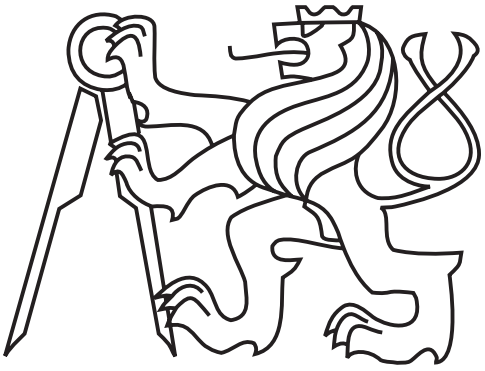
Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Control Engineering



Diploma Thesis

Message schedule verification for Profinet IRT

2015 Author: Bc. Lukáš Halíř

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Control Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Lukáš Halíř**

Study programme: Cybernetics and Robotics
Specialisation: Systems and Control

Title of Diploma Thesis: **Message schedule verification for Profinet IRT**

Guidelines:

1. Design a structure of a software tool to check the qualities of a message schedule for Profinet IRT.
2. Implement the tool and check the validity of the message schedules based on a given set of rules.
3. Study the usage of tool UPPAAL for model checking of time-critical systems.
4. Specify the state machine to control the forwarding of IRT messages in Profinet IO Device.
5. Implement the state machine in UPPAAL.
6. Create a model of a Profinet IRT network with a series of devices (line topology). Check if the behaviour of the network corresponds to a given message schedule.

Bibliography/Sources:

[1] Industrial communication networks - Fieldbus specifications - Part 5-10: Application layer service definition - Type 10 elements. IEC 61158-5-10 ed.3. 08/2014.
[2] Industrial communication networks - Fieldbus specifications - Part 6-10: Application layer protocol specification - Type 10 elements. IEC 61158-6-10 ed.3. 08/2014.
[3] Behrmann, G., David, A., Larsen, K.G. A Tutorial on Uppaal 4.0. 2006.

Diploma Thesis Supervisor: Ing. Pavel Burget, Ph.D.

Valid until the summer semester 2015/2016

prof. Ing. Michael Šebek, DrSc.
Head of Department

L.S.

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, February 20, 2015

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, dne 11. 5. 2015 .....

.....

Podpis

Acknowledgment

At this place I would like to thank my supervisor Pavel Burget for his advices and comments on my work. Also thanks to Ondřej Fiala who helped me with various issues I have encountered during the work on this thesis. Last but not least, thanks to my family and everybody who supported me during my studies.

Abstract

This thesis deals with verification of message schedules in Profinet IRT networks. For the verification two approaches were implemented and described in this thesis.

First approach is to verify the schedule by static check with given set of rules. A test tool with verification rules was implemented in this thesis and several schedules were verified.

In the second approach a model of Profinet network was created and parametrized by the schedule. The model was created in Uppaal software, which performs verification of timed automata systems according to specified requirements. Network with line topology was modeled and verified that it behaves according to the given schedule.

Abstrakt

Tato práce se zabývá ověřováním plánu zpráv v sítích Profinet IRT. Pro ověřování byly v této práci implementovány a popsány dva přístupy.

Prvním z přístupů je oveření plánu statickou kontrolou jeho parametrů podle zadaných pravidel. V této práci byl implementován testovací nástroj se zadanými pravidly a s jeho pomocí bylo ověřeno několik plánů.

V druhém přístupu byl vytvořen model sítě Profinet a byl parametrizován zadaným plánem. Model byl vytvořen v programu Uppaal, který podporuje verifikaci časovaných stavových automatů podle zadaných požadavků. Byla namodelována síť s liniovou topologií a bylo ověřeno její chování podle plánu.

Contents

List of Figures

List of Tables

Listings

Chapter 1
Introduction

This thesis deals with verification of message schedules for Profinet IRT networks. The schedule is crucial for real time applications and if it is miscalculated it might lead to loss of data in the network. The scheduling is a very complex task because a lot of constraints are given by physical limits of used devices and at the same time the schedule should often be optimized to minimize the time needed to propagate the information through the network.

Many approaches exist in the scheduling of the communication and each of them might provide different results. The results should be verified if they really meet the criteria given by the real environment. First idea could be to build a network from real hardware and to observe if it really behaves according to the schedule. Building a network can be a resource demanding task and the costs easily overcome the benefit which is only the verification of the schedule.

The goal of this thesis is to develop software methods to allow verification of message schedules. Two approaches of the verification are presented. The first one is a test tool which performs a static check of schedules. The requirements on the schedule are specified as the test rules and the tool evaluates if the schedule meets all the requirements by verification that no rules are violated.

The second approach is to create a model of the network which can be parametrized by a message schedule. A verification engine can be used to evaluate if the network is capable of following the message schedule.

The thesis is divided into five chapters. The second chapter (Profinet IRT) briefly introduces the reader into Profinet IRT networks and summarizes the knowledge necessary for understanding the following chapters. The third chapter (Test tool) presents how a test tool for the schedule verification was designed and implemented. The internal structure of the tool is described there as well as its user interfaces. Chapter Uppaal describes modeling of Profinet network in Uppaal software and how it was used to verify a message schedule. The last chapter summarizes the results of the thesis and suggests possible future extensions of implemented methods.

Chapter 2
Profinet IRT

Profinet IRT (Isochronous Real Time) is a communication concept based on Ethernet. It was designed for a cyclic data exchange in real time applications which demand fast response times. A typical application is a synchronization of drives in fast processes like milling or printing machines where the drives of individual axes must perform synchronized movements with high precision.

Due to the demands of applications on fast response times, very little data loss in the network should occur. However, Ethernet is not a deterministic environment and neither data delivery nor delivery time is guaranteed. Any message can be delayed at a switch or even dropped as a result of its overload. Therefore Profinet IRT defines extension of Ethernet for real time networks which enforces its deterministic behaviour.

All devices in Profinet networks contain the extended Ethernet switch to communicate with the rest of the network. The switch provides communication services for the transfer and reception of data. The services are used by an application running on the device, which can be a controller or an I/O device.

First of all Profinet IRT introduces mechanisms to synchronize the clocks of individual switches to the same rate. When the clocks are synchronized the communication cycle with a certain length is defined. This cycle is divided into two main phases, red and green. The red phase is reserved for the deterministic real time communication and the green phase is left for other traffic. From this division it can be seen that the real time application can co-exist with non-real time traffic on the same network.

To avoid random drops of frames in the red phase the communication is scheduled with high precision. The schedule ensures that no collisions of the frames occur and every frame is delivered to its receiver at deterministic time.

2.1 Scheduling of communication

The schedule of the communication is calculated in an engineering tool where the network is planned before the data exchange happens. The scheduling algorithm must be aware of the network topology and properties of switches. The properties include propagation delays caused by the finite speed of their operation and they must be taken into account during the planning phase so the schedule is created in a way that the switches are able to follow it. These properties are described in GSDML (General Station Device Markup Language) files [4]. Together with the topology, the algorithm must know what messages are to be sent through the network and their length. The messages are identified by a unique ID which is used by the switches during the forwarding of the frames. The format of the messages is described in section 2.2.

The output of the scheduler is a list of forwarding rules for individual switches in the network. Each rule contains ID of the message to which it relates and the ports where the message is received and transmitted. This combination creates a path of the message through the network. Together with the path, timing information is provided to tell the switch at which time from the beginning of the cycle the transmission should happen. This time is called Frame Send Offset (FSO). Frame Send Offset is defined as the time point when the transmission of the first byte of Ethernet header should start at underlying physical layer. Precise definition of this time point is given in IRT Engineering Guideline [5]. The behaviour of the switches with relation to the calculated schedule is described in section 2.3.

There might be lots of scheduling algorithms with various formats of their output, thus Profinet provides definitions of data records in document [1]. These data records are loaded into the switches during the start-up of the network and determine the behaviour of the switches. For the message schedule verification, data record PDIRData is very interesting because it contains the forwarding rules and the boundaries of the red phase calculated by the scheduler.

2.2 Message format

Messages are sent in standard Ethernet frames with the structure shown in figure 1.

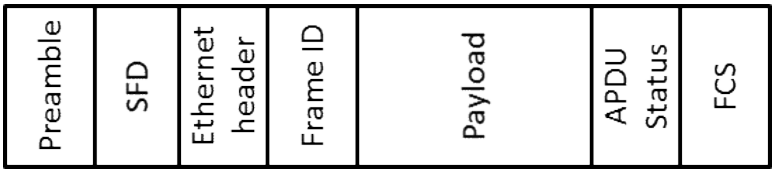


Figure 1: Profinet IRT message

Ethernet frame starts with Preamble, Start Frame Delimiter (SFD) and Ethernet header. It is followed by Frame ID, payload and status of the frame (APDU Status). At the end of the message there is Ethernet Frame Check Sequence (FCS). Frame ID is encoded in the first two octets after Ethernet header. To increase the throughput of the network switches use the cut through mechanism. This approach allows forwarding of a frame while it is still being received. In IRT switches the forwarding is based on Frame ID and once it is received the frame can be forwarded.

To further reduce the propagation delay caused by a switch, Frame ID can be encoded in the first two octets of the destination address in Ethernet header. This approach is called fast forwarding (FFW) and the format of the message is shown in figure 2. With this approach the frame can be forwarded after the first two octets of Ethernet header are received.

There is one more option to reduce the propagation delay. Instead of the long Ethernet preamble (7 octets) the short one can be used (1 octet).

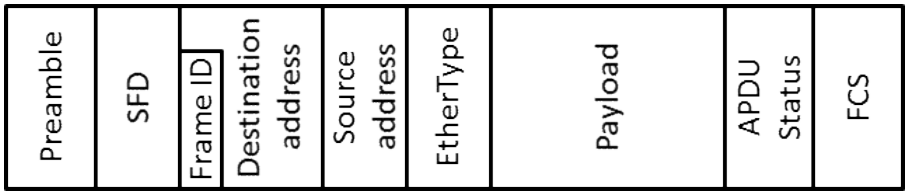


Figure 2: Profinet IRT message for fast forwarding

2.3 Message forwarding

From the switch point of view the messages can be divided into three categories:

- injected
- forwarded
- consumed

Injected messages are not received on any port, but they are injected to the network by the switch. Opposite of this case are received messages, they are not forwarded to any port, they are delivered to the application running on the switch. Forwarded messages are none of the previous, they are received on a port and forwarded to another one according to PDIRData.

IRT switches can be divided into the two types, absolute and relative. They are described below.

2.3.1 Absolute forwarder

Absolute forwarders have the forwarding information in PDIRData for all three categories. When a message is received its Frame ID is found in PDIRData and it is forwarded according to it. If the Frame ID is not found, the message is discarded. Sending of messages is synchronized with local the clock and they are sent according to their FSO.

Absolute forwarders contain PDIRData for the forwarded frames and thus they can have more than two ports and complex topologies can be built with them.

2.3.2 Relative forwarder

Relative forwarders do not contain forwarded frames in their PDIRData and they can have only two ports. Messages received on one port are forwarded to the other port. No timing information is available either and FSO is reconstructed from the time when they are received. Sending of the messages is not synchronized with the local clock and they are sent out at the time relative to their reception.

The relative forwarders can achieve better performance than absolute forwarders with respect to the time a message spends in the switch, but they are limited with two ports.

Chapter 3
Test tool

This chapter describes the design and the implementation of the test tool which verifies the message schedule. The message schedule is stored in PDIRData of individual switches and it is tested to verify that the schedule is valid. As it was described in section 2.1, PDIRData are created from the message schedule calculated by a scheduling algorithm. The tool tests if the transformation was done correctly by comparison of PDIRData with the input and output of the scheduler. The schedule itself is tested if it does not violate equations presented in IRT Engineering Guideline [5]. The equations were transformed into the test rules and the test tool evaluates them to find out if they are violated or not.

The tool provides a graphical interface where the results can be clearly displayed. They can also be saved to an XML file for further processing by another application or they can be printed into a PDF report.

To perform the tests it is necessary to specify the sources of data and additional information. The tool provides a form where the necessary inputs are configured and the configuration can be saved to an XML file and later loaded back. Formats of the input data can vary with their version or an application which generated them. Multiple formats of the input files are supported to enable usage of the tool to as many users as possible and the tool was designed to support possible future extensions of new file formats.

The tool was implemented in C++ programming language. Its implementation was successfully tested on Windows platforms. However, the implementation uses only libraries which are portable between most commonly used platforms and it can be easily ported to Linux or Mac platforms without large changes in the source code.

3.1 Design

The application was required to have both Graphical User Interface (GUI) and Command-line Interface (CLI). It leads to the design where the core functionality of testing was extracted into a testing library and two applications were implemented as an interface for it. The applications are described in sections 3.9 and 3.10 and they use public Application Programming Interface (API) to perform tests or utilize another functionality of the core. The public API, described in section 3.8, can be also used to integrate the library into another application.

The design of the library is shown in figure 3. Its API is called from applications and the requests are passed to the core. The core controls the flow of the program execution and uses individual modules to achieve the requested functionality.

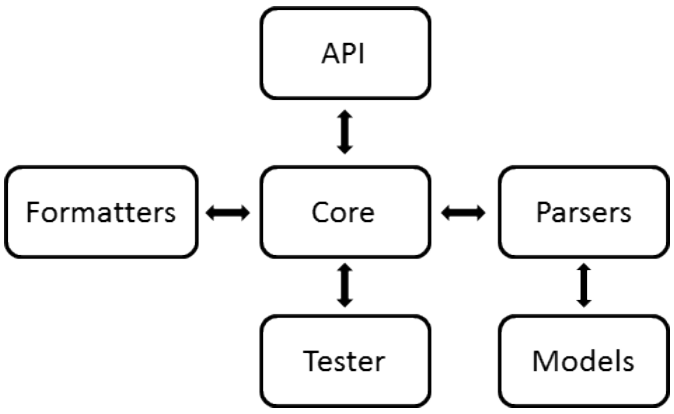


Figure 3: Design of the library

To perform the testing, several inputs must be provided for the library. The inputs are data records for individual switches, input and output of the scheduler and GSDML files describing the switches. Details about individual inputs are given in section 3.2. Every input might be stored in various file formats of different versions. For this reason it was decided to represent each input with a data model in module Models. Individual data models are described in section 3.3.

The data models are created from input files by parsers in module Parsers. They deal with the specifics of the format where the data are stored, and extract them to create the corresponding data models. With this approach the data models are independent of the file format. If a new format of the input data should be supported in the future, it will be sufficient to implement a new parser which will fill the data model and the rest of the library will remain unaffected. The parsers are described in section 3.4.

The implementation of the test rules and their organization was implemented in module Tester, described in section 3.5. The tests usually involve checks of values across the inputs, thus a new data model was designed to perform the testing on. This model is created from data models of individual inputs based on the configuration which is also one of the inputs of the application.

The outputs of the library are the results of the tests. They can be returned as a data structure for immediate processing or as an XML file. For the conversion of the data structures into XML files, module Formatters was designed. Inside this module the results can be formatted into the output format described in section 3.7. Detailed description of this module can be found in section 3.6.

3.2 Inputs

Description of individual inputs for the test tool is given in this section. There are several files needed for the verification of a message schedule, input and output of the scheduler, device descriptions in GSDML and a file containing the data records. For storage of the test configuration a configuration file was designed.

3.2.1 Scheduling Input and Output

These inputs are related to the scheduler of the IRT communication. They contain input and output data of the scheduler. The only supported scheduling library is Lauther Planning Library [6] and therefore only its file formats are supported.

3.2.2 GSDML

GSDML is an XML file where parameters of individual switches are described. Format and the structure are described in its specification [4].

3.2.3 PDIR

This source of data contains necessary data records which are sent by the controller to switches during the application relation (AR) establishment. They are the major point of the interest of this tool because they determine the real behaviour of switches. Necessary data records are following:

- PDIRData [802C]
- PDSyncData [802D]
- PDPortDataAdjust [802F]

The number in brackets is the number assigned to the data record in Profinet specification [2]. This number is used in write request during the AR establishment. PDIRData and PDSyncData are required by the protocol for each switch. PDPortDataAdjust are not mandatory and are used only for advanced features of individual ports.

Two file formats are supported, first one is a packet capture file (pcap) which contains capture of the AR establishment from the real network. The other one is an XML file containing these data.

Pcap

The packet capture can be obtained from already existing network. Libpcap was chosen as the supported file format, it is defined in [12]. This format is supported by WinPcap library, which is used later for parsing the file.

XML

The pcap file can be obtained from already existing networks, thus it is not very suitable for debugging or optimization of the scheduling algorithm. For this reason it was decided to support more flexible format which will bypass the need of a real network. An XML file format was introduced and an example is in the listing 1. Element *domain* contains attribute startup with allowed values legacy or advanced based on the startup mode used by all switches in the network. This element contains one or more elements *device*, one for each device in the network. Elements *record* contain data record mentioned above, formatted in hex strings as described in APDU (Application Protocol Data Unit) abstract syntax in Profinet specification [2].

```
<irtcheck>
  <pdir>
    <domain startup="legacy">
      <device name="device1">
        <record number="802C">0123456789ABCDEF</record>
        <record number="802D">0123456789ABCDEF</record>
        <record number="802F">0123456789ABCDEF</record>
        <record number="802F">0123456789ABCDEF</record>
        </device>
      </domain>
  </pdir>
</irtcheck>
```

Listing 1: PDIRdata XML format

3.2.4 Configuration

This file was designed to store the configuration of a single test case. An example of the file is shown in listing 2.

```
<irtcheck>
  <configuration>
    <sources>
      <file parser="lauther-in" path=".xml" type="schedule-in"/>
      <file path=".xml" type="schedule-out"/>
      <directory path="/" type="gsdml"/>
      <inline type="pdir"/>
    </sources>
    <domain controller="PN-IO" startup="legacy">
      <device gsdml=".xml" ip-address="192.168.0.1" name="device1.
          name" type="DEVICE_A"/>
      <device gsdml=".xml" ip-address="192.168.0.2" name="device2.
          name" type="DEVICE_A"/>
    </domain>
  </configuration>
</irtcheck>
```

Listing 2: Configuration XML format

Element *sources* contains information about the source files used for the testing. It can contain three types of child nodes. Node *file* holds the information about a single source file, node *directory* can be used for specification of a directory which contains several GSDML files. The last type which can be used is *inline* source for input of data records together with the configuration file. In this case the data records are specified in the configuration file and formatted as it is described in paragraph 3.2.3.

For every source it is necessary to specify path to the file, its type and parser to be used. Attribute *type* says what kind of input is specified, available options are schedule-in, schedule-out, gsdml, pdir. Allowed values for attribute parser are dependent on the type. Available combinations are in table 1. It is also possible to use option auto as the name of the parser, in that case the library will try to figure out the correct parser.

Input	Available parsers
Scheduling Input	lauther-in-v2, lauther-in-v4
Scheduling Output	lauther-out
PDIR	pcap, xml
GSDML	gsdml

Table 1: List of available parsers for individual input files.

Element *domain* holds the information about the global parameters and individual devices. The tested startup mode and the name of the controller are specified there. For every device in the network is created a node device, in its attributes there are configured device name, IP address and assignment to the Device Access Point (DAP) in GSDML.

3.3 Models

The data models are representations of the inputs inside the application. For every input a corresponding data model was created. One additional data model, Domain model, was designed to perform the tests on.

3.3.1 Configuration

This model is a representation of the configuration file described in 3.2.4. Its class diagram is shown in figure 4.

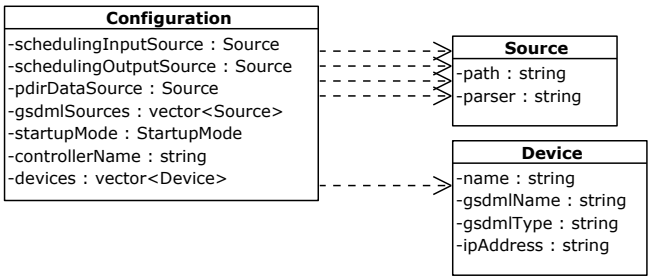


Figure 4: Model of the configuration file

The configuration file is represented by class Configuration. It contains single sources of scheduling input and output and the source of data records. There might be multiple GSDML files used and therefore they are stored in a vector. Attribute *startupMode* holds the tested startup mode and it is represented by enumeration StartupMode which can have values Advanced or Legacy. Name of the controller is stored in attribute *controllerName* as a string. Attribute *devices* contains objects of class Device to represent devices in the network.

Class Device contains name of the switch from scheduling input and output in attribute *name*. This name is mapped to Device Access Point (DAP) inside GSDML file by attributes *gsdmlName* and *gsdmlType*, where *gsdmlName* is file name of GSDML and *gsdmlType* is DAP identifier.

Class Source represents a single source file. It holds path to the file in attribute *path* and name of the parser in attribute *parser*.

3.3.2 Scheduling Input

This model represents the input for the IRT scheduler. The overall structure is shown in figure 5. In total it holds topology of the network with parameters of the switches and a list of the messages which are sent through the network together with their sender and receiver.

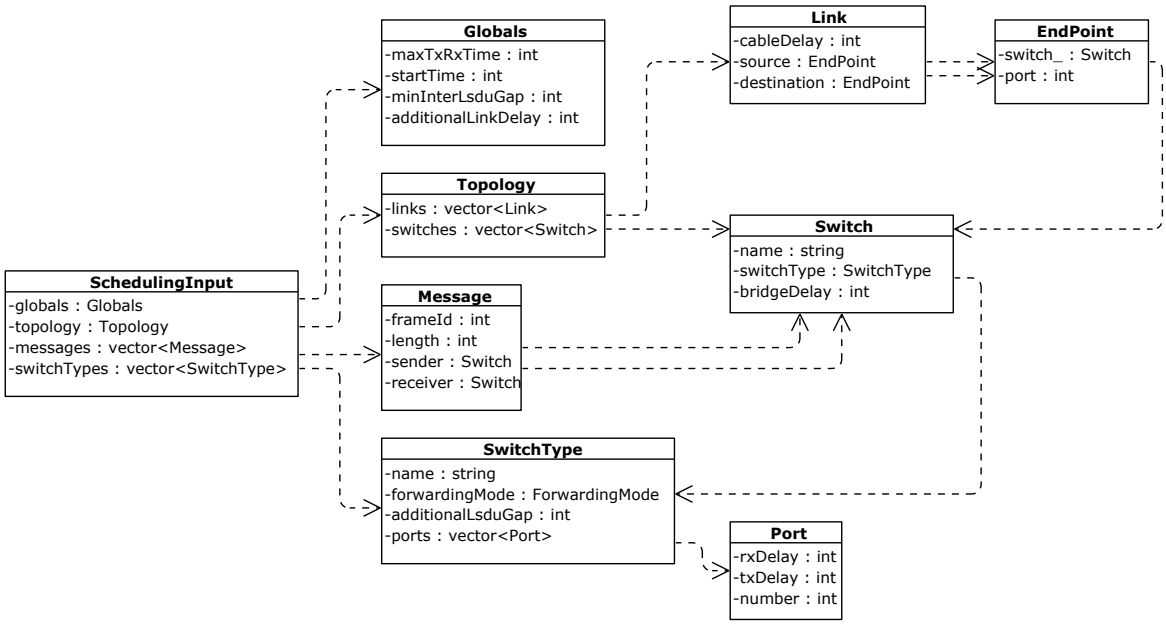


Figure 5: Model of the input for the scheduler

The scheduling input is represented by class SchedulingInput. It holds global parameters for the scheduling algorithm, topology of the network, list of messages to be sent and list of switch types.

Global parameters are stored in class Globals. Attribute *maxTxRxTime* is the time when all the messages must be completely received. Time when the first message can be sent is in attribute *startTime*. Minimum gap between the frames on a single port is in attribute *minInterLsduGap*. The gap is measured from the end of the frame to the start of the preamble of the consecutive frame. Attribute *additionalLinkDelay* represents a time reserve which has to be added to the planned FSO (Frame Send Offset) to cover the possible inaccuracy of clocks of two neighbouring switches.

The topology is represented by class Topology and contains list of switches and links between them. A link is characterized by two endpoints and the delay which it introduces to the communication. An endpoint is a combination of switch and port number where the link is connected.

Individual switches are represented by class Switch which holds a unique name of the switch, switch type and its bridge delay. Bridge delay characterizes a delay which is necessary for the switch to create a forwarding decision for a single frame. The real Profinet networks are often built from many devices of the same kind. The common properties of the switches were extracted to a separate class describing the switch type.

The switch types are characterized by class SwitchType. It stores forwarding mode represented by an enumeration which can have two values Absolute and Relative. Attribute *additionalLsduGap* characterizes an additional gap which has to be added to *minInterLsduGap* from the global parameters to obtain the true gap required by the switch.

The ports are represented by class Port. They contain delays introduced by receiving or transmitting parts. The ports are identified by ascending numbers starting from number one.

Messages, which have to be scheduled, are represented by class Message. It stores length of the data which will be carried by the message, its sender and receiver. Messages are identified by their frame ID which is unique and is generated by engineering tool in ranges specified by the Profinet specification.

3.3.3 Scheduling Output

This model represents the output of the IRT scheduler. Structure of the model is shown in figure 6.

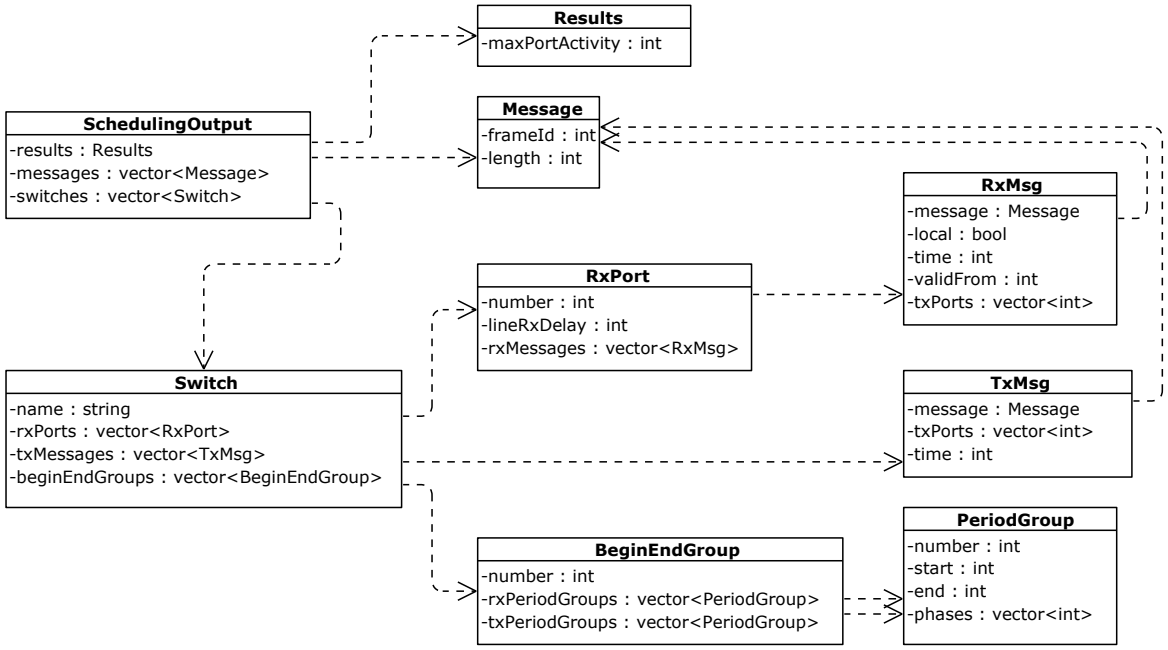


Figure 6: Model of the output of the scheduler

The scheduling output is represented by class SchedulingOutput. It stores calculated time schedule of the messages and boundaries of the red period for every port.

The schedule for one switch is represented by class Switch. It contains the name of the switch which is the same as the name of the corresponding switch in the scheduling input. The forwarding rules are stored separately for each port where the frame should be received in attribute *rxPorts*. Locally generated frames are stored in attribute *txMessages*. Switch also contains boundaries of the red period in attribute *beginEndGroups*.

The boundaries of the red period are represented by class BeginEndGroup. It is present for every port of the switch and therefore it is identified by the port number. The red period may differ for the receiving and transmitting part of the port, but for both cases it is represented by class PeriodGroup. It stores the beginning and end of the period and a list of phases for which it is valid.

Each port, where a frame is received, is represented by class RxPort which contains a list of received frames represented by class RxMsg. Class RxMsg contains the forwarding information for the switch. It contains a list of ports, where the message should be sent, in attribute *txPorts*. This list may be empty if the message is consumed, in this case attribute *local* should be set to true. Attribute *time* represents the time when the message should be transmitted on the port.

Locally generated frames are represented by class TxMsg. The meaning of the attributes is the same as in the case of class RxMsg.

3.3.4 Profinet Data Units

This model contains representation of selected data records required for the testing. The data units are modeled according to the specification [1], the structure and naming of the classes and attributes follows the specification as well. Totally four data units were modeled, ArblockReq, PDIRData, PDSyncData and PDPortDataAdjust.

Record PDIRData was fully implemented in the extent that is defined in the Profinet specification and its structure is show in figure 7. Names of the attributes correspond to the names in the specification. Remaining data records were implemented only partially, because some of their attributes were not necessary for the application.

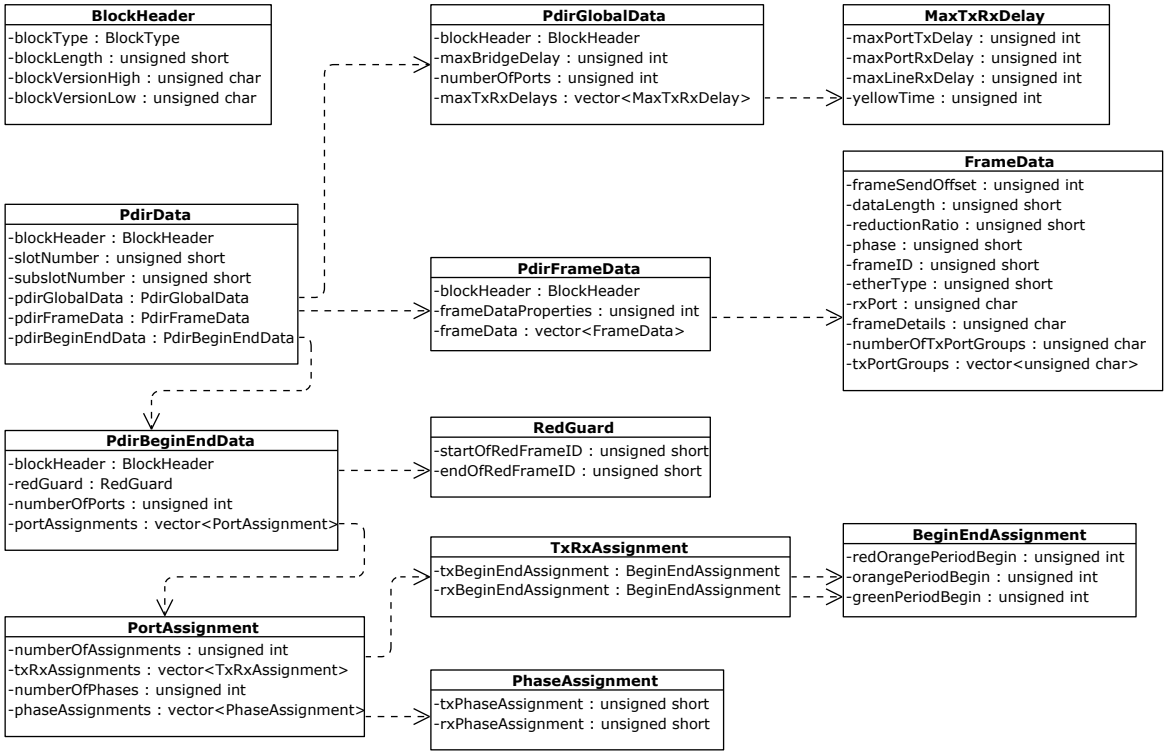


Figure 7: Model of data record PDIRData

3.3.5 PDIR Data Library

This model is a collection of Profinet Data Units for every switch in the network as described above. The structure of the model is shown in figure 8.

It contains a list of switches, where each switch holds a complete set of data records. A switch is identified by attribute *id*. It can contain device name or IP address, the way of identification depends on the source of the data. In case of XML file it is the device name, in case of pcap file it is the IP address.

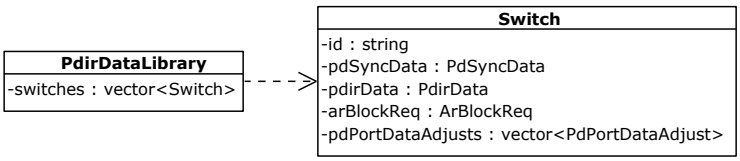


Figure 8: Model of the data records library

3.3.6 GSDML Library

Collection of the GSDML files specified in the configuration is represented by model GSDML Library. Its structure is in figure 9.

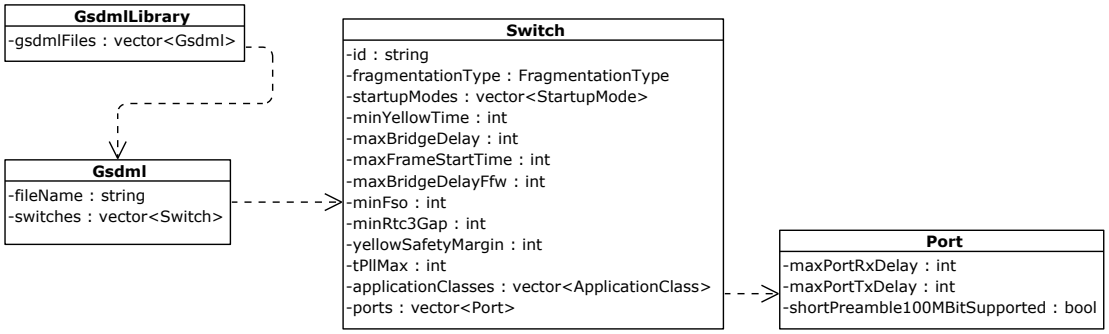


Figure 9: Model of the GSDML files

A single file is represented by class Gsdml and it is identified by its file name. The file can contain one or more Device Access Points (DAP) which are represented by class Switch. Port specific attributes are stored in class Port.

Names of the attributes and their meaning are the same as in their definition in GSDML specification [4].

3.3.7 Domain Model

This model brings together previously mentioned data models of input files. It creates a suitable data structure for the testing. Class diagram of this model is shown in figure 10.

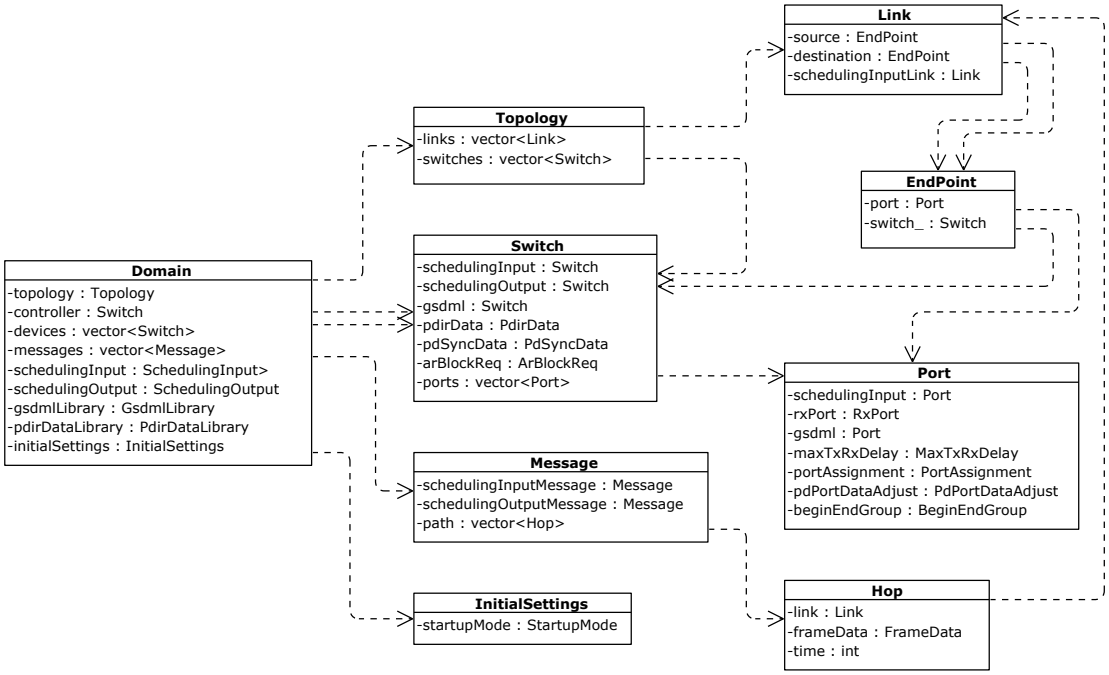


Figure 10: Model of the whole network

The network is represented by class Domain. It contains topology of the network, controller and devices. It also contains original data models of input files.

The topology is represented in the same way as it is represented in the data model of the scheduling input.

Class Switch represents a controller or a device. The class contains corresponding switch objects from other data models as well as their data records. The difference between a controller and a device is that for the controller the data records and GSDML file are not present in the inputs.

For some test cases it was necessary to know the path of the message. The path is reconstructed from the scheduling output and it is represented by a list of hops for each message. A hop is combination of the link where message travels, and the time when the transmission starts.

3.4 Parsers

Module Parsers presents the interconnection between the input files and data models. Every instance of data model described in section 3.3 is created by a parser from the input file. One exception is the Domain model which is not created from a file but from other models.

For access to the data in XML or pcap files, third party libraries were used. These libraries parse the file and provide its content in data structures. Library libxml2 [7] was used to work with XML files and library WinPcap [8] was used with pcap files. The libraries have a C API, which is not very convenient to use in C++, therefore it was decided to implement wrappers with minimal API needed for the tool. This approach is illustrated in figure 11. When a parser of data model needs to access a file, the wrappers XmlParser or PcapParser are used to do so.

The design where the libraries are directly used only by the wrappers, gives an advantage in case that the library has to be changed to another one for any reason. In the code of the tool it will result only in reimplementation of the wrapper. Other classes will remain intact.

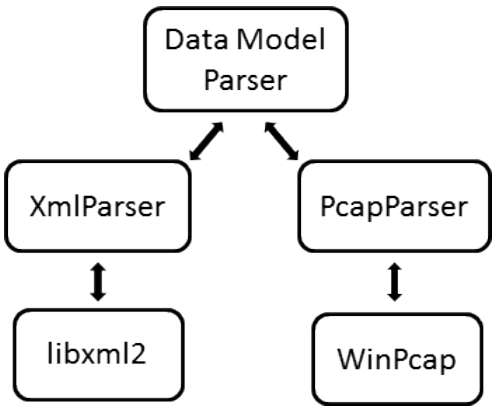


Figure 11: Wrappers for third party libraries

3.4.1 PcapParser

PcapParser provides access to data in a pcap file. It parses the file and calls a user specified callback function for every frame in the file. The content of the frame is not interpreted and it is passed to the callback function in terms of raw byte sequence. Interpretation of the frame is the responsibility of the user of the parser.

3.4.2 XmlParser

XmlParser allows access to the contents of an XML file and its validation against XML schema [14]. It parses the input file and creates an internal tree structure of XML elements. The root element of the tree is returned to the user as an object of class Node.

Class Node provides access to its data and children data via evaluation of XPath expressions [13]. Result of XPath evaluation can be returned as another object Node or an array of them.

3.4.3 Parsing of data models from files

For every data model an abstract class was defined. This class defines the only one method **Parse()** which returns the desired model. The implementation of this method is left for derived classes which in this method implement conversion from a specific file format to the data model.

This approach is useful in case that there exists more than one file format to describe the data to fill the data model with. For every format a parser is created but further in the code the data can be handled in the same way thanks to the interface class.

3.4.4 Integration of parsers into the library

Individual parsers of data models are not accessed directly from the library. For every data model there exists a proxy class which takes path to the file and name of the parser from the configuration

file. Based on the name it creates an instance of the correct parser and uses it to create the model. If the name of the parser is specified as "auto", it tries to estimate the correct parser to be used. This estimate is based on the extension of the source file and validation by xsd, it can be extended by any other suitable approach. In figure 12 an example of integration into the library is given. When PDIR data model is needed the PdirParser is called with the file name and the name of the parser. It decides whether to use parser from pcap or XML file based on the file extension. This parser is then used to obtain the data model.

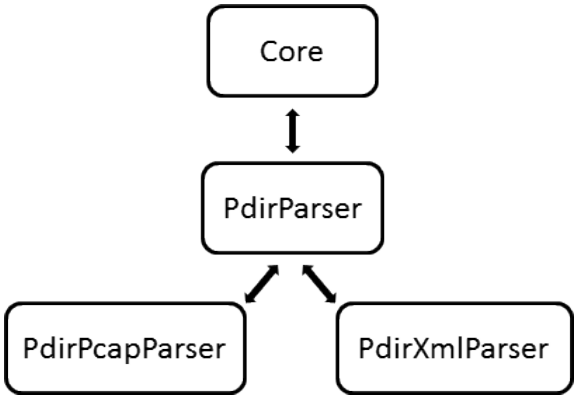


Figure 12: Integration of parsers into the library

3.4.5 Parsing of the domain model

Parsing of the domain model was implemented in class DomainParser. The input can be path to the configuration file or its data model. When the processing is complete, an instance of the domain model is returned.

In case that a path to the file was provided it is parsed to obtain the configuration model. Based on the information inside the other input files are parsed and their data models are obtained.

From the data models of input files the domain model is created. The topology is based on the scheduling input and paths of the messages are created from the scheduling output. During the parsing it is necessary to find corresponding switches in different models. It is done using the information from the class Device of the configuration model.

Class Device of the configuration model has attributes *name*, *gsdml*, *type* and *ipAddress*. Mapping of these attributes to corresponding attributes in other models is in table 2.

Configuration	Corresponding
name	SchedulingInput::Switch::name
	SchedulingOutput::Switch::name
	(PdirDataLibrary::Switch::id)
gsdml	GsdmlLibrary::Gsdml::fileName
type	GsdmlLibrary::Switch::id
ipAddress	PdirDataLibrary::Switch::id

Table 2: Mapping of attributes from device configuration to other models

3.5 Tester

This section describes Tester module which is the most important part of the tool. Individual tests, which are organized in test suites, are implemented in this module. The tests were implemented as independent classes which know nothing about the data models. They are parametrized by the test suite which extracts the necessary values from the domain model.

The design of this module allows adding of new tests by implementing a new class and integrating it into the existing suite. It is also possible to add a new suite and use subset of all tests to achieve testing of specific parts of the IRT schedule.

3.5.1 Class Test

For the implementation of individual test rules abstract class Test was designed. It extracts common behaviour of all tests and provides a standard environment to implement individual tests.

Constructor

The constructor has two arguments

- startupMode
- warning

Argument *startupMode* is the tested startup mode from the configuration file. It is optional with default value Advanced.

A test can generate three types of results, passed, error and warning. If the test rule is not fulfilled, an error or warning is generated. If the argument *warning* is set to true, the test generates warning. It is also optional to specify this argument, default behaviour is to generate error when the rule is violated.

Warnings were implemented because a violation of some test rules is not critical for the operation of the network.

Attributes

- startupMode
- warning
- args

All the attributes are protected and accessible only to derived classes. Attributes *startupMode* and *warning* are stored arguments from the constructor. Attribute *args* contains textual representation of test arguments. It is a map with string keys and values. Every key-value pair represents one test argument. Name of the argument is in the key and value contains textual representation of the argument value. This attribute should be filled in the constructor of the derived class.

Abstract functions

- ID()
- Check()
- PassMessage()
- FailMessage()
- Explanation()

These functions must be implemented by derived classes and their implementation defines the test. This approach allows a more declarative way of the test implementation.

Function **Id()** returns a unique name of the test. For more readable code organization it was decided that classes implementing the test will have the same name as is the ID of the test. Therefore this function actually returns the name of the derived class.

The test rule is implemented by function **Check()**. It returns true if the rule was fulfilled or false if it was violated.

Functions **PassMessage()** and **FailMessage()** return textual representation of the test result. If function **Check()** returns true, **PassMessage()** is used. In opposite case the text returned by function **FailMessage()** is used.

Explanation of the test rule is returned by function **Explanation()**. It is independent of the test result.

Functions

The only public function of this class is function **Run()**. It uses abstract functions above and attributes to evaluate the test rule and construct data structure TestResult that is described in section 3.5.4.

3.5.2 Example test implementation

Here an example of implementation of a test is given. The test checks if a port uses a short preamble in advanced startup mode and it generates warning.

The class is named ShortPreamble and its constructor is in listing 3.

```
ShortPreamble::ShortPreamble(bool shortPreambleUsed, StartupMode
    startupMode)
  :Test(startupMode, true), shortPreambleUsed(shortPreambleUsed)
{
  args["[PDIRData].ShortPreambleUsed"] = shortPreambleUsed ? "True" :
      "False";
}
```

Listing 3: Constructor

It has two arguments *startupMode* and *shortPreambleUsed*. At the first place the base constructor is called with given startup mode and true for the warning argument.

Passed argument *shortPreambleUsed* is stored in the attribute of the same name. The body of the constructor converts it to the textual representation and saves it into the attribute *args* of the base class.

The test rule is implemented in function **Check()** in listing 4.

```
bool ShortPreamble::Check() {
  if (startupMode == Advanced) {
    return shortPreambleUsed;
  }
  return true;
}
```

Listing 4: Test rule implementation

In listing 5 the implementation of the remaining abstract functions is shown. Function **Id()** returns the name of the class in compliance with the convention introduced in section 3.5.1. Other functions return messages characterizing the test.

```
string ShortPreamble::Id() {
  return "ShortPreamble";
}

string ShortPreamble::FailMessage() {
  return "Short preamble is not used.";
}

string ShortPreamble::PassMessage() {
  return "Short preamble is used in advanced mode. In legacy mode
     this test has no meaning.";
}

string ShortPreamble::Explanation() {
  return "Every port of the device should use short preamble in
     advanced startup mode. Short preamble is not supported in legacy
      startup mode.";
}
```

Listing 5: An example implementation of the functions in a test

3.5.3 Test suite

The test suite is the entry point to the module of Tester. Schema of the suite is in figure 13. It works with the domain model and extracts attributes from it to run the tests. Results of individual tests are grouped together using GroupResults, described in section 3.5.4, to increase their readability. For example tests performed on a switch are grouped together. Inside such a group another group for each port may exist and so on. As soon as all tests are done the Suite returns SuiteResult.

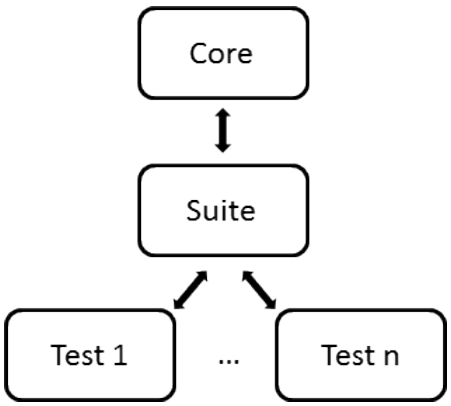


Figure 13: Structure of the suite

Every suite is derived from base class Suite which defines function **Run()** which has to be implemented. This function returns SuiteResult and implements organization and evaluation of the tests. List of all implemented tests and their organization is in appendix A.

3.5.4 Results

Test results are represented by a tree structure shown in figure 14. The root of the tree is class SuiteResult. Its branches are formed by GroupResults and TestResults are leafs.

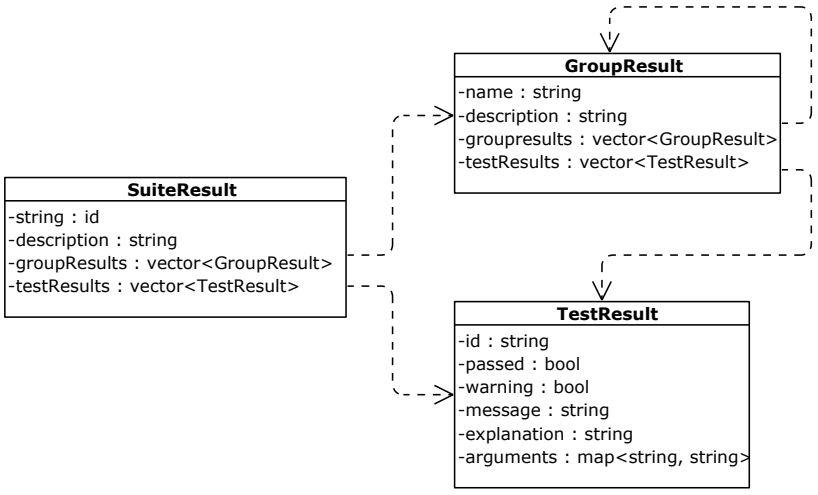


Figure 14: Structure of test results

SuiteResult contains ID of the suite which generated the result. Inside the suite result there can be multiple group results and test results.

Class GroupResult is very similar to SuiteResult but it does not have a counterpart in control structures as tests and suites. Therefore it does not contain any ID, but only a name which does not have to be unique.

TestResult is a representation of the result of a test. It contains ID of the test and messages generated by the test. Attributes *passed* and *warning* indicate whether the result is passed, error or warning. Table 3 shows interpretation of all possible combinations of attributes *passed* and *warning*.

passed	warning	result
true	any	passed
false	false	error
false	true	warning

Table 3: Interpretation of TestResult attributes

3.6 Formatters

In this section the formatters from the data structures to XML strings are described. Totally two formatters were implemented. One for the configuration model and one for the data structure of the results. For the generation of an XML string, boost library [9] was used. It has a module property tree which can be used for building of trees and it also supports its serialization into XML format.

3.6.1 Formatting of the results

Formatting of the results was implemented in class ResultsXmlFormatter. The input for this class is an instance of SuiteResult described in section 3.5.4 and the output is an XML string with format described in section 3.7.

Structure of the formatting is shown in figure 15. For each class of the results a separate function was implemented. The suite is formatted in function **FormatSuite()**. Every group inside the suite is formatted using function **FormatGroup()**. This function can recursively call itself for formatting the subgroups, or it calls function **FormatTest()**.

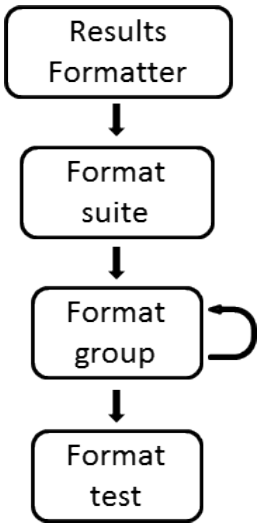


Figure 15: Results formatter

3.6.2 Formatting of the configuration

Configuration is formatted in class ConfigurationXmlFormatter. It takes the configuration model described in section 3.3.1 and formats it into XML string described in section 3.2.4.

For the formatting of every class of the configuration model a separate function was implemented. Figure 16 shows the principle of the formatting. The configuration is processed by function **FormatConfiguration()** which uses functions **FormatSource()** and **FormatDevice()** to format the contents of the class.

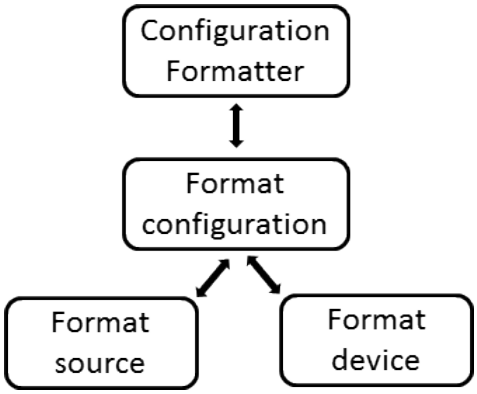


Figure 16: Configuration formatter

3.7 Output

To be able to work with the results in another application, an XML file was designed to store the results. Format of the file follows the data structure from section 3.5.4. Example of the file is in listing 6.

```
<irtcheck>
  <results>
    <suite description="..." id="IrtCheckSuite">
      <tests errors="1" passed="0" warnings="0">
        <group errors="1" name="Globals" passed="0" warnings="0">
          <test id="SendClock" message="Sendclock value is invalid."
             result="error">
            <explanation>Startup mode must be Advanced if SendClock
               is less than 250000.</explanation>
            <arguments>
              <argument id="[Config].Startup" value="Legacy"/>
              <argument id="[SchedulingInput].SendClock (=2*[
                 SchedulingInput].MaxTxRxTime)" value="10000"/>
            </arguments>
          </test>
          <test>
           <!-- another test -->
          </test>
          <group>
            <!-- nested group inside Globals-->
          </group>
        </group>
      </tests>
    </suite>
  </results>
</irtcheck>
```

Listing 6: Output XML with results

The suite result in element *suite* contains its ID and description. The test results are under element *tests* which contains overall number of passed tests, warning and errors.

The *group* element has its name, description and number of tests inside the group according to their results. Inside the group, there can be another subgroup or individual test results.

Each test result contains its ID, result and a short message. There is also an explanation how the test was evaluated, and a list of arguments with which it was run.

3.8 API

This section describes the public interface which can be used to integrate the library into other applications. The library was designed to be used in an as simple as possible way. Therefore no initialization and clean-up are needed after the work with the library is done.

```
static string IrtCheck(const string& configurationFile, const string&
    outputFile);
```

Listing 7: Main method of the public API

The listing 7 shows the main method of the API. As inputs it takes path to the configuration file and path where the results should be saved. The method returns a short string message summarizing the results.

The method implementation first parses the configuration file and creates a domain model based on the information provided inside. The model is then used to perform the tests by the Tester module and the results are formatted in Formatter. The formatted results are then saved to a file. From the data structure returned by the Tester a short summary is created and returned to the caller of the API function.

```
static SuiteResult Check(const string& configurationFile);
static SuiteResult Check(const Configuration& configuration);
```

Listing 8: Methods to run tests and to return the resulting data structure

Methods in listing 8 take the path to the configuration file or the data model and perform the tests. The results are returned back in the data structure.

The program flow is the same as in the previous case but the results are returned in a raw-data structure.

```
static string FormatToXml(const SuiteResult& suiteResult);
static string FormatToXml(const Configuration& configuration);
```

Listing 9: Methods format data structures into XML strings

To save the data structures returned by other API methods into XML, methods in listing 9 can be used. They use module Formatters to serialize the input data structures into the XML string.

```
static Configuration GetConfiguration(const string& path);
```

Listing 10: Method to get data model of the configuration file

To obtain a data model of the configuration a method in listing 10 can be used. This data model can be modified in the application and saved or used to run tests by other API methods.

3.9 Command Line Interface

The command line interface can be used for performing tests in a very fast way. The interface was designed to pass data through XML files, which also allows to integrate the library into another application through this interface.

Usage

irtcheck -c CONFIG_FILE -o OUTPUT_FILE -r REPORT_FILE

Options

- -c, [–config]. Path to the configuration file.
- -o, [–output]. Path to the file where the results are saved.
- -r, [–report]. Path to the file where the PDF report is generated.

Options -o and -r can be omitted, in that case the tool will show a short summary of test results in the command line.

The PDF report is generated from data structures returned by module Tester using library wxPdfDocument [11]. An example of a report is given in appendix B.

3.10 Graphical User Interface

The graphical user interface (GUI) was designed to enable easy usage of the testing library. It provides an interface for work with configuration files in Configuration dialog and to view the test results in form Results view. The user interface was implemented using wxWidgets library [10].

3.10.1 Configuration dialog

The configuration dialog provides an interface to work with configuration files described in section 3.2.4. Window of the dialog is shown in figure 17.

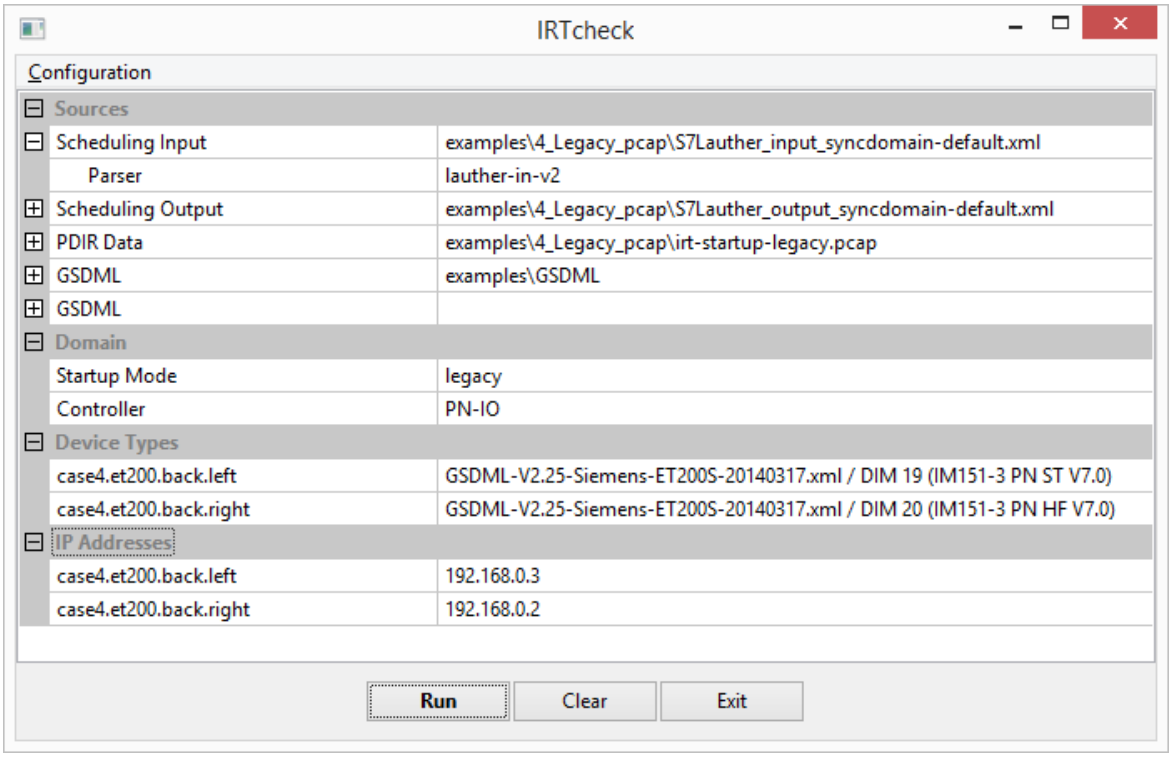


Figure 17: Configuration dialog

New configuration files can be created here and saved, later they can be loaded and modified. The dialog is separated into four categories. In category *Sources* the user can specify the input files and their parsers.

In category *Domain* it is possible to select tested startup mode from options legacy and advanced. Name of the controller is selected by the user from a dropdown menu which is filled by the available switches utilized in the scheduling input file.

Category *Device Types* creates mapping from device names to their Device Access Points in GSDML. The user is again given a set of possible choices in a dropdown menu which is filled from the specified GSDML files in category *Sources*.

IP addresses of the devices are assigned in category *IP addresses*. This category is visible only in case that the data records are parsed from a pcap file. This file is parsed and possible IP addresses are available to the user in a dropdown menu. Otherwise the knowledge of the IP addresses is not necessary.

The configuration dialog uses model-view-controller architecture to implement the necessary functionality. The visible part view catches events from the user and notifies the controller about it. The controller makes the required changes in the underlying model and if necessary, it notifies the view back to update its information from the model.

3.10.2 Results view

The form in figure 18 shows results of the performed tests. The look of the window is data driven by a tree data structure returned from the library, described in section 3.5.4.

The results view is separated into tabs whereas each of them shows a different part of the tree. The failed tests and warnings are the point of the most interest, thus they are shown in the first two tabs. They are followed by the results of the individual groups under the suite result. In the figure those groups are in tabs named Globals, Controller, Devices and Links. After the results the configuration, used to perform the tests, can be view. This view is the same as in the case of the configuration dialog but it cannot be modified.

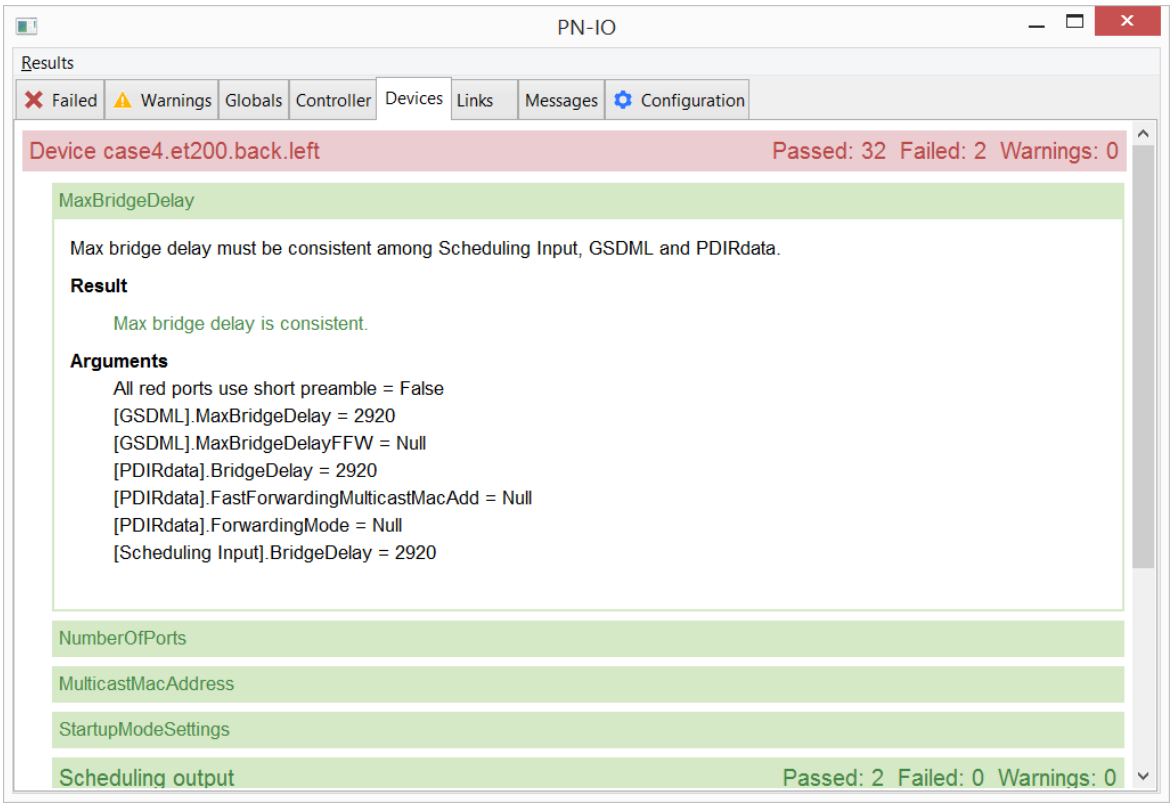


Figure 18: Results view

Group and test results are show in frames which can be expanded to take a look on result details. The groups contain the number of passed tests, errors and warnings in the header. Different types of results are distinguished by their colour, red is for errors, yellow for warnings and green for passed tests. Underlying results are shown after the group is expanded by clicking on the header. These results are indented to visualize the tree structure, every level in the tree has its own indentation level.

Test results have the name of the test in their header which can be also expanded to view the detailed description of the test. In the section with details arguments of the test, explanation and a short message about the result are given. The tests are coloured in the same way as in the case of groups.

The results can be saved to an XML file described in listing 6 from menu Results. A PDF report with the results can be generated also from the menu. An example of the PDF is given in appendix B.

3.11 Verification of message schedules

The implemented tool was used to verify message schedules of several Profinet IRT networks. The network topologies were chosen in a way that they cover the most frequent configurations used in industrial applications. The message schedule was created by the TIA Portal engineering tool and the necessary data were exported to XML files. One real network was verified to show that the tool

is capable of getting tested data records from pcap file with capture of the communication between controller and devices.

The tested network configurations were following:

- Line topology of three absolute forwarders. Data records of this configuration were captured from a real network in the laboratory.
- Tree topology with mixture of absolute and relative forwarders.
- Tree topology with fast forwarding and short preambles.
- Tree topology with fast forwarding, short preambles and dynamic frame packing.

The schedules were verified without an error. That was the expected result because the engineering tool used for the configuration is widely used in industrial applications. Message schedules generated by it could be considered proven by numerous successful network installations. However, as indicated by other manufacturers, having the possibility to check a message schedule together with the parameters that are sent to the individual devices in the topology may help reduce the development and bug fixing time significantly.

Chapter 4
Uppaal

Verification of message schedules does not have to be approached by a static check of parameters as it was described in the previous chapter. A network can be modeled and parametrized by data records for devices and its behaviour can be observed in simulation.

This chapter describes how an IRT network was modeled and verified in tool Uppaal. For understanding how the network looks a brief introduction to Uppaal is given in section 4.1. In the following sections modeling of a single switch and network is described. Requirements on network behaviour were specified and the results of the verification are presented in section 4.4. On top of the network an isochronous application was modeled and verified in section 4.6.

4.1 Introduction to Uppaal

Uppaal is software for modeling, validation and verification of real time systems. They are modeled as timed state machines which can be interconnected into a network with synchronization channels. Formal definitions and deeper knowledge how Uppaal works is given in its tutorial [3] or in help pages of the tool.

A system in Uppaal is modeled as network of timed state machines. Communication between them can be done through synchronization channels or global variables. Once the system is modeled, requirements on its behaviour can be specified in requirements specification language defined by Uppaal, which is a subset of timed computation tree logic (TCTL). Verification engine takes system and its requirement specification and automatically evaluates if the requirements are met.

4.1.1 State machines

Figure 19 shows an example state machine which was implemented for the demonstration in this section. It is an example of a Timer on the right side and it's Tester on the left. Timer waits for a specified time given by Tester and then notifies Tester back when it expires.

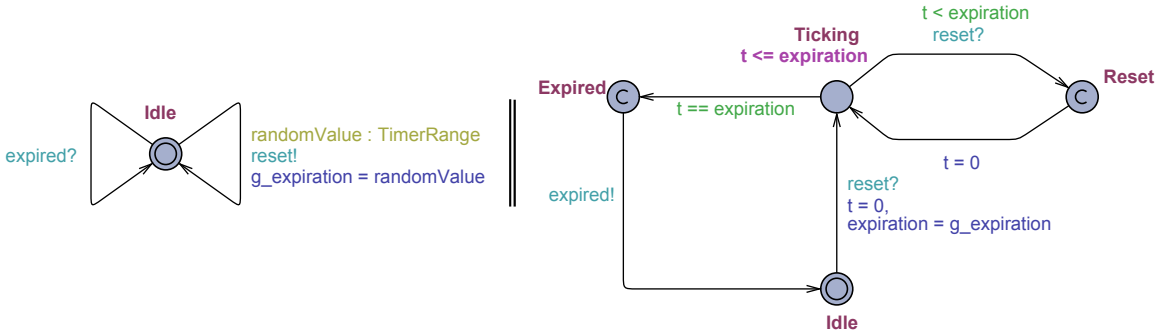


Figure 19: Example of Uppaal state machines

Tester state machine can reset the Timer state machine by triggering **synchronization** signal *reset*. State machine triggering the synchronization uses notation with exclamation mark after the name of the synchronization channel. Synchronized state machine uses notation with question mark to wait for the synchronization.

When Timer is reset a random value, for which the timer will wait, is chosen by **select statement** *randomValue:TimerRange*. Range of the timer is declared in global parameters of the system. The value is saved to the global variable g_expiration by **update statement** *g_expiration = randomValue* and read by Timer when it receives the synchronization signal. The synchronization is an atomic operation which cannot be interrupted and both state machines make the transition synchronously. This implies that the writing and reading of the global variable is also atomic operation and no data races can occur even though the variable could be used by other synchronization channels. This principle with writing and reading to global variable during synchronization can be seen as passing of parameter with the synchronization.

When Timer receives synchronization *reset* it goes to state Ticking, saves value of the timer from parameter to internal variable *expiration* and resets internal clock t to zero. State Ticking has a defined **invariant** $t <= expiration$. This condition must be invariantly fulfilled whenever the state machine is in this state. If the condition is violated and the state machine did not leave the state, an error is reported during the simulation. In state Ticking, Timer can receive another reset synchronization if the clock value is less than the expiration time. This is expressed as a **guard** $t < expiration$. The guard is a condition under which the transition can be done.

When the internal clock reaches the time of the expiration a transition to state Expired can be made. The guard $t == expiration$ forces Timer to make a transition from state Ticking to Expired because this time is the last moment when Timer can stay in state Ticking because of invariant $t <= expiration$. If a transition was not done after this time, it would lead to the violation of the invariant.

States Expired and Reset are called "committed" and marked with C. In the committed states the time is not allowed to pass and they must be left immediately.

4.1.2 Verification

Uppaal provides Requirement Specification Language for the verification of systems. In this language certain criteria can be written by using expressions in table 4. Using these expressions it can be specified how a modeled system should behave during its operation.

The verification engine explores the complete state space of the system and evaluates if the requirements are met or not. For the example from the previous section there is not specified when Tester state machine triggers synchronization *reset*. During the verification, Uppaal explores all possible times when such an event can happen and verifies specified rules for all possibilities.

The verification engine is also capable of generation of diagnostic traces proving or disproving certain verification rule. The diagnostic trace represents a sequence of timed transitions of the system during the verification. This trace can be used for diagnostic purposes or as a proof that a requirement is met or not.

Expression	Interpretation	Description
E<> p	Possibly	There exists a path where p eventually holds
A[] p	Invariantly	For all paths p always holds
E[] p	Potentially always	There exists a path where p always holds
A<> p	Eventually	For all paths p will eventually hold
p –> q	Leads To	Whenever p holds q will eventually hold

Table 4: Uppaal requirement specification language

4.2 IRT Switch

This section describes how an IRT switch was modeled. As it was described in chapter 2, an IRT switch provides services, for the communication with other devices, to the application running on a device. It sends or receives data and delivers them to the application.

The goal of the thesis is a verification of messages schedules in red period of the communication cycle, thus only the behaviour of a switch in this period was implemented. The behaviour in the green period was neglected and not implemented.

It was decided to implement relative and absolute forwarder. The implementation does not take into account fast forwarding and dynamic frame packing, but the state machines can be extended with this functionality if it is necessary in the future.

The state machines simulating behaviour of one switch are the same for every switch in the network, the difference is in their parameters. The parametrization of state machines was done by defining several structures and their instantiation in global declarations of Uppaal. These structures are described in section 4.2.1.

After the description of the data structures the state machines implementing the switch are described in details. They are based on the protocol state machines described in the Profinet standard [2]. Only such protocol machines necessary for the verification of the message schedules were implemented and their state tables were reduced to eliminate the interactions with not implemented state machines. Such an approach is suitable for the purpose of this thesis. However, if a complete

verification of the network behaviour was to be done it would be necessary to implement complete state machines with all the interactions as well.

4.2.1 Switch parameters

Switch parameters are defined by structure Switch in table 5. This structure contains attribute *bridgeDelay* which represents the maximum time needed by the switch to forward a frame. Flag *relative* indicates whether the switch is a relative or absolute forwarder.

Attribute name	Description
bridgeDelay	Bridge delay of the switch
relative	Flag if the switch is relative

Table 5: Structure Switch

Ports are parametrized by data structure Port in table 6. It contains parameters for transmitting and receiving parts. The delays introduced to the communication by port operation are described by attributes *rxDelay* and *txDelay*. Boundaries of the red period are described by attributes *redOrangePeriodBegin* and *greenPeriodBegin*, where *redOrangePeriodBegin* is the beginning of the red period and *greenPeriodBegin* is its end.

Part	Attribute name	Description
Receiving part	rxDelay	Physical delay
	rxRedOrangePeriodBegin	Beginning of red period
	rxGreenPeriodBegin	End of red period
Transmitting part	txDelay	Physical delay
	rxRedOrangePeriodBegin	Beginning of red period
	rxGreenPeriodBegin	End of red period

Table 6: Structure Port

Forwarding rules for switches are defined by structure FrameData in table 7. For every forwarded frame it contains a port where the frame is received and a port where it is transmitted. If attribute *rxPort* is zero it means that the switch is the sender of the message. If the attribute *txPort* is zero, the switch is the recipient of the message.

The data structure also contains the time when the transmission should happen. In a relative forwarder this data structure exists only for injected and received frames. It does not exist for forwarded frames.

Attribute name	Description
id	Frame ID
fso	Time when the frame is sent at txPort
rxPort	Port where the frame is received
txPort	Port where the frame is forwarded

Table 7: Structure FrameData

Two global constants are defined for the whole network. They are listed in table 8. *JITTER* is the maximal allowed difference between clocks between two neighbouring switches. This parameter is used to verify that the network behaves as it is expected under the worst possible cases of clock desynchronization. Constant *SEND_CLOCK* is the length of the simulated communication cycle.

Name	Description
JITTER	Maximal allowed difference between clocks of neighbouring switches.
SEND_CLOCK	Length of communication cycle

Table 8: Global constants of the network

4.2.2 Frame definition

Frames sent through the network are defined by the data structure shown in table 9. It exists for every message and contains its frame ID and data length. Attributes *sender* and *receiver* are not used in control of the switch state machines. They are used during the verification of the message schedule to compare the configured sender/receiver from this structure with the true ones from the simulations.

Attribute name	Description
id	Frame ID
dataLength	Length of frame payload in bytes
sender	Sender of the frame
receiver	Receiver of the frame

Table 9: Structure Frame

4.2.3 Overview of state machines

A switch consists of several state machines which control the frame flow through the switch. The structure of the state machines forming the switch is in figure 20.

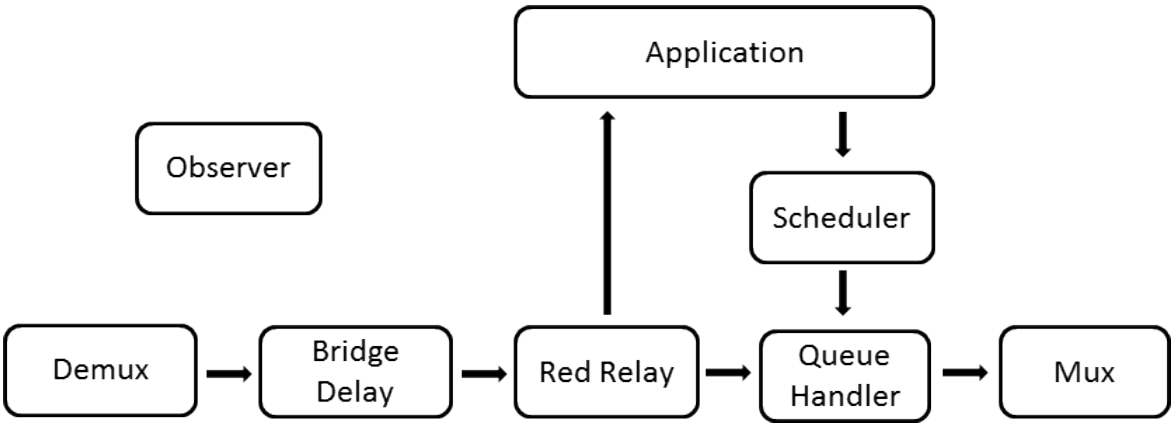


Figure 20: Interconnection of state machines

The receiving part of every port is modeled by state machines Demux and Bridge Delay. Demux guards reception in the red period and Bridge Delay simulates the time necessary to make a forwarding decision.

The transmitting part of the port is controlled by state machines Queue Handler and Mux. Mux has the similar functionality as Demux, it ensures that frames are transmitted only in the red period. It also prevents the switch from attempts to send more than one frame at a time. Queue Handler controls timely correct sending of the frames according to their frame data.

State machine Red Relay provides the forwarding between individual ports. It can also notify application of the switch that a frame was received if the switch is the recipient of the message.

Frames, which have to be sent by the switch, are scheduled for communication from the application. State machine Scheduler accepts the data and puts the frame to the queue of a port where it has to be sent.

For the verification purposes an observer state machine was designed to monitor the state of the communication.

4.2.4 Demux

Demux controls reception of frames from a link. It uses three channels to synchronize with other state machines. *M_UNITDATA_ind* synchronizes Demux with Link, described later in section 4.3.1. The synchronization is triggered when the first byte of MAC address reaches Demux. By channel *DMUX_RED_RELAY_ind* Demux indicated to Bridge Delay that a frame was received. It is

triggered when frame ID is received and a forwarding decision can be done based on it. The frame can be discarded by *discard* event which is used to notify observer Message, described in section 4.3.2.

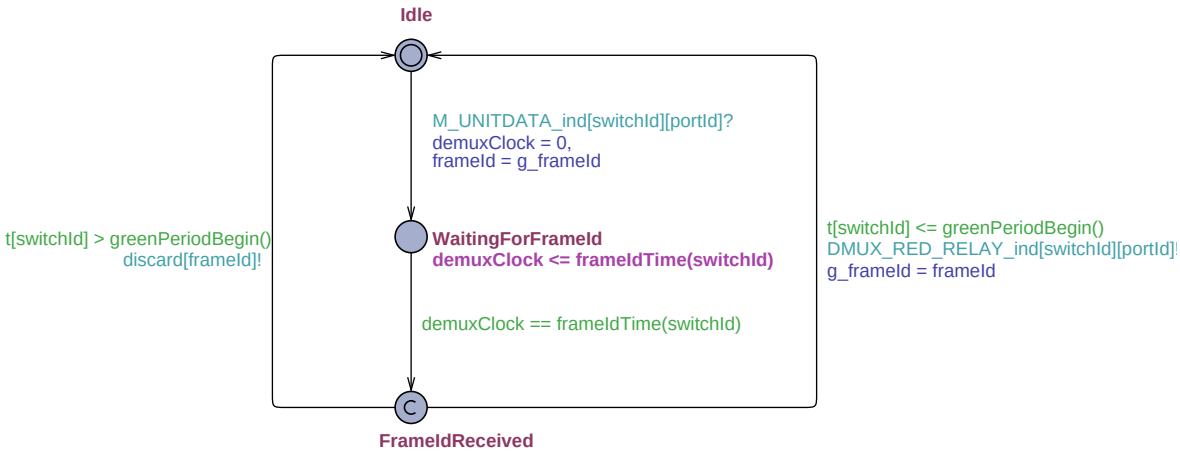


Figure 21: Demux state machine

Crucial for the acceptance of a frame is the reception time of frame ID. In state WaitingForFrameId, Demux waits for time when it is received.

The frame ID reception time is calculated by function **frameIdTime()**. In case of normal forwarding, this time can be calculated by equation 1.

$$T_{IDreception} = (L_{Ethernet} + L_{frameID}) * 80 \quad [\text{ns}] \tag{1}$$

$T_{IDreception}$ is the time when frame ID is received from the moment when first byte of destination address of Ethernet header reached Demux. $L_{Ethernet}$ is the length of Ethernet header without 802.1Q tag which is 14 bytes. $L_{frameID}$ is length of frame ID field which is 2 bytes.

After this time the decision is made. If the frame ID was received inside the red period the frame is passed to Bridge Delay for further processing. End of the red period is equal to the beginning of green period and is returned by function **greenPeriodBegin().**

4.2.5 Bridge Delay

The Bridge Delay state machine simulates the time which is needed by a switch to make the forwarding decision. The state machine is shown in figure 22. The state machine is notified by channel *DMUX_RED_RELAY_ind* from Demux. When the decision time passes it notifies Red Relay by event *bridgeDelay_ind*.

The decision time is calculated by function **decisionTime().** For absolute forwarder it is calculated by equation 2.

$$T_{decision} = T_{maxBridgeDelay} - T_{IDreception} \quad [\text{ns}] \tag{2}$$

$T_{maxBridgeDelay}$ describes the maximum time needed by the switch to forward a frame. This time is measured from the beginning of reception by Demux to the beginning of transmission by Mux. Parameter $T_{IDreception}$ was defined in equation 1.

Relative forwarders do not know the FSO of forwarded frames and it must be calculated from the known parameters. This calculation is described in IRT Engineering Guideline [5] and in the simulation it is implemented in this state machine by function **decisionTime()**. After the decision time passes the frame is forwarded by other state machines to the other port than it was received without any additional delay.

The calculation is done by the function in equation 3.

$$T_{decision} = T_{maxBridgeDelay} - T_{IDreception} + T_{lineRxDelay} - T_{lineDelay} \quad [\text{ns}] \tag{3}$$

In this equation $T_{maxBridgeDelay}$ and $T_{IDreception}$ were defined above. $T_{lineRxDelay}$ is the configured delay of the link where the frame was received, and $T_{lineDelay}$ is its real value used in the simulation.

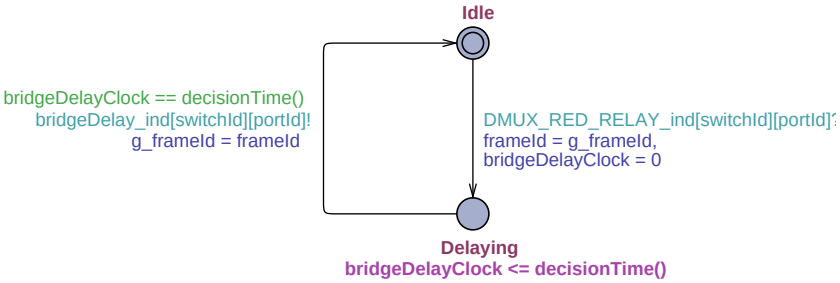


Figure 22: Bridge Delay state machine

4.2.6 Red Relay

Red Relay controls bridging of frames between ports of the switch. The forwarding rules are stored in data structures FrameData, described in table 7. It is notified from Bridge Delay by channel *bridgeDelay_ind* when the time necessary for forwarding decision passes. Based on the forwarding decision the frame is discarded, put to the queue of a port by channel *QueueHandler_req* or it can be delivered to the application by channel *RED_RELAY_Data_ind*. In case that the frame is delivered to the application, the observer is notified by channel *delivered*. The state machine of Red Relay is shown in figure 23.

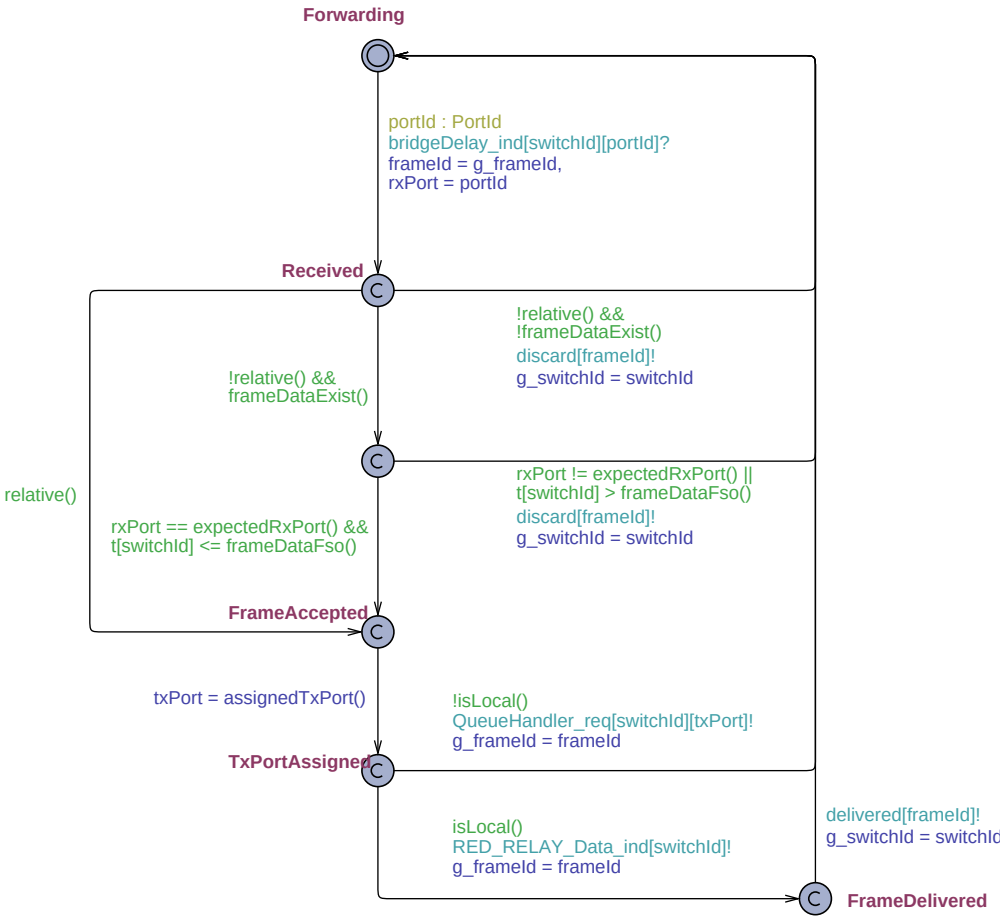


Figure 23: Red Relay state machine

For every frame received by an absolute forwarder, corresponding frame data must exist. The existence of the frame data is evaluated by function **frameDataExist()**. Moreover reception port and time is checked against the frame data. The expected reception port from the frame data is obtained by function **expectedRxPort()** and compared with the real reception port. If the frame

was received on a different port than it is specified in the frame data, it is discarded. If the frame was received too late to be forwarded without violation of its FSO it is also discarded.

For a relative forwarder only the frame data of the consumed frames are available. Other frames are automatically forwarded to the other port than they were received at.

4.2.7 Queue Handler

Queue Handler controls correct transmission of forwarded frames on a port. The implementation of the queue is different for the absolute and the relative forwarder. In the absolute forwarder the queue is ascendingly ordered according to the frame send offsets (FSO) of the frames in the queue. For the relative forwarder the queue is implemented as a pipe and frames are sent to Mux whenever the Mux is able to send a frame.

Three channels are used for synchronization with other machines. *QueueHandler_req* puts a frame into the queue. *QueueNotEmpty* is triggered at the time when the frame has to be sent. Mux informs Queue Handler by channel *MuxFree_ind* when transmission of the next frame can be done. In figure 24 implementation of the queue for an absolute forwarder is shown.

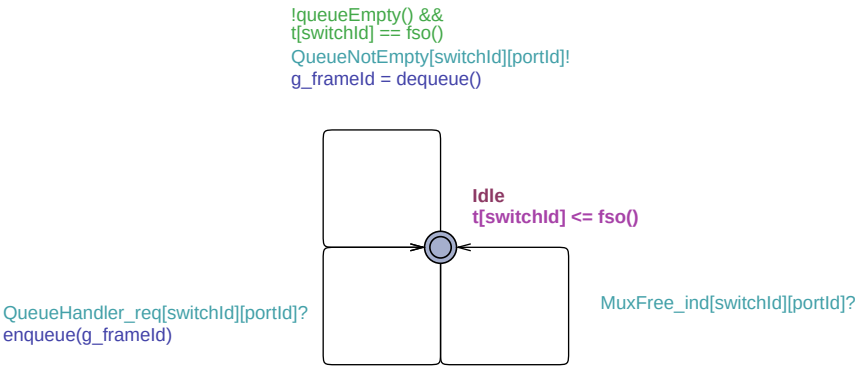


Figure 24: Queue Handler state machine for absolute forwarder

The frames are put into the internal queue representation by function **enqueue()**. The queue is ascendingly ordered according to the frame send offsets (FSO) of the frames. FSO of the first frame is obtained by function **fso()**. Function **dequeue()** removes the first frame from the queue.

Implementation of the queue for a relative forwarder is depicted in figure 25. The frames are sent immediately when they are put to the queue or right after the Mux state machine of the port finished transmission of the previous frame.

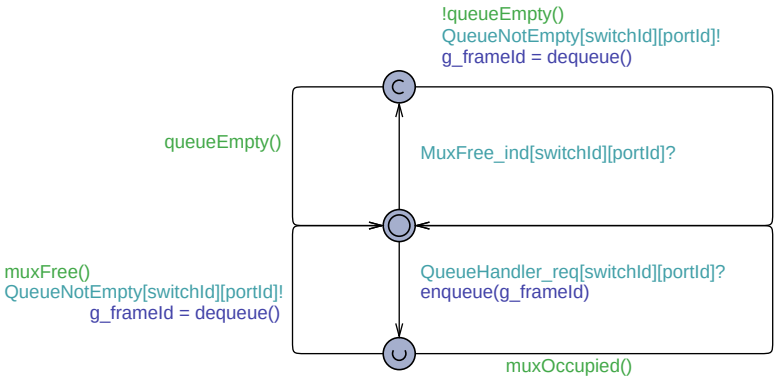


Figure 25: Queue handler state machine for relative forwarder

4.2.8 Mux

State machine Mux controls the transmission of the frames on a port. It is notified from Queue Handler by channel *QueueNotEmpty* when the frame transmission should start. The beginning of

the transmission is signaled via channel *M_UNITDATA_req* to the Link state machine. After the transmission is finished a corresponding Queue Handler state machine of the port is notified by channel *MuxFree_ind*. This signal is relevant only for relative forwarders, absolute forwarders ignore this signal and send the frames at their frame send offset. The Mux state machine is in figure 26.

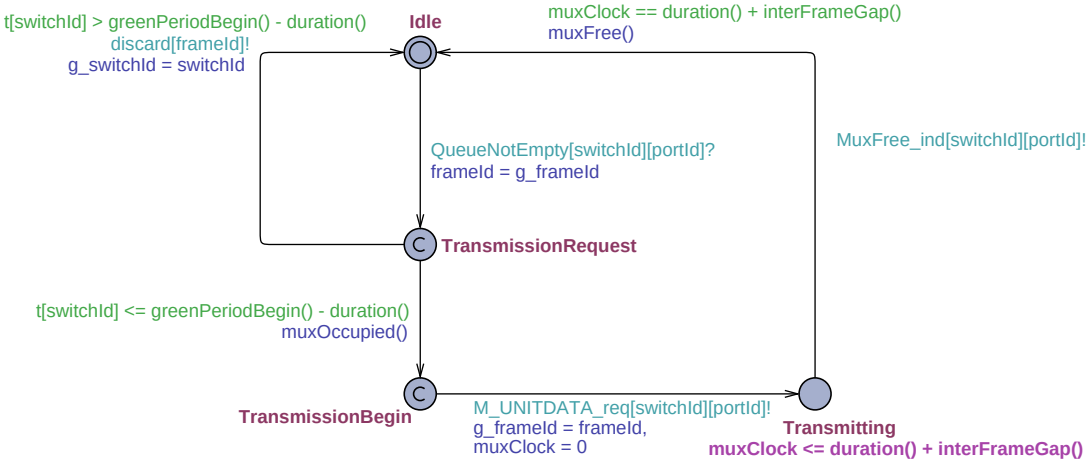


Figure 26: Mux state machine

When the transmission of a frame is requested by Queue Handler it is decided whether the frame can fit into the red period. If not, the frame is discarded. For the time when the frame is being transmitted the state machine is blocked in state Transmitting. In this state Mux cannot receive requests from Queue Handler and blocks transmission of other frames. The time, when the port is blocked for another transmission, is calculated by equation 4.

$$T_{blocked} = T_{duration} + T_{gap} \quad [\text{ns}] \tag{4}$$

In this equation $T_{duration}$ is the duration of the frame transmission and T_{gap} is the minimal gap between two consecutive frames. Parameter $T_{duration}$ is calculated by equation 5.

$$T_{duration} = (L_{ethernet} + L_{frameID} + L_{data} + L_{APDUstatus} + L_{trailer}) * 80 \quad [\text{ns}] \tag{5}$$

Parameters $L_{ethernet}$ and $L_{frameID}$ were defined in equation 1. L_{data} is the length of the frame payload defined in the data structure of the frame. $L_{APDUstatus}$ is the status field of the payload and it is 4 bytes long. $L_{trailer}$ is the length of Ethernet trailer, which is 4 bytes long.

Parameter T_{gap} from equation 4 is calculated by equation 6.

$$T_{gap} = T_{minGap} + (L_{premable} + L_{SFD}) * 80 \quad [\text{ns}] \tag{6}$$

In this equation T_{minGap} is the minimal gap between IRT frames and value 1120 ns is used. Parameters $L_{preamble}$ and L_{SFD} are the length of Ethernet preamble and start frame delimiter which in case of a long preamble give 8 bytes in total.

4.2.9 Scheduler

Scheduler controls timely correct transmission of locally generated frames. The message is scheduled for the transmission when its data are ready. It is signaled by channel *PPM_Set_Data_req* from application and the observer is notified by channel *scheduled*. When the frame should be sent, it is put into Queue Handler by channel *QueueHandler_req*. Diagram of the state machine is shown in figure 27.

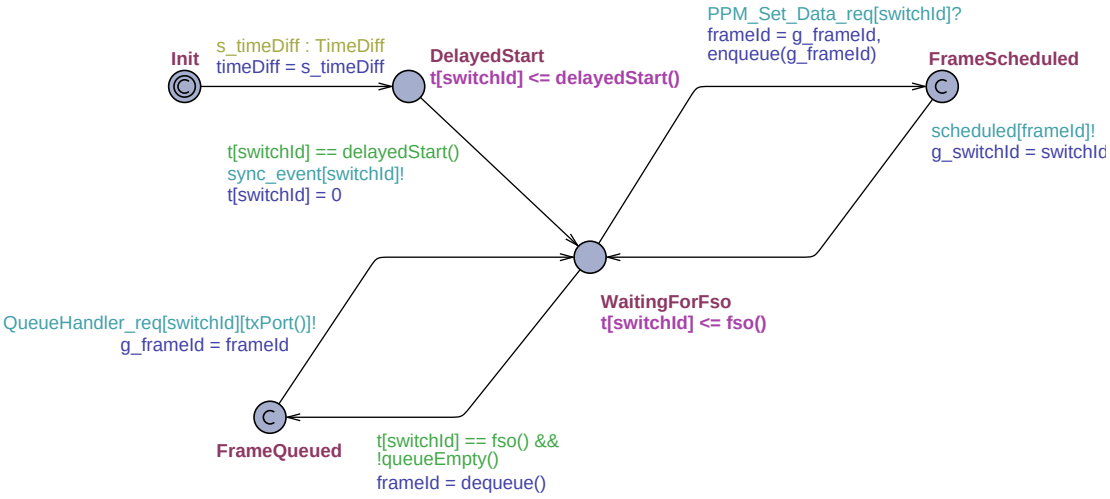


Figure 27: Scheduler state machine

The clocks of switches are not synchronized absolutely. A small difference is allowed to occur between two neighbouring switches. This difference is randomly chosen at the beginning of the simulation by Scheduler state machine and the clocks of individual switches are desynchronized by this value. Verification engine takes all possible combinations of clock difference between switches and verifies that the requirements hold under all conditions.

When a new communication cycle begins the application is notified by synchronization channel *sync_event*. This synchronization is later used in an isochronous application which uses it to control the operation with inputs and outputs.

4.3 Network

The network was modeled using the switches described in the previous section. The switches are interconnected by full duplex cables which are modeled as pairs of links, one for each direction. The links introduce a delay to the communication caused by the finite speed of signal transmission.

The links are defined by structures with attributes in table 10.

Attribute name	Description
sourceSwitch	Source endpoint
sourcePort	
destinationSwitch	Destination endpoint
destinationPort	
lineRxDelay	Configured line delay
linkDelay	Real line delay

Table 10: Structure Link

4.3.1 Link

This state machine models a link between ports of switches. When the frame transmission begins Link is notified by channel *M_UNITDATA_req* from Mux. The cable introduces a delay to the communication which is calculated by equation 7.

$$T_{link} = T_{tx} + T_{cable} + T_{rx} \quad [\text{ns}] \tag{7}$$

T_{link} is the total delay introduced by the link. T_{tx} is the delay of the transmitting port, T_{cable} is the delay of the cable and T_{rx} is the delay of the receiving port. After the time T_{link} passes, Demux of the receiving switch is notified by channel *M_UNITDATA_ind*. Diagram of this state machine is in figure 28.

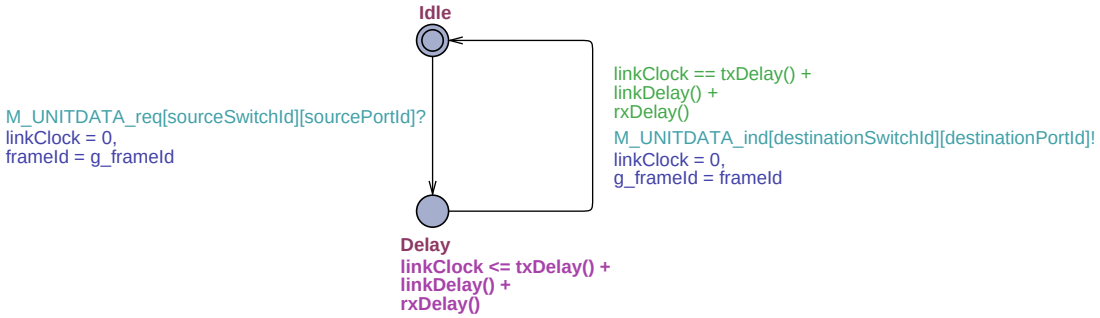


Figure 28: Link state machine

4.3.2 Verification rules for message schedule

The verification of the message schedule was done for one communication cycle. An observer state machine Message, which is shown in figure 29, was designed to monitor the state of the communication. It represents a single message sent through the network. The Scheduler notifies the observer by channel *scheduled* at the time when the message is scheduled for the communication. The message can be delivered to its receiver which is indicated by Red Relay via channel *delivered* or it can be discarded which is signaled through channel *discard*. When a transition happens in the state machine it internally stores the parameter of the synchronization event for the verification purposes. The parameter of all the events is identification of the switch which triggered the synchronization.

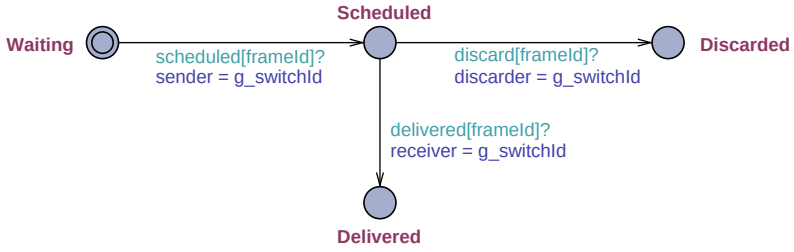


Figure 29: Observer of the network behaviour

To make the network working it was necessary to simulate the application on all switches. It was done by implementing a state machine which consumes the data from Red Relay and provides data for Scheduler. Its diagram is shown in figure 30.

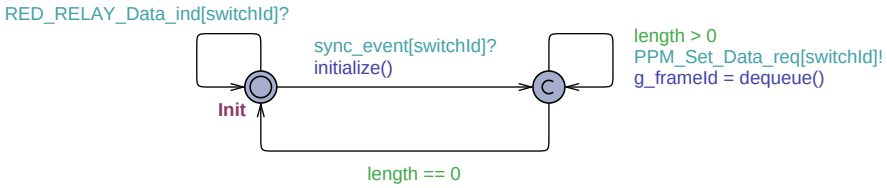


Figure 30: Application state machine

At the beginning of the cycle it puts all generated frames to Scheduler by channel *PPM_Set_Data_req*. The frames are generated from the forwarding rules of the switch.

After all the frames are put to the Scheduler the application waits for the messages to be received. It is notified about this event from Red Relay by channel *RED_RELAY_Data_ind*.

To verify that the network behaves according to the message schedule, several rules were defined and they are described below.

Delivery

This rule was designed to verify that all messages are delivered under all possible conditions. The Uppaal representation is in listing 11. It checks that all observing state machines Message get into state Delivered and therefore all the messages are delivered.

```
A<> forall(frameId : FrameId)
  Message(frameId).Delivered
```

Listing 11: Delivery of message

Sender

This rule verifies that messages are sent by correct switches. When a message is scheduled for the communication its state machine gets into state Scheduled and ID of the sending switch is stored in variable sender. This ID is checked against the configured ID stored in data structure Frame in table table 9. Uppaal representation of this rule is in listing 12.

```
A<> forall(frameId : FrameId)
  Message(frameId).sender == getSender(frameId)
```

Listing 12: Sender of the message

Receiver

Verification that a message was received by a correct switch is done in similar way as in the case of verification of message sender. The ID of the receiving switch is stored in variable receiver of the state machine Message. It is checked against the configured receiver in Frame structure. The implementation of this rule is in listing 13.

```
A<> forall(frameId : FrameId)
  Message(frameId).receiver == getReceiver(frameId)
```

Listing 13: Receiver of the message

Timing

This rule checks if messages are sent according to their calculated frame send offset. When a frame is started being transmitted state machine Mux is in state TransmissionBegin. The time when Mux is in this state is compared with the time in the frame data. In listing 14 there is the implementation of this rule for an absolute forwarder.

```
A[] forall(switchId : AbsoluteForwarderId) forall(portId : PortId)
  MUX(switchId, portId).TransmissionBegin
  imply
  t[switchId]==getFrameData(switchId,MUX(switchId,portId).frameId).fso
```

Listing 14: Timing of messages for absolute forwarder

The implementation of this rule is different for a relative forwarder, because the forwarding rules exist only for the locally generated and received frames. For a relative forwarder only the timing of the frames, where the forwarding rule exists, is verified. The implementation is in listing 15.

```
A[] forall(switchId : RelativeForwarderId) forall(portId : PortId)
  MUX(switchId, portId).TransmissionBegin
  and
  getFrameDataExist(switchId, MUX(switchId, portId).frameId)
  imply
  t[switchId]==getFrameData(switchId,MUX(switchId,portId).frameId).fso
```

Listing 15: Timing of messages for relative forwarder

Path

Besides the timing of the messages it is also necessary to verify that the frames are forwarded through the configured path. The implementation is in listing 16. It is done in a similar way as the timing verification is done. Port ID of Mux is checked against the configured forwarding rules.

```
A[] forall(switchId : AbsoluteForwarderId) forall(portId : PortId)
  MUX(switchId, portId).TransmissionBegin
  imply
  portId==getFrameData(switchId, MUX(switchId, portId).frameId).txPort
```

Listing 16: Path of messages for absolute forwarder

In case of a relative forwarder the verification is done only when the forwarding rule exists for the frame. Implementation for relative forwarder is in listing 17.

```
A[] forall(switchId : RelativeForwarderId) forall(portId : PortId)
  MUX(switchId, portId).TransmissionBegin
  and
  getFrameDataExist(switchId, MUX(switchId, portId).frameId)
  imply
  portId==getFrameData(switchId,  MUX(switchId, portId).frameId).txPort
```

Listing 17: Path of messages for relative forwarder

4.4 Results of network verification

Implemented state machines were used to simulate an IRT network of 5 switches with line topology. Topology of the network is shown in figure 31.

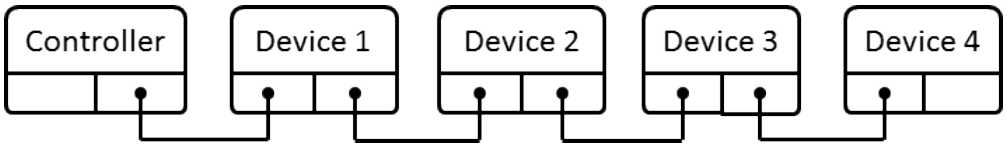


Figure 31: Topology of the network

The network consists of one controller and four devices. Controller and devices 3 and 4 are absolute forwarders, whereas devices 2 and 3 are relative forwarders.

The message schedule was created in a way that controller sends four messages, one for each device. Every device sends one message back to the controller. This can be understood as writing of outputs to devices and reading of inputs from devices. Detailed description of the schedule is given in appendix C.

By evaluating of the requirements described in section 4.3.2 it was found that the network behaves accordingly to the calculated message schedule. When the requirements are met, it can be concluded that the message schedule is feasible for the network. The messages are forwarded along the planned paths and without violating the planned timing. No messages are dropped either and all are delivered to their receivers.

If there was a mistake in the configuration the model would behave differently from the schedule and some messages could be dropped. In that case the schedule would not be feasible and a mistake in the scheduling algorithm would be discovered.

During the verification a disadvantage of this approach occurred. Even for small networks with ten devices verification takes a lot of time. A more important problem is the memory consumption which exceeded the capacity of the computer used, and the verification could not be done. This disadvantage could be overcome by optimizing of the state machines and reducing their possible states.

4.5 Isochronous application

In section 4.3 a dummy application was used to verify the message schedule. A natural next step is to model a more complicated application on top of the network which ensures delivery of the messages between switches.

It was decided to model the isochronous application described in Profinet specification [1]. It is divided into controller and device applications. The controller sends outputs to devices and devices send their inputs to the controller. Exact time when the inputs are scanned and written to the process is defined. This time is the same for all devices in the network and is related to the communication cycle.

Time constraints of the application are shown in figure 32 as taken from standard [1]. Time when the inputs are scanned is described by parameter *T_IO_Input*. Time when the data are available for communication is described by *T_IO_InputValid*. These parameters describe the time before the end of the communication cycle, the data are sent in the next cycle.

Outputs are described by parameter *T_IO_OutputValid* and *T_IO_Output*. The first one is the time when the data are available from the communication. Time *T_IO_Output* is the time when the outputs are written to the controlled process.

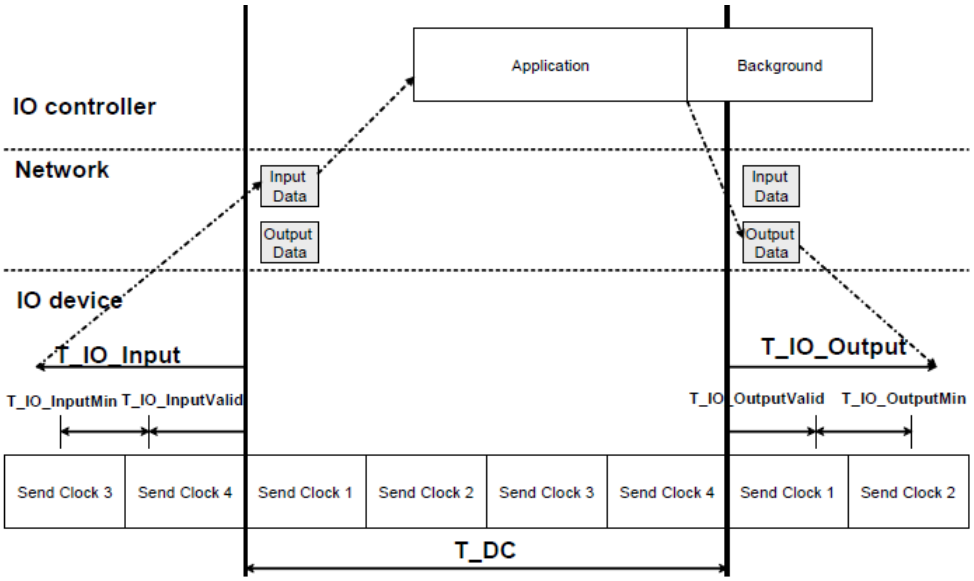


Figure 32: Isochronous application

As it was written the isochronous application is divided into controller and device application. Interconnection of individual state machines is shown in figure 33. The device application is modeled by state machines Isom In, Isom Out and Device, which are described in Profinet specification [1]. They synchronize the application with the communication and control the time when the application interacts with the process. The controller application utilizes the similar mechanism to synchronize the application with the communication, however in this thesis the device applications were focused. The implemented controller application sends and receives I/O data and does not precisely synchronize with the communication. The individual state machines are described in sections below.

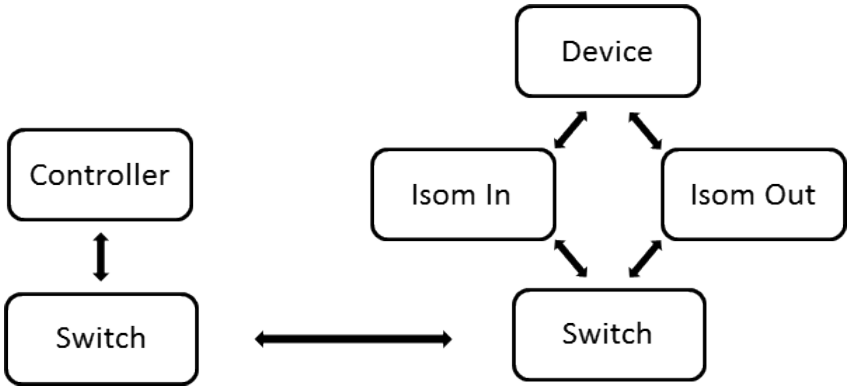


Figure 33: Isochronous application structure

4.5.1 Definition of isochronous application

The isochronous application is defined by data structures as in the case of the network definition. The data structures representing the inputs and outputs were defined as well as the time constraints when the data are scanned or written to the process.

The data structure describing an input of one device is in table 11. It contains ID of the input and frame ID of the message which carries the data through the network.

Attribute name	Description
id	Id of the input
carrier	Frame ID of the frame which carries the input data

Table 11: Structure Input

Outputs are described by the data structure in table 12. The attributes are the same as in the case of the input. One more attribute is defined and it is the device which shall receive the output.

Attribute name	Description
id	Id of the output
carrier	Frame ID of the frame which carries the output data
device	Device where the output data should be set

Table 12: Structure Output

Several time constraints are defined to synchronize data handling in devices. They are listed in table 13.

For the synchronization of the outputs with the communication cycle there are defined constants *T_IO_OutputValid* and *T_IO_Output*. Constant *T_IO_OutputValid* is the time from the beginning of the communication cycle when the output data must be received by a device. *T_IO_Output* is the time measured from the beginning of the communication cycle when the output data are written to the process.

Constant *T_IO_Input* is the time when the input data are scanned from the process. Constant *T_IO_InputValid* is the time when the input data are available for the communication. Both times are measured before the end of the communication cycle.

Name	Description
T_IO_OutputValid	Time when the outputs are available from communication
T_IO_Output	Time when the outputs are set
T_IO_InputValid	Time when the inputs are available for communication
T_IO_Input	Time when the inputs are scanned

Table 13: Global constants of isochronous application

4.5.2 Extending switch to multiple cycles

The network was verified in one communication cycle, because of that it was necessary to extend the switch functionalities to multiple cycles. The only state machine which needed modification was Scheduler which generates *sync_event* for the application. Its extension is shown in figure 34. When the time of the switch reaches the length of the cycle, it is reset to zero and *sync_event* is triggered to notify the application about the beginning of the next communication cycle.

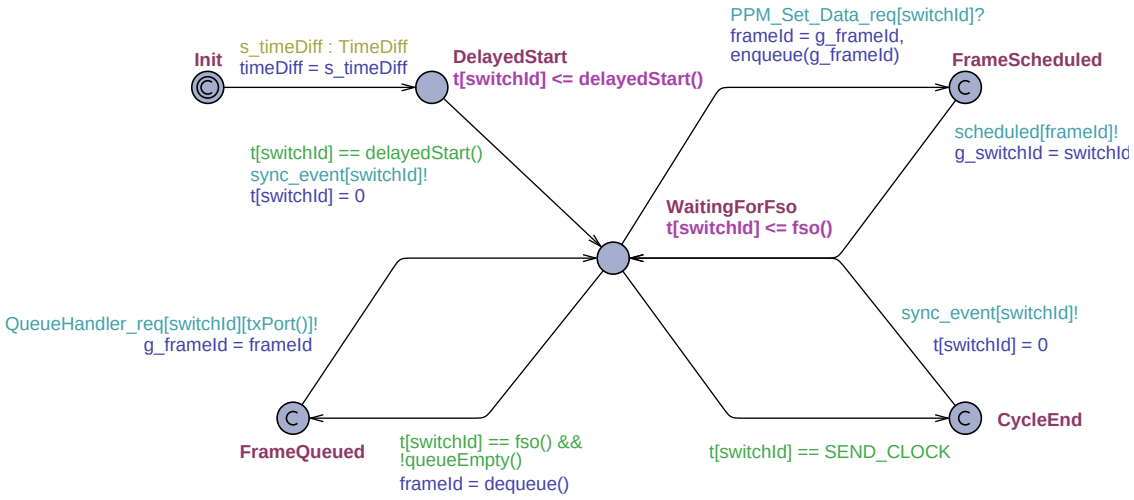


Figure 34: Extended Scheduler state machine

4.5.3 Controller

The controller sends outputs and receives inputs from the network devices. At the beginning of the communication cycle it schedules all the outputs for the communication by channel *PPM_Set_Data_req*. Then it waits to receive the input data from devices. The reception of data is indicated by channel *RED_RELAY_Data_ind* from Red Relay. The input data are stored to be able to check if all inputs were received at the end of the communication cycle. The diagram of the state machine is in figure 35.

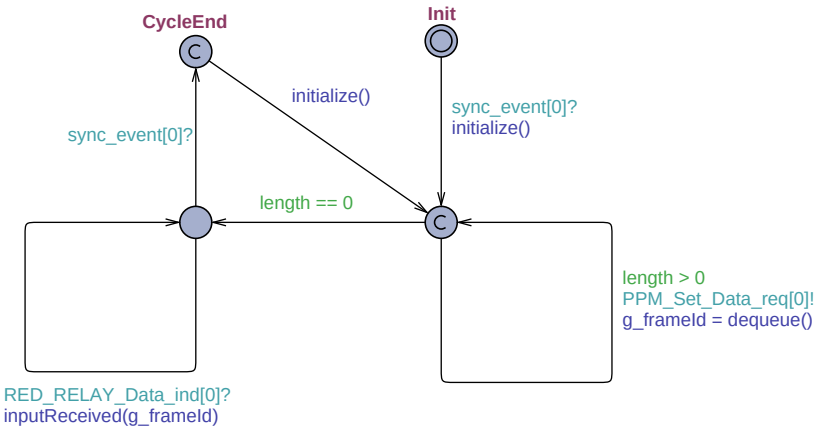


Figure 35: Controller state machine

4.5.4 Isom In

Isom In controls the time when the inputs of the device are scanned. When time T_IO_Input is reached the inputs are scanned by channel *SYNCH_IN*. Scanned input data are sent to the controller in next communication cycle by channel *PPM_Set_Data_req*. Diagram of the state machine is in figure 36.

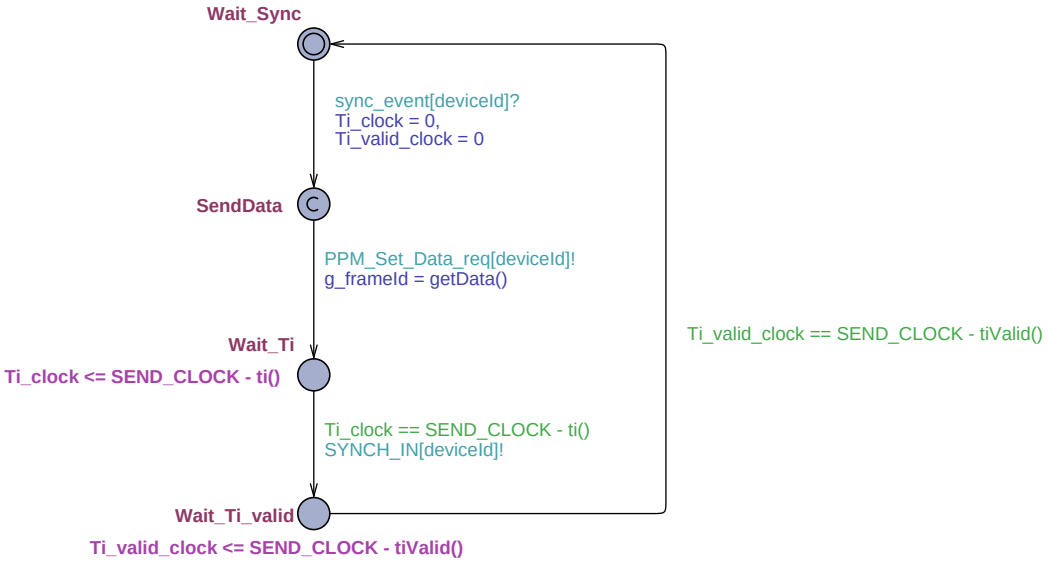


Figure 36: Isom In state machine

The state machine is synchronized with the beginning of the communication cycle by channel *sync_event*. Immediately after the start of a new cycle the data from the previous cycle are scheduled for the communication. Input data for the next cycle are scanned at time T_IO_Input. At this time the application reads the inputs from the process and transfers them to the communication memory. The transfer must be done before time T_IO_InputValid passes. After this time the data must ready for communication in the next cycle.

4.5.5 Isom Out

Isom Out controls the time when outputs are set on the device. It receives output data from the switch by channel *RED_RELAY_Data_ind*. For this state machine it is important that it receives message with output data before time T_IO_OutputValid passes. When the time reaches T_IO_Output received data are written to the process by channel *SYNCH_OUT*. Diagram of this state machine is in figure 37.

The state machine is synchronized with the communication cycle by channel *sync_event*. When a new cycle begins Isom Out waits for the reception of new output fata. If the data are not received before time T_IO_OutputValid passes, the state machine increments its internal counter of cycles without received data. This counter is decremented when the data are received in the future. When the counter reaches a certain limit a flag indicating that the maximum number of cycles without output data was reached. When the data are received before time T_IO_OutputValid, they are transferred from the communication memory to the application. The transfer must be completed before time T_IO_Output passes because at this time the outputs are set to the process.

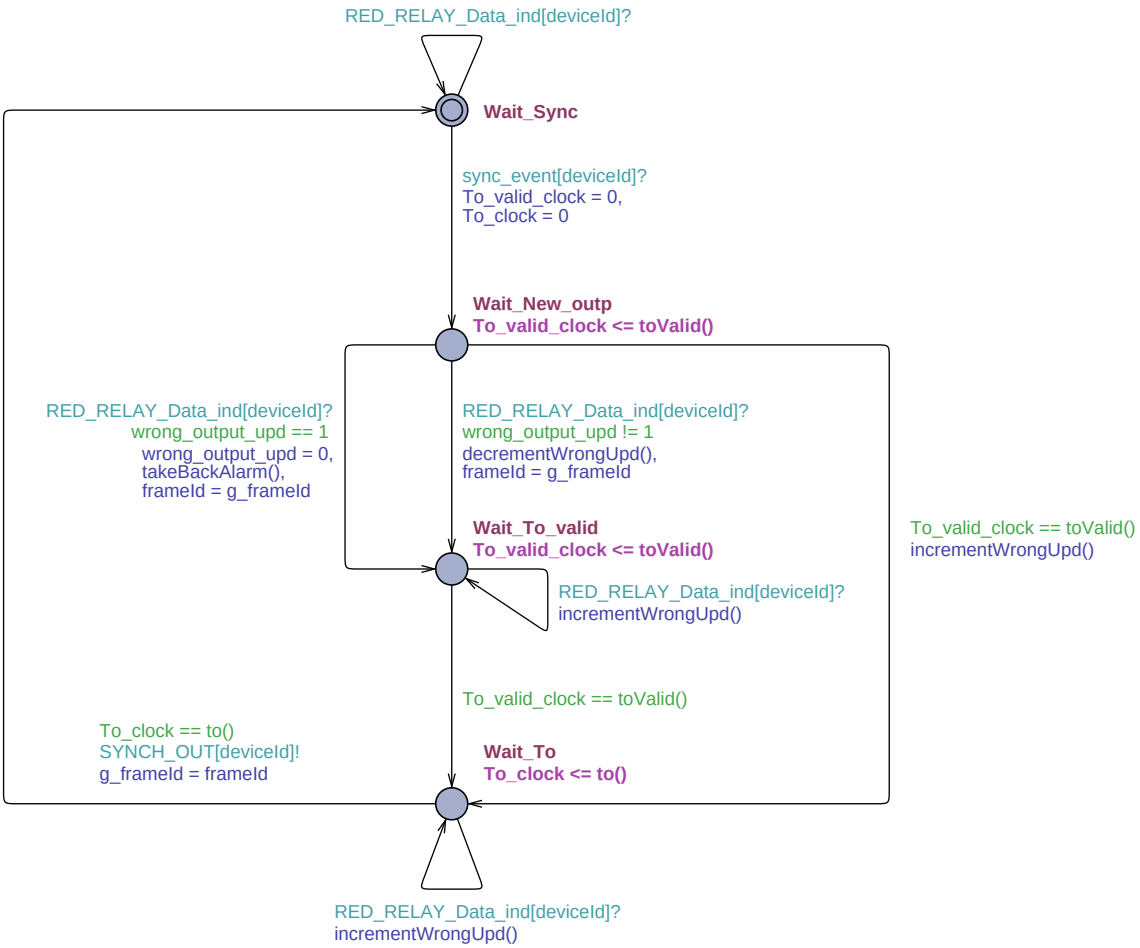


Figure 37: Isom Out state machine

4.5.6 Device

The Device state machine represents an IO device which can scan inputs and write outputs to the controlled process. It is notified from Isom In by channel $SYNCH\_IN$ when the scan of inputs should be done. Writing of the outputs is controlled from Isom Out state machine by channel $SYNCH\_OUT$. When the output data are written to the process the state machine is in state OutputsWritten, this state is used for verification purposes to checks if the data were written at correct time. State machine Device is shown in figure 38.

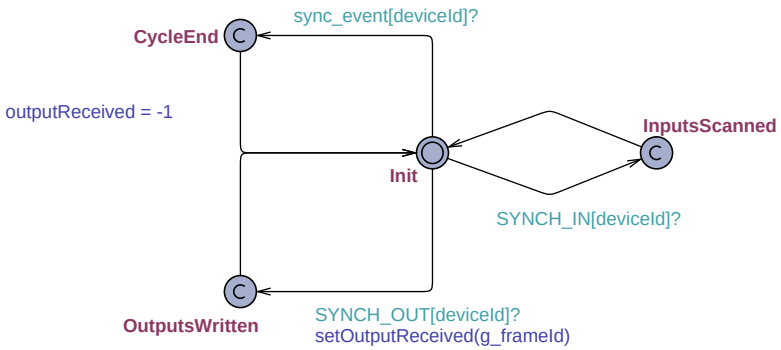


Figure 38: Device state machine

4.5.7 Verification of isochronous application

Inputs received

This rule checks that all defined inputs reach the controller before the end of the communication cycle. Implementation of the rule is in listing 18. Controller stores received inputs in an internal array which can be queried for presence of an input by function **hasReceived()**.

```
A[] Controller.CycleEnd
  imply
  forall(inputId : InputId) Controller.hasReceived(inputId)
```

Listing 18: Inputs received

Outputs received

Verification that the outputs were written by correct devices is done by rule in listing 19. It checks that the output data received by the device match the defined output in the corresponding structure.

```
A[] forall(deviceId : DeviceId)
  Device(deviceId).CycleEnd
  imply
  Device(deviceId).outputReceived == getOutputByDevice(deviceId).id
```

Listing 19: Outputs received

Output timing

This rule verifies that the outputs are written to the process at time T_IO_Output. Implementation is in listing 20. When outputs are written the Device state machine is in state OutputsWritten and the time must be equal to T_IO_Output.

```
A[] forall(deviceId : DeviceId)
  Device(deviceId).OutputsWritten
  imply
  t[deviceId] == T_IO_Output
```

Listing 20: Output timing

Input timing

This rule checks whether the input data are scanned from the process at time T_IO_Input before the end of the communication cycle. When the inputs are scanned the Device state machine is in state InputsScanned and the time must be equal to T_IO_Input before end of the cycle. Implementation is in listing 21.

```
A[] forall(deviceId : DeviceId)
  Device(deviceId).InputsScanned
  imply
    t[deviceId] == SEND_CLOCK - T_IO_Input
```

Listing 21: Input timing

4.6 Results of isochronous application verification

On top of the network described in section 4.4, an isochronous application was built. The application was implemented by the state machines described in section above. The application uses the message schedule verified in section 4.4 to transport the inputs and output from the controller to the devices.

The requirements on the behaviour specified in section 4.5 were evaluated by the verification engine and they have been fulfilled. The inputs are scanned at the correct time by every device and they are delivered to the controller. The outputs are delivered to every device and written to the process.

Chapter 5
Conclusion

The goal of this thesis was to develop methods for message schedule verification in Profinet IRT networks.

Two approaches of the verification were implemented in this thesis. First one was the approach with a static check of the schedule parameters according to the given set of rules. I have implemented a test tool with the rules and the graphical user interface for displaying the results. Using the tool, I have tested and successfully verified several schedules.

The work on the first approach was partially done within a project one semester before the actual diploma thesis project started. The design and the implementation of approximately two thirds of the work were done during that semester. During the thesis itself I have performed extensive tests with various topologies and discovered a few errors in the rule definitions as well as in the implementation.

In the second part of the thesis I have created a model of a network in software Uppaal. It was created from the state machines described in Profinet specification and parametrized by the message schedule for the line topology. I have specified the requirements on the behaviour and using the verification engine I verified that the network behaves according to the message schedule. During the verification, a difference of clocks of neighbouring switches was taken into account and the schedule was verified under conditions simulating the behaviour of real switches.

On top of the verified network model I have created an isochronous application to demonstrate that the network is capable of delivering data for a time critical application.

The work on the second approach was done completely within the diploma thesis project. The implementation of the state machines of the IRT switch as well as of the isochronous application was possible to be finished in time mainly because of my previous experience with the first approach.

In comparison of both approaches every possible situation that can happen in the network should be taken into account during the creation of the set of rules for the test tool. If any situation is not covered in the rules the test tool will not be able to discover possible malfunction of the network. The verification in Uppaal models the network behaviour, based on the specification of Profinet, which is parametrized by a schedule. The rules implemented in the test tool are encoded in the structure of the state machines and does not have to be specified explicitly.

During the verification of message schedules in Uppaal a significant disadvantage of this approach arose. The verification engine has to explore a complete state space of the modeled system and it can take a significant time to verify a schedule for large networks. On the other hand this approach is capable of exploring the behaviour under non ideal circumstances like different values of the local clocks.

The model of isochronous application can be extended in the future with a model of a device backplane bus. This would simulate the transfer of data from the communication part of the device to the physical outputs. By this approach it could be possible to verify feasibility of an isochronous application configuration for a certain class of devices with a common backplane bus.

As it was mentioned the verification of large networks can take a lot of resources (time and memory). A reduction of the state space of individual state machines could be done to speed up the verification.

References

[1] Industrial communication networks - Fieldbus specifications - Part 5-10: Application layer service definition - Type 10 elements. IEC 61158-5-10 ed.3. 08/2014.

[2] Industrial communication networks - Fieldbus specifications - Part 6-10: Application layer protocol specification - Type 10 elements. IEC 61158-6-10 ed.3. 08/2014.

[3] Behrmann, G., David, A., Larsen, K.G. A Tutorial on Uppaal 4.0. 2006.

[4] GSDML. Technical Specification for PROFINET IO. Version 2.31 – Date March 2014.

[5] PROFINET IRT Engineering. Guideline for PROFINET. Version 1.3 – February 2014.

[6] IRT Engineering IRT. Planning DLL Usage. Guideline for PROFINET. Version 4.0.1 – Date September 2014.

[7] Libxml2 [library]. Version 2.9.2, October 2014. Available at: `http://www.xmlsoft.org`.

[8] WinPcap [library]. Version 4.1.3, March 2013. Available at: `http://www.winpcap.org`.

[9] Boost [library]. Version 1.57.0, November 2014. Available at `http://www.boost.org`.

[10] wxWidgets [library]. Version 3.0.2, October 2014. Available at: `http://www.wxwidgets.org`.

[11] wxPdfDocument [library]. Version 0.9.4, August 2013. Available at: `http://wxcode.sourceforge.net/components/wxpdfdoc`

[12] Libpcap File Format [online]. July 2013. Available at: `http://wiki.wireshark.org/Development/LibpcapFileFormat`.

[13] XML Path Language (XPath) [online]. Version 1.0, November 1999. Available at: `http://www.w3.org/TR/xpath`.

[14] W3C XML Schema Definition Language (XSD) [online]. Version 1.1, April 2012. Available at `http://www.w3.org/TR/xmlschema11-1`.

Appendix A List of implemented tests

In this appendix is given a list of implemented tests in the test tool. Their structure is in figure 39.

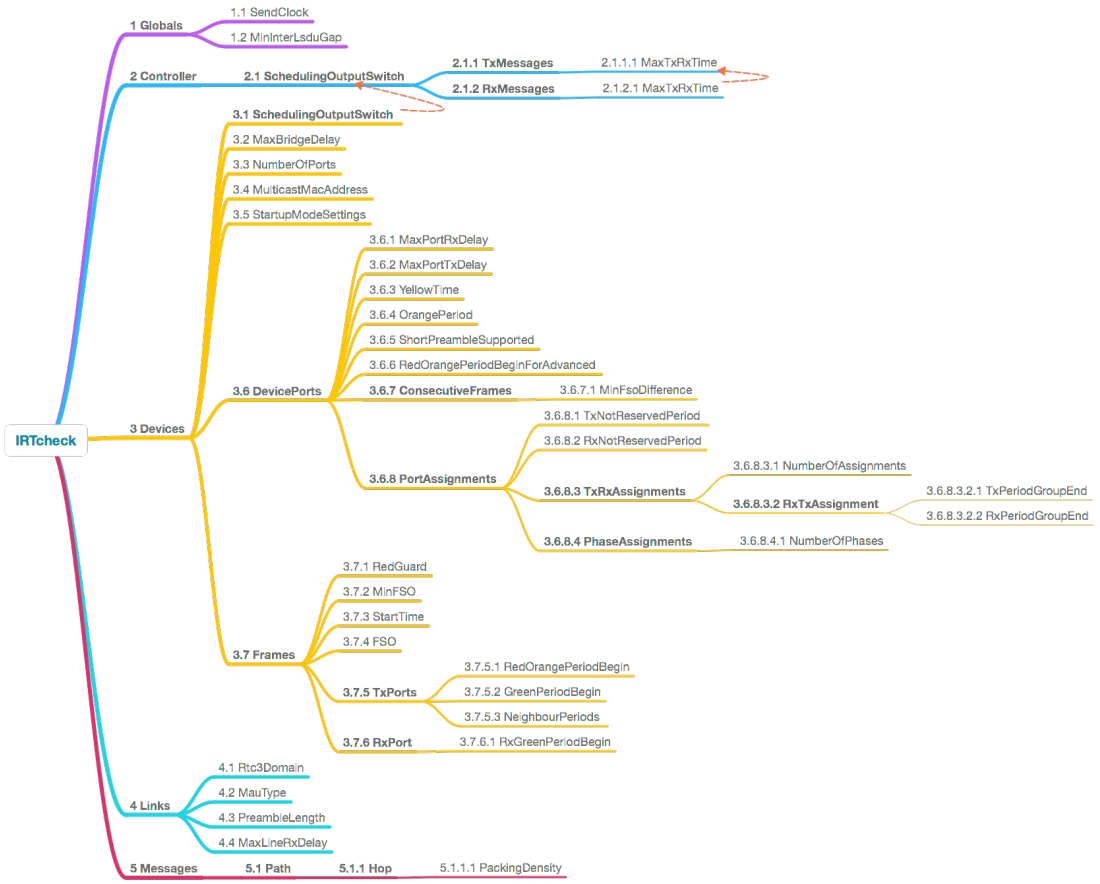


Figure 39: Structure of implemented tests

SendClock

Startup mode must be Advanced if SendClock is less than 250000.

MinInterLsduGap

MinInterLsduGap must be equal to the maximum of MinRTC3_Gap of all devices in the domain.

MaxTxRxTime

Frame transmission must end before MaxTxRxTime. EndOfFrameTransmission is computed as Time + (SA[6] + DA[6] + EtherType[2] + FCS[4]) * 80.

MaxBridgeDelay

Max bridge delay must be consistent among Scheduling Input, GSDML and PDIRdata.

NumberOfPorts

Number of ports must be consistent among scheduling input, GSDML and PDIR data.

MulticastMacAddress

If startup mode is advanced Multicast MAC address must be set to RTC3 multicast address or FFW multicat MAC address.

StartupModeSettings

Device must support tested startup mode.

MaxPortRxDelay

Max port rx delay must be consistent among scheduling input, GSDML and PDIR data.

MaxPortTxDelay

Max port tx delay must be consistent among scheduling input, GSDML and PDIR data.

OrangePeriod

TX/RXOrangePeriodBegin must be equal to TX/RXGreenPeriodBegin. ReservedIntervalBegin/End must be set to zero.

ShortPreambleSupported

If PreambleLength is set to Short in PDPortDataAdjust it must be supported in GSDML.

RedOrangePeriodBeginForAdvanced

Tx/RxRedOrangePeriodBegin must be zero for advanced startup mode.

MinFsoDifference

Difference of two consecutive frames must be larger or equal to MinFSODifference. MinFSODifference = AdditionalLsduGap + MinRTC3_Gap + (PreambleLength + SFD + DataLength) * 80.

TxNotReservedPeriod

TX Period beginnings must be set to zero in phases without reserved interval.

RxNotReservedPeriod

RX Period beginnings must be set to zero in phases without reserved interval.

NumberOfAssignments

Number of assignments must be the same in PDIRBeginEndData, SchedulingOutput Tx and Rx groups.

TxPeriodGroupEnd

TxOrangePeriodBegin, TxGreenPeriodBegin and end of TxPeriodGroup from Scheduling output must be the same.

RxPeriodGroupEnd

RxOrangePeriodBegin, RxGreenPeriodBegin and end of RxPeriodGroup from Scheduling output must be the same.

NumberOfPhases

Number of phases must be one of 1, 2, 4, 8 or 16.

RedGuard

All FrameIDs must be within RedGuard.

MinFSO

Frame send offset must be larger or equal to MinFSO.

StartTime

Frame send offset must be larger or equal to StartTime.

FSO

Frame send offset must be equal to the time from scheduling output.

RedOrangePeriodBegin

Start of frame transmission must be larger or equal to TxRedOrangePeriodBegin.

GreenPeriodBegin

Frame transmission must end before GreenPeriodBegin. EndOfFrameTransmission is computed as FSO + (SA[6] + DA[6] + EtherType[2] + FCS[4]) * 80.

NeighbourPeriods

Local TxGreenPeriodBegin must be smaller or equal to Adjacent RxGreenPeriodBegin.

RxGreenPeriodBegin

Frame ID must be received before beginning of green period.

Rtc3Domain

Two adjacent ports must belong to the same domain.

MauType

MAUType must be the same or missing for both adjacent ports.

PreambleLength

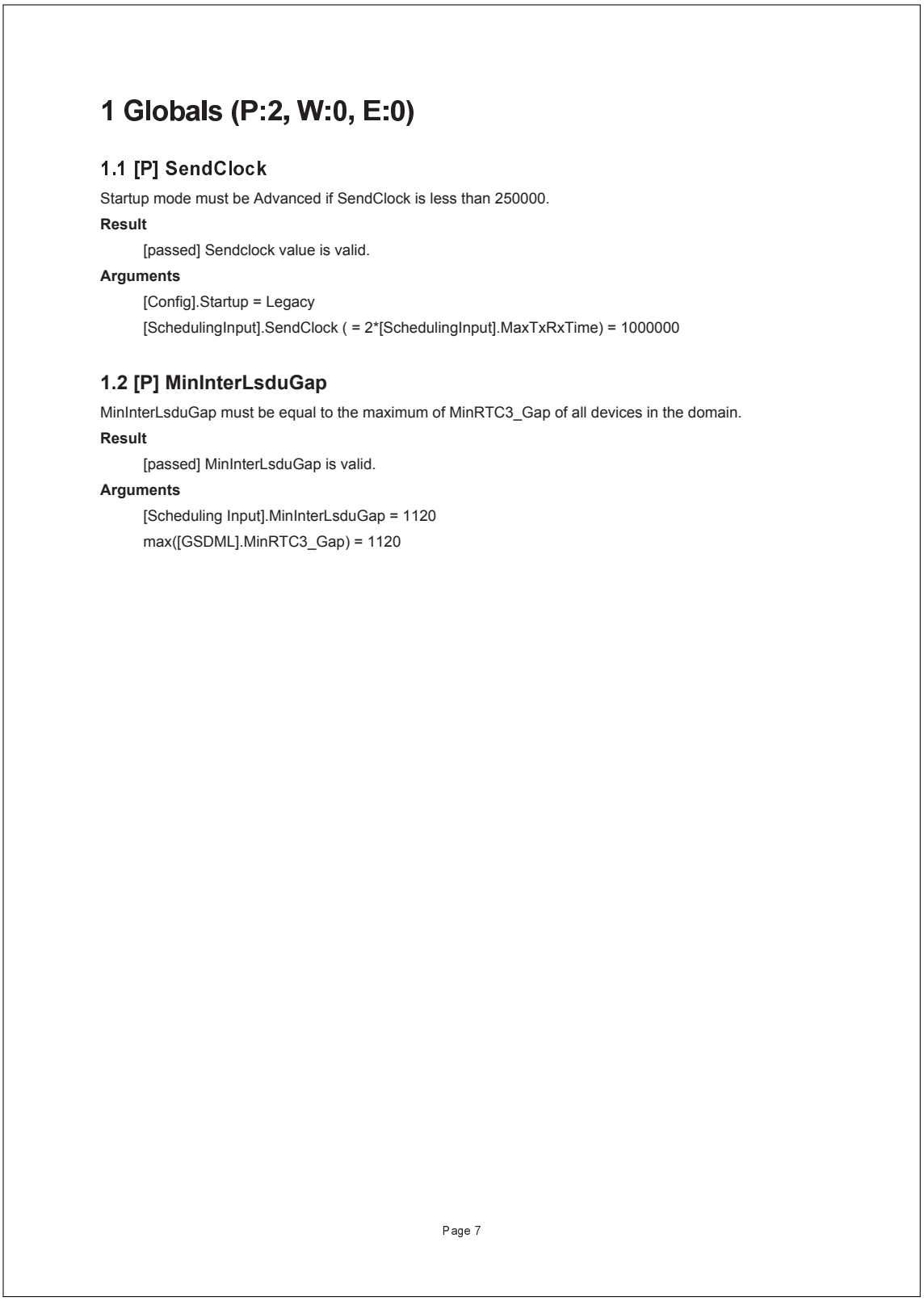
PreambleLengths must be the same or missing for both adjacent ports.

PackingDensity

Preamble lengths along the math of the message should be set to Short.

Appendix B Example of PDF report

This appendix gives example how PDF report generated by the test tool looks. Results of two tests are shown in figure 40.



1 Globals (P:2, W:0, E:0)

1.1 [P] SendClock

Startup mode must be Advanced if SendClock is less than 250000.

Result

[passed] Sendclock value is valid.

Arguments

[Config].Startup = Legacy

[SchedulingInput].SendClock (= 2*[SchedulingInput].MaxTxRxTime) = 1000000

1.2 [P] MinInterLsduGap

MinInterLsduGap must be equal to the maximum of MinRTC3_Gap of all devices in the domain.

Result

[passed] MinInterLsduGap is valid.

Arguments

[Scheduling Input].MinInterLsduGap = 1120

max([GSDML].MinRTC3_Gap) = 1120

Page 7

Figure 40: Example of PDF report

Appendix C Network verification

This appendix describes the setup of the network which was verified in chapter 4. The network consists of five switches in the line topology, it is shown in figure 41.

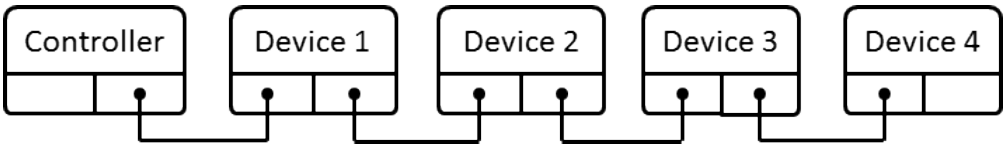


Figure 41: Topology of the network

Each switch has assigned ID from table 14.

Switch name	Switch ID
Controller	0
Device 1	3
Device 2	4
Device 3	1
Device 4	2

Table 14: Switch parameters

The parameters of the switches are listed in table 15.

Switch ID	relative	bridgeDelay
0	false	2920
1	false	2920
2	false	2920
3	true	2188
4	true	2188

Table 15: Switch parameters

The parameters of individual ports are in table 16.

			Switch ID				
			0	1	2	3	4
Port 1	rx	delay	333	374	374	198	198
		redOrangePeriodBegin	0	0	0	0	0
		greenPeriodBegin	0	22535	20349	30363	25995
	tx	delay	1217	280	280	6	6
		redOrangePeriodBegin	0	0	0	0	0
		greenPeriodBegin	0	25462	20768	31760	29248
Port 2	rx	delay	333	374	374	198	198
		redOrangePeriodBegin	0	0	0	0	0
		greenPeriodBegin	30019	20342	0	26640	24128
	tx	delay	1217	280	280	6	6
		redOrangePeriodBegin	0	0	0	0	0
		greenPeriodBegin	30760	20775	0	28603	24235

Table 16: Port parameters

In table 17 there are listed the forwarding rules for individual switches.

		frameId	fso	rxPort	txPort
Switch ID	0	256	25640	0	2
		258	18760	0	2
		260	11880	0	2
		262	5000	0	2
		257	9379	2	0
		259	16259	2	0
		261	23139	2	0
		263	30019	2	0
	1	261	13462	0	1
		260	22535	1	0
		262	15655	1	2
		263	20342	2	1
	2	263	15648	0	1
		262	20349	1	0
	3	257	5000	0	1
		256	29363	1	0
	4	259	9368	0	1
		258	24995	1	0

Table 17: Forwarding rules

Table 18 contains definition of individual messages. The columns Sender and Receiver contain IDs of the switches which send or receive the message.

Frame ID	Data length	Sender	Receiver
256	40	0	3
257	40	3	0
258	40	0	4
259	40	4	0
260	40	0	1
261	40	1	0
262	40	0	2
263	40	2	0

Table 18: Frame definitions

The links are defined in table 19. They are defined by switch ID and port ID of the source and destination of the link.

Link ID	sourceSwitch	sourcePort	destinationSwitch	destinationPort	lineRxDelay	linkDelay
0	0	2	3	1	120	120
1	3	1	0	2	120	120
2	3	2	4	1	120	120
3	4	1	3	2	120	120
4	4	2	1	1	120	120
5	1	1	4	2	120	120
6	1	2	2	1	120	120
7	2	1	1	2	120	120

Table 19: Link definitions

Table 20 contains the constant of the network.

Constant	Values
JITTER	1000
SEND_CLOCK	1000000

Table 20: Constants

Appendix D Verification of isochronous application

This appendix describes the setup of the isochronous application which was verified in chapter 4. The same network as in the case of verification of IRT network was used. Table 21 contains a list of the inputs. For every input a message carrying it is defined.

Input ID	Carrier
0	258
1	259
2	261
3	263

Table 21: Definition of inputs

Table table 22 contains a list of the outputs with messages carrying them. Every output has assigned a device where it should be received.

Output ID	Carrier	Device
0	256	3
1	258	4
2	260	1
3	262	2

Table 22: Definition of outputs

Table 23 contains a list of constants which were used for the verification.

Constant	Value
T_IO_OutputValid	70000
T_IO_Output	80000
T_IO_InputValid	100000
T_IO_Input	110000

Table 23: Definition of outputs

Appendix E Content of attached CD

In table 24 there are listed folders on the attached CD and the description of their content.

Directory	Description
pdf	This thesis in PDF format
uppaal	Uppaal projects with Profinet IRT network and the isochronous application.

Table 24: Content of attached CD