName)
10   | (dr1.TopicParams.Topic == dw2
      .TopicParams.TopicName))
11   &(dw1.entityId == writerEnt )
12   &( dw2.entityId == writerEnt );
13   dw1.order | dr1.order = 1, dr2.
      order | dw2.order = 2;
14   )
15 </constraint>
```

```
4  if Packet.SUBMSG contains
       DATAWSUB
5  if SEARCH HashKey(Packet.
       TOPICPARMS.TOPICNAME, Packet.
       dstIP)
6  Bind Packet.srcIP, Packet.
       entityID
7  else
8  SrcIPPu = Packet.SrcIP
9  entityIDPu = Packet.entityID
10 endif
11 Key = HashKey(Packet.TOPICPARMS.
       TOPICNAME), packet.srcIP
12 endif
13 if Packet.SUBMSG contains
       DATARSUB
14 if SEARCH HashKey(Packet.
       TOPICPARMS.TOPICNAME, Packet.
       dstIP)
15 Bind Packet.srcIP
16 else
17 SrcIPSu = Packet.SrcIP
18 endif
19 Key=HashKey(Packet.TOPICPARMS.
```

```
      TOPICNAME), Packet.SrcIP
20 endif
21
22 BIND
23 RTPS Packet.SUBMSG contains
      DATAWSUB Or
24 RTPS Packet.SUBMSG contains
      DATARSUB
25 if RTPS Packet.SUBMSG contains
      DATARSUB
26 SEARCH HashKey(Packet.TOPICPARMS.
      TOPICNAME, Packet.DstIP)
27 {SPLIT}
28 SrcIPSu = Packet.SrcIP
29 Key=HashKey(Packet.TOPICPARMS.
      TOPICNAME), Packet.DstIP
30 endif if RTPS Packet.SUBMSG
      contains DATAWSUB
31 SEARCH HashKey(Packet.TOPICPARMS.
      TOPICNAME, Packet.DstIP)
32 {SPLIT}
33 SrcIPPu = Packet.SrcIP
34 entityIDPu = Packet.entityID
35 Key=HashKey(Packet.TOPICPARMS.
```

```
        TOPICNAME), Packet.DstIP
36  endif
37
38  EVALUATE
39  RTPS Packet.SUBMSG contains
        DATASUB and
40  IsUnicastCommunication(Packet.
        SrcIP, Packet.DstIP)
41  SEARCH HashKey(Packet.TOPICPARMS.
        TOPICNAME), Packet.SrcIP
42  EVAL Packet.SrcIP,
43  Packet.TOPICPARMS.entityid,
        packet.DstIP
```

7.3 Evaluation and Mapping Summary

We have compared our three syntaxes based on four DSL design guidelines. Java-based syntax shows best adherence with these four design guidelines. Mapping between Java-based syntax and low-level DSL shows that Java-based syntax has all relevant information to derive the low-level DSL.

Chapter 8

Summary and Future Works

8.1 Summary

In this thesis, we have extended SCL with three new syntax to represent complex network constraint scenarios. These network constraints include multiple packets from more than one protocols. The scenarios relevant to the constraint representation include relations between the packet attributes, packet order and environmental variables. We designed our syntaxes with the goal to address following three fundamental design requirements.

1. All relevant information to generate the low-level DSL
2. Cover the constraint scenarios
3. Adhere to the DSL design guidelines

Our syntaxes are based on three different languages from three different paradigm of programming language. First syntax is based on Java expression, which uses the SCL modules as object reference and the relationship between the packet attributes are similar to Java arithmetic expressions. Second language is the QUEL which is

a declarative language used for querying structured data. The third is the Prolog a declarative language which is used for logic programming. The three styles gives different perspective for writing network constraints. Each syntax posses distinct challenge in terms of using network data for representation.

Each of the syntaxes represent three constraints for the RTPS and the IGMP protocols generated using the ATC simulation. These constraint cover the widest range of constraint scenarios as found in Hassan's work [15]. These constraint check capability and limitation of our syntax in representing possible constraint scenarios.

Compared to the low-level DSL, three syntaxes are simpler and more flexible in terms of their structure and representation. Three syntaxes use less line of codes to represent same constraints compared to the low-level DSL. We have compared our three syntaxes based on the four DSL design guidelines to evaluate our syntax. The four criteria of the DSL design guidelines used are redundancy, number of lines of code, code repetition and the number of elements. After comparing the three syntaxes, we have found that the Java-based syntaxes is better compared to the other two syntaxes. The mapping of the Java-based syntax and low-level DSL shows that the proposed syntax has all the relevant information required for the low-level DSL transformation.

Finally, concept of environmental variables is proposed for the first time as the part of the SCL syntax. Environmental variables reduce the complexity of the constraint representation for some of the constraints as shown by second constraint in Section 4.2. Similarly, we have introduced pass and fail annotation in order to simplify the constraint representation, as mentioned in Section 3.3.

8.2 Future Work

There are scenarios which are not relevant to our current IDS constraint but might come up in future. we have mentioned few of those constraint scenarios where our current syntaxes representations would not be sufficient to represent corresponding constraints. A protocol such as the RTPS has multiple submessages in single packet. The number of submessages and their order within a packet is not predetermined. A RTPS protocol constraint for a single packet which requires either the number of submessages or their order cannot be represented currently in our three syntaxes.

There is a possibility of including the scenario for the destroy stage describer in Hassan's low-level DSL [15]. This might require some external information on top of packet specification to determine when to destroy a constraint tree. The future work should focus on using Java-based syntax to auto-generate the low-level DSL for the constraint engine.

Bibliography

[1] 2013 report on cyber security. *Center for Internet Security Annual Report*, 2013.

[2] R Bacher. Computer aided power flow software engineering and code generation. In *Power Industry Computer Application Conference, 1995. Conference Proceedings., 1995 IEEE*, pages 474–480. IEEE, 1995.

[3] James Ball, Julian Borger, Glenn Greenwald, et al. Revealed: how us and uk spy agencies defeat internet privacy and security. *The Guardian*, 6, 2013.

[4] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[5] Jeffrey D Case, Mark Fedor, Martin L Schoffstall, and James Davin. Simple network management protocol (snmp). Technical report, 1990.

[6] Andrei Costin and Aurélien Francillon. Ghost in the air (traffic): On insecurity of ads-b protocol and practical attacks on ads-b devices. *Black Hat USA*, pages 1–12, 2012.

[7] Dennis Cox and Kip McClanahan. Method for blocking denial of service and address spoofing attacks on a private network, May 18 2004. US Patent 6,738,814.

[8] Michael K Daly. Advanced persistent threat. *Usenix, Nov*, 4(4):2013–2016, 2009.

[9] Ali ElShakankiry. Context sensitive and secure parser generation for deep packet inspection of binary protocols. *Queen's University Thesis*, 2017.

[10] What APT Means To Your Enterprise and Greg Hoglund. Advanced persistent threat.

[11] Dirk Fahland, Daniel Lübke, Jan Mendling, Hajo A Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal. Declarative versus imperative process modeling languages: The issue of understandability. *BMMDS/EMMSAD*, 1(29):353–366, 2009.

[12] James P Farwell and Rafal Rohozinski. Stuxnet and the future of cyber war. *Survival*, 53(1):23–40, 2011.

[13] William C Fenner. Internet group management protocol, version 2. 1997.

[14] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. A model-based approach for robustness testing. In *TestCom*, volume 5, page 333. Springer, 2005.

[15] Md Siam Hasan. A constraint-based intrusion detection system for private networks. *Queen's University Thesis*, 2017.

[16] Klaus Havelund, Rahul Kumar, Chris Delp, and Bradley Clement. K: A wide spectrum language for modeling, programming and analysis. In *MODELSWARD*, pages 111–122, 2016.

[17] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

[18] Chuanming Jing, Zhiliang Wang, Xia Yin, and Jianping Wu. A formal approach to robustness testing of network protocol. *Network and Parallel Computing*, pages 24–37, 2008.

[19] Rauli Kaksonen. A functional method for assessing protocol implementation security (licentiate thesis). espoo. 2001. technical research centre of finland. *VTT Publications*, 447:128.

[20] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design guidelines for domain specific languages. *arXiv preprint arXiv:1409.2378*, 2014.

[21] Anneke G Kleppe, Jos Warmer, Wim Bast, and MDA Explained. The model driven architecture: practice and promise, 2003.

[22] Cris Kobryn. Uml 2001: a standardization odyssey. *Communications of the ACM*, 42(10):29–37, 1999.

[23] Paul J Leach and Dilip Naik. A common internet file system (cifs/1.0) protocol. Technical report, Internet-Draft, IETF, 1997.

[24] John W Lloyd. Practical advtanages of declarative programming. In *GULP-PRODE (1)*, pages 18–30, 1994.

[25] Torsten Lodderstedt, David Basin, and Jürgen Doser. Secureuml: A uml-based modeling language for model-driven security. *UML 2002 The Unified Modeling Language*, pages 426–441, 2002.

[26] Michael R Lyu and Lorrien KY Lau. Firewall security: Policies, testing and performance evaluation. In *Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International*, pages 116–121. IEEE, 2000.

[27] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding bgp misconfiguration. In *ACM SIGCOMM Computer Communication Review*, volume 32, pages 3–16. ACM, 2002.

[28] Sylvain Marquis, Thomas R Dean, and Scott Knight. Scl: a language for security testing of network applications. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 155–164. IBM Press, 2005.

[29] Michael C McCord. Lmt: a prolog-based machine translation system. *extended abstract), in Nirenburg (1985, these references)*, pages 179–182, 1985.

[30] Michael C McCord. Design of lmt: A prolog-based machine translation system. *Computational Linguistics*, 15(1):33–52, 1989.

[31] Michael James Michaud. Unintented use of omg data distribution service in real-time mission critical distributed systems. *Royal Military COllege of Canada Thesis*, 2017.

[32] C Mohan. Ibm's relational dbms products: features and technologies. In *ACM SIGMOD Record*, volume 22, pages 445–448. ACM, 1993.

[33] David Moore, Colleen Shannon, Douglas J Brown, Geoffrey M Voelker, and Stefan Savage. Inferring internet denial-of-service activity. *ACM Transactions on Computer Systems (TOCS)*, 24(2):115–139, 2006.

[34] John Moy. Ospf version 2. 1997.

[35] Jun Na and V Rajaravivarma. Multimedia file sharing in multimedia home or office business networks. In *System Theory, 2003. Proceedings of the 35th Southeastern Symposium on*, pages 237–241. IEEE, 2003.

[36] Gerald Neufeld and Son Vuong. An overview of asn. 1. *Computer Networks and ISDN Systems*, 23(5):393–415, 1992.

[37] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004.

[38] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

[39] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Notices*, volume 23, pages 199–208. ACM, 1988.

[40] Lee Rainie, Sara Kiesler, Ruogu Kang, Mary Madden, Maeve Duggan, Stephanie Brown, and Laura Dabbish. Anonymity, privacy, and security online. *Pew Research Center*, 5, 2013.

[41] Silvio Ranise and Cesare Tinelli. The smt-lib standard: Version 1.2. Technical report, Technical report, Department of Computer Science, The University of Iowa, 2006. Available at www. SMT-LIB. org, 2006.

[42] Lawrence A Rowe and Michael R Stonebraker. The postgres data model. *Readings in object-oriented database systems*, pages 461–473, 1990.

[43] Fares Saad-Khorchef, Antoine Rollet, and Richard Castanet. A framework and a tool for robustness testing of communicating software. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1461–1466. ACM, 2007.

[44] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[45] Petr Smolik, Zdenek Sebek, and Zdenek Hanzalek. Orteopen source implementation of real-time publish-subscribe protocol. In *Proc. 2nd International Workshop on Real-Time LANs in the Internet Age*, pages 68–72, 2003.

[46] Michael Stonebraker, Erika Anderson, Eric Hanson, and Brad Rubenstein. *QUEL as a data type*, volume 14. ACM, 1984.

[47] Michael Stonebraker, Jeff Anton, and Eric Hanson. Extending a database system with procedures. *ACM Transactions on Database Systems (TODS)*, 12(3):350–376, 1987.

[48] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The design and implementation of ingres. *ACM Transactions on Database Systems (TODS)*, 1(3):189–222, 1976.

[49] Michael Stonebraker and Greg Kemnitz. The postgres next generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

[50] Michael Stonebraker and Lawrence A Rowe. *The design of Postgres*, volume 15. ACM, 1986.

[51] Michael Stonebraker, Lawrence A Rowe, and Michael Hirohama. The implementation of postgres. *IEEE transactions on knowledge and data engineering*, 2(1):125–142, 1990.

[52] Colin Tankard. Advanced persistent threats and how to monitor and deter them. *Network security*, 2011(8):16–19, 2011.

[53] Juha-Pekka Tolvanen and Steven Kelly. Domain-specific modeling: Enabling full code generation. *Wiley-IEEE Computer Society*, 444:231, 2008.

[54] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+: defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93. ACM, 2003.

[55] Bruce Trask, Dominick Paniscotti, Angel Roman, and Vikram Bhanot. Using model-driven engineering to complement software product line engineering in developing software defined radio components and applications. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 846–853. ACM, 2006.

[56] Jos Warmer and Klasse Objecten. The future of uml. *OMG Information Day, Amsterdam*, 2001.

[57] Yorick Wilks, Xiuming Huang, and Dan Fass. *Syntax, preference and right attachment*. Computing Research Laboratory, New Mexico State University, 1985.

[58] Shu Xiao, Sheng Li, Xiangrong Wang, and Lijun Deng. Fault-oriented software robustness assessment for multicast protocols. In *Network Computing and Applications, 2003. NCA 2003. Second IEEE International Symposium on*, pages 223–230. IEEE, 2003.

[59] Songtao Zhang, Thomas Dean, and Scott Knight. A lightweight approach to state based security testing. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 28. IBM Corp., 2006.

[60] Songtao Zhang, Thomas Dean, and Scott Knight. Lightweight state based mutation testing for security. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 223–232. IEEE, 2007.

Appendix A

SCL

A.1 IGMP:SCL

```
1  IGMP DEFINITIONS ::= BEGIN
2
3  EXPORTS PDU;
4
5  PDU ::= ( V3Report | Query | V2Report | V2Leave)
6  <transfer>
7     Callback
8  </transfer>
9
10 Query ::= SEQUENCE {
11    type INTEGER (SIZE 1 BYTES),
12    maxRespTime INTEGER (SIZE 1 BYTES),
13    checksum INTEGER (SIZE 2 BYTES),
14    groupAddr INTEGER (SIZE 4 BYTES),
```

```
15     -- v3Add     set of V3Addition (SIZE DEFINED) -- really need opt (0
        or 1)
16     v3Add V3Addition (SIZE DEFINED) OPTIONAL
17  }
18  <transfer>
19     Back{type == 17} -- 0x11
20     Forward { EXISTS(v3Add) == PDUREMAINING }
21  </transfer>
22
23
24  V3Addition ::= SEQUENCE {
25     resvSQRV INTEGER (SIZE 1 BYTES),
26     QQIC INTEGER (SIZE 1 BYTES),
27     numSources INTEGER (SIZE 2 BYTES),
28     srcAddrs SET OF SOURCEADDRESS (SIZE CONSTRAINED)
29  } (ENCODED BY CUSTOM)
30  <transfer>
31     Forward{CARDINALITY(srcAddrs)  == numSources}
32  </transfer>
33
34  V2Report ::= SEQUENCE {
35     type INTEGER (SIZE 1 BYTES),
36     maxRespTime INTEGER (SIZE 1 BYTES),
37     checksum INTEGER (SIZE 2 BYTES),
```

```
38     groupAddr INTEGER (SIZE 4 BYTES)
39 } (ENCODED BY CUSTOM)
40 <transfer>
41     Back{type == 22} -- 0x16
42 </transfer>
43
44 V2Leave ::= SEQUENCE {
45     type INTEGER (SIZE 1 BYTES),
46     maxRespTime INTEGER (SIZE 1 BYTES),
47     checksum INTEGER (SIZE 2 BYTES),
48     groupAddr INTEGER (SIZE 4 BYTES)
49 } (ENCODED BY CUSTOM)
50 <transfer>
51     Back{type == 23} -- 0x17
52 </transfer>
53
54 V3Report ::= SEQUENCE {
55     type INTEGER (SIZE 1 BYTES),
56     reserved INTEGER (SIZE 1 BYTES),
57     checksum INTEGER (SIZE 2 BYTES),
58     secondReserved INTEGER (SIZE 2 BYTES),
59     numGrps INTEGER (SIZE 2 BYTES),
60     groupRecordInfo SET OF GROUPRECORD (SIZE CONSTRAINED)
61 } (ENCODED BY CUSTOM)
```

```
62 <transfer>
63    Back{type == 34} -- 0x22
64    -- Forward{LENGTH(groupRecordInfo)  == (PDULENGTH - (SIZEOF(type) +
       SIZEOF(reserved) + SIZEOF(checksum) + SIZEOF(secondReserved) +
      SIZEOF(numGrps)))} -- This really should be a cardinality
      constraint on numGrps
65    Forward{CARDINALITY(groupRecordInfo) == numGrps}
66 </transfer>
67
68 GROUPRECORD ::= SEQUENCE {
69    recordType INTEGER (SIZE 1 BYTES),
70    auxDataLen INTEGER (SIZE 1 BYTES),
71    numSources INTEGER (SIZE 2 BYTES),
72    groupAddr INTEGER (SIZE 4 BYTES),
73    srcAddrs SET OF SOURCEADDRESS (SIZE CONSTRAINED)
74 } (ENCODED BY CUSTOM)
75 <transfer>
76    Forward{CARDINALITY(srcAddrs) == numSources}
77 </transfer>
78
79 SOURCEADDRESS ::= SEQUENCE {
80   srcAddr INTEGER (SIZE 4 BYTES)
81 } (ENCODED BY CUSTOM)
82 END
```