Modeling and implementation of an interface adapter between wide format printers and finishers

Tirupalathurai Kannan, B.

Award date:
2013

Department of Mathematics and Computer Science

Software Engineering and Technology Group

Modeling and Implementation of an Interface Adapter between Wide Format Printers and Finishers

Master Thesis

Author: **B. Tirupalathurai Kannan**

Supervisors:

Dr. Lou Somers (TU/e)

Ing. Stephan Derks (Océ)

Venlo, August 2013

Abstract

Interface implementation between different software components usually takes a considerable amount of time and effort. Océ faces a similar problem in the area of wide format printers. The wide format printers produced by Océ have to be interfaced with finishers from Océ and other companies. Finishers are devices which are connected to printers and perform ancillary activities like folding and stacking of papers with sheets delivered from a printer. The project aim was to develop a generic adapter which takes care of the electrical and functional interface incompatibilities between Océ wide format printers and finishers. The software adapter was designed using a model based approach which reduces the functional interface implementation time. The main contributions of the project are architecture and design for the adapter, design of a generic protocol between Océ wide format printers and the adapter, selection of a modeling approach for the adapter design and a prototype implementation of the adapter (deployed on an interface board) to test the feasibility of the design.

Keywords: Océ wide format printers, finishers, adapter, modeling approach

Dedicated to my beloved parents who have always been a source of inspiration and motivation in every step of my life.

Preface

This thesis is a result of the graduation project as part of the curriculum for the degree MSc in Embedded Systems at TU/e. The project was carried out within the Software Engineering and Technology group of the Mathematics and Computer Science department of the TU/e in collaboration with the System and Development department in the Océ-Technologies. The project duration was from February 2013 to August 2013 at Océ-Technologies, Venlo. The project had a preparation phase from December 2012 to February 2013 as part of the MSc curriculum.

Barath Tirupalathurai Kannan
August, 2013

Acknowledgments

I wish to thank the TU/e and the management of Océ-Technologies for providing me with the scholarship to finance my studies in the Netherlands. I would like to thank the management of Océ-Technologies for providing me opportunity to participate in a summer internship, master thesis project and other activities related to the company. My special thanks to Lou Somers who has been guiding me in my summer internship and my master thesis. His expertise, practical approach and risk analysis helped me to foresee most of the risks at an earlier stage of the project. My sincere thanks to Stephan Derks who has guided me in my master thesis. He gave me good amount of freedom to work in the project and was willing to accept new ideas from me. I will miss my weekly meetings with my supervisors.

The working environment in Océ has been very amicable and there are several colleagues who helped me in my project. I thank Erik Schoenmakers for helping me with the generic protocol design and the ASD. I thank Louis Van Gool for helping me with the ASD and interface language. I would like to thank Henri Hunnikens, Ralph Woltering, Gert Voets, Sidney Laracker, Bart van der Meulen, Harald Schwindt, Richard van Enckevort, Robert Jacobs, Rob Janssen, Kim Leeks and Hans Derks who facilitated my successful completion of this work in different capacities which space will not permit me to elaborate then.

I would like to give warm regards to Reinder J. Bril and Tom Verhoeff who gave valuable feedback during my kickoff and midterm presentations which helped me to improve the project. I extend my regards to Arjan Mooij from the Embedded Systems Institute who gave initial insight into several modeling approaches using Petri nets and the ASD.

Last, but not least, I would like to thank my parents and friends. Without your support and prayers it would not have been possible for me to succeed in life.

Barath Tirupalathurai Kannan
August, 2013

Table of Contents

List of Figures

List of Tables

List of Acronyms

CRC	Cyclic Redundancy Check
PC	Personal Computer
CPU	Central Processing Unit
XML	Extensible Markup Language
IPC	Inter Process Communication
CASE	Computer-aided Software Engineering
SEA	Specification of Elementary Activities
SBS	Sequence based Specifications
DPC	Deferred Procedure Call
ASD	Analytical Software Design
SDL	Specification and Description Language
OS	Operating System
USB	Universal Serial Bus
ECB	Ethernet Control Model
TFTP	Trivial File Transfer Protocol
IP	Internet Protocol
GUI	Graphical User Interface

Chapter 1

Introduction

Wide format printers operate on a continuous sheet of paper. Finishers perform ancillary activities like folding, and stacking with sheets coming out of the printer. The wide format printers from Océ need to interoperate with various types of finishers from Océ and other companies. The interface specification between a printer and a finisher involves mechanical, electrical and functional descriptions. The mechanical interface description decides upon the physical connection between the printer and the finisher. The electrical interface description decides upon the type of communication, communication media and electrical connectors. The functional interface specification decides upon the functional interface protocol employed between the printer and the finisher. The interface specification differs from one finisher to another. The interfacing of a printer and a finisher takes a considerable amount of time and effort due to the varied interface specification across finishers. The objective of this thesis is to find a solution for this interfacing problem with respect to the electrical and functional interface between Océ wide format printers and finishers.

This chapter emphasizes the aforementioned problem and related concepts in a detailed manner. Section 1.1 explains the background domain concepts necessary to understand the problem. Section 1.2 explains the problem description in detail. Section 1.3 explains the goals associated with this project. Section 1.4 explains the approach taken in solving the problem. Section 1.5 explains the organization of rest of the report.

1.1 Background

Océ is a leading player in the area of wide format printing systems. The products are primarily printers and copiers and may consist of other modules like an external PIM (paper input module), a scanner and a finisher. Figure 1.1.1 shows a printer connected to a finisher. The printer configures the finisher for handling finishing jobs and signals the finisher about the arrival of the sheets to the

finisher. The behavior of the system can be compared to a master and slave relationship where the printer is the master sending commands to the finisher and the finisher is the slave responding to the printer's commands.

Figure 1.1.1: Printer connected to a finisher

1.1.1 Wide Format Printers

Wide-format printers are generally printers with a print width between 17" and 100". These are used to print banners, posters, Architecture/Engineering/Construction (AEC) diagrams, computer aided design (CAD) diagrams and geographic information system (GIS) data. These generally use a roll of print material rather than individual sheets and may incorporate hot-air dryers to prevent prints from sticking to themselves as they are produced [7]. An example of a wide format printer, the Océ plot wave 350 printer is shown in Figure 1.1.2. The Océ plot wave printer produces up to six A1 or D-size plots per minute.

Figure 1.1.2: Océ plot wave 350 printer

1.1.2 Wide Format Finishers

Wide format finishers perform post-printing actions such as folding or stacking. These are manufactured by Océ or by other companies. These have to be connected mechanically and electrically with a printer in order to operate. These interact with the printer through a functional interface protocol. A printer can set finishing specification like folding length for the sheets through the functional interface. The interfacing of finishers with printers takes a considerable amount of time and effort due to the fact that finishers can be from other companies with different interface requirements. Some of the finisher's vendors do not provide complete dynamic specification of the functional interface making it tougher for the printer integration. An example of a wide format finisher, the Océ 4311-fullfold finisher is shown in Figure 1.1.3. The Océ 4311 finisher produces folded copies and drawings that are accurate up to the millimeter scale.

Figure 1.1.3: Océ 4311-fullfold finisher

1.1.3 Current Setup

Figure 1.1.4 shows the simplified current setup of the printer-finisher interfacing. Printer type X, here is any wide format printer from Océ and finisher type Y is any wide format finisher from Océ or other companies. The term 'type' is mentioned here to stress the differences in the interface across finishers. The term 'X' represents any number to signify an individual printer. The term 'Y' represents any number to signify an individual finisher. The setup is simplified because some printers can interface with more than one type of finisher. The important point to stress here is that the most of the current wide format printers from Océ can interface with only one type of finisher. Currently, a printer can interface with only one finisher at a time.

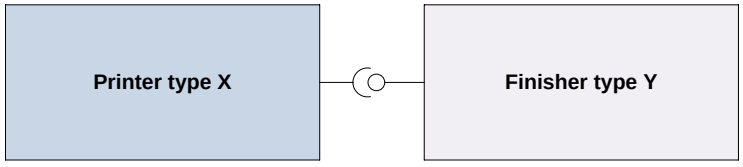


Figure 1.1.4: Simplified current setup of the printer-finisher interfacing

The current generation of finishers uses RS-232 or RS-232 variants (RS-232 with an additional pin for power control) or serial or parallel connectors for electrical connectivity. The finishers from other vendors have their own functional interface specification which has to be used for printer-finisher interfacing. Due to the above mentioned reasons, the printers can interoperate with only one or two types of finishers. The problem with this approach is that the finisher usage is restricted to some printers. The interface implementation with a new finisher involves re-work every time. Océ wide-format printer's electrical interface will be changed in the future to universal serial bus (USB) connectors to bring uniformity at the level of electrical interface and to support increased data rates. This change should be accompanied with the support for existing finishers since these are already interfaced with the printers. The idea is that the printers should be able to interface with all finishers. First, this demands a hardware converter for interfacing different electrical connectors. A typical example justifying the above statement is that the converter for USB to RS-232 variants is not readily available in the market. Second, there is a necessity for a functional interface converter to support different functional interfaces across finishers.

1.2 Problem Description

Océ faces a challenge in interfacing their printers with finishers from Océ and other companies. Finishers communicate with printers using specific functional and electrical interfaces. The current printer's software has been designed to interface a particular finisher. This approach limits the interoperability of finishers with printers and affects the software development time needed for the interface implementation. The approach also affects the maintainability of the printers with a necessity to change the printer's software for every finisher related software patches. The requirement is that any type of Océ wide format printer must be able to communicate with any type of finisher. The idea is to come up with a model based framework which eases the integration (with respect to time and effort) of the existing finishers and robust enough to integrate future generation of finishers. The approach should also improve the maintainability of the printer's software.

Business Motivation: Océ sells printers and finishers separately to its customers. Currently, customers pay for the printer which includes the price for finisher's interfacing hardware and software. This situation is unwarranted since some customers do not require finishers but they have to pay extra money indirectly. This affects the competitive pricing of Océ printers in the market. The new framework must ensure that the finisher software is not part of the printer in order to conform to this business interest. Another motivation is to reduce the additional time and effort spent in the interface implementation which ultimately adds to the cost of the printer. The ambition is to make the design reusable as much as possible for finisher integration thereby gaining an advantage on the cost.

Problem Statement: *To propose a model based framework for interfacing Océ wide format printers with any type of wide format finisher and to prototype a model based framework to test the feasibility.*

1.3 Goals

Given the problem definition in the previous section, the goals of the projects are the following:

- Propose a solution for printer-finisher interfacing at the product level
- Identify the present and future interface requirements of finishers
- Construct an architecture that supports finisher interface requirements
- Specification and design of an interface protocol introduced due to the new architecture
- Investigate different model based approaches for the adapter software design
- Identify a suitable modeling approach for the project
- Prototype using the modeling approach to test the feasibility of the modeling approach

The scope of the thesis is limited to wide format printers and not cut-sheet printers. The cut-sheet printers work on individual sheets of paper like A4 and A3. The cut-sheet printers have different functional interface requirements for finisher operations.

Note: The term 'adapter' refers to a finisher adapter which takes care of the electrical and functional interface incompatibilities between the printer and finishers. This terminology will be used in rest of the document.

1.4 Approach Taken

This section explains the research approach employed in this project. The global solution for the problem, research challenges considered in the project and the project life cycle are discussed in detail.

1.4.1 Proposed Solution

The problem requires a hardware and software converter to translate different finisher interfaces. The proposed solution is to deploy an adapter, a combination of hardware and software converters. The adapter takes care of both the electrical connection incompatibilities and functional interface differences. Figure 1.4.1 shows an adapter which can interface different types of printers with different types of finishers (adapter can interface only one printer/finisher at a time).

The interface between the printer and the adapter remains the same irrespective of the finisher. The interface between the adapter and the finisher is based on the specific finisher protocol. The location of the adapter was decided from the business motivation. Hence, the plan was to run interface software in a separate hardware board outside the printer with its own operating system.

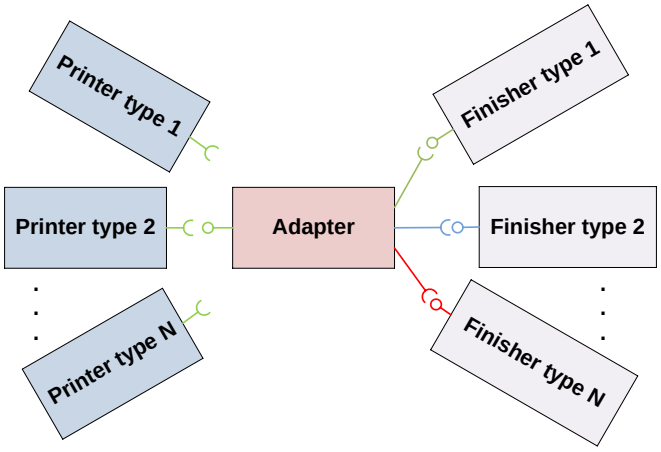


Figure 1.4.1: Proposed solution for the printer-finisher interfacing

The introduction of an adapter between printers and finishers introduces two interfaces following different protocols which are explained below:

Generic protocol: Interface protocol that specifies the functional interface between all Océ wide format printers and the proposed adapter. This terminology will be used in the rest of the document for referring to the printer-adapter functional interface protocol.

Specific protocol: Interface protocol that specifies the functional interface between a finisher and the proposed adapter. This interface protocol is the same as the finisher protocol of this finisher. This terminology will be used in the rest of the document for referring to the finisher-adapter functional interface protocol.

The following advantages are achieved in the proposed solution:

- There is a clear separation of concerns. The finisher interfacing software is isolated from the printer software.
- The customer has to pay only the correct amount of money based on his/her requirements.
- The printer remains numb to the changes in the finisher. This is due to the introduction of the generic protocol between the printer and the adapter.
- The technology and the tooling can be selected irrespective of the printer's technology. For example, the adapter software is aimed to be developed using a model based approach which reduces development time and effort.
- The maintenance of the adapter software is simpler. More finisher related patches can be delivered without changing the printer's software.
- There is no necessity to change the printer's hardware for finisher interfacing since the electrical connectivity with the adapter is fixed.

Alternative Solution: The alternative solution is to run the adapter software in the printer and to have a separate hardware converter outside the printer. The adapter software can be made as a plug-in which can be incorporated when required. This solution conforms to the business interests

and tackles the problem. But the solution does not provide the same clear separation of concerns as the proposed solution. The patches related to the finisher will be incorporated in the printer's software thereby not improving the maintainability of the printer's software. But, this solution avoids the additional marshalling/unmarshalling performed between the printer and the adapter in the proposed solution which in turn improves the timing performance.

1.4.2 Research Challenges

The previous section shows the proposed solution and the implementation of this solution poses research challenges at various levels in the project. The approach followed in solving these research challenges is mentioned briefly in this section. The following research challenges were considered in the thesis:

- *Making the adapter software architecture to handle different finisher capabilities*
  Motivation: The purpose of making an adapter is to use it for present and future generations of finishers. The adapter architecture should be robust enough for changes in the finisher capabilities. For example, one of the changes expected in the future is the use of multiple finishers. The identification of expected future capabilities is crucial in achieving this type of architecture.
  Approach: Different architectural views were created to understand the fit with the finisher requirements. Alternative architectures were considered to check the validity of the proposed architecture. Chapter 4 explains different architectural views of the adapter software.

- *Making the adapter software architecture suitable for model based design*
  Motivation: The solution is aimed at incorporating a model based design. The architecture should provide a clear separation of concerns indicating the exact location to fit in model based software in the architecture. This helps in the unbiased evaluation of the modeling tools.
  Approach: The focus of the model based adapter lies in the logical message conversion ignoring bit level information. The architecture was made to provide clear separation of marshalling/unmarshalling blocks from the message conversion block. Chapter 4 explains different building blocks of the adapter software when it is deployed.

- *Making the generic protocol design to fit with the existing finisher protocols*
  Motivation: The existing protocols differ from one finisher to another. The generic protocol has to fit the existing protocols. The protocol design has to be simple since the effort spent in the adapter state machine depends on the design of this protocol.
  Approach: A top-down approach was employed where in a generic protocol was proposed based on the domain requirements. Then, the protocol was checked for the fit with existing

Note: Marshalling is the process of gathering data and transforming it into a byte stream before it is transmitted over a network and unmarshalling is the reverse process of converting byte stream back to the original data.

finisher protocols. Appendix C: “Generic protocol for adapter interface” contains detailed sequence diagrams showing the mapping of the generic protocol with different finisher protocols.

- *Making the generic protocol design robust for future changes in the finisher protocols*
  Motivation: The changes in finisher capabilities in the future will introduce changes in the finisher protocols. The adapter software must be able to handle these changes without the necessity to modify the generic protocol messages. This is a critical factor to consider while coming up with a generic protocol otherwise considerable re-work has to be done on the printer and adapter interface protocol state machines which will dilute the purpose of having a generic adapter.
  Approach: The generic protocol was classified into different categories with some sections for the expected future capabilities. The interrelation between different sections was considered during the protocol specification. For example, to support multi finisher capabilities in the future requires a finisher identifier in all the messages. Chapter 4 briefly explains the generic protocol design decisions. The complete information regarding the generic protocol is available in the Appendix C: “Generic protocol for adapter interface”.

- *Analysis of complexities for message conversion in the adapter software*
  Motivation: The adapter has to perform message conversion in order to interface different type of printers and finishers. The identification of message conversion complexities is critical in verifying the feasibility of a generic adapter.
  Approach: The different types of message conversion complexities were identified before the generic protocol specification in section 2.2.4. Then, the conversion complexity threshold was determined in section 4.2.2 after the generic protocol specification.

- *Handling of concurrency in the adapter*
  Motivation: The adapter has to interface with a single printer and one or more finishers at the same time. This introduces the question of handling concurrent messages from different components. This question has to be answered starting from the architecture to the implementation.
  Approach: The adapter has different marshalling/unmarshalling components for the printer and finishers. The message conversion module implementation gives priority to messages from the printer. If there is a concurrent message both from the printer and finishers, then printer message is processed immediately and the finisher messages are placed in a queue. Chapter 3 and chapter 6 explain the concurrency handling in the adapter.

- *Verification of generic protocol consistency*
  Motivation: The generic protocol will be implemented as a separate state machine in the printer and the adapter. Both the state machines have to be consistent with the protocol state machine. This is necessary to avoid run time inconsistencies like deadlock and livelock.

Approach: The modeling selection criteria in chapter 4 included this aspect before choosing the modeling approach. The modeling tool is used to verify the generic protocol consistency.

- *Selection procedure of a modeling approach for the adapter*
  Motivation: Model based software is a very generic term and the project aims to find software suitable for the design of adapter software with certain message conversion complexities.
  Approach: The selection procedure was divided into three broad categories comprising of model based aspects, quality aspects and engineering aspects. The details are explained in chapter 4.

- *Handling incomplete dynamic behavior of the finisher protocol specification*
  Motivation: The finisher protocol specification especially from other vendors sometimes has an incomplete dynamic specification. This means the adapter software should implement a mechanism to report the inconsistent protocol behavior.
  Approach: The approach employed is to have a strict finisher protocol state machine in the adapter and to use additional component to detect the occurrence of illegal behavior. Chapter 6 explains about the details of this additional component.

- *Selection of scenarios for the prototyping of the adapter software*
  Motivation: The aim of the project is to test the feasibility of the model based framework which includes the architecture, protocol and model based approach for the adapter. The scenarios chosen for testing the feasibility must include all message conversion complexities which make the feasibility more concrete.
  Approach: The scenarios were selected based on message conversion complexities especially to test different message conversion complexities. The scenarios which require different threads to communicate were also tested. Chapter 6 explains about the choice of scenarios for prototyping the adapter software.

- *Testing strategy employed to test the entire setup*
  Motivation: The main objective of the thesis is to come up with a generic adapter. The testing of the adapter is possible only with printer and finisher modules. The project should also look into different possibilities of testing the adapter.
  Approach: The adapter software was made to run in a prototype board with its own operating system. The printer and the finisher run in the PC connected to the board via USB. Ethernet over USB is used for communication between different modules. Printer and finisher stubs were created to test the functionality of the adapter. Section 6.1 explains the test setup used in this project.

1.4.3 Project Life Cycle

The 'Agile' software development methodology was followed in the project. Figure 1.4.2 gives the overall picture of the tasks involved in the project but the flow was not strictly followed. The major activities in the project are given at the top block with internal activities at the bottom. There are several activities carried out in parallel especially among modeling approach selection, generic protocol design and prototyping phases in making the adapter software. This approach helped to reduce the risks at an earlier stage in the project. For example, the testing of the prototype board and testing of the architecture in the board was carried out before the modeling approach was selected. This ensured that there were no setbacks at the last stage of the project due to lack of experience in handling the hardware.

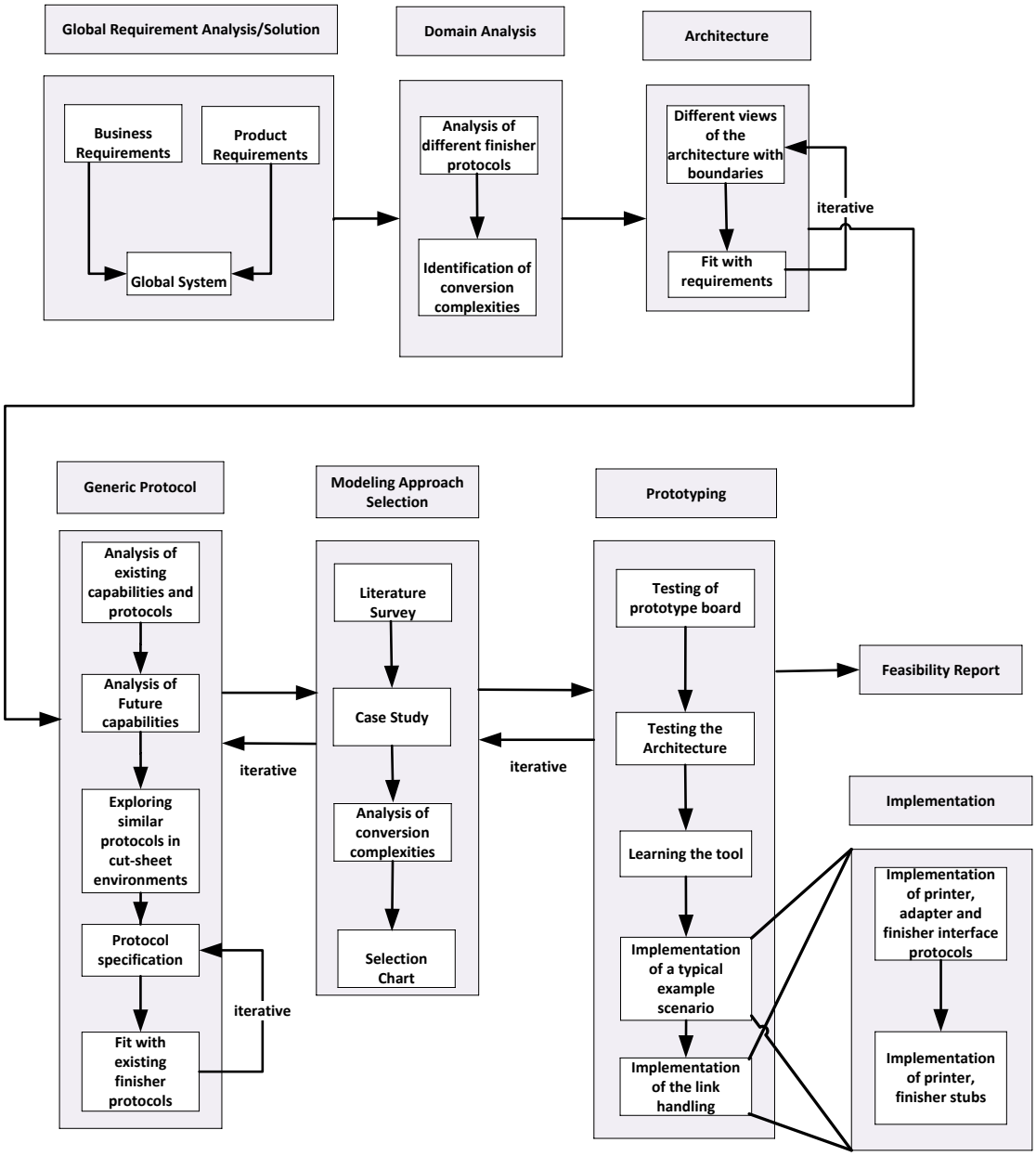


Figure 1.4.2: Project life cycle

The major phases of the project are listed below:

Global Requirement Analysis/Solution: This phase is the initial phase in the project. This phase is explained in the previous sections of this chapter.

Domain Analysis: This phase is explained in the chapter 2 in a detailed way. This phase comprises of understanding different finisher protocols and identification of type of complexities involved in the message conversion.

Architecture: Chapter 3 explains the details behind this phase. This phase involved identification of a suitable architecture for the adapter software.

Generic Protocol: The generic protocol specification and design are performed in this phase. Chapter 4 describes the various aspects involved in designing the generic protocol.

Modeling approach selection: Chapter 4 explains this phase in a detailed manner. This phase involves comparing different modeling approaches and the identification of a suitable modeling tool for designing the adapter software.

Prototyping: Chapter 6 describes the details of this phase. This phase involves the actual prototyping to verify the feasibility of the solution.

1.5 Report Organization

The rest of the chapters in the report are organized as follows. Chapter 2 describes the domain analysis carried out in the thesis. Chapter 3 construes the architecture of the adapter software in a detailed manner. Chapter 4 describes the generic protocol and the modeling approach selection in a detailed way. Chapter 5 explains the analytical software design (ASD) in a detailed manner. Chapter 6 describes the prototyping carried out in the project. Chapter 7 explains the conclusions derived from the project and future possibilities. The UML diagrams shown in different chapters follow UML 2.2 standard.

Chapter 2

Domain Analysis

This chapter explains the different activities performed in the domain analysis phase. Finisher protocols, both from Océ and from other companies were analyzed. The finisher protocol from another company has an incomplete dynamic functional interface specification. The identification of different finisher capabilities and framing of message categories is essential in understanding the complexities involved in the adapter logic and framing the architecture of the adapter software.

Section 2.1 lists the finisher capabilities which include the present capabilities and expected capabilities in the future. Section 2.2 explains the finisher protocol analysis carried out to understand the nature of different finisher protocols. Section 2.3 describes the interface adapter solutions available in the cut-sheet environment. This chapter is concluded in section 2.4 which outlines the important findings of this chapter.

2.1 Finisher Capabilities

The finisher capabilities represent the behavior and functionalities of the finisher. The present capabilities can be understood from the existing finisher protocols. The expected future capabilities are obtained from discussion with the stakeholders especially architects of the Océ wide format printers.

2.1.1 Present Capabilities

The present finishers behave like a slave to the printer and carry out jobs assigned by the printer. Most of the finishers has their own power supply and can be turned on/off independently. For some finishers, an additional power signal has to be received from the printer in order to turn on/off. The establishment of a connection with the printer happens by sending link establishment messages. Once the finisher is connected to the printer, the finisher can perform sheet related tasks

sent by the printer. The printer can configure the finisher differently for every job sent. The finisher can report errors to the printer and vice-versa. The finisher can also work under the direct control of the operator without the requirement for a printer. The tasks performed by the wide format finishers include the following:

- *Stacking* – creating a pile of sheets
- *Folding* – bending the sheets by applying pressure
- *Stickers* – pasting labels in the sheets
- *Turn table* – changing the orientation of the sheets

2.1.2 Future Capabilities

Three future capabilities that are expected in the future were identified from discussion with the stakeholders. The expected future capabilities are software download, power cycling and multiple finishers.

Software Download: This is a feature by which the device can update the firmware. The device can either download the firmware directly (internet or USB device) or get the firmware through another device like a printer. There are two types of software downloads that needs to be incorporated. The two types are adapter software download and finisher software download. Figure 2.1.1 shows the process involved in the two software download features. For the adapter software download, the adapter firmware will be downloaded by the printer from external source like a USB device or internet and then transferred to the adapter. The finisher software download feature is currently not available in finishers. This feature has three steps, the printer downloads the finisher's firmware from an external source, then transfers it to the adapter and finally the adapter transfers it to the finisher. The generic protocol and the adapter software must ensure that this feature can be achieved in the future.

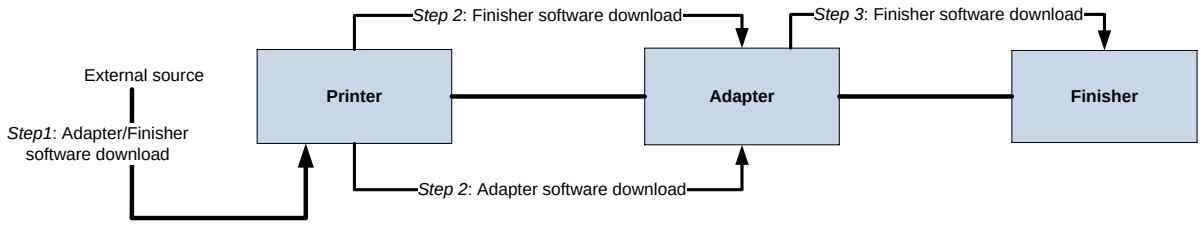


Figure 2.1.1: Software download feature

Multiple finishers: In the multiple finishers' environment, finishers are stacked one behind the other allowing finishing tasks to be performed continuously on the sheets. The adapter software should be able to communicate with all the finishers. The current generation of finishers does not support this multiple finisher functionality. In the domain of graphical art printing, there is a possibility of this feature being incorporated in the future. At a conceptual level, the adapter software and the generic protocol must be able to handle multiple finishers. Figure 2.1.2 shows an example case where three finishers are stacked one behind another in the order of their

numbering. The adapter can communicate with all the finishers independently and there is a communication link between the immediate finishers. The details regarding the support for the multiple finishers will be explained in chapter 3 and chapter 4.

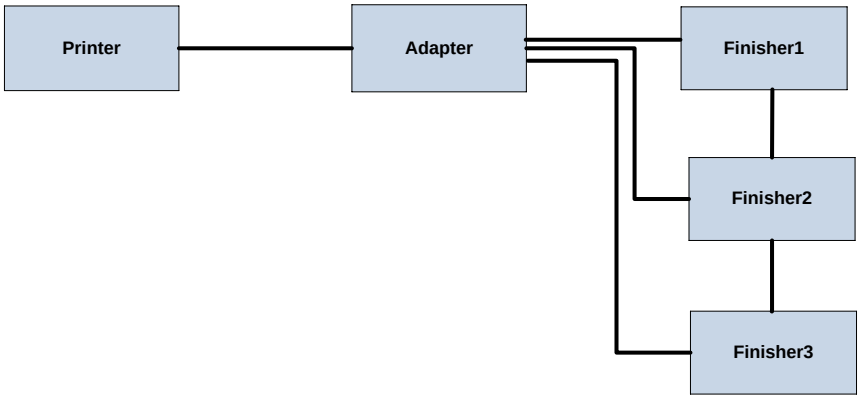


Figure 2.1.2: Multiple finishers feature

Power Cycling: This is a concept to save energy consumption in electrical devices. It is the concept of changing power states based on the usage of the device in order to conserve power. Power cycling can be done both for the finisher and the adapter. The scenarios when the finisher/adapter must be power cycled must be identified in order to power cycle the finisher/adapter. The generic protocol and the adapter software must ensure that this feature can be included in the future.

2.2 Finisher Protocol Analysis

This section explains different message formats, protocol details like electrical interface, message categories and expected message conversion complexities for the adapter software.

2.2.1 Message Formats

There are two different message types available in the finisher interface protocols and they are explained as follows:

Logical messages: Raw functional interface messages independent of the electrical channel related information. This terminology will be used in the rest of the document. Appendix A: “Message list for various finishers” provides a complete list of logical messages with parameters and their range for different finishers.

Low level messages: Functional interface messages with framing details specific to the electrical communication channel.

The logical level message format used in the analyzed finishers is shown below:

$$\{Identifier, Data\ length, Data\ array\}$$

The identifier represents the type of the message, direction of the message and expected functionality. There are two types of messages possible: event messages and data messages. Event messages contain only an identifier and they initiate an action in the receiver. Data messages contain data along with the identifier. The data array has a maximum range beyond which the data length parameter's value cannot be set.

Examples of different low level message formats are shown below:

ID	Size	Data 1	Data 2	Data 3	…	Data n

Figure 2.2.1: Low level message format type 1

Len	ID	Size	Data 1	Data 2	Data 3	…	Data n	CRC

Figure 2.2.2: Low level message format type 2

A low level message format is shown in Figure 2.2.1. The first parameter – ID corresponds to the identifier in the logical message format, the second parameter – size corresponds to the data length in the logical message format and the data block corresponds to the data array in the logical message format. Another low level message format is shown in Figure 2.2.2. The first parameter – Len represents length of the entire message and this parameter is included since the size of a CRC varies from one message to another. The last parameter is the cyclic redundancy checks (CRC) which is a check sum used to verify the integrity of the message. The rest of the message parameters are same as in the low level message format type 1.

From the message format analysis of finishers, it can be seen that there are two levels of message conversion needed in the adapter software. First level, conversion of logical messages transferred between the finisher and the printer. Second level, conversion of low level messages transferred between the printer and the finisher. These two messages have to be dealt differently in the adapter software. The number of logical messages used in the protocol ranges from 25 to 50. The number of arguments in logical level in each message varies from 0 to 8. The one of the focus of the thesis is to work on the logical message conversion in the adapter software.

2.2.2 Protocol Details

This section explains several aspects of the finisher protocols: electrical interface, maintenance of connectivity, paper handling procedure, error handling procedure and incomplete dynamic specification case.

Electrical interface: The finisher interface protocol specification contains electrical and functional details. There is a dedicated communication channel between the printer and the finisher. The current finishers communicate using RS-232, RS-232 variants, serial connector and parallel connector. The baud rate, number of data bits, parity bit availability, stop bit value and other parameters depend on the type of the connector used. The communication links are expected to

work without disturbances and there is no retry mechanism is available in the communication protocol (low level protocol specific to the connector RS-232).

Maintenance of connectivity: The communication request can be started by the printer or finisher depending on the protocol. Once the link has been established between the printer and the finisher, the maintenance of the link would be done by using one of the two methods mentioned below:

- Link check messages: These messages are transferred continuously between the printer and the finisher in order to ensure the working of the link. Once the link is broken, it can be understood from the non-arrival of these messages.
- Timeout for every reply message: For every message sent from the printer and finisher, a timer is started and the reply is expected to arrive within a certain time period. If the timer expires, then the link is assumed to be broken.

Both the methods have their advantages and disadvantages. In the link check method, there would be overcrowding of messages in the channel during peak traffic time. But the loss of link can be detected immediately once the link has been broken. In the timeout method, the loss of link can be detected only after a message has been sent but there would be no overcrowding of messages in the channel.

Paper handling: Every sheet of paper is identified by a unique identifier present in the message. There are two methods to handle paper related messages in the finisher protocols**.**

Parallel paper task handling: Figure 2.2.3 illustrates the parallel paper task handling feature in finishers. The finishers are informed in advance about the sheets with their finishing procedure that they are going to handle before the sheets actually arrive.

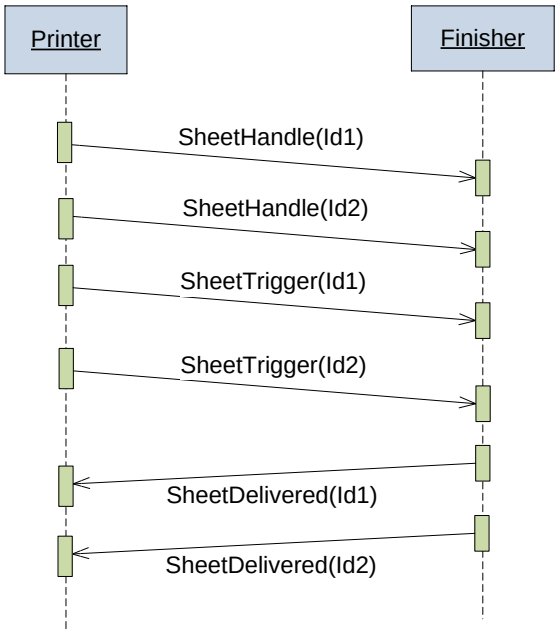


Figure 2.2.3: Parallel paper task handling

There is a maximum limit on the number of such handle messages that can be sent to the finisher before the sheets are delivered by the finisher. The actual arrival of sheets is indicated by the printer through a sheet trigger message. Several such triggers can be sent for which the handle message has been sent already. The finisher replies to the printer once the sheet has been delivered.

Sequential paper task handling: Some finishers do not have the above feature. Figure 2.2.4 illustrates the sequential paper task handling feature in such finishers. They expect the sheet trigger message about the arrival of a sheet immediately after the handle message (indicating the finishing procedure). The finisher completes the particular sheet and sends delivered message back to the printer. Then, finisher gets ready to receive another handle message from the printer.

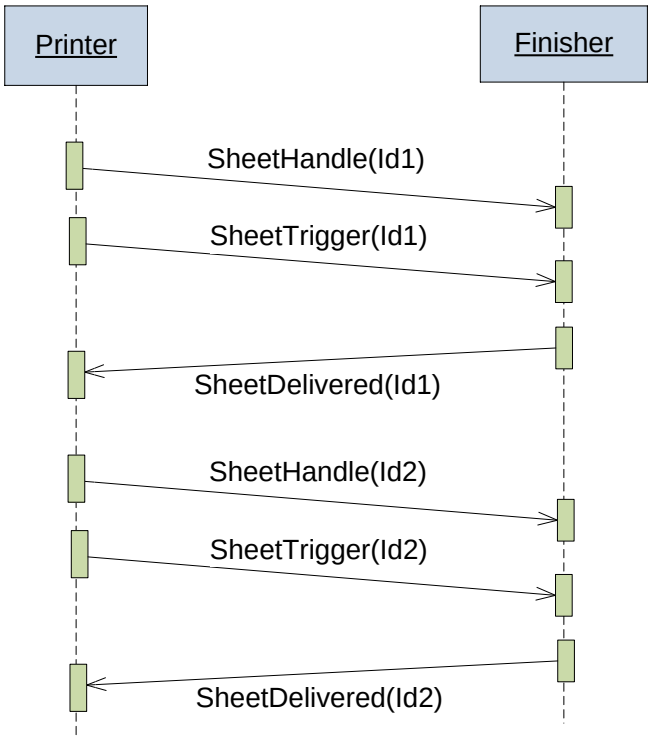


Figure 2.2.4: Sequential paper task handling

Error handling: In case of communication hardware errors (framing, parity bit violation), size check failures and identifier failures, the finisher reports a communication error and tries to re-establish the connection. The range checks for incoming messages are performed in the functional interface software and if there are interface violations then these errors are reported to the printer. Errors related to a paper jam are reported by the finisher when it occurs, the procedure to resolve the paper jam is sent to the printer, the operator gets notification about the error from the printer and finally, when the paper jam gets cleared it is sent to the printer. The errors related to the paper jam in the printer are sent to the finisher.

Figure 2.2.5 shows an example error handling procedure followed in case of a paper jam. The finisher tells the printer to inform the operator that an error has occurred and the type of the error

is a paper jam. Then, the finisher informs the printer to open the door1. Once the jammed paper has been removed from the finisher by the operator, the finisher informs the printer to close the door1. Finally, the finisher informs that the error has been cleared.

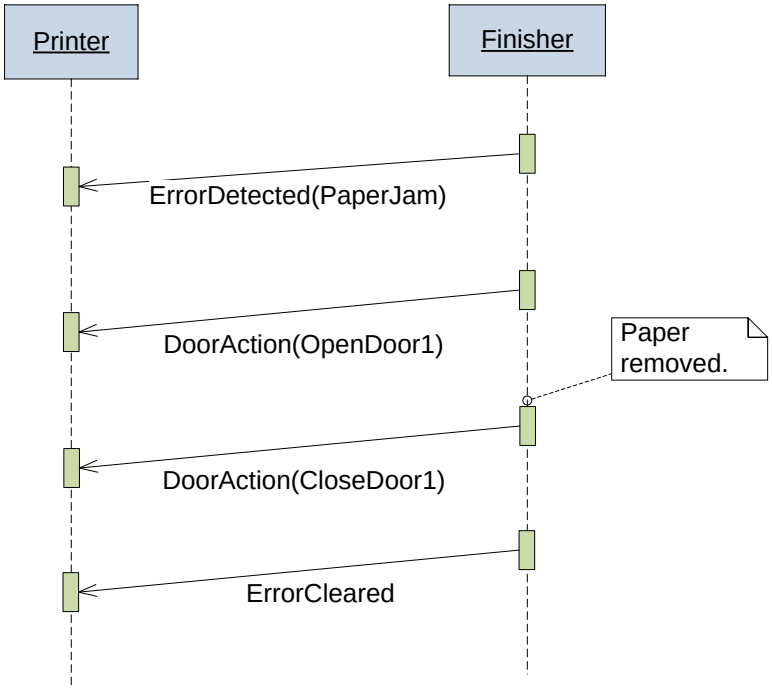


Figure 2.2.5: Error handling procedure for a paper jam

Incomplete dynamic specification: A finisher which has to be integrated with an Océ printer via adapter in the future is a finisher from another company. The dynamic behavior of its interface was not captured completely in the protocol specification, and logging of messages is not possible from the finisher side. There is a difference in working of the finisher with the serial interface and parallel interface. The adapter must be able to incorporate a parallel interface if required. This shows the necessity to design the adapter software keeping in mind that the finisher protocol specification is incomplete in nature.

2.2.3 Message Categories

The message category refers to the purpose of a message. Each message will fit into one message category. The rationale behind fitting each message to one category is that the isolation of scenarios becomes simpler providing separation of concerns. The categories are framed based on the activities happening in the finisher and they are inclusive of the expected future capabilities. The message category list is shown below:

- **Link handling:** This refers to the messages responsible for establishing connection, maintaining the connection and termination of the connection between the printer and the finisher.

- **Paper handling:** This refers to the messages responsible for fetching details of the finisher, configuring finishing properties for every paper and sending triggers for arrival of jobs to the finisher.
- **Operational state management:** This refers to the messages responsible for the maintenance of operational states of the finisher. For example, the online state where the printer can send finishing jobs and the offline state where the operator has direct control over the finisher and the finisher are decoupled from the printer.
- **Error handling:** This refers to the messages responsible for reporting errors both from the finisher and the printer and messages sent to resolve these errors.
- **Power state management:** This refers to the messages responsible for the maintenance of the power states in the finisher. These messages can be used to get/set the power state of the finisher and messages sent to initiate the sending of power signals to start the finisher.
- **Diagnostics management:** This refers to the messages responsible for testing performed by a service engineer to test the behavior of a finisher. These messages are sent from the printer and the reply is sent from the finisher.
- **Software download management:** This refers to the messages that are sent to update the firmware of the finisher and adapter.
- **Multiple finishers' mode management:** This refers to the messages that are used to configure the printer and finishers to operate in multiple finishers' mode.

2.2.4 Message Conversion Complexities

The finisher protocol reveals the expected complexities in message conversion of the adapter software. The exact complexity can be understood only after designing the generic protocol and mapping it to specific finisher protocols. The complexity classification was done to understand different complexities possible in the message conversion of the adapter software. The three different complexities expected in the adapter software are the following:

- **Data**: This type of complexity is the complexity involved in the parameter or data conversion from generic protocol messages to specific protocol messages and vice-versa. This will not be complicated and the reason is that the parameters in the finisher protocol messages include commands, status information, and configuration information. The parameters do not include information with a large amount of data, like for example a JPEG image. There is no interrelation between parameters present in several messages for the same message type, for example video streaming.
- **Control**: This type of complexity is the complexity involved in the number and type of messages during message conversion from generic protocol messages to specific protocol messages and vice-versa. The challenge in designing the adapter software lies mostly in the control part. For example, a specific combination of messages from the printer to the adapter may be converted to another combination for a finisher. The most complex scenario would be identified after the specification of the generic protocol.

- **Timing**: This type of complexity is the complexity involved in the adapter software in handling the timing requirements from both the software modules for message conversions. The timing related messages usually occur not in the happy case scenarios for finisher protocols. Hence, the normal operation of the adapter will not require much timing information make it simple. The timing information has to be incorporated as timeouts to handle unexpected behaviors in the protocol for example loss of a communication link.

2.3 Cut-sheet Printer Interface Solutions

The cut-sheet printer environment comes under the category of a related domain for wide-format printers. The printer-finisher interface implementation in the cut-sheet printer environment was investigated to find suitable techniques that can be applied in the wide format environment. The important fact is that the concept of an interface adapter is well established in the cut-sheet environment. The solution is not applicable straightaway since the finisher capabilities, electrical interface and the functional interface protocols are different.

Common interface protocol: One of the techniques used in the cut-sheet environment related to the interface adapter is the usage of the common interface protocol. The protocol is used for the communication interface specification of the printer with other devices. If some external device is not following the protocol but needs to communicate with the printer, then the device is connected to the interface adapter. The printer addresses the device as a normal device following the common interface protocol. The printer's commands will be sent to the interface adapter and the adapter translates the message and then sends it to the external device. An example of the common interface protocol usage is shown in Figure 2.3.1.

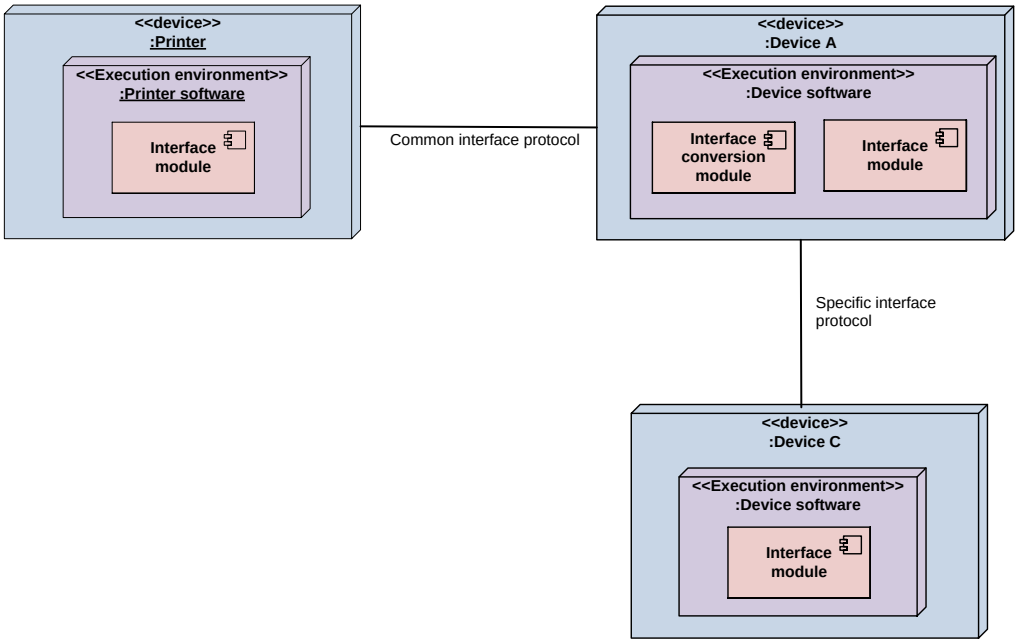


Figure 2.3.1: Common interface protocol

The printer can communicate with device A and device C directly using the interface protocol. There is a specific protocol interface between device A and device C. The messages sent to the device C are received by device A and converted before reaching the destination. In this way, the printer can communicate with the device C as if the device was following the common interface protocol. The example explained here is a proxy pattern, a software design pattern.

Logical message format: The common interface protocol uses a similar type of logical message format as that of the wide format finisher protocols. An example of the logical message format used in the common interface protocol is shown below:

$$\{Priority, Direction, Sub-protocols, Command, Mode, Node\ Id, Data\ array\}$$

'Priority' tells whether the message has high or low priority, 'Direction' tells whether the message is from the printer to the finisher or vice-versa, 'Sub-protocols' tells the message category of the message and this is applicable only in application mode, 'Command' is the identifier used to map the functionality of the message, 'Mode' describes whether the device operation is in download or application mode, 'Node Id' describes the node identity of the printer or the finisher, 'Data Array' is the fixed data array which is the data transferred from printer to devices and vice-versa. The concept of 'Node-Id' is to incorporate multiple finishers.

Message Categories: These helps to identify messages for the generic protocol since the common interface protocol use similar type of messages. An example message category list in the cut-sheet environment is shown below:

- System: Messages to set up and maintain communication with the finisher.
- Status: Messages related to controlling the operational status of the finisher.
- Action: Messages related to the handling of sheets. This does not include the sheet trigger messages.
- Trigger: Indicates the (non-) arrival of sheets. These messages are used for timing of sheets.
- Error: Messages related to reporting and handling of errors.
- Information: Messages related to information exchange between the printer and the finisher.
- Diagnostics: Messages related to the execution of diagnostic tests.
- Development Support: Messages to support development and analysis.

Working of multiple finishers: The cut-sheet printers handle multiple finishers and the common interface protocol can support them. Figure 2.3.2 shows the paper handling procedure in the cut-sheet environment with multiple finishers. There are two types of messages: a 'prepare' message to inform the finisher about the type of action required on a specified sheet and a 'trigger' message to inform the finisher about the actual arrival of a sheet. Prepare message is sent from the printer to all the finishers informing the finishing details and the trigger message is sent only to the immediate node in the paper path. Here, the paper path is follows: printer, finisher1, finisher2 and finisher3. Hence, the finishers have to implement part of the common interface protocol related to the 'trigger' message.

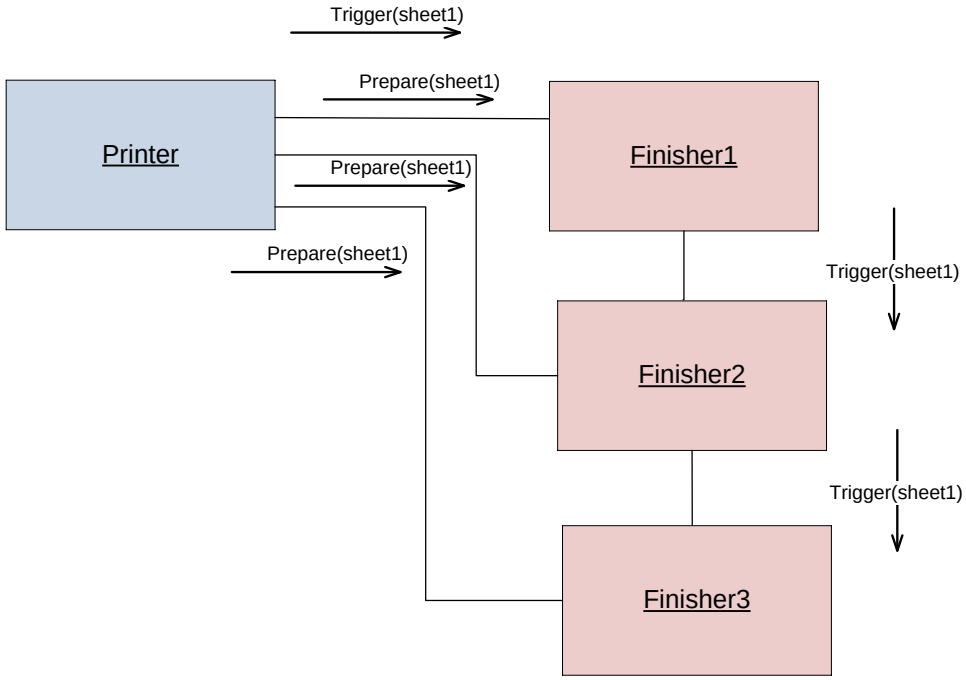


Figure 2.3.2: Paper handling procedure in cut-sheet environment with multiple finishers

2.4 Conclusion

In this chapter, a domain analysis was performed to understand the finisher capabilities and different finisher protocols. The printer-finisher behavior reflects a master and slave relationship. The expected future capabilities of the wide format finishers include software download, multiple finishers and power cycling. The finisher protocols comprise mostly of a query and reply based messages. The message conversion of the adapter will be limited to logical messages of finishers to test the feasibility of the adapter design. The number of logical messages in the protocol is limited to around 50 and the number of parameters in each message is limited to around 8. The finisher protocol messages can be grouped into eight different categories. This shows that the generic protocol messages will be grouped into these eight categories. The message conversion complexity for the adapter is expected to be in the control part. This chapter also discusses an example interface adapter solution in the cut-sheet printer environment. The common interface protocol is used by the printer to communicate with different devices. The implication is that the common interface control can be used as a reference for designing the generic protocol in the wide format environment. The multiple finishers' support using the common interface protocol would help in framing a working procedure for wide format printers.

Chapter 3

Architecture

This chapter explains different architectural views of the proposed adapter. The different architectural views (naming and scope) were framed based on the architecture specification procedure followed in Océ. The first step was to identify different stakeholders of the project and the project requirements which are explained in section 3.1. The next step was to create different architectural views in order to map different requirements. The conceptual view of the adapter is explained in section 3.1. Section 3.2 explains the logical view. The component view of the adapter is explained in section 3.5. The deployment view of the adapter is described in section 3.4. The boundaries of the adapter architecture are explained in section 3.6. Section 3.7 concludes the chapter with important details from different sections. The specification of the architecture is not done in an extensive manner since the idea is to identify the different software blocks and their responsibilities. This activity can be taken up as future work.

3.1 Requirement Analysis

The requirements are classified into four categories: functional, platform, process and non-functional requirements. The important requirements are functional and process requirements since the idea is to test the feasibility of the adapter software. The MoSCoW[1] prioritization scheme was followed in order to assign priorities on the requirements.

Main Stakeholders:

- *Software Architects (SA)* are interested in the architecture, the modeling approach applied for designing the software and the business aspects of the adapter.

Note: MoSCoW[1] "is a prioritization technique used in business analysis and software development to reach a common understanding with stakeholders on the importance they place on the delivery of each requirement - also known as MoSCoW prioritization or MoSCoW analysis" (see http://en.wikipedia.org/wiki/MoSCoW_Method)

- *Print System Architects (PSA)* whose focus points are the product evolution and the roadmap of printers are interested in the impact of the adapter on mechanical interfaces.
- *Software integrators (SI)* are interested in the integration aspects of the adapter software with printers and finishers.
- *Software developers (SM)* are interested in the design of the adapter software and modeling tools used in the project.

Functional Requirements:

FR_1 The adapter software must be able to communicate with the printer using the generic protocol [input/output]

FR_2 The adapter software must be able to communicate with different finishers using their specific protocol [input/output]

FR_3 The adapter software must be able to perform message conversions based on the state machine/state machines [processing]

FR_4 The adapter software must be able to handle timing issues for messages based on the state machine/state machines [timing/synchronization]

FR_5 The adapter software should be able to implement power cycling feature in the future [general]

FR_6 The adapter software should be able to incorporate multiple finishers in the future [general]

FR_7 The adapter software should maintain a file regarding the finisher capabilities. The details of the file can be provided online or offline [data]

Platform Requirements:

The prototype board and the test step were chosen to enable rapid prototyping of the adapter software. Hence, the adapter software was implemented in a prototype board with CPU capability which is overkill for running the adapter software. Driver software for electrical interfacing and transport layer stack were provided by Océ. The details of the prototype board are given in the Appendix B: “Océ prototype board specification”.

Process Requirements:

These were the procedures followed like tools, programming languages in realization of the software. The one of the focus point of the thesis is the selection of the modeling approach. Section 4.4 contains the selection criteria for the modeling approach that explain the process requirements in a detailed manner.

Non-functional Requirements:

NFR_1 Maintainability: The adapter software should be able to implement software download feature in the future.

NFR_2 Robustness: The adapter software should be able to implement link check mechanism feature to maintain the link between a printer and the adapter.

NFR_3 Extensibility: The architecture should give guidelines on handling different versions of the adapter software in the adapter and the printer interface software in the printer.

3.2 Conceptual View

This view represents the black box view of the adapter software describing its context and the user's view. Figure 3.2.1 shows the conceptual of the adapter with its interfaces. The adapter interfaces a single printer with one or more finishers.

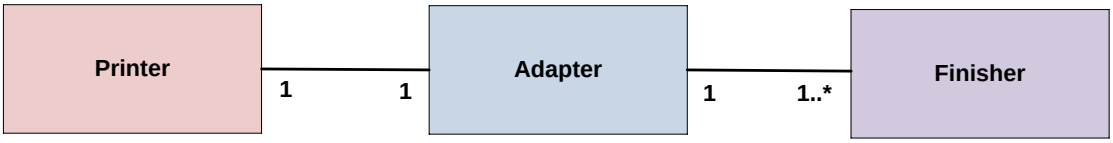


Figure 3.2.1: Conceptual view

The components and their interfaces are defined as follows:

Components:

Printer:

- Printers are Océ wide format printers.
- The printer behaves like master sending instructions to finisher/finishers.

Adapter:

- Adapter should support present and future wide format printers and finishers.
- Adapter converts the generic protocol to the specific protocol of the finisher connected to the adapter and vice-versa.
- As a design constraint, the adapter should not have any functional requirement for printer-finisher operations. The adapter behaves like a transparent device performing message conversions and does not tries to emulate the printer behavior.

Finisher:

- Finishers are Océ developed wide format finishers and wide format finishers from other companies.

Interfaces:

Printer-Adapter: Generic interface between an Océ wide format printer and the adapter.
Adapter-Finisher: Specific interface between the adapter and finisher/finishers.

3.3 Logical View

This view represents the white box view of the adapter software describing the top level design. Figure 3.3.1 shows the different packages with its interfaces and also classes within these packages. The dotted boundary shown in the figure is done to indicate the focus area of the architecture in the thesis.

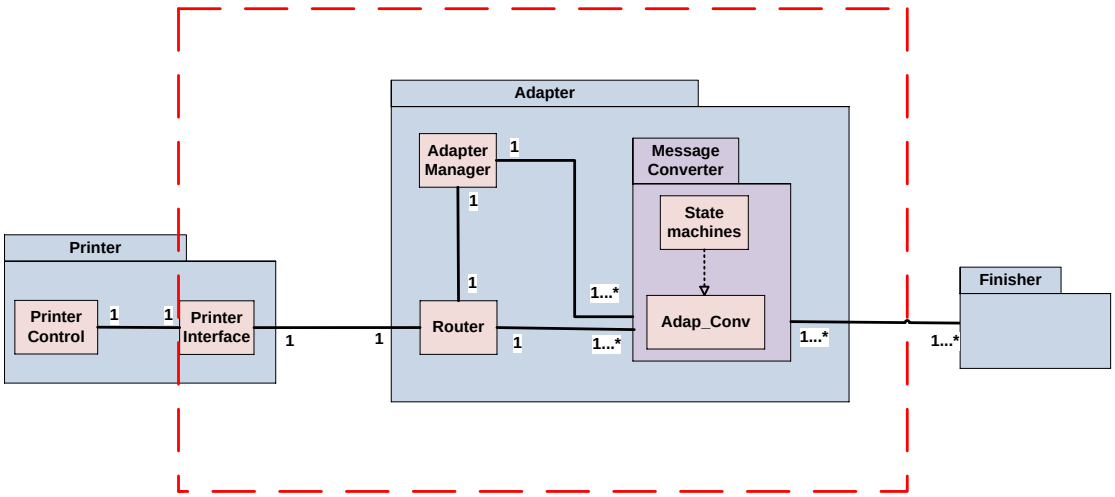


Figure 3.3.1: Logical view

The different blocks: packages and classes are explained below:

Packages/Classes:

Printer Control:

- This module interacts with the printer interface module.
- This module decides a printer action based on the message from the printer interface module. This module acts as a proxy class interfacing printer interface with other modules of the printer.

Printer Interface:

- This module connects the printer with the adapter.
- This module handles exceptions made by the adapter.

Router:

- This module is present to support multiple finishers.
- This module performs routing of messages from the printer to a specific message converter or adapter manager based on the identifier present in the message.

Adapter Manager:

- This module interacts with the adapter hardware
- This module takes care of the non-functional adapter requirements like software download and link maintenance

Message Converter:

- Adap_Conv module converts generic messages from the printer to specific messages of the finisher and vice-versa.
- The conversion is based on the state machine or state machines available
- Adap_Conv module handles exceptions made by the finisher
- Adap_Conv module handles interface versioning
- Each message converter is independent of other message converters

Finisher:

- Finisher is viewed as a black box.

Interfaces:

Printer control-printer interface:

- This interface represents the commands received from the printer control and response send to the printer control.
- This definition is not part of the thesis.

Printer interface-router:

- This interface carries the generic protocol messages which can be for the adapter manager or a message converter

Router-adapter manager:

- This interface carries the software download and link maintenance messages

Message converter-adapter manager:

- This interface represents interaction between the adapter manager and message converters carrying messages to indicate loss of link, start of download or hardware failures

Router-Message converter:

- This interface carries the generic protocol messages for a specific finisher

Message converter-finisher:

- There are one or more interfaces available between the message converter of the adapter and finishers which carries specific protocol messages. In other words, the number of message converters running is same as the number of finishers.

3.4 Deployment View

This view describes the mapping of the adapter implementation on processing nodes. There are two types of deployment views discussed: real deployment view which corresponds to the actual set up and prototype deployment view which corresponds to the prototype set up.

Real deployment view: Figure 3.4.1 shows the real deployment view. The components related to the printer software: printer control, printer interface and printer generic run in the printer hardware. The components related to the adapter: adapter manager, adap_pri, router, message converter and adap_fini run in the adapter hardware. The finisher components run in the finisher hardware. The electrical interface between the printer and the finisher will be USB and the electrical interface between the adapter and the finishers can be any serial communication based electrical media. The printer generic component takes care of marshalling/unmarshalling for the printer with respect to generic messages. The new software modules added in the adapter due to the deployment will be explained in the next section.

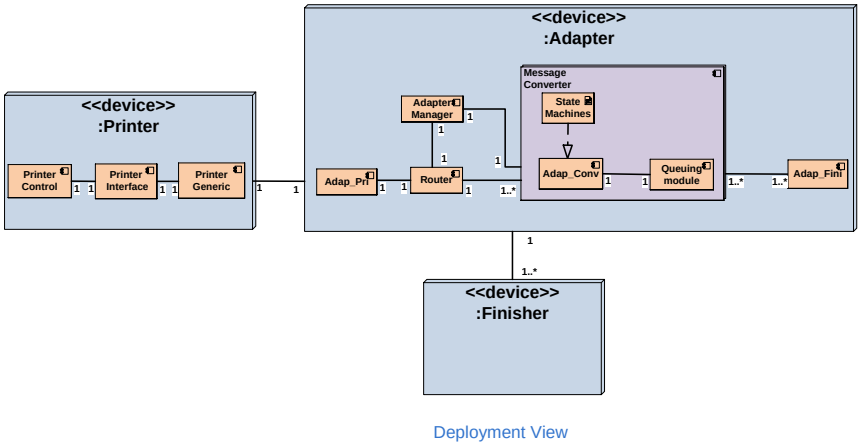


Figure 3.4.1: Real deployment view

Prototype deployment view: Figure 3.4.2 shows the prototype deployment view. The adapter was deployed in a prototype board. The printer and the finisher modules were deployed in a personal computer (PC). The deployment was chosen to ease the process of prototyping and testing. The electrical interface between the PC and the prototype board is USB with support for Ethernet. The exact test up used in the project will be discussed in section 6.1.

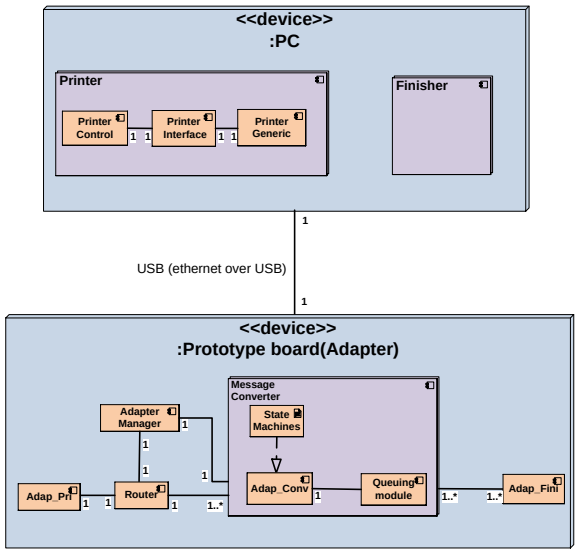


Figure 3.4.2: Prototype deployment view

3.5 Component View

This view describes the adapter software in terms of its implementation components for the real deployment discussed in the previous section. The component view is split into two views: static view which shows the composition of adapter software in terms of compile time organization of its components and dynamic view which shows the composition of adapter software in terms of threads when it is running.

Static view: Figure 3.5.1 shows the component static view of the adapter software. The adapter software executable is composed of the following components: adap_pri, adapter manager, router, message converter and adap_fini.

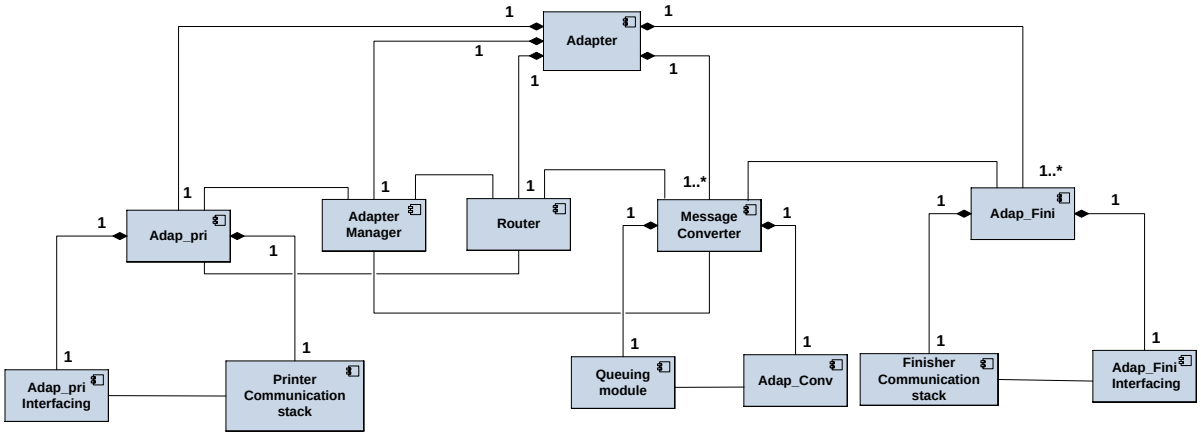


Figure 3.5.1: Component static view

The components which are added due to the deployment are explained below:

Adap_pri:

- This component performs marshalling and unmarshalling of generic protocol messages
- This component is responsible for interfacing the printer with the adapter at a low level
- This component is composed of printer communication stack necessary to perform electrical communication and interfacing module necessary to interact with the router

Adap_fini:

- This component performs marshalling and unmarshalling of specific protocol messages
- This component is responsible for interfacing the specific message converter with the finisher at a low level
- This component is composed of finisher communication stack necessary to perform electrical communication and interfacing module necessary to interact with the message converter

Queuing module: This module takes care of the queuing of messages from the finisher side.

Reason for queuing: The adap_pri and adap_fini components are simple marshalling/unmarshalling components without any queuing. The message converter is expected to handle concurrent messages from both the printer and the finisher. The introduction of queue in the message converter solves the problem of handling concurrent messages. The queue has been introduced in the finisher side which implicitly means that the priority is given to printer messages when there are concurrent messages. Thus, making the message converter to handle messages in a sequential manner.

Alternative architecture: The alternate architecture that has been analyzed was to split the adap_conv into two modules adap_pri_conv and adap_fini_conv running in different threads realized from a common state machine. The adap_pri_conv takes care of converting the generic messages to specific messages. The adap_fini_conv takes of converting the specific messages to generic messages. The rationale behind this approach is to differentiate the two types of message conversions happening inside the adapter. The advantage is that the sequential message converter would be converted to a concurrent one but still there would be synchronization required between these threads. The handling of messages in real time would be more difficult due to this additional synchronization. The actual implementation of this message converter would be more complicated since the effort required would be twice the effort needed for the previous adapter during specification. Hence, the previous adapter architecture has been chosen for the design.

Dynamic view: Figure 3.5.2 shows the component dynamic view of the adapter. The adapter process is composed of the following threads: adapter manager, router, message converter, finisher queuing, adap_fini and adap_pri. The individual threads are explained below:

- Adapter manager: Thread responsible for handling jobs related to the hardware, link handling procedure with the printer and adapter software download
- Router: Thread responsible for handling routing of generic messages
- Message converter: Thread responsible for handling generic protocol to specific protocol message conversion and vice-versa
- Finisher queuing: Thread responsible for handling queuing of finisher messages before sending it to the message converter
- Adap_pri: Thread which listens for incoming messages from the printer and performing unmarshalling of these messages
- Adap_fini: Thread which listens for incoming messages from the finisher and performing unmarshalling of these messages

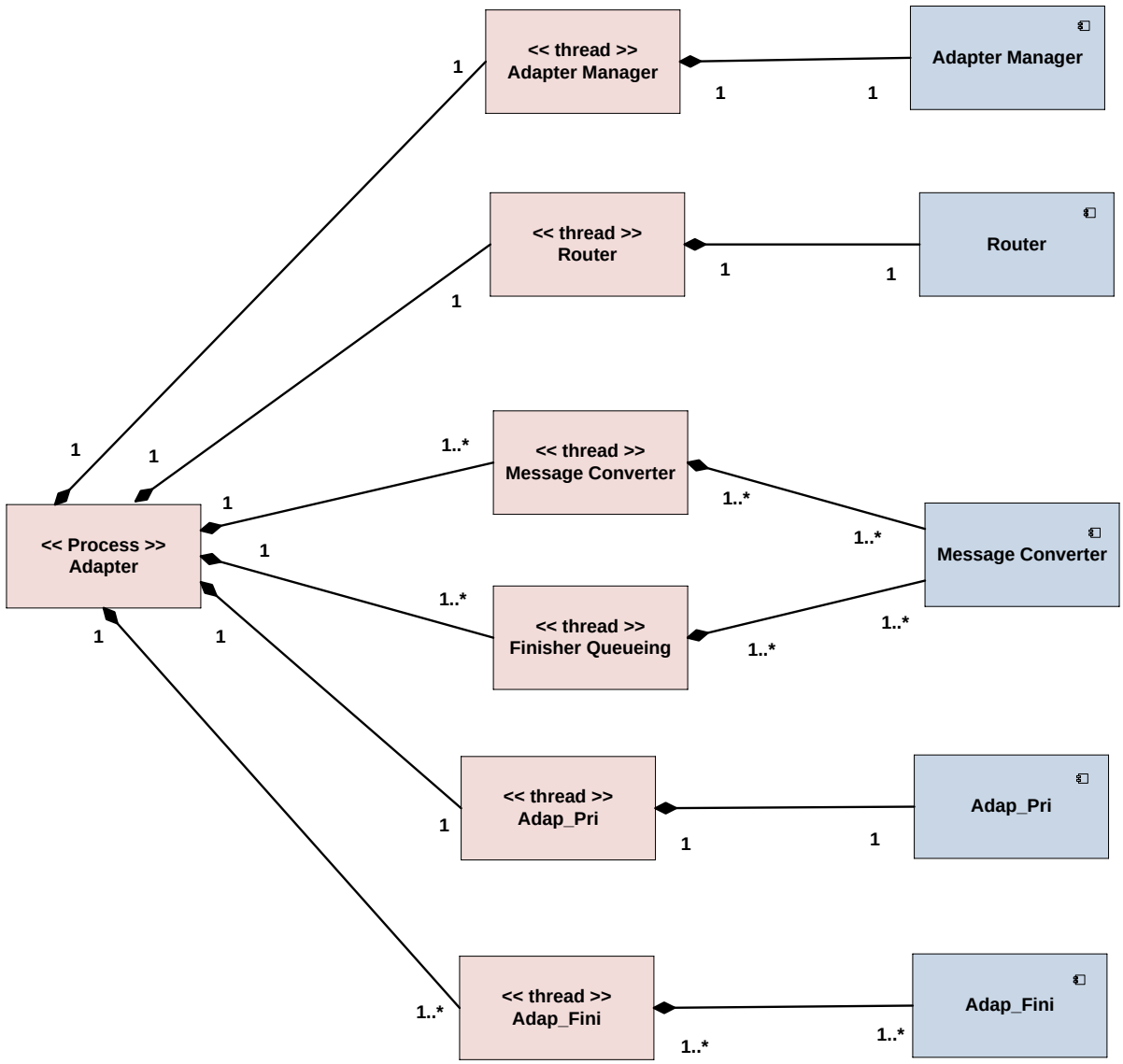


Figure 3.5.2: Component dynamic view

3.6 Boundaries of the Architecture

The boundaries of the architecture represent the scenarios for which the architecture is applicable for the adapter software. The various scenarios are listed below:

- The basic premise behind the architecture is that the printer-finisher relationship is similar to master and slave relationship. The decision on queuing at the finisher side of the adapter is based on this premise. When there is a concurrent message at the adapter from both the printer and the finisher, the priority is given to handling the printer message and the finisher message has to wait in the queue. If this master-slave relationship is changed then the decision on queuing has to be checked again for the new system.

- The realization of state machines to adap_conv component can be done at time of compilation or execution. This is a design choice which can be taken based on the constraints of the design tool.
- The selection of a single state machine or set of state machines for the realization of adap_conv module is a design choice.
- There is a clear separation between message converter and other components. Only the message converter and adapter manager is intended to be generated by a modeling approach.
- The current architecture considers the adapter as a separate entity (hardware and software) receiving commands from the printer and the finisher. If the adapter has to become part of another device, then separate feasibility analysis has to be carried out to verify the validness of the architecture. For example, if the adapter has to interact with a controller of mechanical paper flow device between the printer and a finisher.

3.7 Conclusion

This chapter describes different architectural views for the adapter based on the requirements of the adapter software. First, the main stakeholders of the project were identified and the requirements were framed. The functional and the process requirements are important to test the feasibility of the adapter software. The architectural views give stage by stage transformations, starting from a conceptual model to the final deployment of the adapter software. The initial framing of the adapter architecture in the project was important because the modeling approach fit can be checked with the architecture. Otherwise, there might be a tendency to frame the architecture based on the modeling approach which might result in less efficient architecture with poor separation of concerns. This fit was considered as a selection criteria in section 4.4 before selecting a modeling tool. The one of the scope of the project involves realization of a message converter using a suitable modeling approach, the low level marshalling or unmarshalling components can be realized using pre-existing communication stack to ease the prototyping procedure which will be discussed in section 6.3.

Chapter 4

Generic Protocol and Modeling Approach Selection

This chapter discusses the generic protocol and modeling approach selection. These belong to the generic protocol phase and modeling approach selection phase of the project respectively. Section 4.1 describes a typical adapter scenario for the adapter. Section 4.2 briefly explains the concepts involved in the generic protocol specification. Section 4.3 lists the tools investigated for modeling the adapter. The selection criteria used to evaluate various modeling tools are explained in section 4.4. Section 4.5 explains the modeling selection chart used to choose the modeling approach for designing the adapter software. Section 4.6 gives the conclusions derived from the chapter and recommendations to Océ regarding modeling tools.

4.1 Typical Adapter Scenario

Section 2.2.4 explains the complexities expected in the adapter software. In order to explore the modeling tools and perform case studies, a concrete example is required involving these complexities. The typical scenario includes a combination of all the complexities expected in the adapter. The adapter is expected to handle several such scenarios in reality. The typical adapter scenario designed is shown in Figure 4.1.1. The printer sends a $X(i)$ message; the adapter converts the message to $A(i+1)$ and sends it to the finisher. The finisher replies with message B and the adapter sends a $C(i)$ message and the finisher replies with a D message. Finally, the adapter converts D to Y and sends Y to the printer.

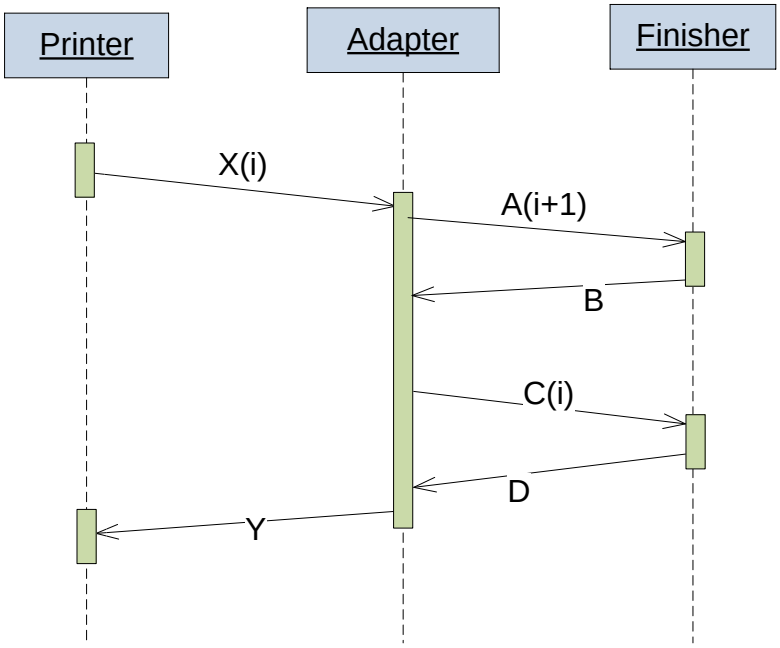


Figure 4.1.1: Typical adapter scenario

State machines: Figure 4.1.2 shows the state machines for the typical adapter scenario given above. The center state machine is the design state machine for the adapter, the left state machine is the interface state machine corresponding to the printer-adapter interface and the right state machine is the interface state machine corresponding to the adapter-finisher interface. These state machines completely describe the typical adapter scenario for the adapter and the above sequence diagram is a trace of these state machines.

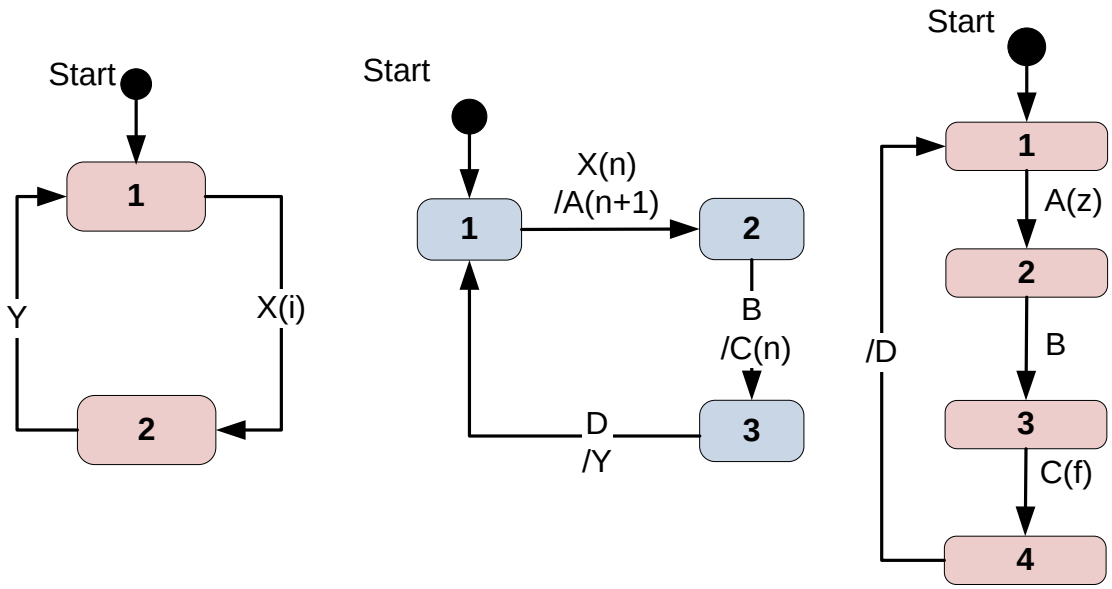


Figure 4.1.2: State machines for the typical adapter scenario

Message complexities: The control complexity is the transformation from a simple $X(i)$, Y message pair to $X(i)$, $A, B, C(i)$, D, Y messages. The data complexity is the parameter transformation

happening in the message $X(i)$ to $A(i+1)$. There is also memory required in the adapter for remembering the value of i. The timing complexity is not included in this scenario but can be included by incorporating timeouts for each reply message. For example, if the intended reply is not obtained in the specified time interval then an alternate message will be sent.

4.2 Generic Protocol

In this section, the proposed generic protocol is explained briefly. The complete details of this protocol are available in the Appendix C: "Generic protocol for adapter interface". The message categories for the generic protocol are same as the finisher protocol message categories defined in section 2.2.3. The protocol specification contains list of messages under each category, message parameter details, timeout behavior of messages, sequence diagrams capturing various scenarios, sequence diagrams showing mapping of the generic protocol with specific finisher protocols and state machines required for the implementation. The sequence diagrams and state machines that are described for the complete system involving the printer interface, adapter and the finisher interface.

4.2.1 Domain Concepts

This section explains the domain concepts involved in the protocol. This comprises node identifiers present in the message, protocol state naming and its context, timeout behavior for messages, delay calculation carried out for timeouts, design decision on clocks and starting procedure involved in some finishers. These concepts explain the top level design decisions taken in the protocol.

Node_Id: In the multiple finisher's environment there is more than one finisher connected to the adapter. The finishers are usually connected back to back with one another. The adapter has to interface the printer with all the finishers. Node_Id (present in all messages) uniquely identifies a device in the multiple finishers' environment. The value starts from 0 which represents the adapter and it is incremented by 1 for every finisher addition.

Consider a paper flow path given in order $\{printer, finisher1, finisher2, and\ finisher3\}$. The electrical connectivity is as follows: the printer is connected to the adapter, the adapter is connected to all the finishers, finisher1 is connected to finisher2 and finisher2 is connected to finisher3. 'Node_Id=0' represents the adapter, 'Node_Id=1' represents the 'finisher1', 'Node_Id=2' represents the 'finisher2' and 'Node_Id=3' represents the 'finisher3'.

Protocol States: The generic interface protocol has to be implemented as separate machines in the printer and the adapter. States were divided into two categories: stable states and transition states. Stable state is one in which the protocol remains unless there is a trigger received from external actors like printer control module or adapter manager module. Transition states are temporary states when moving from one stable state to another state. States can be also composite states. Every state corresponds to a mode in the printer or adapter. There are three modes in the printer or adapter: 'APPLICATION' mode in which normal messages related to finishing are transferred,

'DOWNLOAD' mode in which the adapters' firmware is updated and 'F_DOWNLOAD' in which the finisher's firmware is updated.

The stable states were divided into operational states and power states which are explained below:

Operational states**:** There are eleven operational states in the proposed state machine of the printer/adapter.

- Started: In this state, printer/adapter has been powered ON.
- Idle: In this state, connection has been established between the printer and the adapter.
- Link check: This state is a parallel state to all other operational states except the 'started' state. This state is responsible for the maintenance of the link between the printer and the adapter.
- Connected: In this state, a connection has been established between the printer, adapter and finisher.
- Standby: In this state, the adapter is waiting for commands from the printer and it is ready to move to other states like 'online' and 'offline'.
- Online: In this state, the finisher can receive commands for paper handling.
- Offline: In this state, the finisher is in manual mode executing instructions directly from the operator. The connectivity between the adapter and the finisher remains.
- Error: This state is entered whenever an error occurs in the adapter or printer or finisher and it has been reported to the printer. Once the error has been cleared, the printer and the adapter returns back to their original state.
- Download: The firmware of the adapter is updated in this state.
- F_Download: The firmware of the finisher is updated in this state.
- Multi-Mode: The connection topology for multiple finishers' environment is set in this state.

Power States**:** There are two power states in the proposed state machine of the printer/adapter.

- Normal Power: In this state, the device operates in the normal power.
- Low Power: In this state, the device enters low power mode in order to conserve power using the concept of power cycling.

Message Timeout Behavior: After sending/receiving some messages, a timer is started. Timeout refers to a specified time period that will be allowed to elapse in a device (printer or adapter) after which a specified action takes place. The action can be cancelled by occurrence of another event before the timer expiry. There are some terms involved in specifying the timeout behavior which is listed below:

- Start trigger: The sending or reception of this message starts the timer.
- Time period: This represents the time duration after which some action (state change or sending a message) will happen due to an expiry of the timer.
- Stop event: This is an incoming message which stops the timer expiry caused by the start trigger.

- Elapsed action: This represents a state change or sending of messages when the time period value of the timer has elapsed.

Delay Calculation: Timeout for various messages have to adhere to the delay introduced by the transmission media and marshalling/unmarshalling software components. Suppose, the printer sends a message to the adapter and expects a reply in T1 sec then the adapter has to reply back within T1-delay sec as shown in Figure 4.2.1. This scenario is equally valid when the adapter sends a message to the printer. This delay from now on will be called as system_delay and is calculated by the formula given below. The formula has two times the sum of various delays considering the delays incurred in both directions. The unit of system_delay is milliseconds.

$$system\_delay = 2 * (max(marshalling\ delay) + max(unmarshalling\ delay) + max(transmission\ delay))$$

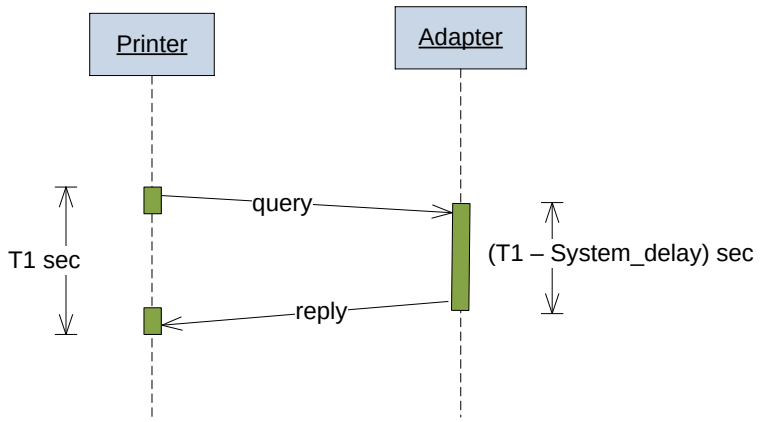


Figure 4.2.1: Impact of system_delay in timeout calculation

Design decision on clocks: The clocks present in the printer and the adapter will not be synchronized. The rationale behind this decision was to avoid the additional complexity of synchronization thereby making the design simple. Moreover, there is no rationale to synchronize both of these clocks.

Starting finishers: Usually, finishers are started without any signaling from the printer. But for some finishers there is a requirement for power signaling from the printer in order to turn on the relay present in the finisher. The procedure employed was to send power signals from the adapter to finishers based on the detection of a hardware link. This procedure was specified as part of the protocol in the power handling message category.

4.2.2 Analysis of Complexities in Message Conversion

The two message categories link handling and paper handling have been defined completely with state machines in the protocol specification. The other categories like error handling, operational state management and multiple finishers' mode management were partially defined in the protocol specification. The rest of the protocol definition can be taken up as future work. The reasons for the selection of categories are as follows:

- The link handling category state machine contains several state transitions and transitions to states responsible for other categories like software download, involves parallel states like link check, interaction with other actors like adapter manager and printer control, contains control complexity in message conversions, contains time complexity for unexpected failure scenarios, and contains limited data complexity. This category is the basic category which needs to work in the adapter before moving to other categories in real-time.
- The paper handling category represents the most important feature in the finisher's functionality. This category enables the finisher to accept sheets from the printer and perform finishing operations on them. The state machine implementation poses challenges in the handling of multiple instances based on message parameters and contains a higher level of data complexity than other categories.
- The other categories primarily involves state machine specification with control complexity, restricted number of transitions to other states, timing complexity for unexpected scenarios and limited data complexity.

These two categories will ensure that the finisher can perform paper related operations provided there are no errors. These categories include most of the complex scenarios possible in the adapter operation and would ensure the feasibility of the generic protocol. Thus, these two categories were selected for complete specification for verifying the feasibility of the generic protocol.

4.3 Tools Investigated

This section explains the different modeling tools explored in the project. There was an initial scan made in the project and the following six tools were identified: Petri nets, ASD, Event-B, Rational Rose – Real Time, interface language and specification description language (SDL). The SDL tool was not considered further after the scan since the approach was very domain specific. The other tools were evaluated based on selection criteria. There was a small case study done in Petri nets and ASD to understand them in detail.

4.3.1 Petri nets

This modeling approach [1] involves specification (drawing) of Petri nets in order to synthesize an adapter (Petri nets). This approach has been applied for real world problems with good results [8]. The specification involves the following steps:

- Behavioral interface models: Petri net models of software modules that needs to be integrated
- Transformation rules: Elementary operations that the adapter can perform. This is the high level message conversion specification.
- Behavioral property: Properties (like absence of deadlocks) that are required in the integrated system.

Tool chain: Figure 4.3.1 shows the tool chain used in the Petri nets approach.

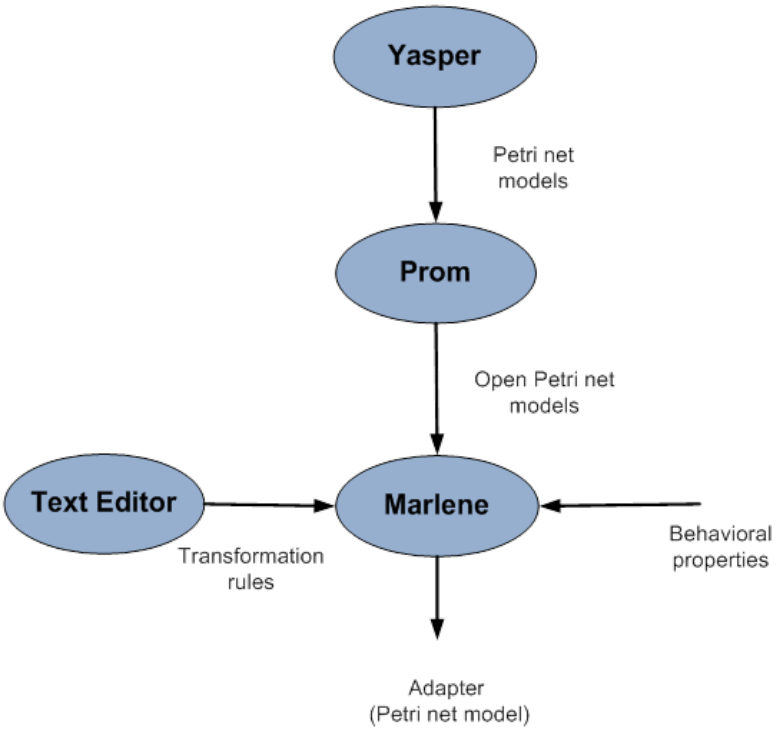


Figure 4.3.1: Tool chain

The steps involved in the adapter synthesis using the tool chain are explained below:

- Specify the Petri models to be integrated using Yasper [11] tool.
- Convert them to open Petri net models [13] [14] (an extension of the Petri net model) using Prom tool [12].
- Specify the transformation rules using a text editor
- Open Petri net models, transformation rules and behavioral properties are the inputs needed for the Marlene tool [9] to generate an adapter which is again an open Petri model
- The Marlene tool can be specified to generate either a synchronous or an asynchronous adapter

Architecture of the Marlene tool: Figure 4.3.2 shows the architecture [9] of Marlene tool which is the core adapter. Paper [1] introduces a modeling approach using the Marlene tool. The modeling approach comprises of front ends which are hand written modules which can perform low level message transformations and a core adapter. The core adapter comprises of an engine and a controller [2] [3]. The engine converts the transformation rules and the controller satisfies the expected behavioral properties. The controller synthesis is carried out using a tool called 'WENDY' [10]. The engine and the controller together constitute the adapter in the tool. The procedure followed in the adapter (Petri net model) synthesis using this tool: a Petri model for the engine was generated for the adapter and a Petri net model for the controller was generated for the adapter.

The final adapter is again a Petri model obtained by combining these two Petri net models. The tool and the model it produces have the same components (controller and engine).

This architecture resembles the proposed architecture of the adapter shown in section 3.4. The front ends can be modeled as adap_pri and adap_fini and the core adapter can be modeled as adap_conv module.

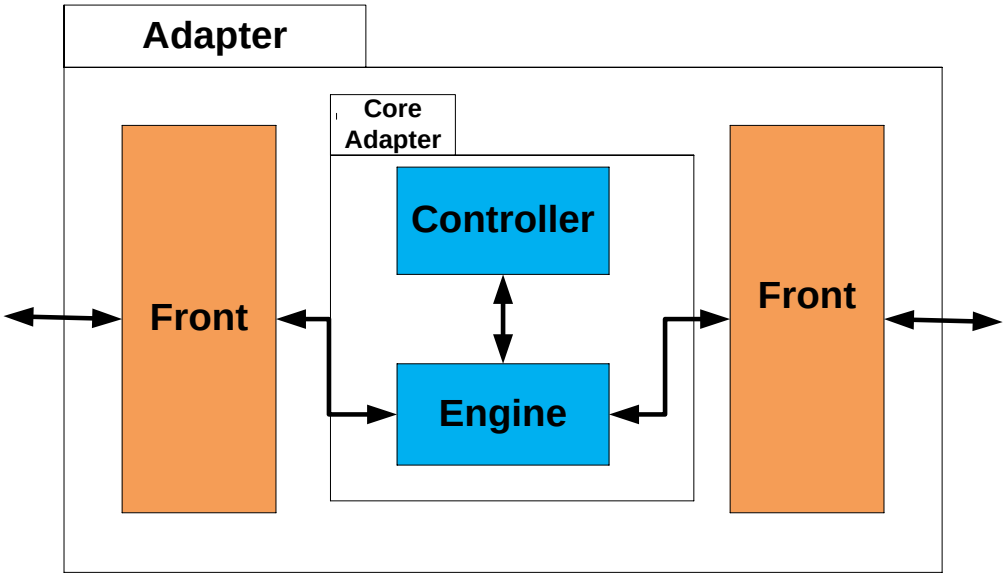


Figure 4.3.2: Architecture of the tool

Related work: In the past, there was an internal case study done at Océ using 'MARLENE' to generate an adapter between printers and finishers based on Petri nets. Two types of incompatibilities (optional acknowledgement message from the finisher and reset/cancel request handling messages) have been considered in this case study. There were issues related to timing and multiple pages which resulted in no or unstable adapters from the tool. From the analysis, it was observed that the practical usage of the tool seems to be limited due to the above mentioned reasons for this specific case study.

Case study: The case study performed was to synthesize the adapter (adap_conv) for a typical adapter scenario given in section 4.1 without the parameter conversions in order to simplify the case study. The case study was not a direct evaluation using the tool but understanding the behavior of the tool. The case study results were produced using a drawing tool. The behavioral input models for the printer and the finisher and the synthesized adapter are shown in Figure 4.3.3. The behavioral models were specified using Petri nets for the printer and the finisher modules. The behavioral property for the adapter is the absence of deadlocks. The intermediate open net models were not included in Figure 4.3.3 and only the final model of the adapter adap_conv is shown. The transformation rules specified for the adapter were the following:

$$X \rightarrow A, B \rightarrow C, D \rightarrow Y$$

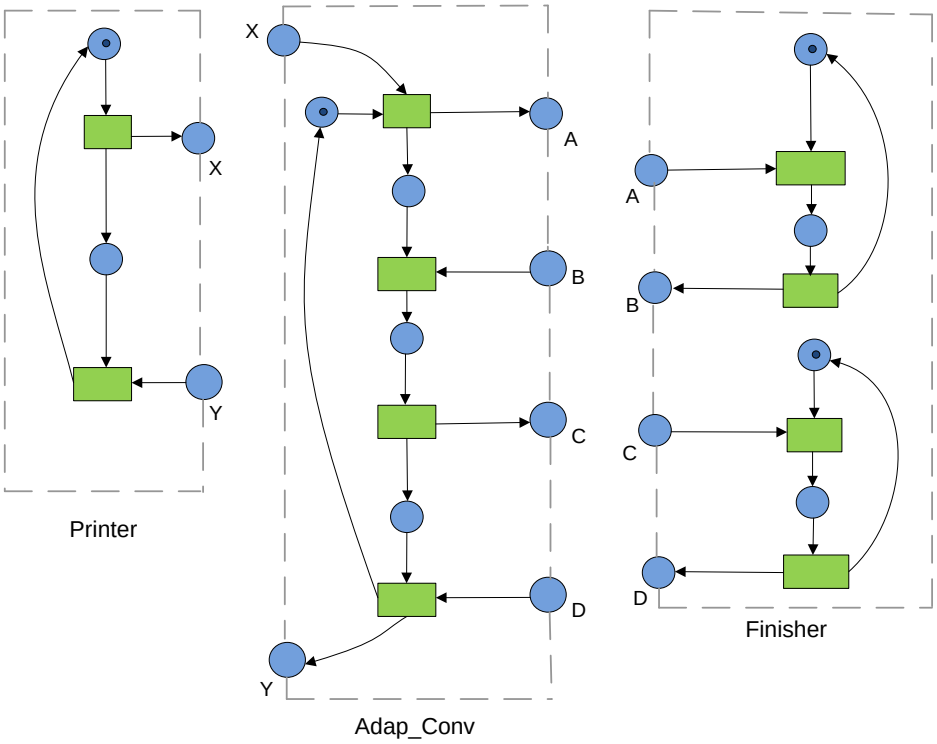


Figure 4.3.3: Behavioral input models and the synthesized adap_conv module

In this modeling approach, interface messages are places in the boundary. The Petri net contains places which are represented by circles and transitions which are represented by rectangles. The arrows denote the pre-condition or post-conditions for the transitions. There are tokens which can occupy a place. A token moves from one place to the other due to the trigger of a transition. A transition is triggered whenever there is a token in each of these places with an incoming arrow and no token in all the places with an outgoing arrow. The result is that the token is moved to places with all the outgoing arrows from the transition.

4.3.2 ASD

The Analytical Software Design [5] software abbreviated as 'ASD: Suite' is a design tool from Verum Software Technologies. The suite enables the software designer to draw system level state diagrams and verify the consistency between these diagrams using formal method analysis. The ASD top level functionality can be understood from the Figure 4.3.4. The ideas behind the functionality diagram are that the model verification is performed through iterations and the generated code and the model are equivalent as claimed by ASD. The ASD allows design errors to be corrected in an iterative manner. Once the design has been found to be error free according to ASD, the code can be generated using the ASD suite. The paper [6] argues that the ASD reduces design errors but the tool has performance issues. The paper has proposed a discrete-event simulator for the performance evaluation of the ASD like structured software. The verification engine used by ASD is FDR2 (Failures-Divergences Refinement) which is a refinement checking software tool [15].

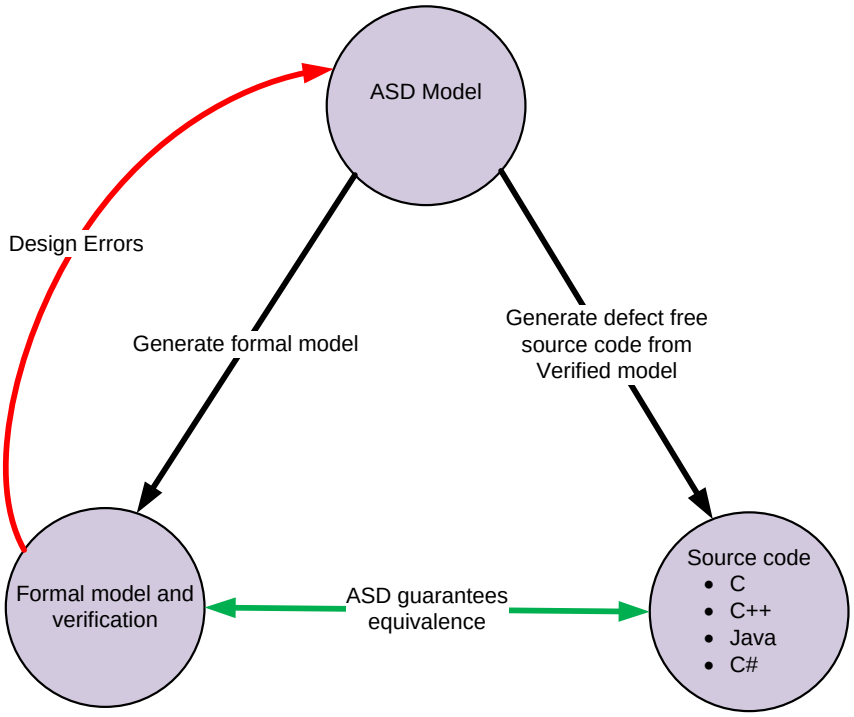


Figure 4.3.4: ASD functionality

The ASD tool comprises two components: a front end component (GUI provided by ASD) and a back end component. The input specification for the software design is done from the front end component which runs on the Windows platform. The verification of the design and code generation is done in the back end servers connected via VPN to the frontend. The design is specified through a front end interface called 'ModelBuilder' using two model types: interface model and design model. The models are basically state machines of the software represented in the form of a table in separate files. The design model is analogous to a software component with internal behavior and the interface model is analogous to an interface definition between different software components. The interface model takes care of the coupling between two design models and coupling between a design model and a foreign component. The detailed explanation of the two model types and its usage will be discussed in section 5.1.

The basic verification that the tool performs is to check whether a design model and its associated interface model are consistent with one another. The verification is done separately for an interface model. For a design model, it is checked how the design model fits with other interface models. The verification is an iterative process where the designer tries to correct the error scenarios by updating the model. The tool aids in this aspect by providing information highlighting the cause of the error using sequence diagrams. The detailed explanation about the model verification will be discussed in section 5.2.

The code can be generated once the verification has been successful with ASD. It is possible to generate code in the following programming languages: C, C++, Java, C# and TinyC. There are two thread model types available for generating code: single threaded model and multi-threaded model. In order to interface foreign components with the ASD generated code, there are stubs

which can be generated. These stubs provide an interface to integrate the ASD code with foreign code. The detailed explanation about the model verification will be discussed in section 5.3.

There was a case study done using ASD for the typical adapter scenario shown in 4.1. The case study is not explained here since the section 6.4 contains implementation of three typical adapter scenarios and one of them is the case study. The case study was not a direct evaluation using the tool but understanding the behavior of the tool. This means that the case study results were ASD tables but specified without using ASD.

4.3.3 Event-B

The Event-B approach uses set theory as a modeling notation. This modeling approach uses formal methods for system-level modeling and analysis [16]. This method uses refinement procedure to split the system into different abstraction levels and uses mathematical proof to verify consistency between different refinement levels [16]. Rodin platform [20] which is based on eclipse is the IDE used for the toolset implementation and various tools can be incorporated as plug-ins in the framework. The toolset aims at developing a model based software from capturing software requirements till the implementation of the software. Pro-B is a model checker and animator that can be used with the Rodin tool [19]. There are formal methods based code generation tools that are proposed for this approach [17] [18]. Figure 4.3.5 shows the various plug-ins of Rodin toolset with example tool implementation given in brackets (please refer to [22] for more details regarding individual tools).

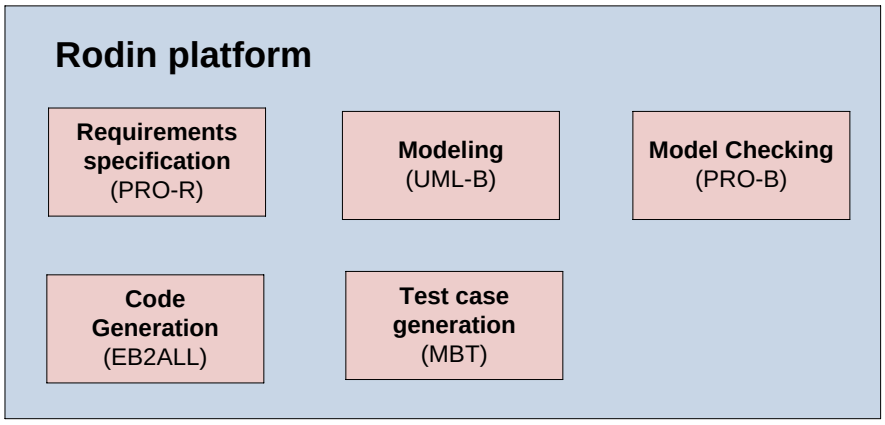


Figure 4.3.5: Event-B toolset

The tool was identified at a later stage in the project and due to time constraints there was no case study performed using this tool.

4.3.4 Rational Rose – Real Time

The Rational Rose-RealTime (RoseRT) is a UML-based computer-aided software engineering (CASE) tool developed by IBM corporation that supports the development of real-time embedded software. It uses real-time object-oriented modeling methodology [21]. Software is built using a

combination of active entities called capsules and passive entities. Capsules communicate with each other by message passing via ports. The behavior of a capsule is modeled by means of a state diagram. Passive entities correspond to regular data classes in object-oriented languages like C++. RoseRT also performs automatic code generation for different target platforms. RoseRT supports code generation for C, C++ and Java programming languages.

The tool is the commonly used software development tool at Océ. There was no case study performed using this tool since the tools' functionality was known and it would not be an interesting option for Océ.

4.3.5 Interface Language

The Interface Language shown in Figure 4.3.6 is an internal tool (prototype) developed at Océ. The tool is used for specifying interface state machines using XML format and generating the code for interface verification. The tool can be fed with sequence diagrams in order to verify their consistency with the state machine. This feature helps in improving the specification of the state machine. If the generated state machine is integrated with the interface implementation environment, then the tool can perform runtime verification. The runtime verification represents identification of the type of an error and the module responsible for causing the error. The run time traces can be verified with the interface state machine model. This tool uses IPC mechanism to generate logs which in turn is used for sequence diagrams' verification and runtime verification.

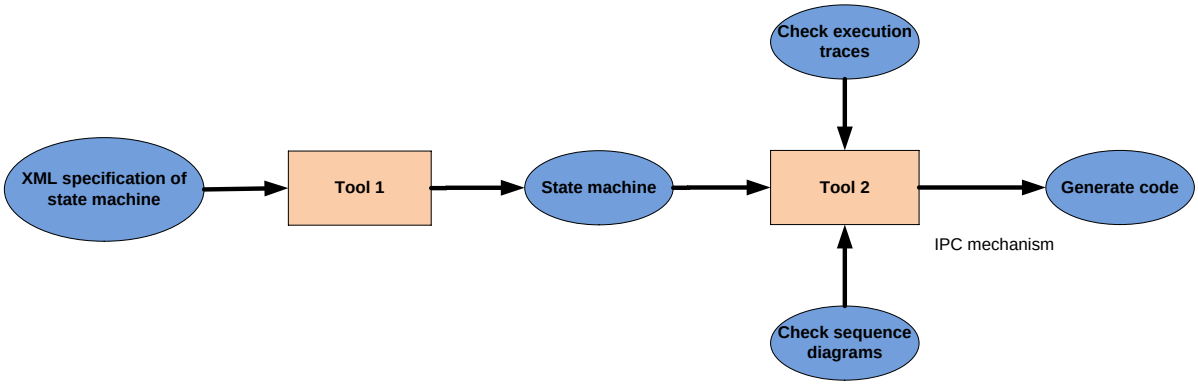


Figure 4.3.6: Interface language functionality

The functionality of the tool is summarized as follows:

- To refine input state machine specification using an iterative procedure for automated generation of class diagrams and state diagrams and sequence diagram verification
- To generate sequence diagrams from a given failure interface runtime trace with indication of the failing transition

The tool cannot be used for designing the adapter software but it can be used for designing the adapter interface protocols. The advantage would be verification of the interface state machines during design time and runtime. This requires an IPC mechanism to be implemented as a

communication stack for interfacing the printer and the adapter. If other communication stack is used, then the integration of this tool with the communication stack is required. Due to the above mentioned reasons, there was no case study done using this tool.

4.3.6 SDL

The approach using specification and description language (SDL) is explained in this section. The paper [4] explains the method to design highly reusable protocol software for software defined radios. The paper identifies the commonalities among different wireless communication systems. A generic protocol software stack was created from the commonalities. Then another software stack called standard-specific supplements was added to incorporate the new system specific features. The software specification was carried out using the SDL. A common state machine was drawn incorporating the common flow and the system specific flow. This method has an advantage of software reusability. But the paper is highly specific to the software defined radios and does not give any recommendations for other systems in identifying the commonalities. Hence, the approach was not considered further and there was no case study performed using this tool.

4.4 Selection Criteria

The selection criteria [27] [28] were used to analyze different modeling tools and to select the modeling tool for designing the adapter software. These were grouped into three categories: model based criteria, quality metrics and engineering methods. An additional category would be business interests of Océ. These categories will give clear focus areas for the selection of a tool. Some of the criteria might belong to more than one category, and then the criteria are placed in the most suitable category. The idea behind the custom tool which is mentioned in the chart is to generate code based on the specification of state machine of the adapter software.

Model based criteria: These denote the modeling concepts supported in various tools.

- Adapter synthesis: This criterion tells whether a tool can synthesize the adapter state machine based on high level specification of the input models with transformation rules. The synthesized adapters' state machine can be directly used for implementation.
- Adapter modeling/System modeling: This criterion tells whether a tool fits the proposed architecture for designing the adapter software. The tool should accept system level state machines or Petri nets or other models as inputs in order to design the adapter.
- Interface modeling: This criterion tells whether a tool can be used to model the interfaces of the adapter. The tool accepts interface state machines or Petri nets or other models as inputs in order to model the adapter interfaces.
- Code generation: This criterion tells whether a tool can generate code from the design specification.
- Model checking: This criterion tells whether a tool can verify system invariants. An invariant is a condition that can be relied upon to be true for a given system. This feature helps to verify the design for unexpected behaviors.

- Test case generation: This criterion tells that the tool can generate test cases based on the system model. The tool should have provisions to update the test cases when the system model is changed.

Quality metrics*:* These determine the quality of the outputs produced by a modeling tool.

- Sequence trace verification: This criterion tells whether a tool has a provision to accept example sequences as inputs and validate them with the global state machine. This feature will help the software designer to test the system boundary conditions.
- Design verification: This criterion tells whether a tool can verify the design for interface violations and runtime inconsistencies like deadlocks and livelocks. This verification helps to identify design errors at an earlier stage in the software life cycle.
- Interface protocol consistency: The interface protocol is implemented as different software components. This criterion tells whether a tool can check the interface protocol implementation for consistency. This will ensure the protocol implementation is without runtime inconsistencies like race conditions.
- Dynamic foreign interface error isolation: This criterion isolates the source of the interface error and the type of protocol violation that has occurred during runtime. For example, if the finisher is from other companies then the interface between the adapter and the finisher is a foreign interface.
- Consistency between design and code: This criterion tells whether a tool can give guarantee that the design and the code are consistent. The code should reflect the design for both static and dynamic specifications.
- Nature of input specification: This criterion explains the nature of the input specification required for the tool. If the criterion is ‘detailed’ then it covers all the dynamic scenarios of the input specification and if the criterion is ‘not detailed’ then it does not cover all the dynamic scenarios of the input specification.
- Counter example generation: This criterion explains whether a tool can generate example scenarios for indicating a design failure with its type and reason for their occurrence.
- Counter example visualization: This criterion explains whether a tool can display counter examples using some visualization technique. This helps the designer to understand the error scenario in a better way.

Engineering methods: These criteria tells whether a tool can fit into the Océ tool chain, cost issues, support for a tool and documentation.

- Documentation: This criterion tells whether a tool has sufficient documentation.
- Development environment: This criterion describes the development environment (tools) used for the specification and execution of the tool.
- Target environment: This criterion describes the target environments (operating system) supported for the generated code.
- Input specification method: This criterion describes the input specification procedure like file format.

- Requirements specification: This criterion tells whether a tool provides support for capturing the requirements. This may help to map the design and requirements in a single tool.
- Tool licensing: This criterion explains whether there is licensing required for the tool and if it is required then the type of licensing necessary.
- Tool availability: This criterion tells whether a tool would be available for the next 3 years. The availability is predicted based on the organization delivering the tool. If the tool is obtained from a University then the availability is marked as 'Not Sure' and if the tool is obtained from Océ or from other companies then the availability is marked as 'Available'.
- Tool support: This criterion tells about the technical support available for the tool. If the item is mentioned as 'Internal' then the tool is developed in Océ. If the item is mentioned as 'External' then the support is available from other company. If the item is mentioned as 'University' then the support is available from University.

4.5 Modeling Selection Chart

Table 4.5.1 shows the modeling selection chart used to compare different modeling tools. The important outcomes of this comparison are the following: custom tool can be made for a specific purpose with code generation feature but it will be of limited usage for future possibilities, interface language tool can be used for the interface verification and not for designing the adapter, Petri nets based tool provide adapter synthesis possibility but the synthesis to code transformation has to be done manually and also the synthesis for this adapter is not needed since the mapping can be done manually and Event-B tool provides several possibilities on the modeling front but the tool needs knowledge of set theory and the tool is still in the phase of development with support from various Universities. Considering the above factors, ASD and Rose can be considered for the adapter implementation. ASD has an edge in the model verification where it verifies detailed dynamic scenarios unlike Rose. Rose has been used in other projects in Océ and testing ASD for the adapter software was interesting to Océ. Thus, ASD was selected for modeling the adapter software.

Selection Criteria Type	**Selection Criteria**	**Petri nets**	**ASD**	**Interface language**	**Rational Rose – RT**	**Custom Tool**	**Event-B toolset**
Model Based Criteria	*Adapter synthesis*	Yes	No	No	No	No	No
	Adapter modeling/ System Modeling	Yes	Yes	No	Yes	Yes	Yes (PRO-B)
	Interface modeling	Yes	Yes	Yes	Yes	No	Yes (PRO-B)
	Code generation	No	Yes (C, C++, C#, Java)	Yes	Yes (Java, C++, C#, Pascal)	Yes	Yes (C, C++, Java And C#) (EB2ALL)
	Model checking	No	No	No	No	No	Yes (PRO-B)
	Test case generation	No	No	No	No	No	Yes (MBT)
Quality	*Sequence trace verification*	No	No	Yes	No	No	Yes (PRO-B)

Metrics	*Design verification*	Yes	Yes	Yes	Yes	No	Yes (PRO-B)
	Interface protocol consistency	Yes	Yes	Yes	No	No	Yes (PRO-B)
	Dynamic foreign interface error isolation	No	No	Yes	No	No	No
	Consistency between design and code	No	Yes	Yes	Yes	No	Yes (EB2ALL)
	Nature of input specification	Detailed	Detailed	Detailed	Not detailed	-	Detailed + Requires knowledge of set theory
	Counter example generation	No	Yes	No	No	No	Yes (MBT)
	Counter example visualization	No	Yes (Sequence diagrams)	No	No	No	Yes (MBT)
Engineering Methods	*Documentation*	Yes	Yes	Yes	Yes	Yes	Yes + not complete
	Development environment	Yasper (Petri net models), Prom (Open Petri net models), Marlene (Adapter generation)	ASD Environment	XML (for input specification), Python (verification tool)	Rose environment	-	Rodin -Eclipse based
	Target environment	-	Windows, LINUX, Vxworks	Windows, LINUX requires IPC	UNIX, Windows NT/2000, and real-time operating system	-	Not sufficient information available
	Input specification method	Input models using Petri nets and transformation rules	State machine in a table format	State machine in a XML format	State machine diagrams	-	System behavior using set theory
	Requirements Specification	No	No	No	No	No	Yes (Pro-R)
	Tool licensing	Not required	Usage based licensing	Not required	Required	Not required	Not required
	Tool availability	Not sure	Available	Available	Available	Available	Not sure
	Tool support	University	External Company (Verum)	Internal	External Company (IBM)	Internal	University + External Company (Formal Mind for Pro-B, Pro-R)

Table 4.5.1: Modeling selection chart

4.6 Conclusion and Recommendations

This chapter explains two important aspects of the thesis: generic protocol and modeling approach selection. The link handling and paper handling categories were selected for the detailed specification since these were identified as the complex message categories. The final outcome of the generic protocol phase was that the design of a generic protocol for wide format finishers is feasible. The analysis of various modeling tools were done to select a tool for designing the adapter software. The selection chart shows comparison of different tools for the selection criteria. ASD tool was chosen for designing the adapter software. There were technical and business reasons behind this selection. One of the reasons behind the selection is that the Océ had interests in understanding the tool. The other reason is that the tool requires complete specification of dynamic scenarios initially during design phase which will result in fewer errors at the time of implementation. The generated software can also be integrated into Océ development environment. The recommendations to Océ are as follows: Event-B in the future may come up with an input specification procedure which may not require knowledge of set theory for input specification and there has been continuous development on various tools used in this toolset. Event-B development looks promising and Océ can keep track on the changes happening in this tool. The integration of interface language with inter-process library can be taken as a separate task within Océ.

Chapter 5

ASD

This chapter explains the software design procedure using ASD and related concepts. Section 5.1 explains the modeling procedure followed in the ASD. Section 5.2 explains the design verification done by ASD. Section 5.3 explains code generation and integration aspects of ASD. Section 5.4 describes the conclusion drawn out from various sections in the chapter. Please read section 4.3.2 before reading this chapter in order to understand this chapter in a better way. For a detailed understanding of the ASD, please refer to ASD documentation [24].

5.1 Modeling using ASD

ASD enables the software designer to draw system level state diagrams and verify the consistency between these state diagrams using mathematical analysis. ASD is a component-based technology encompassing a mixture of ASD components and foreign components [24]. ASD component includes design model and interface model. Foreign components are third party software which needs to be integrated with ASD. An example for ASD components in an adapter is shown in the Figure 5.1.1.

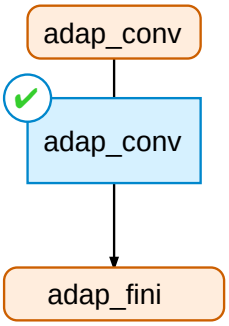


Figure 5.1.1: ASD components of an adapter

This model comprises one design model (adap_conv) at the center and two interface models namely adap_conv and adap_fini at the top and bottom respectively. An interface model is used to capture the external visible behavior of an ASD component. A design model is used to capture the internal behavior of an ASD component. Here, the design model represents message conversion happening in the adapter, adap_conv interface represents the generic interface of the adapter with the printer and adap_fini interface model represents the specific interface of the adapter with a finisher.

Component interactions: An important aspect about ASD modeling is that the ASD model is asymmetric with hierarchical layers. The messages that flow from top layer to bottom layer are executed synchronously and messages that flow from bottom layer to top layer are executed asynchronously (using queue). Figure 5.1.2 shows the interactions between two components where component 1 is at a higher layer and component 2 is at a lower layer [23]. Component 1 can make a synchronous function call to component 2 and get a synchronous reply for that function call. Component 2 can also send messages via queue to component 1. Components at the same level should not interact with each other. This mechanism prevents the occurrence of deadlocks in ASD model. This concept shows there is a difference between interfaces at the top of a design model and interfaces below the design model. The design model implements one interface model at a layer above it and can use several interface models that are at a bottom level. For example, adap_conv design model implements the adap_conv interface model and uses the adap_fini interface model.

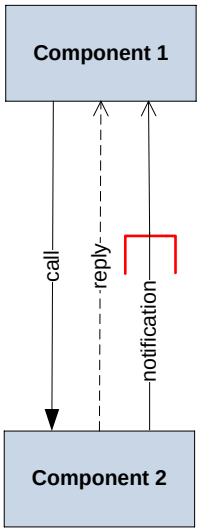
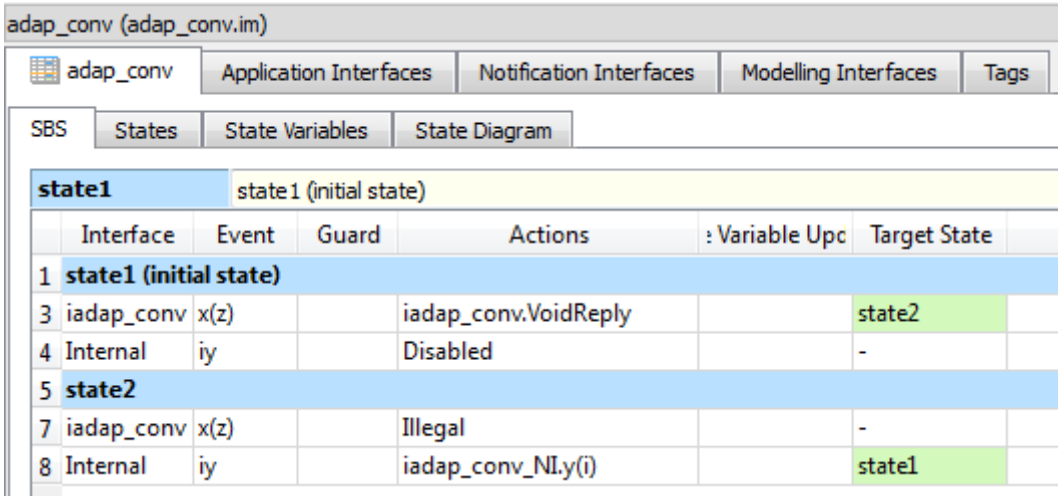


Figure 5.1.2: ASD component interactions

Input specification: The input specification is done through front end software (GUI provided by ASD). Interface models and design models in ASD contain state machines described in a table format. Figure 5.1.3 shows an example of ASD interface model (adap_conv) input specification. Figure 5.1.4 shows an example of ASD design model (adap_conv) input specification table (table is not shown completely). The contents of the ASD table (interface model and design model) are the following: sequence based specifications (SBS), various states in the model, state variables and state

diagram (which is generated by ASD for graphical visualization of the SBS). SBS is a table which is used for defining the state machine of the model. The interface model additionally contains interface definitions. The design model additionally contains the implemented service (interface model) and list of used services (interface models).


adap_conv (adap_conv.im)

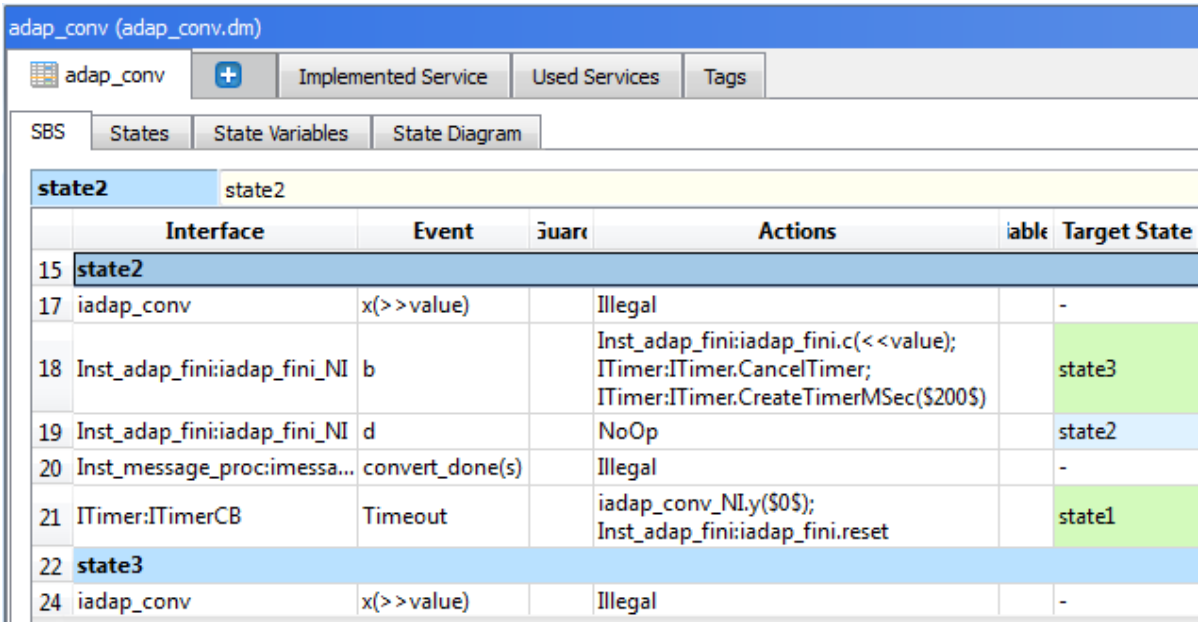
adap_conv | Application Interfaces | Notification Interfaces | Modelling Interfaces | Tags

SBS | States | State Variables | State Diagram

state1 | state1 (initial state)

	Interface	Event	Guard	Actions	: Variable Upc	Target State
1	**state1 (initial state)**					
3	iadap_conv	x(z)		iadap_conv.VoidReply		state2
4	Internal	iy		Disabled		-
5	**state2**					
7	iadap_conv	x(z)		Illegal		-
8	Internal	iy		iadap_conv_NI.y(i)		state1

Figure 5.1.3: Example of an ASD table (interface model)


adap_conv (adap_conv.dm)

adap_conv | Implemented Service | Used Services | Tags

SBS | States | State Variables | State Diagram

state2 | state2

	Interface	Event	Guarc	Actions	iable	Target State
15	**state2**					
17	iadap_conv	x(>>value)		Illegal		-
18	Inst_adap_fini:iadap_fini_NI	b		Inst_adap_fini:iadap_fini.c(<<value); ITimer:ITimer.CancelTimer; ITimer:ITimer.CreateTimerMSec(200)		state3
19	Inst_adap_fini:iadap_fini_NI	d		NoOp		state2
20	Inst_message_proc:imessa...	convert_done(s)		Illegal		-
21	ITimer:ITimerCB	Timeout		iadap_conv_NI.y(0); Inst_adap_fini:iadap_fini.reset		state1
22	**state3**					
24	iadap_conv	x(>>value)		Illegal		-

Figure 5.1.4: Example of an ASD table (design model)

The SBS contains the following fields which have to be filled by a software designer:

Interface: This field shows the interface corresponding to an event in the next column. The interface model and design model both use the interfaces defined in an interface model. There are three types of interfaces that can be used by an interface model: application interface, notification interface and modeling interface. There are two types of interfaces that can be used by a design model: application interface and notification interface. The different interfaces are explained below:

- Application interface: This interface contains client requests and reply events for a component. The client requests are synchronous messages coming from a higher layer to the model and the reply events are synchronous messages sent as a reply to call events.
- Notification interface: This interface contains notification events for a component. The notification events are asynchronous messages coming from a lower layer to the model. These messages are stored in a queue present in design model before reaching the model.
- Modeling interface: This interface contains modeling events for a component. The modeling events are used for model verification and these events do not occur in real-time. The trigger messages coming from lower layers in an interface model are modeled using modeling events.

Event: This field contains the set of incoming events that can occur in a state.

Guard: This field contains a guard expression which must be evaluated as true for an action to happen for a received event. The guard expression can contain only state variables and constants with mathematical operators. The guard field can be empty.

Actions: This field comprises of a list of outgoing actions which can be client requests, notification events and other actions. The other actions include 'illegal' which denotes the event is not allowed, 'NoOp' which denotes that no action will be performed on receiving an event and 'Disabled' which is similar to 'illegal' but it is used for modeling events.

State variable updates: The value of a state variable is changed in this field. This field can be empty.

Target state: This field specifies to which state a transition will take place.

Designing complete software: In order to design the complete software using ASD, the various components present in the software have to be identified first. The various components are ASD components and foreign components. The next step is to create and describe interface models using SBS. The final step is to create and describe design models using SBS. The foreign components can be interfaced with ASD through an interface model. Figure 5.1.5 shows an example of a complete adapter model for the typical adapter scenario described in section 4.1. The top level component acts as a client and uses the services of a lower component since synchronous requests can be made by the top level component. The lower level component acts as a server and provides a service to the top level component since low level components send only reply events and asynchronous notification events. The software used for marshalling/unmarshalling can be interfaced via adap_conv and adap_fini models, the code for message processing can be interfaced via message_proc interface model and ASD timer runtime code can be interfaced with ITimer interface.

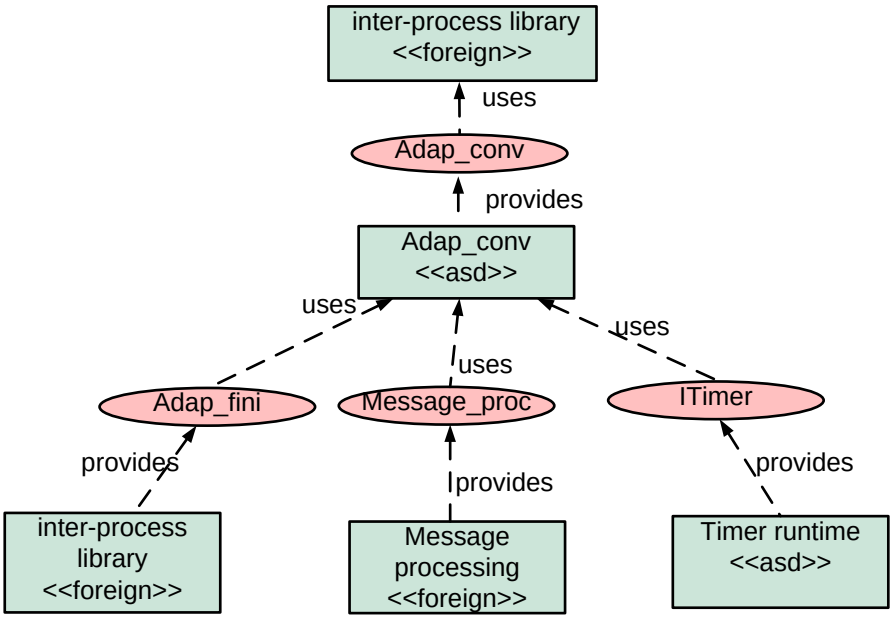


Figure 5.1.5: Example of a complete ASD model of the adapter

Handling of message parameters: Messages comprising of events and actions can have parameters except modeling events. The parameter type, initial value and range are specified based on the programming language for which the code has to be generated. The reply event can have only a single reply value or void reply as parameter. The passing of parameters from events to actions and across rule cases (rule case denotes a single row in SBS) is possible in ASD. In real-time, the passing of parameters across rule cases represent a memory capability of the software. The processing of message parameters cannot be done in design models and it has to be performed in a foreign component. This means that if there are decisions based on message parameters in a state machine, then these decisions are made in foreign components and communicated to a design model as notification events.

Timers: ASD provides built-in timers for interfacing with an ASD model and the timer provided is a one shot timer. The one shot timer executes a timeout only once after the set time period value of the timer has expired. ASD allows usage of multiple timers simultaneously based on the design requirements. Timer events can be incorporated in the ASD model using the built-in ITimer interface. There is a cancel event available in the interface model which cancels the expiry of the timer and ensures that the expiry does not occur after cancellation.

Yoking: This is an approximation made by ASD in order to avoid queue buffer size violations for the queues present in the design model. This refers to restricting the number of notification events that can arrive in a queue. The yoking value is specified per notification event if required and the value should be specified based on the analysis of timing behavior of the system and arrival and service rates of the queue as claimed by ASD. Improper usage of this concept may lead to run-time complications.

5.2 Design Verification using ASD

Verification procedure: The model verification (deadlocks, livelocks and modeling error check) of an interface model is performed independently. For the verification of a design model, ASD first verifies all the associated interface models and then the design is verified. This procedure ensures that a design model is compliant to rules imposed by all the associated interface models. Figure 5.2.1 shows an example of design model (adap_conv) verification checks performed by ASD. Figure 5.2.2 shows an example of interface model (adap_conv) verification checks performed by ASD. If the model verification for the complete software is successful then design models are shown with a green color tick mark in the frontend software otherwise with a red color cross mark in the GUI. The ASD indicates the error scenario using sequence diagrams. Figure 5.2.3 shows an interface violation captured by ASD. If the error is clicked in a sequence diagram then the software shows a corresponding rule-case in SBS. The error is caused since the adap_conv design model did not expect the convert_done message in that state but the message_proc interface model is allowed to send the message in that state. The design model does not follow the conditions set by the interface model. This feature in ASD helps to debug error scenarios in a faster way.

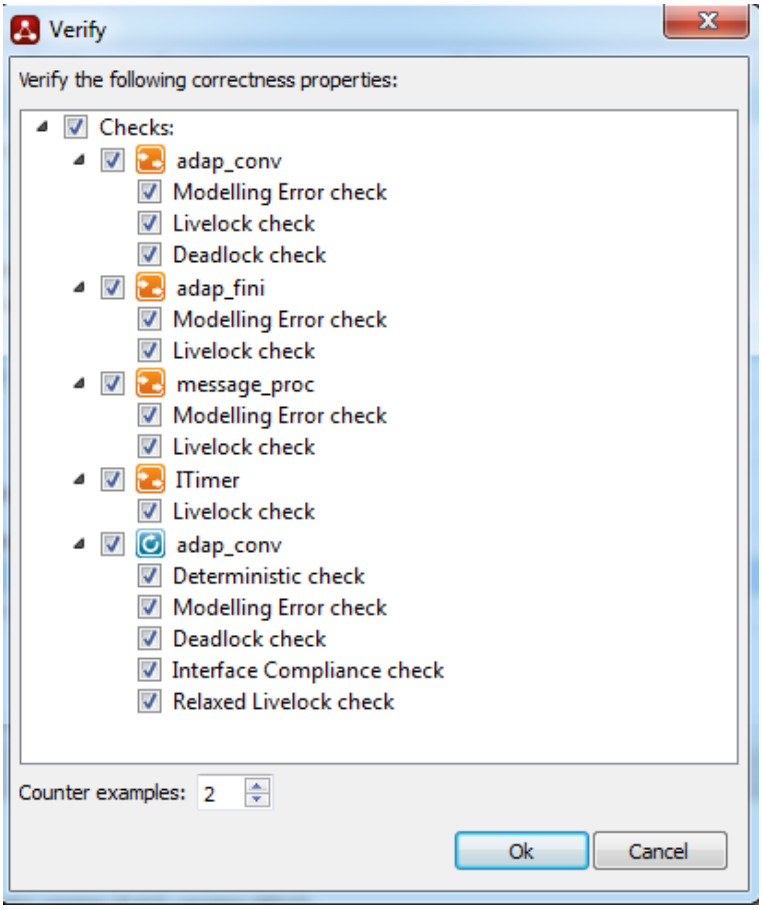


Figure 5.2.1: Example of design model verification using ASD

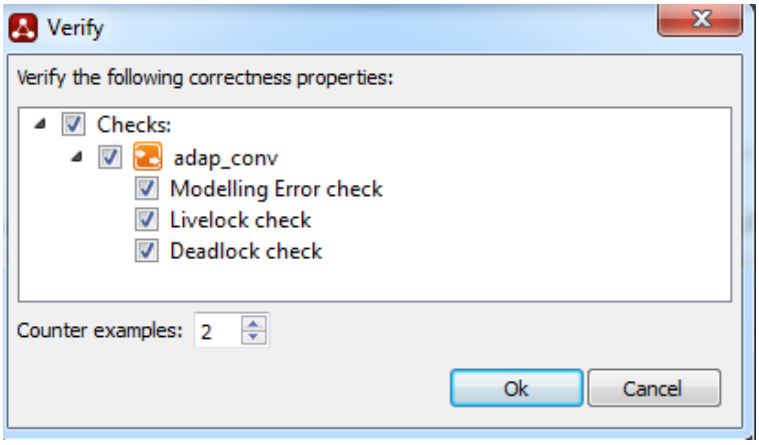


Figure 5.2.2: Example of interface model verification using ASD

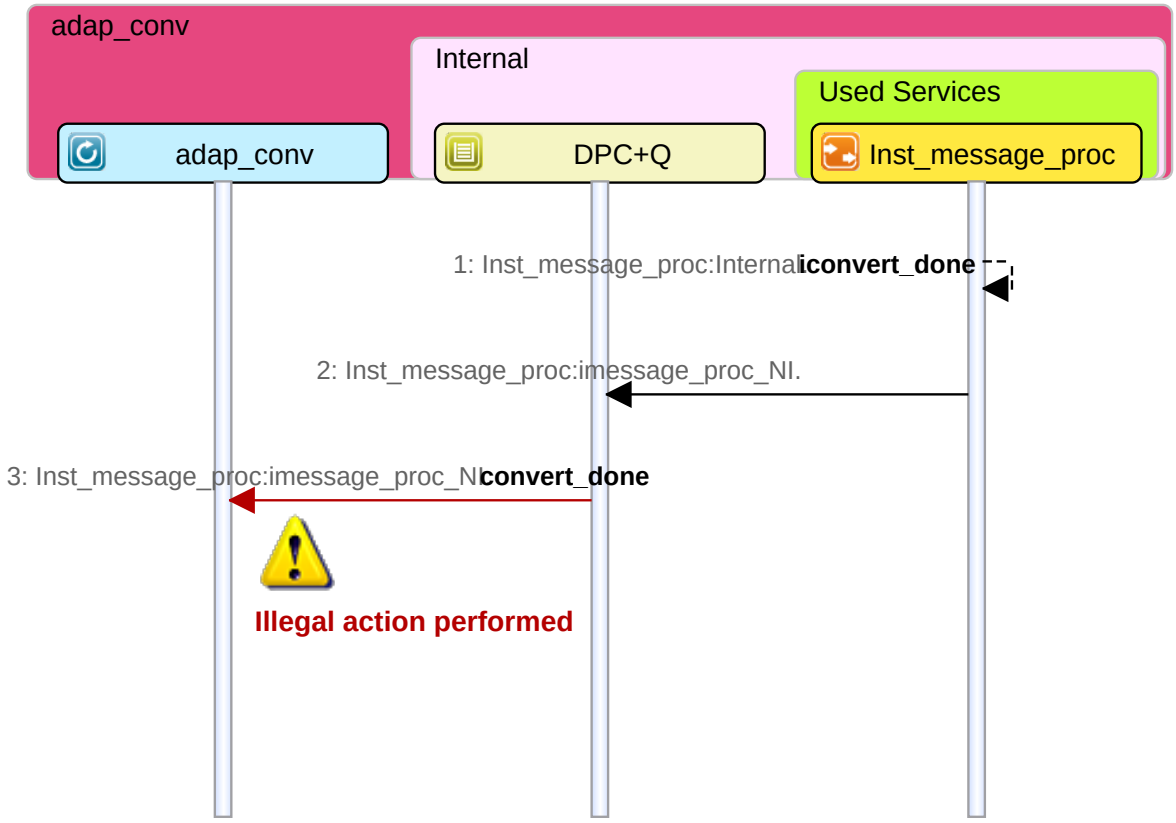


Figure 5.2.3: Example of an interface violation captured by ASD

Types of errors: This describes the different errors identified by the ASD during model verification. ASD does not verify the contents of parameters present in messages and is transparent to them. The types of errors verified by ASD are explained below:

Presence of deadlocks: A deadlock is a condition where both the components interacting with each other do not perform any action since they expect an event from one another. ASD checks for the presence of deadlocks in interface models and design models.

Presence of livelocks: A livelock is a condition where in a component is busy with its internal behavior and does not respond to external triggers. ASD checks for the presence of livelocks in interface models and design models.

Interface compliance: The ASD checks whether a design model complies with all the associated interface models.

The rest of the error verifications can be grouped under modeling error checks:

- **Determinism**: The ASD expects a design model to be deterministic without possible happening of concurrent events by definition. In SBS, only one rule case should be selected every time during state transitions.
- **Guard completeness**: The ASD checks whether all the cases of a guard expression evaluation are present in a state. The guard expressions include arithmetic (addition and subtraction), logical and comparison operators. For example, if a rule-case has $flag > 1$ as guard expression for an event named 'checking' then another rule-case must have $flag \leq 1$ as guard expression in the same state for the same event. This ensures all the alternate flows are captured by the ASD.
- **Range violations**: The ASD expects the designer to specify a range for every state variable used and it checks whether the range of the variable is maintained correctly in the SBS.
- **State invariant violations**: State invariants are expressions that can be used at the beginning of a state and have to be evaluated as true whenever the state occurs in the state machine. The ASD checks this behavior.
- **Queue buffer violations**: The ASD checks that the queue buffer size is not violated due to notification events.

5.3 Code Generation and Integration

This section explains code generation and integration aspects of the ASD. Code should be generated once the design has been verified.

Code generation: Code can be generated for an ASD model in different programming languages like C, C++, Java or TinyC. The ASD generates one header file for every interface model and one header file and one source file for every design model.

Stub generation: Figure 5.3.1 shows an example of stub code generation using ASD. Stubs are skeleton code (source file) which contain routines to interface with a foreign interface (interface model).

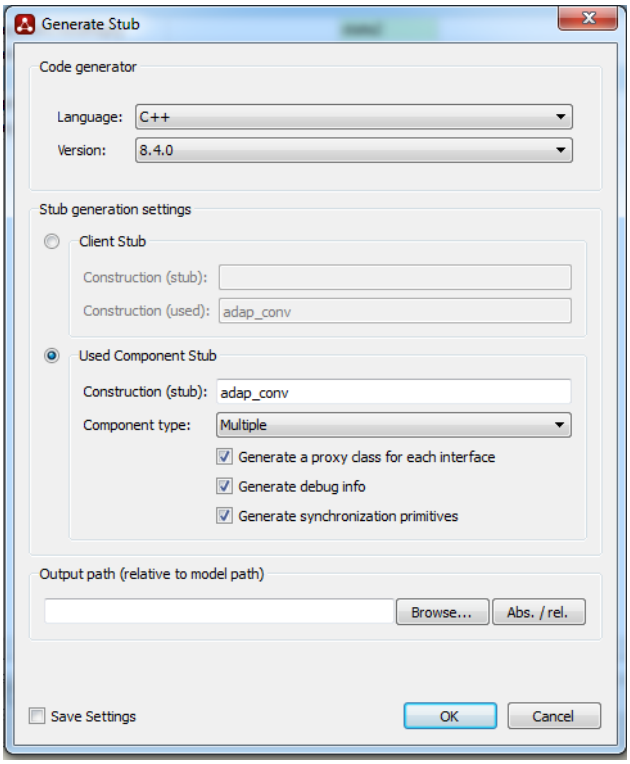


Figure 5.3.1: Example of stub generation using ASD

There are two types of stubs based on the location stub with respect to the interface from which it is generated from: client stubs which are at a higher level to the interface model and used component stubs which are at a lower level to the interface model. In Figure 5.1.5, adap_pri interface model can be interfaced through a client stub with inter-process library and adap_fini interface model can be interfaced through a used component stub with inter-process library.

ITimer integration: The interface model of the ITimer is provided by ASD. But the design model is not available to the designer. The code for the ITimer model (design + interface) can be downloaded directly from ASD.

Compilation: The building of an executable requires the following components: ASD files, foreign component files, runtime files, boost libraries in case of C++ and a thread library. The run-time files are downloaded from ASD which includes ITimer files.

Threads: There are two types of thread models, a multi-threaded model and a single-threaded model which can be specified during the code generation for design models. In the multi-threaded model, there is a separate thread for every design model instance created by a client. The thread implements a deferred procedure call (DPC) server for performing queuing mechanism in order to handle notification events. In the single threaded model, the client thread is reused for handling notification events.

Run-time issues: The ASD expects a foreign component to adhere strictly to the interface definition. If there are run-time violations of the interface then the generated ASD software will crash. The ASD code has an in-built logging facility which can be used for verifying the reason for a crash. If the

foreign is bound to produce interface errors then an ‘armour’ component can be used to protect the generated ASD software as claimed by ASD [25]. The ‘armour’ component acts as a middle layer isolating the foreign component from the generated ASD software. If there are non-compliant interface messages, then these are filtered and logged by the ‘armour’ component and not sent to the generated ASD software. The ‘armour’ component has to be created by the software designer like any other ASD component.

5.4 Conclusion

This chapter contains three aspects of ASD: modeling, verification and integration of the generated code. The important concept to remember is that the ASD model is asymmetric. A modeling of an adapter software using ASD will first involve identifying the hierarchical levels in the software. In adapter model using ASD, printer interface was placed at the top of the message converter and finisher interface was placed at the bottom of the message converter giving priority to printer messages. ASD is transparent to contents of message parameters and does not verify them. The use of foreign components for parameter processing will introduce additional messages in the state machine. The important verification that the ASD performs is to check consistency between design model and its associated interface models. This forces the designer to think of all the dynamic scenarios during the design phase. In the GUI, sequence diagrams are not shown during the normal design but only when there is a verification error which is less helpful for a software designer. The next chapter contains more information on the experience with ASD for scenarios used for prototyping.

Chapter 6

Prototyping

This chapter describes the activities performed in the prototyping phase. Section 6.1 shows the test setup used for testing the prototype. Section 6.2 describes the initial testing performed to understand the prototype board. Section 6.3 describes the testing carried out to understand the adapter architecture with interfacing through inter-process software. Section 6.4 explains the implementation of a typical scenario using ASD. Section 6.5 explains the implementation of link handling section of the generic protocol using ASD.

6.1 Test Setup

Figure 6.1.1 shows the prototype set up used in the project. The adapter software was deployed separately in an existing Océ prototype board. The printer software and finisher software were deployed in a PC. The PC and the board were connected by an USB cable. The idea behind running printer software and finisher software in the same device was to ease the prototyping with respect to networking issues since the finisher is viewed as a black box in reality and the purpose of finisher software inclusion was to test the adapter software.

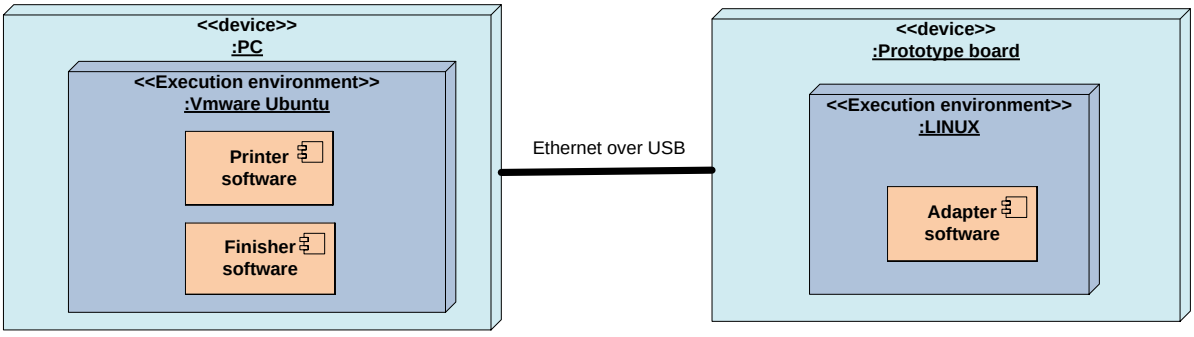


Figure 6.1.1: Prototype test setup

Prototype board: The details of the prototype board are given in Appendix B: “Océ prototype board specification”. The board has an ARM based microprocessor. The board runs a custom version of the LINUX operating system (OS) and the boot loader takes the operating system image from an USB drive connected to the board. The operating system image, other software and other files were placed in a specific directory structure inside the USB flash drive in order to enable the boot loader to understand the structure. Files placed in the USB flash drive can be read during execution. There is a serial connector in the board which enables the programmer to log in to the terminal of the board. The board has another USB slot for connecting a USB cable which was used as the physical communication channel for adapter, printer and finisher.

Virtual machine: The printer software and finisher software were deployed in a Vmware virtual machine installed in the PC. The PC had a Windows 7 operating system and Vmware had an Ubuntu operating system. The idea behind this deployment was that the real deployment will use LINUX OS and the compilation environment for both types of deployment would be similar. Another reason is that there were different software packages (inter-process library, ASD) used in the project and the integration time would be faster if both the deployments use same type of operating system. The finisher software and printer software runs as a separate executable connected by the adapter software.

Connectivity: There was only one physical connection and two logical connections present in the prototype set up. The physical connection was between the PC and the prototype board was established using USB. In the Windows operating system, Ethernet support over USB was provided by installing a Theyscon USB Ethernet control model (ECM) device driver. This step was necessary to incorporate socket based communication between software components in a network. The two logical connections were a socket connection between the printer software and the adapter software and a socket connection between the adapter software and the finisher software.

Programming language: The C++ programming language was selected for implementing the software components. This programming language is the de-facto programming language used in Océ for implementing printer software and offers the benefits of object oriented techniques.

Compilation environment: There was no pre-defined compilation environment required for Vmware Ubuntu since the operating system is LINUX based and the code can be directly compiled on the host (Ubuntu). The target (board) environment has no direct compilation environment. Hence, an Océ cross-compiler environment (based on MinGW) was setup for compiling the code from Windows to LINUX running in ARM.

6.2 Testing the Prototype Board

The initial testing performed was to compile a simple program and execute it in the target. The executable was kept in the USB drive and the execution was performed using a serial interface. There was also a trivial file transfer protocol (TFTP) connection established between PC and board in order to update adapter software without physically removing the USB drive.

Ping test: The next step was to test the connectivity between the host and the target. There were fixed IP addresses configured for all the devices. The connectivity was checked by sending a ping request. There was an initial problem with ping tests from the host to the target. The Vmware had to be set in bridged mode in order to make the ping work correctly.

Polling server: The testing procedure followed was to perform emulation of the printer software and finisher software using test stubs. The test stubs were connected to a client which uses socket based communication. Initially, there was a client established in the host and a server was established in the target. The client and the server communicated via sockets (SOCK_DGRAM based). The server implemented a chat functionality receiving datagram messages and sending them back to the client. The next procedure was to test the complete set up of the adapter where in the adapter is expected to handle asynchronous messages from the printer and the finisher. Hence, a polling server was established in the adapter which polls for messages from two clients based on port numbers. There were two port numbers defined: one for the printer and other for the finisher. The socket used here was based on 'SOCK_STREAM' and the adapter does translation of messages. Figure 6.2.1 shows the initial testing of the prototype board and this testing procedure helped to clear most of the hardware issues and networking issues.

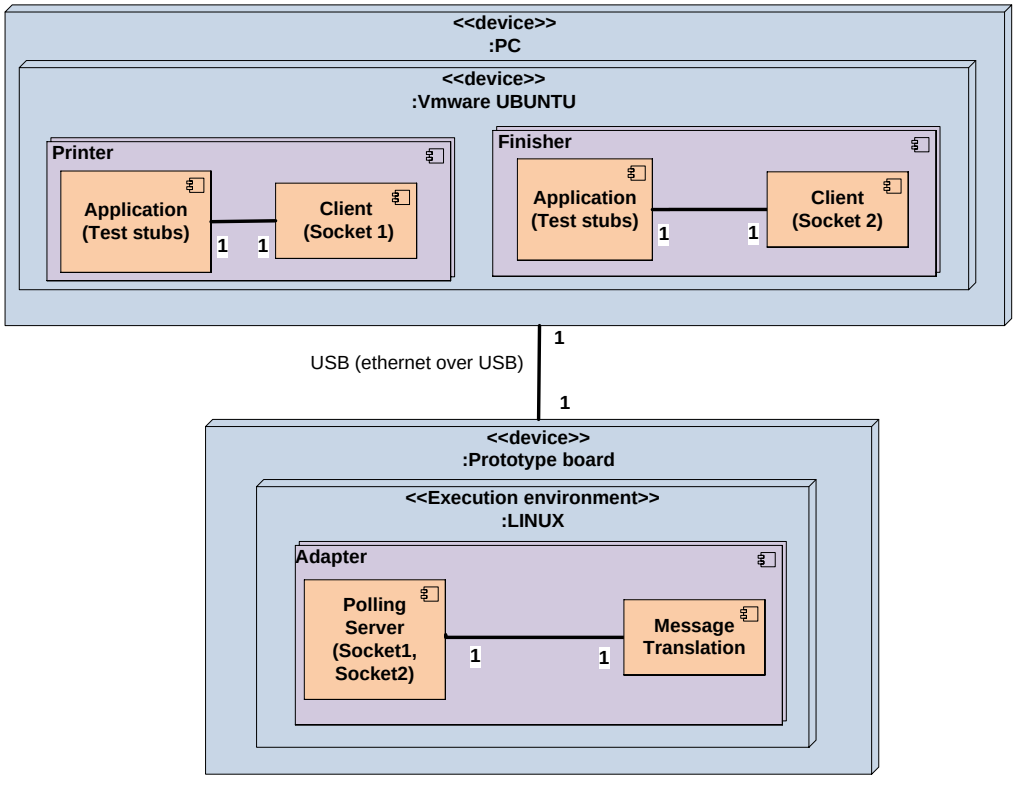


Figure 6.2.1: Initial testing of prototype board

6.3 Interfacing with Inter-process Communication Library

The one of the focus points of the project was verification of the adapter software for logical messages. The proof of concept could be achieved only with marshalling and unmarshalling components. The low level conversion software was an open choice in the thesis and an existing internal inter-process communication library from Océ was selected in order to implement the low level software. The inter-process library is used for providing asynchronous socket based communication across peers in a network. The reason behind the selection was to aid the prototyping phase since low level socket implementation will take additional time.

Protocol stack: Figure 6.3.1 shows the protocol stack of the complete system comprising of printer, adapter and finisher. The protocol stack for the printer was meant only for finisher interfacing. The application layer contains different message categories which were defined in the generic protocol and specific protocols. The inter-process library acts a middle layer enabling application messages to travel in a network and reach other devices through lower layers. The lower layers contain driver software for the communication channel. There are two types of scenarios possible: the real scenario in which inter-process stack used only in the printer and the adapter since finisher is considered as a black box and the prototype scenario wherein the finisher also contains inter-process stack layer since this facilitates the speed of prototyping.

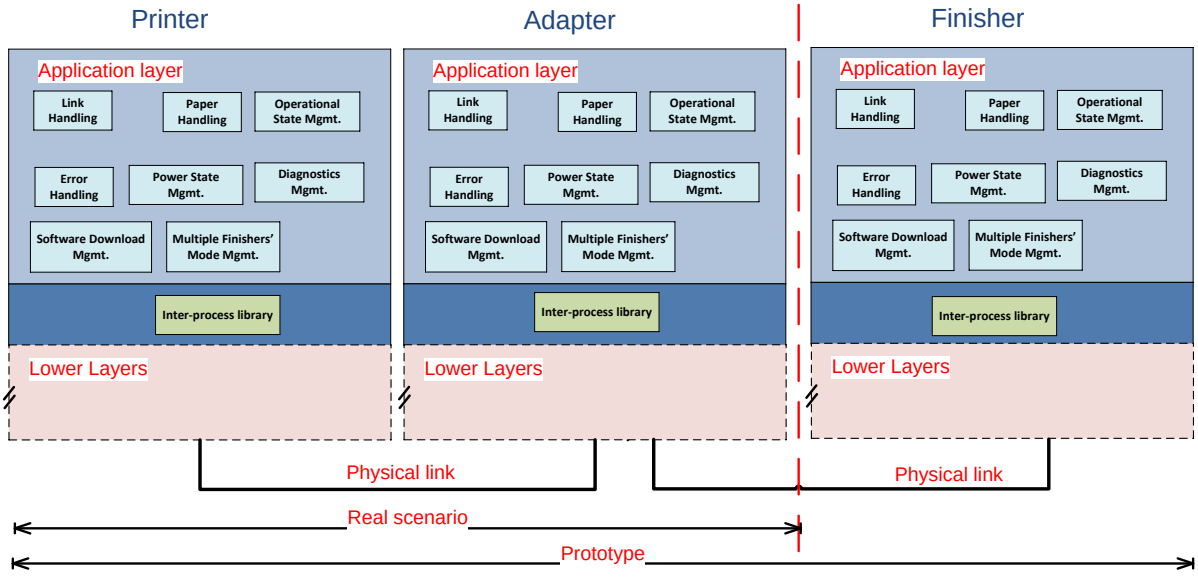


Figure 6.3.1: Protocol stack

Functionality: Figure 6.3.2 shows the code generation process of inter-process tool. The parser is fed with interface specification file in XML format and existing input style-sheet files. The specification of interfaces using the library for the typical adapter scenario is given in Appendix E: "Inter-process communication specification for the prototype of typical adapter scenarios". The interface specification comprises of signals which are interface messages and parameter type

definitions. There is only one parameter allowed per message but multiple parameters can be composed as a structure. The output files from the parser are a source C++ file and a header C++ file. These files contain classes for server and client implementation.

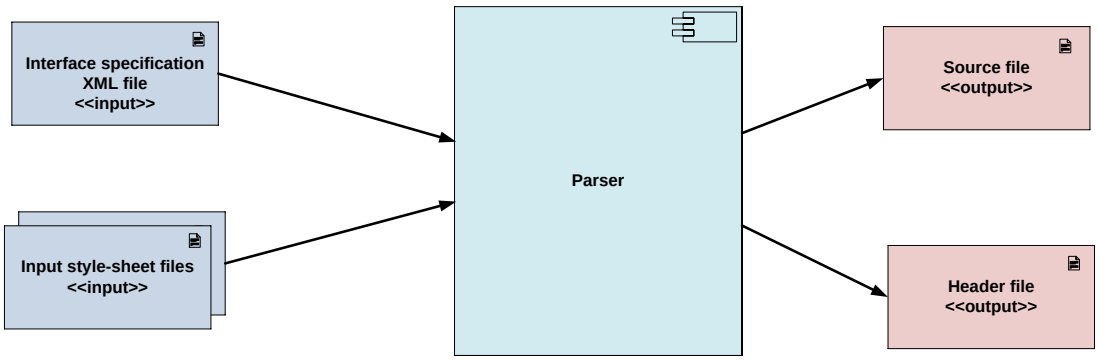


Figure 6.3.2: Tool functionality

Compilation: After the generation of files, client and server parts of the code can be integrated to an application. The client or server code needs the inter-process library for compilation. An application developed using inter-process tool would require inter-process tool generated files, the inter-process library, inter-process library header files and application files for the compilation.

Execution: The execution of an application using the inter-process library required a network topology file as an input during run-time. This file contained name of the application, name of the peer, interface identifier, transport layer details like transport identifier, IP address and port number. The library creates an additional thread in order to listen to incoming messages from an interface.

Interfacing: Figure 6.3.3 shows the testing of adapter architecture performed using the inter-process library.

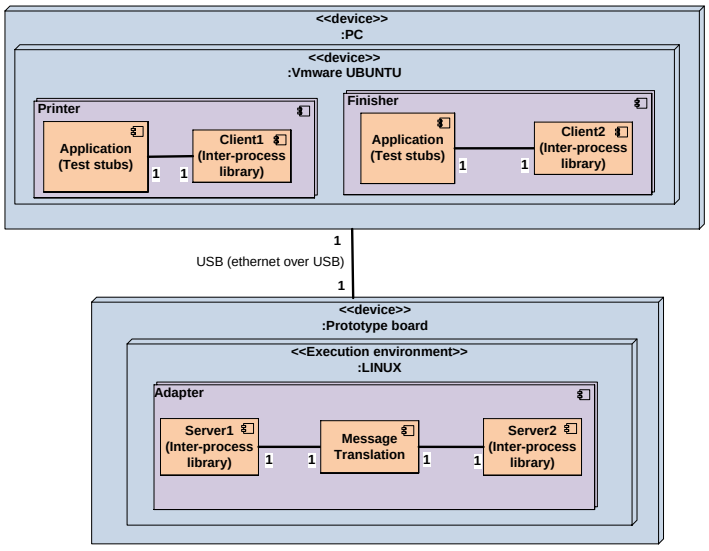


Figure 6.3.3: Testing the adapter architecture using the inter-process library

This procedure was followed to test the adapter architecture using the library but without model based components. This procedure was used to evade any risks due to the integration of the library. The testing was the same as that of the testing performed using the polling server except that there were two servers handling printer and finisher messages separately enabling concurrent reception of messages.

6.4 Implementation of Typical Scenarios

In order to test the feasibility of the adapter, it was necessary to prototype a design which had all the message conversion complexities present. The typical adapter scenario mentioned in Section 4.1 contains such complexities. This adapter scenario with inclusion of timeouts was chosen for prototyping. There were three scenarios: adapter scenario 1, adapter scenario 2 and adapter scenario 3.

Adapter scenario 1: Figure 6.4.1 shows the first adapter scenario which is same as the typical adapter scenario shown in Figure 4.1.1 but with more details and a minor change. The additional details were that this sequence diagram contains all the components of the adapter and printer and the parameter conversion was changed to $'i + 100'$ from $'i + 1'$ from the typical adapter scenario given in Figure 4.1.1.

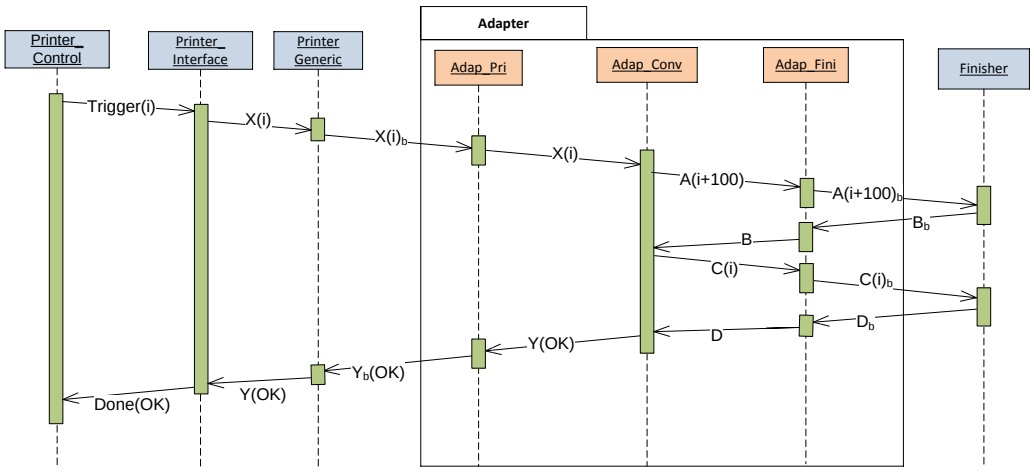


Figure 6.4.1: Adapter scenario 1

Adapter scenario 2: This adapter scenario 2 is shown in Figure 6.4.2 represents an alternate flow happening in the adapter state machine for error cases. This scenario incorporates timer module which makes the state machine more complex than the previous scenario. If the adapter does not receive B message as reply for a $A(x)$ message sent to the finisher, then the adap_conv does a timeout after 300ms in order to send a $Y(FAIL)$ to the printer and sends a $Reset$ message to the finisher. The printer interface module converts $Y(FAIL)$ message to $Done(FAIL)$ and sends it to the printer control module. Initially, the $Reset$ message was not present and it was added to tackle an 'interface violation' error captured by the ASD verification procedure. The interface violation error occurs since B message might be received in another state after some delay and the message

is not expected in that state. The $Reset$ message is used to bring the interface state machines in the adapter and finisher to the initial state where in B message is ignored.

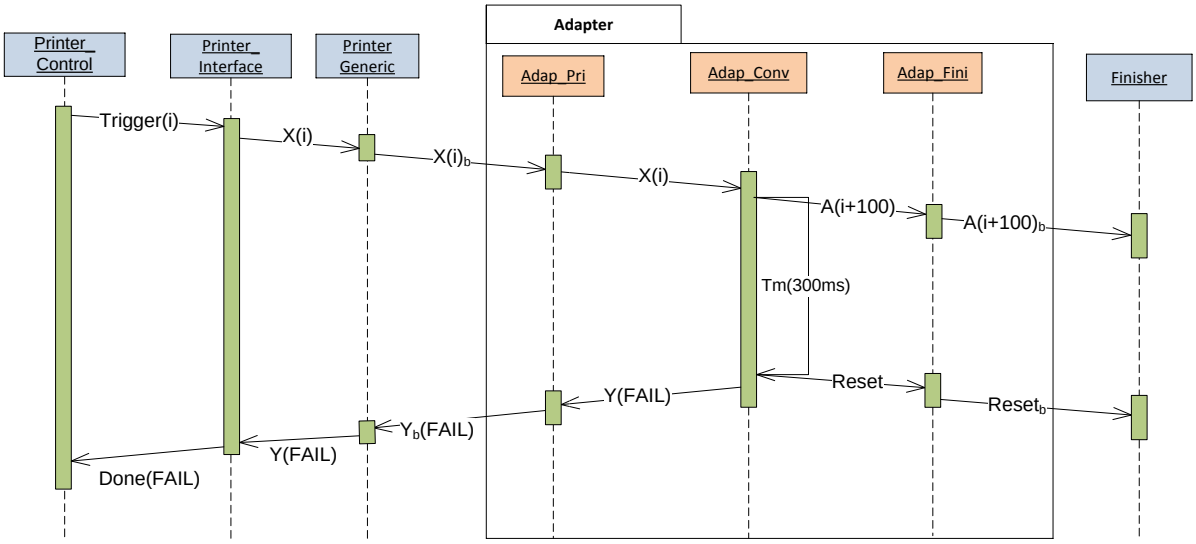


Figure 6.4.2: Adapter scenario 2

Adapter scenario 3: This scenario shown in Figure 6.4.3 is the same as the previous scenario except that timeout happens for the $C(i)$ message. If the adapter does not receive a D message as reply for a $C(i)$ message sent to the finisher, then the adap_conv does a timeout after 200ms in order to send a $Y(FAIL)$ message to the printer and send a $Reset$ message to the finisher. The printer interface module converts $Y(FAIL)$ message to $Done(FAIL)$ and sends it to the printer control module.

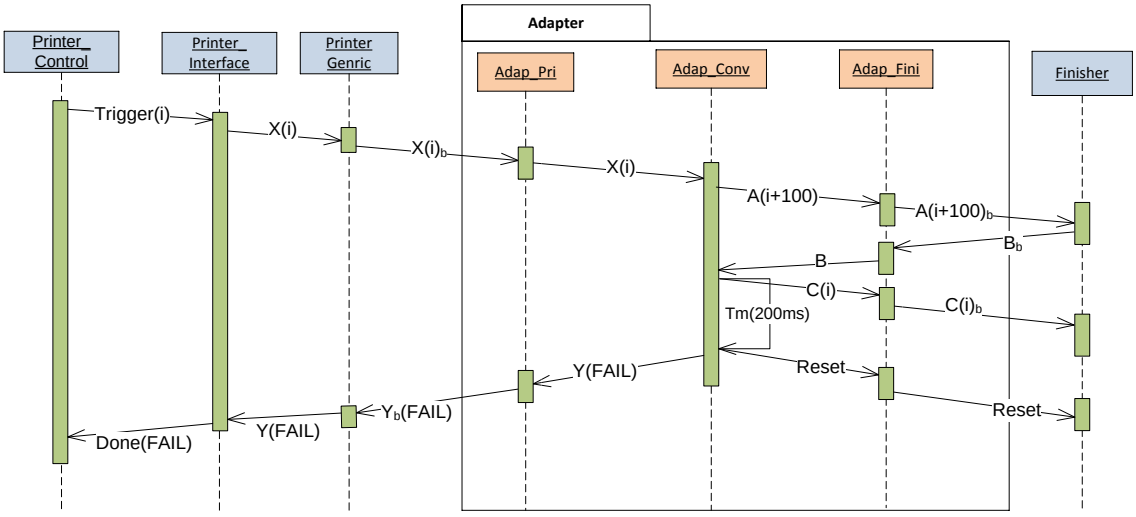


Figure 6.4.3: Adapter scenario 3

State machines: Figure 6.4.4 represents the required state machines for designing the prototype using ASD. Please note that the name given in parenthesis for each state machine is same as the ASD model naming. The three scenarios shown above are a trace of these state machines and the state machines capture the complete expected functionality of the printer, adapter and finisher.

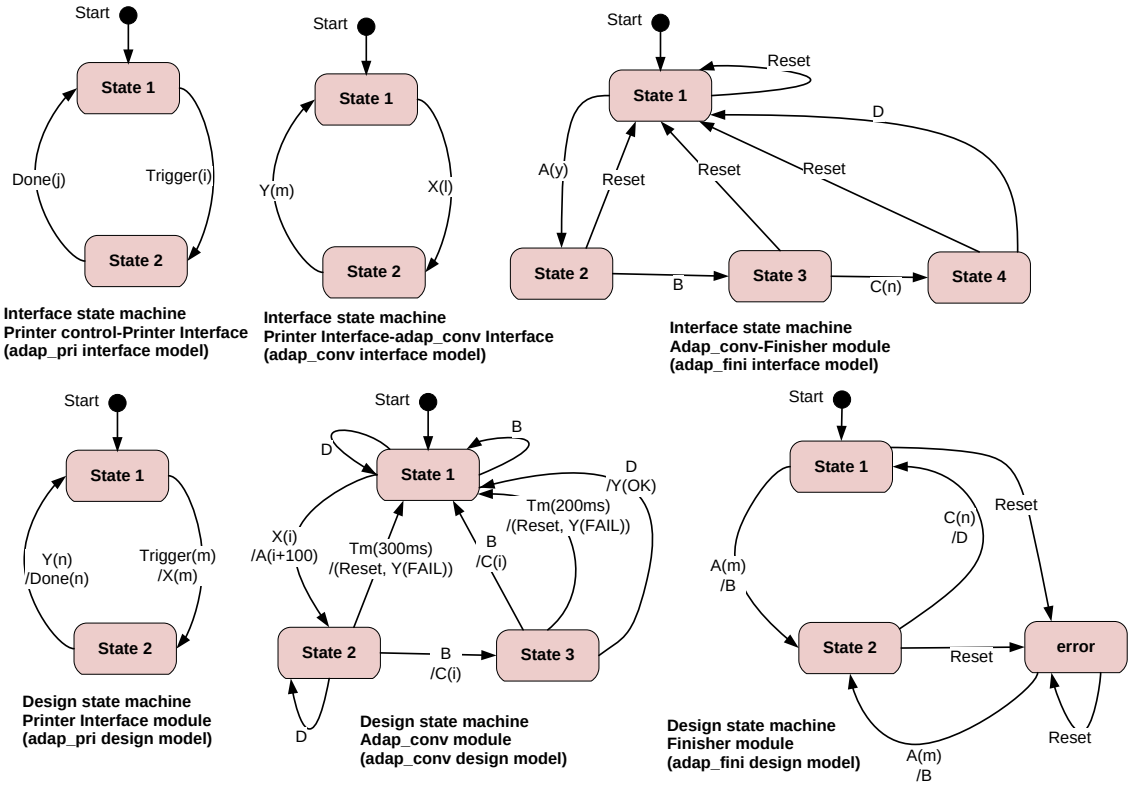


Figure 6.4.4: State machines for typical adapter scenarios

Mapping with deployment view: The components present in the sequence diagrams can be mapped to components in the prototype deployment view shown in Figure 3.4.2. The components such as router and adapter manager were absent in this prototype. The reason behind the removal was to test the basic functionality of the adapter with a message converter. The adapter manager is used for performing non-functional adapter activities like software download and interaction with the hardware. Hence, testing these features was less critical than testing a message converter. The router performs splitting of messages based on message identifiers in an environment with multiple finishers. Since there was only one finisher in the prototype this component was also removed.

ASD components: Figure 6.4.5 shows the ASD components used for designing the prototype. The square boxes with a green color tick mark represent design models and other blocks represent interface models. Printer related components are in the top layer, adapter related components are in the middle layer and finisher related components are in the bottom layer. Please find the detailed ASD state machines (SBS) for typical adapter scenarios in the Appendix D: "ASD specification for the prototypes: typical adapter scenario and link handling". The explanation of these components (since different naming procedure was used for modeling using ASD) with respect to the prototype deployment view in Figure 3.4.2 is given below:

- Adap_pri interface model: This represents the interface between printer control module and printer interface module.
- Adap_pri design model: This represents the internal behavior of printer interface module.

- Adap_conv interface model: This represents the interface between printer interface module and adap_conv module.
- Adap_conv design model: This represents the internal behavior of adap_conv module.
- Adap_fini interface model: This represents the interface between adap_conv module and finisher module.
- Adap_fini design model: This represents the internal behavior of finisher module.
- Message_proc interface model: This represents the interface between adap_conv module and message processing foreign component. This model is included since ASD performs message conversion in foreign components.
- ITimer interface model: This is the built-in interface model provided by ASD for incorporating timer functionality in the design.

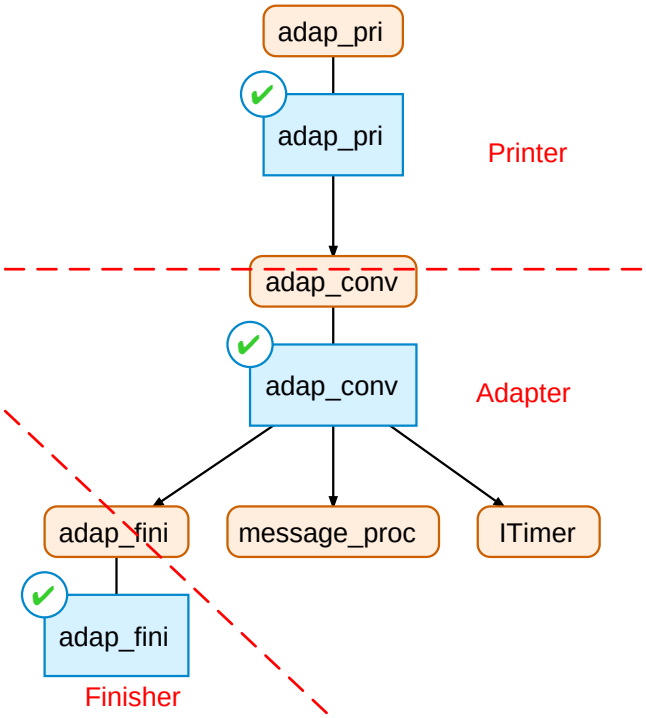


Figure 6.4.5: Components of ASD for typical adapter scenarios

Verification and integration procedure: The ASD models for the prototype were verified and the model was found to be error free by ASD. The verification was done as a complete model incorporating printer, adapter and finisher. The implementation required different state machines in different files. Figure 6.4.6 shows all software components for typical adapter scenarios. Here, printer, adapter and finisher components are split-up and contain their foreign components. Figure 6.4.7 shows the different files generated by ASD.

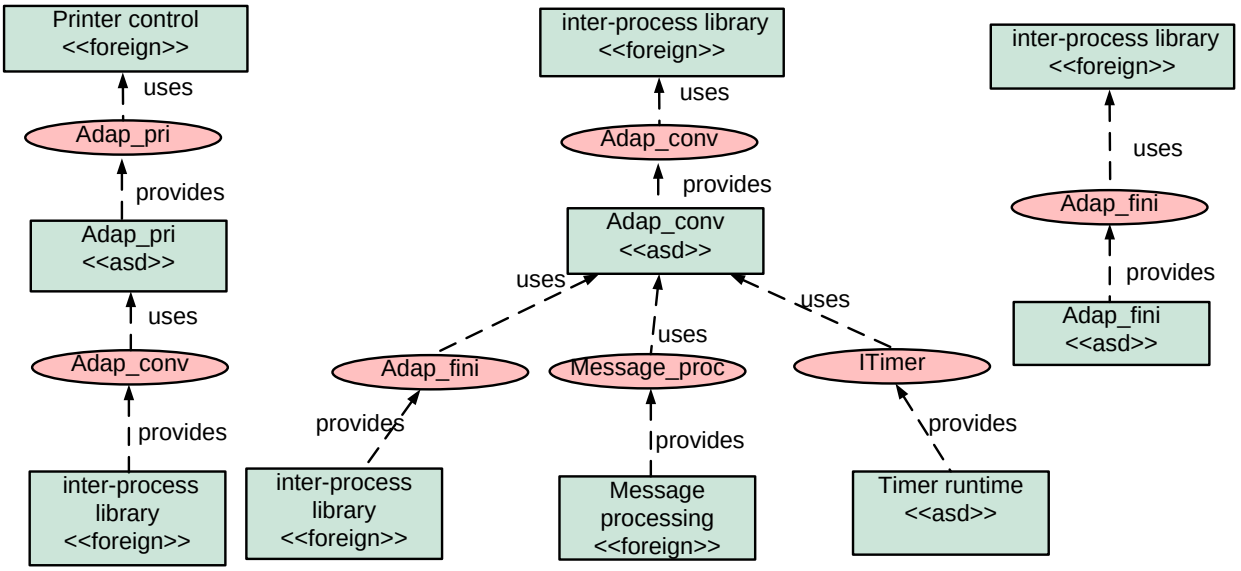


Figure 6.4.6: Complete software components for typical adapter scenarios

Printer

Generated ASD files: adap_convInterface.h, adap_priComponent.cpp, adap_priComponent.h, adap_priInterface.h

Handwritten stub files: printer_control.cpp, printer_control.h, printer_genericComponent.cpp, printer_genericComponent.h

Adapter

Generated ASD files: adap_convComponent.cpp, adap_convComponent.h, adap_convInterface.h, adap_finiInterface.h, message_procInterface.h

Handwritten stub files: adap_finiComponent.cpp, adap_finiComponent.h, adap_generic.cpp, adap_generic.h, message_procComponent.cpp, message_procComponent.h

Finisher

Generated ASD files:adap_finiComponent.cpp, dap_finiComponent.h, adap_finiInterface.h

Handwritten stub files: finisher_specific.cpp, finisher_specific.h

Common

Runtime files:channels.h, configurator.h, context.h, diagnostics.cpp, diagnostics.h, dpc.cpp, dpc.h, passbyvalue.h, trace.h,ucv.h

Figure 6.4.7: ASD generated files

Integration with inter-process library: The input files 'interface_generic.xml' and 'interface_specific.xml' specifies the generic interface messages and specific interface messages for the prototype respectively. These files are available in the Appendix E: "Inter-process communication specification for the prototype of typical adapter scenarios". The parser produced the following output files: 'interface_generic.cpp' and 'interface_generic.h' for 'interface_generic.xml' input file. The parser produces the following output files:

'interface_specific.cpp' and 'interface_specific.h' for 'interface_specific.xml' input file. These output files contain server and client C++ classes necessary for interfacing with ASD handwritten stubs.

With client stubs: The inter-process library can invoke member functions of the client stub (foreign component) class and the client stub can invoke member functions of the library class through shared static objects. The same logic is equally valid for interfacing any other application code with ASD client stubs. Figure 6.4.8 shows an example code snippet for the interfacing of ASD client stubs.

```
//adap_pri.cpp
//ASD client stub - Message sent from the adapter client to the inter-process library

void iadap_conv_NIProxy::y(const asd::value< bool >::type& i)
{
        // Start custom code section
        std::cout<<"Adapter client: Received Y message from the adapter ASD module: "<<i<<std::endl;
        Interface_Generic.Y(Result(i));
        std::cout<<" Adapter client: Sending Y message to the inert-process library : "<<i<<std::endl;
        // End custom code section
}


//AdapterServerImp.cpp
//Inter-process library code - Message sent from the printer to the adapter client

int AdapterServerImpl::X( const interface_generic::Data& a_Arg)
{
        std::cout<< "Inter-process: Received X message from the printer : " << a_Arg.DataValue << endl;
        pri_c.x(a_Arg.DataValue);
        std::cout<< " Inter-process: Sending X message to the adapter : " << a_Arg.DataValue << endl;
        return 1;
}
```

Figure 6.4.8: Code snippet for the interfacing of ASD client stubs

With used component stubs: The integration of inter-process library or any other application with ASD used component stubs for application interface messages can be done in the same manner as with client stubs using shared static objects. The notification events have to be integrated little differently for used component stubs. The notification function calls can be made only from an ASD stub member function and not from outside. Since ASD expects client requests and notification events to be handled in a single thread for used component stubs. For the inter-process library, this is not an option since the library executes in a separate thread for handling incoming messages and has to send notification events to ASD.

The procedure followed to send notification events from the inter-process software to ASD are as follows:

- Callback functions were created for every notification message that needs to be send to ASD

- These callback functions were declared as pure virtual functions in a separate class (‘control class’) which were inherited by ASD and inter-process library classes
- The implementation of the callback functions were made in an ASD class
- A shared static object of the inter-process library class was created in the constructor of ASD and this constructor invoked an inter-process library member function with ‘this’ pointer as parameter. In this member function, the received parameter was stored in a shared static object (‘control object’) of type ‘control class’. This step was done to enable dynamic binding of ‘control class’ member functions
- This ‘control object’ was used to invoke callback functions whenever inter-process library needed to send notification events to ASD
- Then the ‘control class’ implementation in ASD was invoked due to dynamic binding
- These call back functions present in ASD in turn send notification events to ASD

Compilation: Figure 6.4.9 shows the compilation and linking procedure followed for creating an executable.

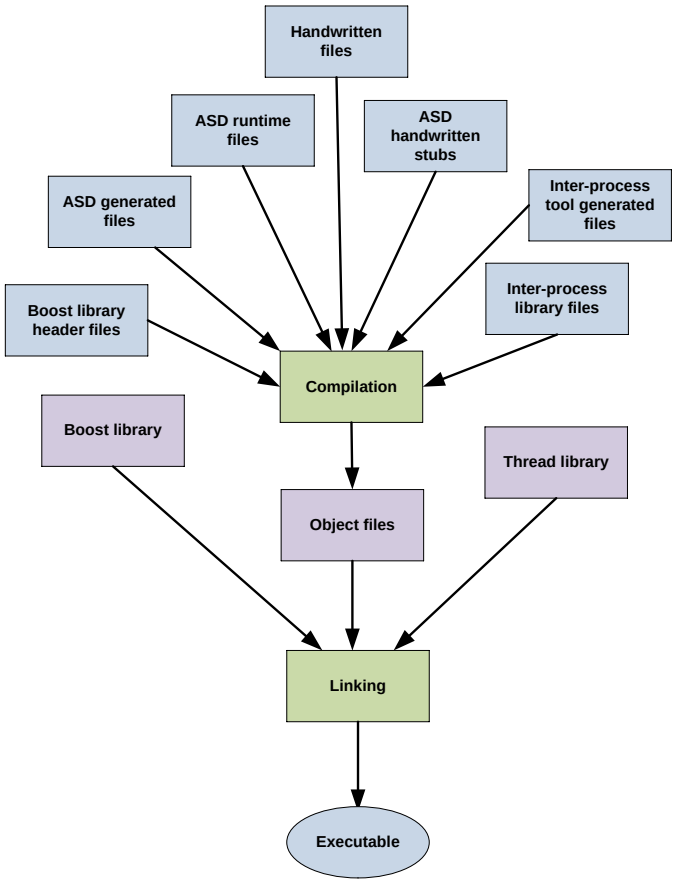


Figure 6.4.9: Compilation and linking procedure followed

The compilation of C++ files generated by ASD requires boost libraries [26] for compilation and linking. The source and header files necessary for boost libraries can be downloaded from the boost organization website [26]. These files have to be compiled using a procedure specified in their

website. The documentation was available for Windows and LINUX. The cross-compilation procedure using 'MinGW' was not available in their documentation. The information available is that the compilation may or may not be successful. The cross-compilation to target was set up from the Windows environment. This cross-compilation of boost library files posed some challenges since it was supported by the documentation. The version of boost library used was boost_1_47_0.

Mapping with component static view: Please refer to component static view shown in Figure 3.5.1 in chapter 3 for understanding this comparison for the adapter. The ASD generated files contain code for implementing the message converter component and the queuing component. Hand written stub files together with inter-process library contain code for implementing the adap_pri and adap_fini components.

Testing: There were three different executable files (printer, adapter and finisher) created. Figure 6.4.10 shows a screenshot of the prototype testing. The prototype was tested by making the printer control module to send $Trigger(i)$ messages for every 2 sec and other values like 1 sec, 100 ms. The prototype was tested for various scenarios by executing only the printer module and adapter module, by executing printer module only, by executing the complete system involving printer, adapter and finisher modules, by disconnecting USB cable and then reconnecting it and by stopping one of the modules. The prototype produced expected results for all the test cases. Thus the prototype proved the feasibility for the adapter software using a modeling approach.

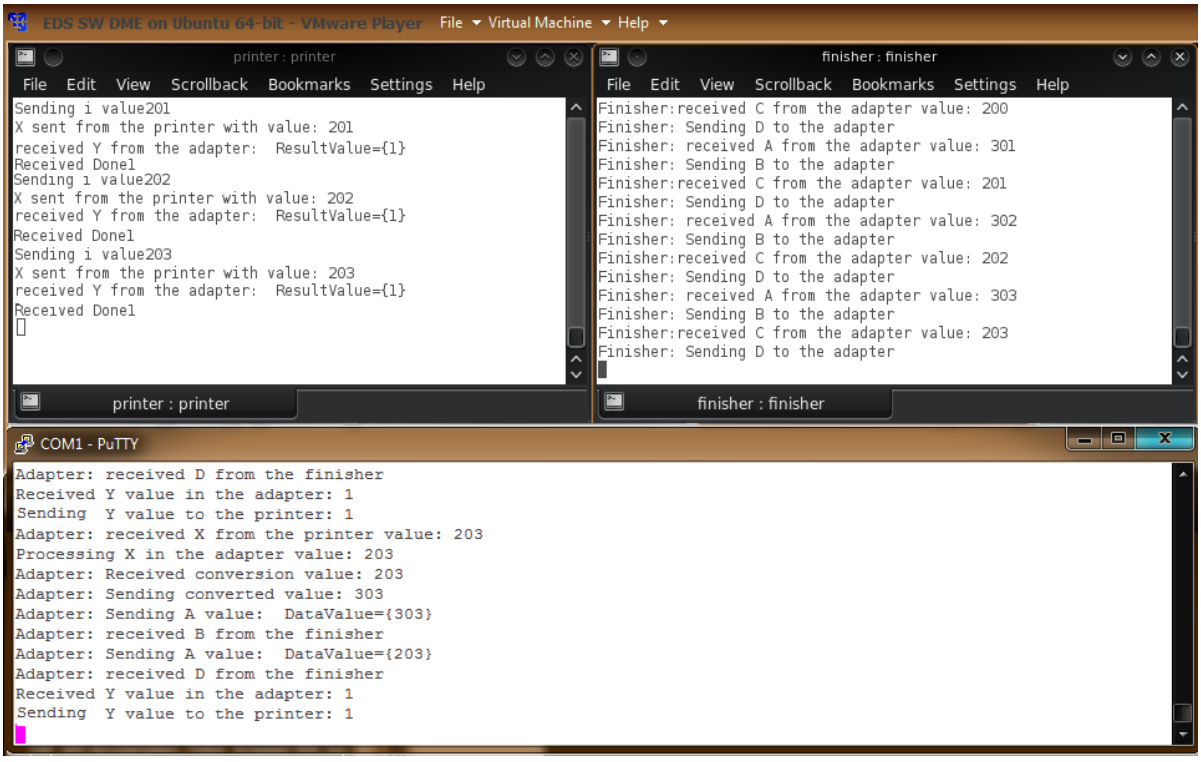


Figure 6.4.10: Screenshot of the prototype testing

Mapping with component dynamic view: Please refer to component dynamic view shown in Figure 3.5.2 in chapter 3 for understanding this comparison for the adapter. There were four threads running in the adapter module: hrtimer_nanosleep, futex_wait_queue_me and two sk_wait_data threads. Thread 'hrtimer_nanosleep' corresponds to adap_conv thread, the two threads for 'sk_wait_data' corresponds to adap_pri and adap_fini threads and thread 'futex_wait_queue_me' corresponds to queuing thread present in message converter module.

6.5 Implementation of Link Handling

For the link handling message category, detailed state machines for printer, adapter and finisher components are available in the Appendix C: "Generic protocol for adapter interface". The idea behind the implementation of these state machines was to explore ASD in a more detailed way. The complexities present in this message category can be understood from section 4.2.2. The ASD modeling could not be completed due to time constraints. But the modeling helped to understand more concepts involved in making ASD. Figure 6.5.1 shows a partial ASD model created for the link handling message category.

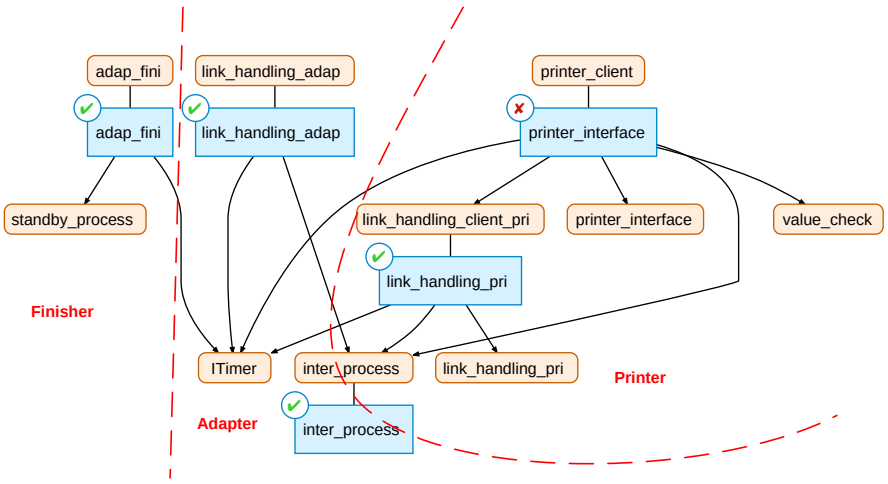


Figure 6.5.1: ASD components for link handling message category (Partial)

The following observations were made during this partial implementation:

The asymmetric nature of ASD has impact on the design. If the system behavior is not asymmetric then modeling becomes tricky. For example in an interface model, if there are only notification events in a state then one of these have to be kept as 'inevitable' event with a yoking value to avoid deadlock conditions. Still, there would be problems related to the interface violations because 'inevitable' event overruled the occurrence of other 'optional' events. The best practice employed was to incorporate an additional application interface event when there were more than one notification event in a state. This changed the original design.

The allowance of single interface model to capture all the messages for a design model from the top layer breaks the communication protocol verification when the protocol state machines would be

implemented in separate devices. For example, in order to check whether an interface model fits two design models (one at the top and another at the bottom), there should be a single interface model used. In reality, the interface model has to contain messages from other actors apart from the primary actors involved in the interface protocol. This forces the interface model to be split in two different interface models with different state machines breaking the verification of interface protocol.

The absence of parameter processing within an ASD design model increases the size of state machines. For example, link handling state machines contain decision making blocks and these have to be processed in foreign components and introduced back as notification events in the design model. This causes the design model to grow in size.

Application interface events execute in the context of the client. Link check functionality needs to be implemented (maintenance of link) as a separate thread. The link check module was modeled as a design model. This would create a DPC server thread since there were notification events to this module. In order to make application interface events to execute in a separate thread, the client connected to the interface model must implement a separate thread. This changed the architecture by introducing an additional DPC thread whenever a component has to execute in a separate thread.

Communication between different threads (design models at a same level) can be done using broadcast messages. For example, in the link handling category the link check module reports 'link failure' which needs to reach other modules running in different threads. This can be modeled using a separate interface model and design model dedicated to receive the 'link failure' message and broadcasting it as notification events to all design models connected to the interface. In order to read a broadcast message, the design model should have subscribed for broadcast messages arriving from an interface. The summary is that there is no peer-to-peer type of communication in ASD.

6.6 Conclusion

This chapter explains the steps involved in developing a prototype in order to check the feasibility of the adapter software using a modeling approach. The test set up was selected to aid the process of prototyping and to test the concept. The selection of an existing inter-process library was a design choice which can be replaced with other software package if required. The integration process involved several challenges especially in integrating boost library files. The prototype implementation showed that having a model based adapter is feasible for the proposed architecture. The prototype implementation also proved the feasibility of the generic protocol. The partial implementation of link handling revealed some design constraints while using ASD. The ASD indeed verifies consistency between interface model and design model which improves the generated software. There was considerable amount of time spent to understand the modeling approach followed in ASD and the type of verifications done by it.

Chapter 7

Conclusions

The problem statement of the master thesis is revisited to understand the match between the given problem and the results attained.

> **Problem statement:** *To propose a model based framework for interfacing Océ wide format printers with any type of wide format finisher and to prototype a model based framework to test the feasibility.*

In the next sections, the main contributions made in the project, limitations and future work, and final outcome of the thesis are discussed.

7.1 Main Contributions

Generic Protocol: The thesis involved a domain analysis to understand existing finisher interface protocols and finisher capabilities. This was done to understand the different complexities in the finisher protocols. The generic protocol was designed to tackle present and future changes in the finisher protocols. Two message categories with maximum message conversion complexities were identified for the complete protocol specification. For these categories, the specification was done in an extensive manner with message details, sequence diagrams capturing various scenarios, interface state machines for printer and finishers, adapter state machine and mapping of different finisher protocols with the generic protocol. This proved the feasibility of having a generic protocol for finisher interfacing.

Adapter architecture: In the thesis, adapter architecture has been proposed to handle different capabilities of finishers and to support a model based approach. Then, different architectural views were framed to understand the fit with stakeholders. The prototype implementation was done based on the architecture and reflections were made to check if the actual implementation meets

the expectations from the architecture. The prototype implementation showed the feasibility of the adapter architecture.

Modeling approach selection: In the thesis, there was an initial survey done to understand different modeling approaches for designing the adapter software. There was a case study performed using Petri nets and ASD to understand these tools in details. Then, there were selection criteria framed to analyze different modeling tools. The selection criteria comprised of model based concepts, quality metrics, engineering methods and business interests. Finally, a selection chart was created to compare different modeling tools based on these selection criteria in order to select a modeling tool for designing the adapter software. The ASD tool was selected based on this chart. The business reason behind the selection is that the Océ had interests in understanding the tool. The technical reason behind the selection is that the tool requires complete specification of dynamic scenarios initially during design phase which will result in fewer errors at the time of the implementation.

Prototyping: The prototype implementation of the adapter software was done using the ASD. The prototype was made based on the adapter architecture. The prototype was used to test the feasibility of the generic protocol, adapter architecture and actual handling of message conversions. The prototype implementation was done for typical adapter scenarios comprising of all message complexities. The prototype was tested for different scenarios and the prototype behavior followed the model specification. The prototype testing proved the feasibility for using the ASD to model the adapter software.

7.2 Limitations and Future Work

The selection of the modeling approach was not done in a strict way. The reason behind was that Océ had interests in understanding the ASD. There might be other tools available that are more suitable for making adapter software. This can be viewed as a limitation in the thesis.

The modeling of adapter software with the ASD requires changes in the original design. The important aspects of ASD include its asymmetric nature, transparency to message parameters, lack of peer-to-peer communication, thread creation procedure and usage of single interface model for a design model at a top level. These attributes are discussed in detail in section 6.5 which impacts the original design of the adapter. This can be a limitation factor which requires consideration before modeling the software using ASD.

The prototype that was created using ASD had strict interface specifications. This means that the foreign components were expected to adhere to these interface specifications. If during runtime, foreign components send faulty messages then it would lead to software crashes. This problem can be solved conceptually by using an ‘armour’ component. The current prototype can be extended with ‘armour’ component and tested with erroneous foreign components. This would give a clear view about deploying the adapter software in reality.

The generic protocol was designed considering different existing finishers and future possibilities. The completion and extension of the generic protocol can have a clear roadmap in five years to make it as an industrial standard for finisher interfacing. This would remove the requirement of a finisher adapter for future finishers.

The Event-B modeling approach seems to be a promising one among the investigated tools since it can be used from the requirement specification to the code generation. There was no case study performed using the tool due to time constraints since the tool was identified at a later stage in the project. A case study could be performed to understand the tool functionality for making the adapter software.

The scope of the project was restricted to wide format printers. The scope can be widened to incorporate cut-sheet printers in the future. This would demand additions in the generic protocol specification. But the architecture for adapter software in principle should remain the same for a possible cut-sheet adapter and other similar devices.

The finisher adapter was envisioned as a separate unit interacting only with printer and finishers. The impact of interaction with other devices like a paper flow controller can be checked in the future to test the applicability of the architecture for handling instructions from other devices.

7.3 Final Outcome

The usage of the generic protocol for interfacing wide format printers and finishers is feasible. The two message categories used for protocol specification were the most complex categories. The specification of other message categories should be also feasible.

The modeling of an adapter software using ASD is feasible. The prototype implementation testifies the previous statement.

The usage of a generic adapter architecture which supports model based software is feasible. The prototype implementation was checked with different architectural views in order to come to this conclusion.

The modeling and verification of an interface protocol using ASD is feasible in some cases. The feasible case is for the prototype implementation of typical adapter scenarios that was modeled and verified using ASD. The difficult case is for the partial ASD implementation of link handling that had problems in the design for having a common interface model.

References

[1] Christian Gierds, Arjan Mooij and Karsten Wolf, “Reducing adapter synthesis to controller synthesis,” IEEE transactions on services computing, vol. 5, no. 1, pages 72-84, january-march 2012.

[2] Wil van der Aalst, Arjan Mooij, Christian Stahl and Karsten Wolf, “Service interaction: patterns, formalization, and analysis,” In advanced lectures of the 9th international school on Formal Methods for Web Services (SFM-09: WS), LNCS 5569, pages 42-88, Springer-Verlag, 2009.

[3] Christian Gierds, Arjan Mooij and Karsten Wolf, “Specifying and generating behavioral service adapters based on transformation rules,” Preprints CS-02-08, Institut für Informatik, Universität Rostock, Germany, 2008.

[4] M.Siebert, B.Walke, “Design of Generic and Adaptive Protocol Software (DGAPS),” Proceedings of the Third Generation Wireless and Beyond, 3Gwireless '01, San Francisco, Calif, USA, June 2001.

[5] ASD: Suite. Online. Accessed 8th January 2013. http://www.verum.com/product/Verification-is-the-Difference.aspx

[6] Ramin Sadre et. al, “Simulative and Analytical Evaluation for ASD-Based Embedded Software,” Measurement, Modeling, and Evaluation of Computing Systems and Dependability and Fault Tolerance Lecture Notes in Computer Science Volume 7201, 2012, pp 166-181.

[7] Wide format printers. Online. Accessed 12th February 2013. http://en.wikipedia.org/wiki/Wide-format_printer

[8] Arjan J. Mooij, Voorhoeve M. “Specification and Generation of Adapters for System Integration” Chapter in the book titled "Situation Awareness with Systems of Systems.", Piërre van de Laar et. al, ISBN: 978-1-4614-6229-3.

[9] Tool MARLENE. Online. Accessed 13th February 2013. http://service-technology.org/tools/Marlene

[10] Tool WENDY. Online. Accessed 13th February 2013. http://service-technology.org/tools/wendy

[11] Tool Yasper. Online. Accessed 14th February 2013. http://www.yasper.org/

[12] Tool Prom 5.2. Online. Accessed 14th February 2013. http://prom.win.tue.nl/tools/prom/

[13] E. Kindler, “A compositional partial order semantics for Petri net components,” in ATPN, ser. LNCS, vol. 1248, 1997, pp. 235–252.

[14] P. Massuthe, W. Reisig, and K. Schmidt, “An Operating Guideline Approach to the SOA,” Annals of Mathematics, Computing & Teleinformatics, vol. 1, no. 3, pp. 35–43, 2005.

[15] Philip Armstrong, Michael Goldsmith, Gavin Lowe, Joël Ouaknine, Hristina Palikareva, A. W. Roscoe, and James Worrell, “Recent developments in FDR,” Proceedings of CAV 12, LNCS 7358, 6 pages, Springer-Verlag, 2012.

[16] Butler, Michael. "Mastering System Analysis and Design through Abstraction and Refinement." (2012).

[17] Edmunds, Andy, and Michael Butler, "Code Generation for Event-B with Intermediate Specification," Rodin User and Developers Workshop. 2009.

[18] Méry, Dominique, and Neeraj Kumar Singh, "Automatic code generation from Event- B models," In Proceedings of the second symposium on information and communication technology, pp. 179-188. ACM, 2011.

[19] Leuschel, Michael, and Michael Butler, "ProB: A model checker for B," In FME 2003: Formal Methods, pp. 855-874. Springer Berlin Heidelberg, 2003.

[20] Abrial, Jean-Raymond, et al. "Rodin: an open toolset for modelling and reasoning in Event-B," International journal on software tools for technology transfer 12.6, (2010): 447-466.

[21] B Selic, "Tutorial: real-time object-oriented modeling (ROOM)," Real-Time Technology and Applications Symposium, 1996. Proceedings., pages 214 - 217, 1996 IEEE

[22] Rodin plug-ins for Event-B. Online. Accessed 29th July 2013. http://wiki.event-b.org/index.php/Rodin_Plug-ins

[23] Hooman, Jozef, Arjan J. Mooij, and Hans van Wezep. "Early fault detection in industry using models at various abstraction levels," Integrated Formal Methods. Springer Berlin Heidelberg, pages 268-282, 2012.

[24] ASD documentation. Online. Accessed 30th July 2013. http://community.verum.com/documentation/user_manual.aspx/9.1.0/overview

[25] Armour component in ASD. Online. Accessed 30th July 2013. http://community.verum.com/Files/Armour_pattern/ForeignClientArmourPattern.pdf

[26] Boost libraries for C++. Online. Accessed 30th July 2013. http://www.boost.org/

[27] Gulati, Ms Samridhi, and Ms Sarita Singh," Analysis of Three Formal Methods-Z, B and VDM," International Journal of Engineering 1, no. 4 (2012).

[28] Frappier, Marc, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar, "Comparison of model checking tools for information systems," In Formal Methods and Software Engineering, pp. 581-596. Springer Berlin Heidelberg, 2010.

Appendix A: “Message list for various finishers”

This is provided as a separate document.

Appendix B: “Océ prototype board specification”

This is provided as a separate document.

Appendix C: “Generic protocol for adapter interface”

This is provided as a separate document.

Appendix D: “ASD specification for the prototypes: typical adapter scenario and link handling”

This is provided as a separate document.

Appendix E: “Inter-process communication specification for the prototype of typical adapter scenarios”

This is provided as a separate document.