PATTERN-BASED DESIGN AND VALIDATION
OF COMMUNICATION PROTOCOLS

By

YOUNGJOON BYUN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2003
Copyright 2003

by

Youngjoon Byun
To my family
ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Beverly A. Sanders, for her excellent advice and advocacy during the course of this work. I would also like to thank the members of my committee, Dr. Jonathan C. Liu, Dr. Steve M. Thebaut, Dr. Joseph N. Wilson and Dr. Mark C. K. Yang, for their revision of this work and their valuable suggestions.

I am indebted to Dr. Yann-Hang Lee for his valuable guidance during the course of my Ph.D. program. My appreciation and special thanks go to Dr. Byung Sun Lee, Mr. Chang-Sup Keum and Ms. Ki-Sook Chung for the kind comments on the case study presented in this dissertation.

Finally, I wish to thank my wife Younji for her support and understanding during all these years.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>x</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Pattern-based Development Methodology</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Pattern Language Overview</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Organization of Patterns</td>
<td>7</td>
</tr>
<tr>
<td>2 RELATED WORK</td>
<td>10</td>
</tr>
<tr>
<td>2.1 Patterns in Communication Systems</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Model Checking using SPIN and PROMELA</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Introduction to SDL and Model Checking SDL System</td>
<td>18</td>
</tr>
<tr>
<td>3 PATTERN LANGUAGE FOR COMMUNICATION PROTOCOLS</td>
<td>22</td>
</tr>
<tr>
<td>3.1 Structural Patterns</td>
<td>22</td>
</tr>
<tr>
<td>3.1.1 Protocol Layer</td>
<td>22</td>
</tr>
<tr>
<td>3.1.2 Mux</td>
<td>26</td>
</tr>
<tr>
<td>3.1.3 Dynamic Handler</td>
<td>27</td>
</tr>
<tr>
<td>3.2 Behavioral Patterns</td>
<td>32</td>
</tr>
<tr>
<td>3.2.1 Communicating Extended Finite State Machine</td>
<td>32</td>
</tr>
<tr>
<td>3.2.2 Timer</td>
<td>47</td>
</tr>
<tr>
<td>3.2.3 Repeated Events</td>
<td>49</td>
</tr>
<tr>
<td>3.2.4 Message Transfer</td>
<td>55</td>
</tr>
<tr>
<td>4 MODEL CHECKING PATTERN-BASED DESIGN</td>
<td>65</td>
</tr>
<tr>
<td>4.1 Modeling Pattern-Based Design in PROMELA</td>
<td>65</td>
</tr>
<tr>
<td>4.1.1 Model Construction from Patterns</td>
<td>66</td>
</tr>
<tr>
<td>4.1.2 Communication Path and Channel Capacity</td>
<td>68</td>
</tr>
<tr>
<td>4.1.3 Timer Handling</td>
<td>70</td>
</tr>
<tr>
<td>4.1.4 Environment Process</td>
<td>73</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.2 Property Specification</td>
<td>73</td>
</tr>
<tr>
<td>4.2.1 Linear Temporal Logic</td>
<td>74</td>
</tr>
<tr>
<td>4.2.2 Property Extraction from Communication Patterns</td>
<td>75</td>
</tr>
<tr>
<td>5 CASE STUDIES</td>
<td>77</td>
</tr>
<tr>
<td>5.1 Case Study: Alternate Bit Protocol</td>
<td>77</td>
</tr>
<tr>
<td>5.1.1 Structural Design</td>
<td>77</td>
</tr>
<tr>
<td>5.1.2 Behavioral Design</td>
<td>78</td>
</tr>
<tr>
<td>5.2 Case Study: ATM UNI Signaling Protocol</td>
<td>82</td>
</tr>
<tr>
<td>5.2.1 Protocol Overview</td>
<td>82</td>
</tr>
<tr>
<td>5.2.2 Structural Design</td>
<td>86</td>
</tr>
<tr>
<td>5.2.3 Behavioral Design</td>
<td>87</td>
</tr>
<tr>
<td>5.2.4 Validation of Pattern-based Design</td>
<td>93</td>
</tr>
<tr>
<td>6 CONCLUSION</td>
<td>99</td>
</tr>
<tr>
<td>APPENDIX</td>
<td>102</td>
</tr>
<tr>
<td>A MODELING OF ALTERNATE BIT PROTOCOL IN PROMELA</td>
<td>102</td>
</tr>
<tr>
<td>B MODELING OF ATM UNI SIGNALING PROTOCOL IN PROMELA</td>
<td>106</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>117</td>
</tr>
<tr>
<td>BIOGRAPHICAL SKETCH</td>
<td>124</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td>1–1</td>
<td>Pattern language for communication protocols</td>
</tr>
<tr>
<td>4–1</td>
<td>Effect of channel capability on memory and CPU for ABP</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1–1</td>
<td>Overview of pattern-based design and validation</td>
</tr>
<tr>
<td>1–2</td>
<td>Organization of pattern language</td>
</tr>
<tr>
<td>3–1</td>
<td>A structure of pattern protocol layer</td>
</tr>
<tr>
<td>3–2</td>
<td>A structure of pattern split protocol layer</td>
</tr>
<tr>
<td>3–3</td>
<td>SDL implementation of pattern protocol layer</td>
</tr>
<tr>
<td>3–4</td>
<td>Example of pattern protocol layer in SSCF at UNI</td>
</tr>
<tr>
<td>3–5</td>
<td>Structure of ABP using split protocol layer</td>
</tr>
<tr>
<td>3–6</td>
<td>A structure of pattern mux</td>
</tr>
<tr>
<td>3–7</td>
<td>SDL implementation of pattern mux</td>
</tr>
<tr>
<td>3–8</td>
<td>A structure of pattern dynamic handler</td>
</tr>
<tr>
<td>3–9</td>
<td>A structure of pattern split dynamic handler</td>
</tr>
<tr>
<td>3–10</td>
<td>SDL implementation of pattern dynamic handler</td>
</tr>
<tr>
<td>3–11</td>
<td>A file transfer server using pattern dynamic handler</td>
</tr>
<tr>
<td>3–12</td>
<td>A call control block using pattern split dynamic handler</td>
</tr>
<tr>
<td>3–13</td>
<td>An example CEFSM represented in STD</td>
</tr>
<tr>
<td>3–14</td>
<td>Pattern basic CEFSM</td>
</tr>
<tr>
<td>3–15</td>
<td>Pattern predicate CEFSM</td>
</tr>
<tr>
<td>3–16</td>
<td>Pattern predicate after action</td>
</tr>
<tr>
<td>3–17</td>
<td>Merge patterns combined from two CEFSMs</td>
</tr>
<tr>
<td>3–18</td>
<td>SDL fragment for pattern basic CEFSM</td>
</tr>
<tr>
<td>3–19</td>
<td>SDL fragment for pattern predicate CEFSM</td>
</tr>
<tr>
<td>3–20</td>
<td>Example of pattern predicate after action for error detection</td>
</tr>
<tr>
<td>3–21</td>
<td>Example of merge patterns</td>
</tr>
</tbody>
</table>
Patterns help to improve software quality and reduce development cost by reusing the experience of experts for recurring problems. In this dissertation, we apply the pattern concept to the development of communication protocols, particularly focusing on the description and validation of message interaction in the protocols. Typically, it is important for designers to capture essential functions of a system at the initial design phase and uncover design errors as early as possible to prevent the errors from affecting later phases. There are many useful patterns for communication systems, but to date they are mainly concentrated on the object-oriented design and implementation. There is little research on the patterns for message interaction and the validation of pattern-based design. We hypothesize that many communication protocols can be developed using a few recurring patterns.

In this pattern-based methodology, we propose a set of patterns to describe the architectural and behavioral specification of communication protocols. A complex protocol can be obtained by composing such patterns. To provide confidence in the design, we suggest a validation method for the design using the SPIN model.
checker. The validation is composed of model construction for the design and identification of desired properties of the system. Then, the model is checked against the properties. The most difficult part of using tools such as SPIN is obtaining the appropriate properties of a system in a formal way against which to check the design. An innovative feature of our patterns is a section that helps the designer obtain the properties in linear temporal logic.

To show the usefulness of our methodology, we perform several case studies. From the methodology, protocol designers can abstract a system in several patterns and uncover design errors before the detailed design and implementation.
CHAPTER 1
INTRODUCTION

When programmers develop software systems, they often find situations similar to those that have arisen in previous developments. A design pattern is a written document providing a generic solution for a recurring problem in a certain context [1, 2]. A design solution that has worked well in a particular situation can be used again in similar situations in the future. Design patterns, therefore, help to improve software quality and reduce development cost through predefined solutions and their reuse.

Patterns are considered to be useful in many types of software systems because they provide generic solutions for common problems. However, it is not always easy for a domain specific designer to find and use the generic solution in a specific application. The solution may be unsuitable for a particular area. Thus, it is frequently useful to have a set of patterns for a specific domain. In this dissertation, we present several patterns for the development of communication protocols. Actually, research and usage of patterns in communication systems are increasing as an emerging area in the design patterns community [3, 4]. Much of the work focuses on the structure of communicating blocks or objects and the relationship among them. Although the patterns are valuable in system developments, our main concern exists in the message exchanges among communicating blocks because those exchanges provide the principles of protocol operation.

Another problem that we handle in this dissertation is the correctness and validation of pattern-based design. Currently, many patterns are concentrated on the design and implementation of software development without consideration of other aspects, for example, patterns for requirements analysis and testing [1].
We suggest a validation technique for a design specification developed from our patterns. Moreover, the property description in each pattern enables designers to capture the system requirements to be checked after the design.

In summary, we suggest a pattern-based development methodology for communication protocols which aims to support high-level design and validation of protocols in the early development phase.

1.1 Pattern-based Development Methodology

The classic life-cycle of software engineering is composed of several phases such as requirements analysis, design, coding, testing, and maintenance [5, 6]. Our concern lies in the initial design phase for the high-level description of a system and its validation. Designers in that phase typically want to capture

![Diagram](image-url)

Figure 1-1: Overview of pattern-based design and validation

the essential functions first; for example, flows of messages and interactions with other systems. Then, they build an abstract system before the detailed
design and implementation. For the specification of the abstract system, we propose a pattern language, a collection of patterns that work together to solve problems in a specific domain. The pattern language is categorized in two groups, structural patterns and behavioral patterns, to address overall architecture and common behavior of a protocol system. Figure 1-1 illustrates the pattern-based development methodology. The patterns are contained in the pattern repository that has an expandable structure with pattern specialization and instantiation.

After analyzing the requirements of a protocol, we devise the architecture of the protocol with structural patterns. The architecture is composed of several blocks along with communication paths between them. A block is an architectural building element of a developing system and can contain other blocks. At this point, blocks are considered to be black boxes. The external interfaces such as communication paths and messages are defined, but the internal details are not.

The execution of the abstract system is acquired in behavioral patterns after the architectural design. The behavioral patterns provide common behavior of a protocol system, focusing on the interaction between blocks. They assist developers in describing the internal behavior of blocks. Each block instance has a state that may change to another state in response to a message input. The response may also trigger additional events such as the generation of output messages. We use a communicating extended finite state machine (CEFSM) to formally describe the behavior. Predicates and timers may be used to describe conditional behavior and timing constraints. Note that we use the event, signal, and message interchangeably in this dissertation.

Protocol designers complete the abstract design of a protocol system by composing the structural and behavioral patterns. It may be necessary to revise the requirements during the design step to fix any unclear requirements. As a result of the design, we have a system design description which sketches an
abstract system of a protocol focusing on message exchanges. The description is a combination of instantiated blocks, communication paths, messages, and other components of patterns used.

After the pattern-based high-level design, we perform a validation of the specified system to validate the correctness and consistency of the design. It is important to uncover them as early as possible to prevent the errors from affecting the later phases. In this dissertation, we suggest a model checking technique for the validation of the design. Model checking is an automatic technique to verify properties of a system by investigating a model of the system [7, 8]. We selected SPIN (Simple Promela INterpreter) as our model checking tool. It was developed at Bell Labs for the analysis and validation of distributed systems, especially of communication protocols [9, 10, 11]. Furthermore, it is freely available from the SPIN website [11].

For the SPIN model checking, we first build a model of the system design description in PROMELA, an input language of SPIN, and identify requirements or properties to be checked from the description. Then, the model is simulated and verified against the properties using SPIN. Usually, the construction of a model is a challenging practice in formal validation because it is crucial to validation result and it must reflect the system to be developed exactly. We provide a translation mechanism to build a model from a pattern-based design specification. The translation is simple because of the correspondence between the pattern elements and PROMELA constructs.

Meanwhile, it is important to identify the properties of a system in the design phase so that they can be used for later testing and maintenance of the system as well as for the design validation. For this purpose, each pattern of our pattern language provides a property specification section to help to describe some properties from the pattern. Consequently properties to be checked at the validation phase
can be obtained at the design phase. The properties include the occurrence of message arrival, ordering in the message interaction, and correspondence or alternativeness of messages. If the validation fails, it is necessary to find the reason and fix the problem. Then, we return to the design step or validation step. Otherwise, refinement and further development follow.

In this dissertation, we show the implementation of a pattern in SDL (Specification and Description Language). SDL is a formal description language for communicating systems recommended by ITU (International Telecommunication Union) \[12\] and is becoming popular in design and implementation of communication protocols \[13, 14\].

### 1.2 Pattern Language Overview

Patterns are rarely standalone. Instead, several related patterns are typically used to solve problems in a specific domain. Table 1–1 presents our pattern language to be used for communication protocols. Detailed description of each pattern can be found in the following sections.

<table>
<thead>
<tr>
<th>Category</th>
<th>Patterns</th>
<th>Variants or Specialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>structural</td>
<td></td>
<td></td>
</tr>
<tr>
<td>protocol layer</td>
<td>split protocol layer</td>
<td></td>
</tr>
<tr>
<td>mux</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dynamic handler</td>
<td>split dynamic handler</td>
<td></td>
</tr>
<tr>
<td>basic CEFSM</td>
<td>predicate CEFSM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>predicate after action</td>
<td></td>
</tr>
<tr>
<td></td>
<td>merge patterns</td>
<td></td>
</tr>
<tr>
<td>timer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>repeated events</td>
<td>timed repeated events</td>
<td></td>
</tr>
<tr>
<td></td>
<td>timed retrial</td>
<td></td>
</tr>
<tr>
<td>behavioral</td>
<td></td>
<td></td>
</tr>
<tr>
<td>message transfer</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unconfirmed sender</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unconfirmed receiver</td>
<td></td>
</tr>
<tr>
<td></td>
<td>confirmed sender</td>
<td></td>
</tr>
<tr>
<td></td>
<td>confirmed receiver</td>
<td></td>
</tr>
<tr>
<td></td>
<td>timed receiver</td>
<td></td>
</tr>
<tr>
<td></td>
<td>timed retrial receiver</td>
<td></td>
</tr>
<tr>
<td></td>
<td>timed confirmed sender</td>
<td></td>
</tr>
<tr>
<td></td>
<td>timed retrial confirmed sender</td>
<td></td>
</tr>
</tbody>
</table>
is given in Section 3.1 and 3.2. Structural patterns are used to represent the abstract architecture of a protocol system which is a combination of communicating blocks, communication paths, and messages. One pattern is able to depict a hierarchical structure by nesting other patterns. In addition to the static architecture expressed in the pattern protocol layer or mux, block instances can be created dynamically using the pattern dynamic handler.

Behavioral patterns help a designer describe internal behavior of blocks identified in structural patterns. We use CEFSM to describe the behavior. One feature of the behavioral patterns is that a state transition diagram (STD) is used to represent the CEFSM. There is no standard notation for pattern description, although for object-oriented systems the Unified Modeling Language (UML) and Object Modeling Technique (OMT) are common [15,16]. We use STD because it is easy to understand and translate into our validation language PROMELA.

A pattern is represented in a particular form to describe the design problem and its solution. There are many variants in the form, but most forms typically have name, context, problem, solution, and example sections. We follow a traditional pattern description form suggested in Buschmann et al. [1], but our form exploits SDL as an implementation language and additionally presents the property specification section to aid validation of a system developed using the pattern. The following enumerates each section of the form:

- **Name**: The name of the pattern. A meaningful word or phrase has to be used to intuitively present the main purpose of the pattern.
- **Context**: A situation in which the problem of the pattern arises.
- **Problem**: General description of problem’s essence.
- **Force**: Various viewpoints and aspects of the problem that should be considered when solving the problem. It describes requirements that must
be considered in the solution, constraints of the problem or context, and
desirable properties of the solution.

- **Solution:** The generic solution to the problem. We use diagrams and CEF-
  SMs to illustrate the static and dynamic aspects of the solution.
- **Property specification:** Properties that can be captured from the pattern
  when it is used in a system design.
- **Implementation:** Guidelines or suggestions to transform the solution into
  SDL. It is usually obtained by one-to-one mapping of pattern elements.
- **Example:** Any known usages of the pattern in real applications.
- **Variants:** Slightly different solutions for similar problems or more specific
  situations.
- **See also:** Any related or similar patterns.

### 1.3 Organization of Patterns

Software reuse is a powerful discipline to create a new software system from
existing ones [17]. Although the concept is simple, it is not easy to find a reusable
artifact that provides high-level abstraction applicable to several developments.
The pattern is a good candidate for software reuse through experienced design ab-
straction instead of real source code. It is believed that design reuse has advantages
over code reuse [18].

![Organization of pattern language](image-url)

**Figure 1–2:** Organization of pattern language
Figure 1-2 shows the organization of our patterns. Note that the behavioral patterns have an expandable hierarchical structure. The pattern basic CEFSM is the foundation of behavioral specification. It is possible to represent more complex and high-level abstraction through merge patterns such as source merge, target merge, and sequential merge. Patterns can also be augmented with the pattern timer and/or repeated events to control timing constraints and repetition of events. Basic CEFSM, timer, and repeated events are called elementary patterns because other patterns are able to be made from these patterns. The current domain of our pattern language is a communication protocol and our concern is concentrated on the behavior of message exchanges between communicating blocks. The patterns of message transfer are specialization of the elementary patterns. For example, the pattern basic CEFSM has a tuple (\{S_s, S_t\}, S_s, \{e\} \cup O, \{< S_s, e, S_t, A, O >\}, V) which means that there are two states \(S_s\) and \(S_t\) with \(S_s\) as an initial state. For an input event \(e\), the pattern moves to \(S_t\) after performing action \(A\) and generating output \(O\). There is a set of variables \(V\) for the action. The formal definition of the pattern is given in Section 3.2.1. From the pattern, we can introduce a specialized event *indication* instead of a general event \(e\) so as to describe a new pattern unconfirmed receiver. The indication is used to inform a receiving entity of the arrival of a message for a request message from a sender. The result pattern has a tuple (\{\(S_s, S_t\}\), \(S_s\), \{*indication*\} \cup O, \{< S_s, *indication*, S_t, A, O >\}, V) which represents the reception of an indication message from a peer without needs of response. Furthermore, it is also possible to describe more specific situations by combining with other patterns. For example, by combing the pattern unconfirmed receiver with the pattern timer, we can get a new pattern timed receiver which is a common situation to take a message in timing constraints.

For the adaptation of patterns in a real system, patterns have to be instantiated. In other words, all states, messages, actions, timers, and predicates have
the concrete names and values before applying to the system. For example, the pattern unconfirmed receiver can be instantiated to be used in an ATM connection such as (\{\text{WAIT, EST}\}, \text{WAIT}, \{\text{AAL\_EST\_ind}\}, \{<\text{WAIT, AAL\_EST\_ind, EST, allocate resource"}, \rightarrow\}, \phi\}). Upon receiving AAL\_EST\_ind at the state WAIT, the CEFSM moves to the next state EST after performing the action *allocate resource*. There are no outputs and variables in this instantiation. Note that the indication at the pattern unconfirmed receiver has the real name AAL\_EST\_ind. As a result, the instantiated pattern is actually used at the design of a system. The concept of specialization and instantiation was introduced by Casati et al. [19] where the authors provide a formal basis for expanding their patterns to deal with the exception cases in workflow design. Structural patterns also need the instantiation such as the behavioral patterns.

Meanwhile, the elementary patterns can be used to make new patterns specific to other domains. For instance, suppose we are going to develop a pattern language for reactive systems. First, we identify and analyze recurring situations in the domain. Then, patterns for the domain are constructed on top of the elementary patterns such as the case of message transfer.

The remainder of the dissertation is organized as follows. In Chapter 2, we begin with the related work and background information on patterns, SDL, and SPIN model checker. Then, each structural and behavioral pattern of the pattern language is described in detail in Chapter 3. Next, in Chapter 4, we present the validation technique of pattern-based design which includes model construction and SPIN usage in the validation. In Chapter 5, we perform case studies on an alternate bit protocol and an ATM signaling system to show the feasibility of our methodology. We conclude in Chapter 6 with a summary of the dissertation and further research.
CHAPTER 2
RELATED WORK

2.1 Patterns in Communication Systems

This section provides the basic concept of a pattern and its relationship to our work. The notion of a pattern was made by a group of building architects who identified recurring problems in building constructions. They presented solutions to the problems in a book for other architects to reuse them in similar situations [20].

Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use this solution a million times over without ever doing it the same way twice.

Software engineers have used the idea in software developments. From the essential techniques and well-proven experience in software design and implementation, developers are able to reuse the successful practices in their further developments. Although the concept of a pattern is simple and has been known for a long time, software patterns have become popular with the introduction of the books of Buschmann et al. [1] and Gamma et al. [2]. See also the patterns home page [21] for the general introduction to the patterns.

As closely related work, Geppert and Rößler [22] and Gotzhein [23] suggested a pattern-based SDL design where a pattern pool maintains a set of SDL patterns that can be selected, adapted, and composed for a new SDL system. In fact, their research provided us with the initial motivation. The SDL patterns are SDL-oriented by presenting SDL semantics as well as SDL syntactic rules in the pattern description. Due to the SDL orientation, their approach makes it possible to generate an executable SDL specification directly from the pattern-based design. Our methodology, on the other hand, can support other formal
description techniques such as Estelle [24] and LOTOS [25] as a design specification language because our patterns are represented in a general CEFSM. Moreover, by performing a validation before the implementation, it is possible to find design errors immediately after the design.

Huni and his colleagues [26] proposed a framework called Conduits+ to address a generalized software architecture for communication protocols. The framework has two types of elements: conduits, protocol independent components, and information chunks, messages that flows through conduits. There are four kinds of conduits in the framework, that is, Protocol, Mux, Adapter, and ConduitFactory. By using a sequence of design patterns of Gamma patterns [2], Conduits+ is able to implement a network protocol with a graph of conduits and information chunks. Since the conduits provide simple and concise architectural elements for network protocols, we use them as a part of our structural patterns. Other patterns for static components of a protocol system are also found in Parssinen and Turunen [27] where the authors present common principles of a communication protocol in protocol system, protocol entity, protocol behavior, and entity interface.

We use the CEFSM to formally describe the behavior of communication protocols. The theoretical background of it is found at the book of Ellsberger et al. [14]. There are several ways to implement a finite state machine (FSM) in patterns [2, 28]. Yacoub and Ammar [28] presents a pattern language that provides a pattern system for FSMs in object-oriented design. They propose a basic FSM pattern and its extensions to handle several design issues such as state transition mechanisms, design structure, state instantiations, exposure of internal state, and the machine type, focusing on the object-oriented implementation. In our case, a CEFSM pattern is depicted in an STD so that the description is simple and straightforward. Furthermore, the CEFSM has an expandable hierarchical structure using the specialization and instantiation. The pattern specialization is a
mechanism for creating a new pattern from an existing one and the instantiation is a finalization of a pattern to adapt it in a concrete situation [19].

Meanwhile, our notation in the behavioral patterns is based on the Statecharts [29], a visual formalism for the specification of reactive systems. A transition of an STD has a label with an input event, predicates, actions, and outputs events where all fields are optional although the input event is almost always necessary. A transition of Statecharts is labeled with an event, a condition, and an action. Statecharts have more descriptive power than our STD through the use of hierarchies of states, orthogonality, and history connectors. However, several experiments such as the case studies presented in Chapter 5 show that our notation is enough for the description of communication protocols.

Faison emphasizes the importance of interaction in software with multiple processes or components [30]. Actually, interactions between communicating entities are essential in the description of behavior in many application domains. He found fundamental patterns in the interactions such as pull interaction and push interaction to describe the way the entities relate to and communicate with each other. He also provided many useful examples in real life and software components. The basic idea of the patterns is also found at our message transfer pattern.

As a similar approach in a different domain, a workflow system uses pattern concept called rule patterns in the design of exceptional handling [19]. An exceptional handling rule is represented with event, condition, action, and output, which is analogous to our behavioral patterns except the timing concept. The authors discovered frequently occurring exceptional situations in workflow design and modeled the abstract activity in an application-independent manner. They also introduced the specialization and instantiation as a formal basis for expanding their patterns.
Patterns can be used not only in software developments, but also in other areas. As an example of patterns that are not relevant to software development, Adams and his colleagues [31] presented several patterns for operations and maintenance of telecommunications systems. They described their experience and expertise for highly available fault tolerant switches. There are also several efforts made to capture patterns for domain-specific applications such as factory automation, e-business, embedded systems, and enterprise application [32, 33, 34].

Our patterns have a section named see also to provide the related patterns and relationship with other patterns. Further related work can be found in this section. For more information on patterns for communication systems, see Buschmann et al. [1], Schmidt et al. [3], and Rising [35, 4].

2.2 Model Checking using Spin and Promela

Model checking is an automated technique to validate correctness of a system by investigating a finite state model of the system [7, 8, 36]. The technique is especially useful for reactive and distributed systems that are characterized by many interactions among processes. A model checker explores all states reachable from an initial state and validates a set of correctness properties on the model.

Model checking typically starts with the construction of a model and the identification of properties to be checked. Then, it validates the properties with an appropriate model checker. A system is usually modeled using a state based description such as communication automata, CSP (Communicating Sequential Processes), Petri Nets, etc. A model must be made as closely as possible to a system so that the validation results for the model reflects the system’s execution exactly. The properties a model checker validates in a model include the reachability of a certain state, safety and liveness of a system, and relative order of events in a system [7]. Several formalisms are used to precisely express the properties. Temporal logic [37] is specifically tailored for the description of properties of system
behavior over time. The two most popular temporal logics are linear time temporal logic (LTL) and computation tree logic (CTL). Formally, a model checker validates whether

\[ M \models \Phi \]

holds, where \( M \) is a model of a system and \( \Phi \) is a property to be required in the system.

Model checking has been successfully used in hardware validation and protocol communication software. There are many tools applicable today such as SMV [38], SPIN, UPPAAL [39], KRONOS [40], etc. We select SPIN (Simple Promela INterpreter) as our model checker for the validation of pattern-based design. It was developed at Bell Labs for the analysis and validation of distributed systems, especially of communication protocols [9, 10, 11]. Furthermore, it is freely available at the SPIN web site [11]. A system is described in a modeling language called PROMELA (PROcess MEta LAnguage) [41]. Given a system model specified in PROMELA, SPIN can simulate the system’s execution. It is also possible to generate a C program to perform an exhaustive exploration of a system’s state space using a depth-first search algorithm. During the simulation and validation, SPIN can check for absence of deadlocks, non-progress cycles, unspecified message receptions, un-executable code, etc. The model is also proven the correctness of system invariant claims and temporal claims expressed in next-time free LTL formulae. If a system model violates a correctness property, SPIN recognizes this and a trace of violation is generated. To cope with the problem of state space explosion, SPIN employs several techniques such as partial-order reduction, state-vector compression, and bit-state hashing. For the simple usage of the tool, a graphic user interface, Xspin, is also provided.

PROMELA is a validation modeling language for SPIN, focusing on the abstraction of message exchange in a system [42, 43]. It has C-like syntax with features
from Dijkstra’s guarded commands and communication primitives from Hoare’s CSP. The important constructs of a PROMELA program are process, message channel, and variable. Processes execute asynchronously, which means there are no assumptions on the relative speed of processes and only one process is executed at a particular point. Each statement in a process is either executable or blocked. If a statement is blocked for some reason, the statement halts until it becomes executable. Processes interact either by message passing via channels or memory sharing of global variables. Communications via message channels are either synchronous (i.e., rendezvous) or asynchronous (i.e., buffered).

The following example [42] presents a sample PROMELA program to show the basic concept of PROMELA. It calculates the factorial value of a number \( n \) at the process `fact`, returning the result through the channel `p`.

```proctype
proctype fact(int n; chan p) {
    int result;
    if
        :: (n <= 1) -> p!1
        :: (n >= 2) ->
            chan child = [1] of {int};
            run fact(n-1, child);
            child?result;
            p!n*result
    fi
}

init {
    int result;
    chan child = [1] of {int};
    run fact(5, child);
    child?result;
    printf("result: %d\n", result)
}
```

A process is defined by a `proctype` with process behavior in it. A special process `init` usually instantiates other processes by using the `run` operator. Channels defined by `chan` are used to transfer data from one process to another. The statement `p!1` means the sending of the constant one through the channel `p`. To receive a value or a message from the head of a channel, a receive statement
expressed in the symbol ? is used such as `child?result`. The channel in the example can store up to one integer value because the size of each channel is declared as one. For synchronous communication, the channel size has zero.

There are three kinds of control flow constructs in PROMELA namely, selection, repetition, and unconditional jumps. The `if` selection contains several execution sequences, each preceded by a double colon. A sequence can be selected only if its first statement is executable. The first statement is therefore called a guard. If none of the guards of the statement is executable, the construct blocks. In the above example, the `if` construct has two sequences. Note that only one sequence can be selected among the sequences. The repetition construct `do` conducts the same mechanism as `if`. But it repeats the construct until it meets the `break`. Another way to terminate the repetition is to jump to a label outside of the statement with `goto`. A label identifies a unique control state and can appear before a statement.

By prefixing a sequence of statements enclosed in parentheses with the keyword `atomic` a user can indicate that the sequence is to be executed as one indivisible unit, that is non-interleaved with any other processes. Meanwhile, SPIN is able to express the correctness properties in the PROMELA statements such as `assert`, `end label`, `progress label`, `accept label`, and `never` claims [44].

The following shows a result of validation using SPIN for the example PROMELA code.

(Spin Version 3.4.16 -- 2 June 2002)
+ Partial Order Reduction

Full statespace search for:
  never-claim - (not selected)
  assertion violations - (disabled by -A flag)
  cycle checks - (disabled by -DSAFETY)
  invalid endstates +

State-vector 124 byte, depth reached 27, errors: 0
28 states, stored
0 states, matched
Internally, SPIN maintains three key data structures [45]: state-vector, depth-first stack, and seen set. The state-vector shows the size of a state which is composed of the value of local and global variables, control flow location of each process, and the contents of message channels. In the example, each state occupies 124 bytes. The depth reached field represents the deepest stack depth reached during the depth first search of the state space. The seen set holds the states already explored during the search. Thus, the stored means the number of states stored in the seen set. The matched is the number of states that were already found in the seen set.

There is a lot of research that exploits SPIN and PROMELA for the validation of communication protocols [46, 44, 47, 48]. Kamel and Leue [47] presented the modeling and validation of the General Inter-ORB Protocol (GIOP) using SPIN. This paper provides useful examples in the tool usage. The authors elicit ten high-level requirements from the text-based specification and formally express them in LTL for the validation. Property extraction from a system requirements specification is common in practice. Note that our approach to capture the system properties from a pattern does not enumerate all potential properties of a system. Nonetheless, this method provides a systematic way to elicit the part of properties of the target system.
D'Argenio and his colleagues [46] described the transfer of files on a lossy communication channel using a bounded retransmission protocol. They present a specification of the protocol in a network of timed automata and a list of properties of the protocol with respect to inputs and outputs. SPIN and UPBAAAL are used for the checking of the properties and shows the importance of the real-time aspects in the protocol validation. For more experience and usages of SPIN in protocol validation and other applications, refer the proceedings of SPIN workshops [11].

Our design provides a visual description of a system in the composition of patterns. In fact, using the visual specification such as STD is a common way to describe a system abstraction before the formal PROMELA modeling [49,50]. Leue and Holzmann [51] suggested v-Promela, a visual and object-oriented modeling language for SPIN. The language is designed to present abstraction and hierarchical layering. The visual notation is able to express both structure and behavior of a reactive concurrent system to acquire software architecture at the early life-cycle stages. They use UML for Real-Time (UML-RT) notation for structural description and adapt many important ideas from Realtime Object-Oriented Modeling (ROOM). Behavior is specified using hierarchical communicating extended finite state machines (HCEFSMs). This paper suggests a translation mechanism from the visual constructs to a PROMELA program. The visual notation is supported by a graphical tool, VIP. On the other hand, Mikk et al. [52] have translated Statecharts into PROMELA using extended hierarchical automata as an intermediate format. Their technique allows a system described in Statecharts to be validated in a linear temporal logic model checker.

2.3 Introduction to SDL and Model Checking SDL System

Our methodology proposed in this dissertation was originally aimed at the initial design of a communication system to be implemented in SDL. SDL skeleton code which outlines the system architecture and behavior could be generated after
the pattern-based design and validation. In this section, we present the basic concept of SDL which is related to our techniques. Further information on SDL is available at [13, 14, 53]. Meanwhile, our pattern repository can be combined with other SDL development methodologies such as SDL+ [54] as a library of reusable artifacts.

SDL is an object-oriented formal language recommended by the ITU Telecommunication Standardization Sector (ITU-T) for the precise and unambiguous specification of event-driven and reactive systems, in particular, communication systems [12]. It describes structure, behavior, and data of a system in formal notation.

The static structure of an SDL system is described by hierarchical blocks. A block can contain other blocks, resulting in a tree structure. A leaf block is made up of one or more processes. Channels and signal routes are used to convey signals between the structural elements. Processes are connected with each other and to the boundary of the nesting block by signal routes. Blocks are connected together by channels. If many signals are transferred on the same channel or signal route, their ordering is preserved. In addition to the static process, the SDL process can also be created dynamically.

The behavior of a system is described by a set of autonomous and concurrent processes. A process is an extended finite state machine, communicating asynchronously with other processes by signals. Each process has an implicit unbounded FIFO input queue where signals are buffered on arrival. Signals are extracted from the input queue in the order of arrival. An expected signal triggers a transition, and the process executes a set of tasks such as an assignment of a value to a variable, a procedure call, and a signal output. After initiating a transition, a signal is removed from the input queue. Each process has a unique address. A signal always carries the address of the sending and the receiving processes. The
destination address may be used if the destination process cannot be determined statically and the address of the sending process may be used to reply to a signal.

In SDL, variables are owned by a specific process and cannot be modified by other processes. The synchronization between processes is achieved using the exchange of signals or remote variables.

Many researchers have investigated the simulation and validation of SDL systems using Spin/Promela \[55,56,57,58\]. Holzmann and Patti \[56\] introduced an experimental validation tool, supertrace, for the validation of an SDL specification. In fact, the tool is an ancestor of Spin and used for AT&T’s 5ESS© switching system with a practical size. They also presented some consideration in the usage of the tool such as closeness of a model and handling of time expiration that are still open questions in current SPIN version.

Bozga and his colleagues \[59\] suggested a validation toolset with an intermediate representation called IF for distributed software systems. The IF validation environment provides a translator from SDL to IF sdl2if so that a system described in IF is able to be validated in several different validation tools such as CADP \[60\], Kronos \[40\], and Tgv \[61\]. Consequently, the environment makes it possible to support several different techniques for one system validation.

Sidorova and Steffen suggested a validation methodology for a large-scale SDL system in which an SDL specification is transformed to a Promela model using the existing tools such as sdl2if, LIVE, and if2pml \[57\]. As the size of a system to be model checked is limited, they use a bottom-up compositional technique which starts from small components and then composes them for larger ones.

Tuominen \[58\] provided a translation mechanism from an SDL-88 dialect to Promela. Bosnacki and his colleagues \[55\] have extended Spin to validate the timing properties of the SDL model. The resulting tool DTSpin allows to quantify the time elapse between events. Meanwhile, Prigent et al. \[62\] have suggested a
validation of the SDL program with save operator which is not handled in other literature. They extended the tool if2pml to get a PROMELA code from IF with save.

One feature in their attempts is that they extract an abstraction model from a completely developed SDL system. Typically, a model from a final system suffers from a state space explosion and cannot be validated as a whole due to the size of the system. As a result, the system is split into several small components and performs a validation from each component [55,57]. In our case, we design a system in high-level from the patterns and obtain a validation model from it. Thus, the model needs relatively small number of states for the validation. Additionally, we can extract several properties of the system during the design phase. Accordingly, the properties can be used at the testing phase after the implementation of the system as well as the validation for the design.

Meanwhile, the patterns discussed in this paper are specific to the communication protocols. However, SDL is not always used for communication systems. It is possible for many SDL systems to be developed using the patterns popular in other domains. Worm [63] presented SDL implementation of the classic patterns such as Observer and Factory. The examples in this paper guide the SDL users how to apply the patterns in an SDL system.
CHAPTER 3
PATTERN LANGUAGE FOR COMMUNICATION PROTOCOLS

3.1 Structural Patterns

3.1.1 Protocol Layer

3.1.1.1 Context

We need to design a complex system such as a communication protocol. Some parts of the system may already exist.

3.1.1.2 Problem

How can we describe the structure of a communication protocol system?

3.1.1.3 Forces

The system is too large and complex to be understood completely. We need techniques to help manage the complexity.

- Decomposition: We decompose the system into several subsystems that can be dealt with more-or-less independently.
- Abstraction: Each subsystem is treated as a black box and specifies the interfaces with other subsystems. Internal details are left for later.
- Reuse: Some of the subsystems may already be available and thus do not need to be designed.

3.1.1.4 Solution

A communication protocol can be designed in layers. Each layer handles problems at a particular level of abstraction. A layer offers services to the higher layer and uses services from the next lower layer. This structure allows a protocol developer to design external interfaces before internal functionality [64, 9].
First, we identify the blocks belonging to a layer and determine the communication paths between the adjacent layers. The layers and communication paths are logical objects that may or may not correspond directly to physical components of network or communication links.

Second, we address the messages between layers and associate the messages with the communication paths. The communication paths show the list of message types that can be sent on the path and the direction of the message flow. Figure 3–1 shows a protocol layer that includes one block, four communication paths, a message list, and the adjacent layers. The internal behavior of the protocol layer block can be designed using other patterns of this pattern language.

### 3.1.1.5 Variant: split protocol layer

Often, a communication layer can be conceptually split into two related functions such as sending and receiving for message transfer [65]. It may be helpful for the designer to consider the two functions separately. Figure 3–2 shows a
structure of the pattern split protocol layer where the *Outgoing* block initiates a communication requested from the upper layer and the *Incoming* block handles messages coming from the lower layer.

### 3.1.1.6 Implementation

The SDL implementation can be obtained directly from the design. SDL has two constructs to describe a system structure: SDL blocks and SDL processes. SDL blocks are pure structuring mechanisms that may contain other blocks and processes while SDL processes contain the specification of behavior.

Typically, the non-leaf blocks of the structural patterns are mapped to SDL blocks and the communication paths between them are mapped to SDL channels. Two types of channels are possible in SDL. One is a delaying channel and another is a non-delaying channel. The selection of the channel is dependent on the situation. Leaf blocks are mapped to SDL processes. In this case, the communication paths are mapped to signal routes which connect processes to other processes and to the channels of their containing block. Messages flowing on communication paths that are mapped to channels or signal routes are called signals in SDL.

Figure 3-3: SDL implementation of pattern protocol layer

Figure 3–3 shows an implementation of the pattern protocol layer of Figure 3–1. The *Protocol Layer* of Figure 3–1 is mapped to an SDL block *ProtocolLayer*. Each communication path is converted to a delaying channel such as $c_1$, $c_2$, $c_3$, and $c_4$. The signals correspond to the messages flowing through the channel.
3.1.1.7 Examples

The example shows a part of the Service Specific Coordination Function (SSCF) for the User-Network Interface (UNI) in the ATM signaling system [66]. The SSCF\_UNI layer provides a mapping function between UNI Signaling layer and Service Specific Connection Oriented Protocol (SSCOP) layer. The basic structure of SSCF\_UNI is an example of the pattern protocol layer as Figure 3–4.

Three kinds of messages such as \textit{AAL\_EST}, \textit{AAL\_REL}, and \textit{AAL\_DATA} are exchanged with the upper layer through the communication paths \textit{UNI2SSCF} and \textit{SSCF2UNI}. \textit{AAL\_EST} is used to establish a connection from the UNI signaling layer in the form of a request and a confirmation such as \textit{AAL\_EST.req} and \textit{AAL\_EST.conf}. It also informs the upper layer that an incoming connection has been established by an indication message \textit{AAL\_EST.ind}. \textit{AAL\_REL} is used to release the connection established. \textit{AAL\_DATA} is used by the upper layer to send a data packet in a request form \textit{AAL\_DATA.req}. \textit{SSCF\_UNI} hands out a received packet to its user in an indication form such as \textit{AAL\_DATA.ind}. The lower interface has similar messages which uses the communication paths \textit{SSCF2SSCOP} and \textit{SSCOP2SSCF}.

As an example of the pattern split protocol layer, we demonstrates a variation of alternating bit protocol (ABP) [67] which provides simple but reliable message...
transfer on a lossy lower layer. Figure 3–5 illustrates the architecture of the

![Diagram of ABP using split protocol layer](image)

Figure 3–5: Structure of ABP using split protocol layer

protocol. The upper interface uses two messages *put* and *get* for a reliable data transmission. The lower interface uses messages *data_req*, *data_ind*, *ack_req*, and *ack_ind* to send and acknowledge messages, allowing for retransmission if necessary to deal with message loss.

### 3.1.1.8 See also

Layers [1], Pattern Half Object + Protocol [68], Protocol Conduit [26], Service Architecture [23], Service Provider Refinement [23], SDL and Layered Systems [69]

### 3.1.2 Mux

#### 3.1.2.1 Context

In a layered design, there may be multiple blocks in an adjacent layer and resolving the destination or source of the messages is required.

#### 3.1.2.2 Problem

How can we resolve the destination or source of a message?

#### 3.1.2.3 Solution

Use a table to map an instance of a block to its address. Figure 3–6 describes the structure of a mux layer where the upper layer has several blocks, while the lower layer has one block. Similarly, the pattern is applicable to the reverse situation.
3.1.2.4 Implementation

Figure 3–7 shows an implementation of the pattern mux where the *Mux_Block* is an SDL process, and communication paths such as *r₁*, *r₂*, *r₃* are signal routes. The process array, *Mux_Table* shows a trivial implementation of the address resolution. It stores every instance identifier, *PID*, with the key *name*. The signals enumerate the messages flowing through the signal routes. Note that the signal routes of the upper interface are joined in one point to be connected with the outside channel.

3.1.2.5 See also

mux conduit [26]

3.1.3 Dynamic Handler

3.1.3.1 Context

A block needs to handle multiple communications at the same time. The expected load will be variable and the system is appropriately sized to handle it.
3.1.3.2 Problem

How can a block be organized internally to service multiple communication requests?

3.1.3.3 Forces

- *Concurrent processing*: To provide a good response time, requests should be processed concurrently.

- *Capacity*: Handling concurrent requests imposes overhead for context switches, resource contention, etc. If too many requests are handled at the same time, the overhead will dominate the computation and the system performance will degrade unacceptably. Therefore the amount of concurrency should be bounded. Finding the optimal bound may be difficult to determine.

- *Static vs dynamic handler creation*: A communication request is serviced by a handler, an instance of a block servicing the request. An important design decision considers when the handler instances should be created. The static approach creates all handlers at the system start-up time and their lifetime extends until the system is shut down. When not servicing a request, the handlers are idle, but still utilizing system resources. On the other hand, an idle handler may be quickly deployed to handle a new request with little overhead.

The dynamic approach creates a handler upon a request and its lifetime spans only as long as necessary to service the request. This approach incurs overhead for handler creation and termination, but there are no idle handlers to unnecessarily consume system resources. Thus the resources used by request handling adapt to the load.

A static approach is useful when the system load is uniformly high. A dynamic approach is better when the system is appropriately sized and the
load is variable. Hybrid approaches combining aspects of static and dynamic handling are also possible.

3.1.3.4 Solution

The context of this pattern leads us to choose a dynamic approach. The solution utilizes a single instance of a static block, Admin, (an example of the Singleton Pattern) which is created at system startup time. The instance waits for a communication request message from adjacent layers. Upon arrival of the request message, Admin serves as a factory and dynamically creates an instance of the block Handler. The Handler instance then services that communication. After the communication has been serviced, the instance is terminated. The maximal number of instances of Handler that can be created is bounded by a fixed number N. If there are more requests than N, Admin must either queue the message, or more typically reject the request. N is determined empirically, or by performance analysis using the expected load on the system.

Figure 3–8: A structure of pattern dynamic handler

Figure 3–8 shows a structure of the pattern. First, the entity Admin waits for a request through the communication paths either p2 or p6. Upon receiving a message, for instance msg2 through p2, the Admin creates an instance of Handler block giving the necessary information for communication, for instance, the address of the requester. The dotted line from Admin to Handler means the creation of an instance. After being created, the instance starts communication through the paths p3, p4, p8, and p9. When the communication ends, the Handler instance informs.
the Admin of termination of service by sending, for example, a message msg5 and ceases to exist.

3.1.3.5 Variant: split dynamic handler

The pattern dynamic handler considers only one type of handler. In communication systems, however, it is common to have handlers in pairs such as the pattern split protocol layer. Figure 3–9 describes a dynamic creation of two type handlers,

![Diagram](image)

Figure 3–9: A structure of pattern split dynamic handler

*Outgoing Handler* and *Incoming Handler*. The block Admin creates one of them depending on the type of a request. All other behavior is similar to the pattern *dynamic handler*. Note that the two handlers may need internal communication between them.

3.1.3.6 Implementation

For the implementation of pattern *dynamic handler*, developers must consider both the structure and the behavior of the blocks in the pattern. Figure 3–10 shows the structure of the pattern where two processes, Admin and Handler, exist with the initial and maximum number of instances. The process Admin has one instance during its life span, while the process type Handler has no instance at startup time and can have the maximum $N$ instances. The processes are connected to the boundary of the block with signal routes which will interact with outside channels. The behavior of the pattern needs the creation and termination of an SDL process instance.
Figure 3–10: SDL implementation of pattern dynamic handler

3.1.3.7 Examples

Figure 3–11 shows a simplified file transfer server using the pattern dynamic handler. In this example, we do not present the interface with the lower layer for simplicity. The server is composed of a block FTP_Admin and a block FTP_Handler. When a user tries to download a file, an event FTP_connect goes to the block FTP_Admin indicating a file transfer trial. The block creates an instance of FTP_Handler to make it possible for the user to download a file from the server. The instance sends a message FTP_connect_ok to indicate that it is ready to receive a command. For the command get with a file name wanted, the FTP_Handler provides the requested file with the message success. After getting the file, the user sends a disconnect message which makes the instance stop after sending the message terminate to the FTP_Admin.

Figure 3–11: A file transfer server using pattern dynamic handler

As an example of split dynamic handler, Figure 3–12 shows a simplified version of call control block composed of Call_Admin, Outgoing_Handler, and Incoming_Handler in a switching system. When a calling party tries a call, a
message $H_{init}$ goes to the block $Call_{Admin}$ indicating that there is a call request from a calling party. The block creates an instance of the block $Outgoing_{Handler}$ in order to make the instance manage the calling party. After generating a message $L_{init}$, the instance waits for a call connection from a called party.

![Call Control Block Diagram](image)

Figure 3–12: A call control block using pattern split dynamic handler

On the other hand, if the block $Call_{Admin}$ receives a message $L_{alert}$ implying that there is an incoming call, it creates an instance of the block $Incoming_{Handler}$ to setup a call connection with the called party. A message $H_{alert}$ is used to indicate a new call is coming. When the called party answers, the messages such as $H_{answer}$, $L_{answer}$, $L_{complete}$, and $H_{complete}$ are transferred.

3.1.3.8 See also

DynamicEntitySet [70], ConduitFactory [26], Pattern Half Object + Protocol [68]

3.2 Behavioral Patterns

3.2.1 Communicating Extended Finite State Machine

3.2.1.1 Context

Many communication systems react to events coming from outside environments. The systems can be modeled by distinct states and transitions. When a system receives an event, it moves from its current state to a new state while performing some actions and providing output signals.

3.2.1.2 Problem

How can we describe the behavior of a communication system?
3.2.1.3 Forces

- **Understandability**: The notation should capture the most important aspects of the behavior of a system in a way that is convenient to express and can be easily understood by a reader.
- **Completeness**: The notation should be able to express all the important aspects of a design.
- **Definedness**: The notation should be well-defined, preferably standardized. A formally defined notation allows the possibility of tool support for analysis, simulation, verification, code generation, etc.
- **Scalability**: The notation should remain tractable for systems with large numbers of states.
- **Ease of implementation**: The formalism should represent the system’s behavior in a way that can be easily mapped into an implementation language.

3.2.1.4 Solution

The behavior of a communication system can be described using a communicating extended finite state machine (CEFSM). A CEFSM is a finite state machine extended with local variables and parameterized communication events indicating communication with another CEFSM [14, 71]. These state machines are very familiar to computer scientists and engineers and meet the criteria discussed in the forces section. In particular, the local variables help with the scalability problem by allowing, for example, an 8-bit counter to be represented by one variable instead of 256 states. Other state based formalisms [29] may provide better support for modularizing large designs at the expense of a more complex notation. A survey of other formalisms that can be used for telecommunication systems design can be found in [72].

When an event is initiated by the environment, the system updates local variables, emits output events and transitions to a new state.
Definition 1 (CEFSM) A CEFSM is a 5-tuple $(S, s_0, E, f, V)$ where

- $S$ is a set of states
- $s_0$ is an initial state
- $E$ is a set of events with their parameter lists
- $f$ is a state transition relation
- $V$ is a set of local variables along with their types and initial values, if any.

For a state, an input event, and a predicate composed of a subset of $V$, the state transition relation $f$ has a next state, a set of output events and their parameters, and an action list describing how the local variables are updated.

As an example, see the following CEFSM:

$$CEFSM = (\{S_1, S_2, S_3, S_4\}, S_1, \{\text{init}, e_1(p_1), e_2, o_1, o_2(p_2, p_3)\}, f, \{x\}),$$

where $f$ has the four elements

$$\begin{align*}
&\langle S_1, \text{init}, S_2, (x := 0), \{o_1\} >, \\
&\langle S_2, e_1(p_1), S_3, (x := x + p_1), \{o_2(x, p_1)\} >, \\
&\langle S_2, e_2, S_4, (\text{"encoding } e_2\"), \{\} >, \\
&\langle S_3, e_2[x = 8], S_4, (\text{\text{\text{\text{-}}}}), \{\} >
\end{align*}$$

This CEFSM has four states, five events, and one integer variable. The input event $e_1$ has a parameter $p_1$, and the output event $o_2$ has two parameters, $p_2$ and $p_3$. The four transitions among the states are represented by the relation $f$. For simplicity, we do not give the types of variables and parameters.

As an internal event, we assume an init event to indicate the start-up signal of the CEFSM. The tuple element $\langle S_1, \text{init}, S_2, (x := 0), \{o_1\} >$ of relation $f$ denotes a transition that moves from $S_1$ to $S_2$ while assigning zero to the variable $x$ and generating the event $o_1$ after the initial signal init. The action list can include the brief activities during the transition in plain English as well as a variable update.
For example, “encoding $e_2$” implies that the machine will encode the $e_2$ received. The actions will be refined in the later development phases.

As a precondition of a transition, we introduce a predicate, a boolean valued expression of the local variables [14]. Upon receiving an event, the CEFSM evaluates the predicate. If the predicate holds, the CEFSM executes the transition. However, if the predicate does not hold, the CEFSM ignores the event and stays at the current state. An empty predicate is defined to be shorthand for true. In the previous example, the transition from $S_3$ to $S_4$ has a predicate $x = 8$.

In this pattern language, we usually represent a CEFSM with a state transition diagram (STD), a directed graph whose vertices correspond to states and whose edges correspond to transitions. Figure 3–13 shows an STD of the previous example. Each state is represented by a circle, and the initial state has a double circle. Each transition is labeled with an input event, action list, and output events. It is denoted by input(parameters)/predicate]/actions/outputs(parameters). The ‘－’ symbol in a transition indicates that there is no corresponding field. Transitions that do not alter the state are represented by an arc that points to itself.

Typically, a complex communication system is designed with a large number of states and transitions. A CEFSM can be expanded by merging with other CEFSMs.

---

**Figure 3–13**: An example CEFSM represented in STD
Definition 2 (Merge of CEFSMs) Let $M_1 = (S_1, s_1, E_1, f_1, O_1, V_1)$ and $M_2 = (S_2, s_2, E_2, f_2, O_2, V_2)$ be two CEFSMs. $M_1 \oplus M_2$, merge of the two CEFSMs, creates a new CEFSM $(S, s_0, E, f, O, V)$ such that

- $S = S_1 \cup S_2$. $S$ is a union of $S_1$ and $S_2$.
- $s_0 = s_1$, the initial state of $M_1$.
- $E = E_1 \cup E_2$
- $f = f_1 \cup f_2$
- $O = O_1 \cup O_2$
- $V = V_1 \cup V_2$

The merge is formed by using the union operation between sets.

3.2.1.5 Pattern: basic CEFSM

This is the basis of the pattern language and is composed of a source state and a target state. The transition relation $f$ has only one element such as

$$basic\ CEFSM = \{<S_s, e, S_t, A, O>\}$$

Note that the source state $S_s$ and target state $S_t$ could be the same state. Figure 3–14 shows the STD of the pattern basic CEFSM.

![Figure 3–14: Pattern basic CEFSM](image)

**Property specification.** In the pattern basic CEFSM drawn in Figure 3–14 (a), the input event $e$ has to occur eventually to initiate the transition. In the case of Figure 3–14 (b), it is necessary for the input event to occur infinitely often to repeat the transition.
- Property: The input event $e$ has to occur eventually.
  - LTL: $\Diamond e$

- Property: The input event $e$ occurs infinitely often.
  - LTL: $\Box \Diamond e$

After the occurrence of the input event, the output event $O$ has to respond to the input so that the transition completes.

- Property: The input event $e$ is followed by the occurrence of output event $O$.
  - LTL: $\Box (e \rightarrow \Diamond O)$

It is important to note that the latter property does not consider the correspondence between the input and output events. Usually, one needs input and output events to alternate. This property can be captured by introducing a global variable $n_c$ initialized to zero to measure the number of events happened [47]. The counter is increased when an input event $e$ happens, whereas it is decreased if an output event $O$ occurs.

- Property: The input event $e$ and output event $O$ occur alternatively.
  - LTL: $\Box (0 \leq n_c \leq 1)$, where $n_c$ initialized to zero is increased after $e$ and decreased after $O$.

Meanwhile, if the state $S_s$ is the initial state of a whole system, the input event $e$ is the first event to have happened in the system. Accordingly, other events cannot proceed the event.

- Property: The event $e$ proceeds all other events $e_o$. None of the available events can happen before $e$.
  - LTL: $\Box (\neg e_o) \lor (\neg e_o \ U e)$

Until now, the properties specified above apply to the entire computation. However, we can constraint the period in which the properties would apply. For example, in Figure 3–14 (a), the input event $e$ must happen between the state $S_s$ and $S_t$. 
• Property: The event $e$ must occur between the states $S_s$ and $S_t$.
• LTL: $\Box((S_s \land \neg S_t) \rightarrow (\neg S_t \lor (e \land \neg S_t) \lor \Box \neg S_t)))$

In addition to the existence property, it is also possible to specify the absence property of an event.

• Property: The event $e$ must not happen before the state $S_s$ and also after the state $S_t$.
• LTL: $(\Diamond S_s \rightarrow (\neg e \lor S_s)) \land (\Box (S_t \rightarrow \Box (\neg e)))$

Since every pattern of our pattern language is based on the pattern basic CEFSM, the properties mentioned in this pattern are also applicable to other patterns. It is worth noting that the properties presented in this section are not a complete set of properties possible in the pattern. The key idea is to capture and deduce the properties required in a system from the visual pattern.

3.2.1.6 Variant: predicate CEFSM

As we mentioned at the definition, a CEFSM can have predicates to control the behavior of the CEFSM [14]. The predicate indicates under what condition the action will be taken. By adding a predicate to the pattern basic CEFSM, we can get the variant predicate CEFSM. Meanwhile, an input event usually has several predicates as a decision point. We, therefore, define the pattern having several predicates.

$$\text{predicate CEFSM} = \{ \langle S_s, e[predicate_1], S_{t1}, A_1, O_1 \rangle, \langle S_s, e[predicate_2], S_{t2}, A_2, O_2 \rangle, \ldots, \langle S_s, e[predicate_n], S_{tn}, A_n, O_n \rangle \}$$

Note that the source state $S_s$ and target states could be the same state. If the predicates are mutually exclusive, then the CEFSM is deterministic. Figure 3–15 shows the STD of the pattern predicate CEFSM.
Property specification. The input event $e$ has to occur eventually and at least one of the predicates must hold at that time to initiate the transition.

- Property: The input event $e$ happens eventually and at least one of the predicates holds.
- LTL: $\Diamond(e \land (\text{predicate}_1 \lor \cdots \lor \text{predicate}_n))$

Furthermore, the output event corresponding to the holding predicate has to respond to the input event.

- Property: The predicate $\text{predicate}_i$ which holds at event $e$ is followed by a corresponding output event $O_i$ where $1 \leq i \leq n$.
- LTL: $\Box(((e \land \text{predicate}_1) \rightarrow \Diamond O_1) \lor \cdots \lor ((e \land \text{predicate}_n) \rightarrow \Diamond O_n))$

In addition to these two properties, it will also be necessary to consider the properties presented at the pattern basic CEFSM from which the new properties are deducible.

3.2.1.7 Variant: predicate after action

At the pattern predicate CEFSM, an input event is followed by predicates to decide the next transition. However, in some cases decisions need to be made after performing some actions. In other words, after performing a sequence of actions for an input, an instance decides its next transition based on the result of the actions.
The instance therefore needs predicates after the actions.

\[ \text{predicate after action} = \{ < S_s, e, S'_s, A_s, O_s > \]
\[ < S'_s, [\neg \text{predicate}_1], S_{t1}, A_1, O_1 > \]
\[ < S'_s, [\neg \text{predicate}_2], S_{t2}, A_2, O_2 > \]
\[ \ldots \]
\[ < S'_s, [\neg \text{predicate}_n], S_{tn}, A_n, O_n > \} \]

Note that the transitions from \( S'_s \) do not have event fields. Figure 3–16 shows the STD of the pattern predicate after action. In fact, this pattern is a sequential

![Pattern predicate after action](image)

Figure 3–16: Pattern predicate after action

merge between the transition from \( S_s \) to \( S'_s \) and the predicate CE FSM. Refer to the pattern sequential merge.

**Property specification.** First, the input event \( e \) must occur. Then at least one of the predicates has to hold in the future.

- Property: The input event \( e \) must happen eventually and at least one of the predicates has to hold after the event.
- LTL: \( \Diamond(e \land \Diamond(\text{predicate}_1 \lor \cdots \lor \text{predicate}_n)) \)

Another property of the CE FSM is an occurrence of an output event following the input and holding predicate.
- Property: The input event $e$ is followed by a holding predicate $\text{predicate}_i$ and the corresponding output event $O_i$.
- LTL: $\Box(e \rightarrow \Diamond((\text{predicate}_1 \land \Diamond O_1) \lor \cdots \lor (\text{predicate}_n \land \Diamond O_n)))$

### 3.2.1.8 Variants: sequential, source, and target merges

Typically, a complex communication protocol is made by composing several CEFSMs to fulfill the required functionality. There are three common types of merge: sequential merge, source merge, and target merge. To introduce these patterns, we classify a state in a CEFSM into either a terminal state or a nonterminal state. A state is called a terminal state if it is not used as a source state in a transition of the CEFSM. Otherwise, it is a nonterminal state.

A sequential merge is a merging of a terminal state of a CEFSM with a nonterminal state of another CEFSM. Figure 3–17 (a) shows the merging of states $S_2$ and $S_3$. The combined CEFSM goes to state $S_4$ through the state $S_{23}$. This situation is common when a system handles a sequential input from the environment.

![Figure 3–17: Merge patterns combined from two CEFSMs](image)

The pattern source merge combines each nonterminal state of two CEFSMs. In Figure 3–17 (b), the initial states, $S_1$ and $S_3$, are combined, and the resulting CEFSM has three states and two transitions. This behavior commonly occurs in a state receiving several potential input events.
The pattern target merge is obtained by combining each terminal state of a
CEFSM. This is usual when two transitions want to stay at the same state after
each transition. Figure 3–17 (c) shows a typical example.

Property specification.} The pattern sequential merge is a combination of
two basic CEFSMs in sequence. Thus, the input events happen in order.

• Property: The input event $e_1$ has to occur eventually to initiate the CEFSM.
  LTL: $\Diamond e_1$

• Property: The input event $e_1$ is followed by the events $O_1$, $e_2$, and $O_2$.
  LTL: $\Box(e_1 \rightarrow \Diamond(O_1 \land \Diamond(O_2)))$

Frequently, it is sufficient to check that the initial input $e_1$ is followed by the final
output $O_2$ such as

• Property: The input event $e_1$ is followed by the output event $O_2$.
  LTL: $\Box(e_1 \rightarrow \Diamond O_2)$

Furthermore, the input $e_2$ cannot happen before the input $e_1$ because $e_1$ is the first
event of the CEFSM.

• Property: The event $e_1$ proceeds the event $e_2$.
  LTL: $\Diamond e_1 \rightarrow (\neg e_2 \cup e_1)$

Note that the second property does not consider the correspondence between each
event. These events have to happen alternatively. This property can be captured
by introducing the global variables $n_1$, $n_2$, and $n_3$ which are initialized to zero. The
variables count the number of events happened \[47\]. The counters are increased
when the events $e_1$, $O_1$, and $e_2$ happen, while they are decreased if the events $O_1$, $e_2$, and $O_2$ occur.

• Property: The events $e_1$, $O_1$, $e_2$, and $O_2$ occur sequentially and alternatively.
  LTL: $\Box((0 \leq n_1 \leq 1) \land (0 \leq n_2 \leq 1) \land (0 \leq n_3 \leq 1))$, where the counters
  are increased when the events $e_1$, $O_1$, and $e_2$ happen and decreased when the
events $O_1$, $e_2$, and $O_2$ occur.
For the case of pattern source merge, the two inputs have to occur to check both the cases.

- Property: The input events $e_1$ and $e_2$ have to occur eventually to initiate the CEFSM.
- LTL: $\Diamond e_1 \land \Diamond e_2$

### 3.2.1.9 Implementation

The basic CEFSM is implemented in SDL by a mechanical one-to-one mapping from its STD. Figure 3–18 shows an SDL diagram fragment for the basic CEFSM of Figure 3–14 (a). Each state of the CEFSM is converted to the corresponding SDL state. A transition is represented by an input signal, a task for the action, and an output signal. When an instance of the pattern receives an input event $e$ with its parameters in the state $S_s$, it performs the task $A$ and generates output signal $O$. Note that if the transition expresses an initialization with the $\text{init}$ internal event, the source state has to be translated to a start symbol. The internal event is not shown at the SDL implementation.

The implementation of pattern predicate CEFSM leads to an SDL decision symbol. Figure 3–19 shows an SDL diagram fragment for Figure 3–15 where there are $n$ predicates for an event $e$. Upon receiving an event $e$, the CEFSM checks the predicates and then performs actions for a true predicate. The pattern predicate after action is similarly implemented as Figure 3–19 with a decision symbol after...
the actions. It is important to note that the state $S'_s$ of Figure 3–16 is not shown in the SDL diagram.

\[ S_{t1} \leftarrow \text{predicate}_1 / A_1 / S_{tn} \]

\[ S_{tn} \leftarrow \text{predicate}_n / A_n / O_n \]

\[ \Rightarrow \]

\[ S_s \leftarrow (\text{predicate}_1) \leftarrow (\text{predicate}_n) \]

\[ e \leftarrow A_1 \leftarrow O_1 \]

\[ \ldots \]

\[ e \leftarrow A_n \leftarrow O_n \]

\[ S_s \]

Figure 3–19: SDL fragment for pattern predicate CEFSM

The implementation of merge patterns can be obtained by straightforward mapping of the resulting STDs. We omit the SDL implementation of the patterns.

3.2.1.10 Example

As an example of the pattern predicate after action, Figure 3–20 shows an error detection method which performs the checksum for an input msg(data). If

\[ \text{wait_data} \]

\[ \text{msg(data)/rst:=checksum(data)/} \]

\[ \text{msg(nok)} \]

\[ \text{msg(ok)} \]

\[ \text{error/} \]

\[ \text{proceed/} \]

Figure 3–20: Example of pattern predicate after action for error detection

the result of the action procedure has zero value, which means that there is no error in the received message, the CEFSM performs the decoding and generates the output message msg(ok). Otherwise, the input message has an error, and an error notification message msg(nok) is sent. Note that the predicates $rst = 0$ and $rst \neq 0$ are evaluated depending on the previous action checksum().
From the property specification of the pattern, we can get the following requirements of the CEFSM.

- Property: The input event $msg(data)$ has to happen eventually and the checksum will have zero or non-zero value for the $data$.
- LTL: $\Diamond(msg(data) \land \Diamond((rst!=0) \lor (rst=0)))$
- Property: The input event $msg(data)$ is followed by one of output events $msg(nok)$ or $msg(ok)$ depending on the result of predicates.
- LTL: $\Box(msg(data) \rightarrow \Diamond(((rst!=0) \land \Diamond(msg(nok))) \lor ((rst=0) \land \Diamond(msg(ok))))$

As an another example, let us suppose we are designing a system that uses the connection oriented communication with other systems. The system handles connection establishment and release requests to setup a connection with a peer system and to release the connection. Figure 3-21 (a) shows the connection setup scenario using the pattern basic CEFSM. After receiving a connection request $EST.req$, the CEFSM performs action “connect”. Then, it notifies the peer that the connection is set up by using the message $EST.conf$. Similarly, the disconnection step is also achieved in the pattern basic CEFSM as Figure 3-21 (b).
In fact, it is possible to handle the two requests at one state. Note that the CEFSMs for connection setup and release have different state names. Before merging the CEFSMs, we rename both wait\_establish and wait\_release to wait\_msg. The merged CEFSM is described in Figure 3–21 (c), and it is an example of pattern source merge.

On the other hand, a connection can be disconnected only after the connection is setup. Figure 3–21 (d) show that the connection CEFSM is sequentially merged with the disconnection CEFSM. The state wait\_release was renamed to established before the merging. The state released can be reached through the state established. The requirements of the result CEFSM can be obtained as follows:

- Property: Eventually, the input event EST\_req has to occur to initiate the connection.

- LTL: ♦EST\_req

- Property: The input event EST\_req is followed by the event EST\_conf, REL\_req, and REL\_conf in sequence. In other words, before the request of connection release REL\_req, the connection was established by EST\_conf.

- LTL: □(EST\_req → ♦(EST\_conf ∧ ♦(REL\_req ∧ ♦REL\_conf)))

- Property: The event EST\_req proceeds the event REL\_req. In other words, before the connection release request, the connection must be established by EST\_req.

- LTL: ♦EST\_req → (¬REL\_req U EST\_req) ce.

- Property: The events EST\_req, EST\_conf, REL\_req, and REL\_conf must happen alternatively.

- LTL: □((0 ≤ n_1 ≤ 1) ∧ (0 ≤ n_2 ≤ 1) ∧ (0 ≤ n_3 ≤ 1))
3.2.1.11 See also

To describe the CEFSM, we use the STD with transitions labeling with an event, predicates, actions, and outputs. The similar notation is used at Statecharts [29], a visual formalism for the specification of reactive systems, where a transition is labeled with an event, a condition, and an action. Statecharts are an extension of our STD to enhance the descriptive power using hierarchy of states, orthogonality, and history connectors.

3.2.2 Timer

3.2.2.1 Context

In an event-driven system such as a communications system, an event may occur later than it is expected, or not happen at all because of transmission delay, lost message, etc. Many event-driven systems employ timing constraints where some action is taken if an expected event does not occur in a given amount of time. We are designing the system in CEFSMs.

3.2.2.2 Problem

How can we model the timing constraints to avoid unbounded waiting for an event?

3.2.2.3 Solution

A CEFSM can be supplemented with a timer and timer-related operations to manipulate the timing constraints [14]. A timer $T$ is an element of variable set $V$ with an associated time value. It has two modes, active and inactive. Initially, a timer is inactive. The unit of a time value depends on the context of an application. There are timer-related operations, set and reset, which are the elements of action list $A$. The operation set$(v,T)$ allocates the time value $v$ to the timer $T$ and makes the timer be in the active mode. Once a timer is set, the timer creates a timer expiration signal after passing the time value unless it has been cancelled by the reset operation. The operation reset$(T)$ stops the timer $T$ and
changes the timer to the initial inactive mode. Typically, if an expected message arrived before a timer expiration signal, the system resets the timer. Otherwise, the system receives the timer expiration signal and then sends an error notification for the lossy message or requests resubmission of the expected message. Note that all these time concepts come from SDL [14].

![Diagram](image)

Figure 3-22: Pattern timer with the expected event \( e \)

Figure 3-22 shows an STD of the pattern timer. First, a timer \( T \) should be set before using it. In the diagram, the transition happens upon the event \( e_0 \). On timer expiration for \( T \), the CEFSM moves to the state \( S_2 \). If the CEFSM receives the expected event \( e \), it resets the timer and performs the remaining transition.

### 3.2.2.4 Property specification

First, the input event \( e_0 \) must occur eventually to initiate the CEFSM.

- Property: The initial input event \( e_0 \) must happen eventually.
- LTL: \( \Diamond e_0 \)

Then, either the intended event \( e \) or time expiration \( T \) has to follow the input event \( e_0 \).

- Property: The input event \( e_0 \) is followed by either an event \( e \) or a time expiration \( T \).
- LTL: \( \Box(e_0 \rightarrow \Diamond((e \land \neg T) \lor (\neg e \land T))) \)
There is a possibility to arrive at the intended event $e$ after $T$ is taken due to the delay of the event. If the delayed delivery of an event is not allowed in the system, we also have to describe this property using the absence property.

- Property: After a timeout $T$, no $e$ occurs.
- LTL: $\square(T \rightarrow \square(\neg e))$

### 3.2.2.5 Implementation

A timer variable in the CEFSM maps to an SDL Timer object. The Timer object must be declared like any other variable in SDL. In addition, the time value is given as part of the declaration. The operations on the Timer object are $\text{set}(\text{Timer})$ and $\text{reset}(\text{Timer})$. Since the time value is given in the declaration, it is not specified in the set operation. Figure 3–23 shows the SDL implementation of the typical timer pattern discussed above.

![SDL implementation of timer pattern](image)

Figure 3–23: SDL fragment for pattern timer

### 3.2.3 Repeated Events

#### 3.2.3.1 Context

In an event-driven system, a single event may not be sufficient to initiate a reaction: an event must occur several times. For example, if a message is transmitted in several packets, all packets must arrive before the further message handling.
3.2.3.2 Problem

How does a communication system handle the repeated events?

3.2.3.3 Forces

- **Timing constraints**: It is necessary to impose timing constraints for the repeated events. There are two kinds of timing constraints such as for the individual event and for all events.

- **Number of repetitions**: In some cases, the number of repetitions is known in advance. In other cases, it is not. Even when the number is not known in advance, there still may be an upper bound on the number of repetitions.

3.2.3.4 Solution

This pattern is a specialization of the pattern predicate CEFSM. First, the CEFSM has an integer variable $c$ which is initialized to one to count the number of occurrences of an event. Predicates $c < N$ and $c = N$ determine the next transition where $N$ is the number of times the event should be repeated. If $c$ is less than $N$, then $c$ is incremented and the state unchanged. Otherwise, the state changes to the the next state and any specified output events are generated. Figure 3–24 shows the solution and its SDL implementation.

![Figure 3–24: Pattern repeated events and its SDL implementation](image-url)
3.2.3.5 Property specification

The input event $e$ has to occur to initiate the CEFSM and the counter $c$ should be one at that time.

- Property: The input event $e$ has to happen eventually and the counter $c$ is one at that time.
- LTL: $\Diamond(e \land (c = 1))$, where $c$ keeps the number of occurrences of event $e$.

Then, the event happens $(N-1)$ times more and then it is followed by an output event $O_2$.

- Property: The input event $e$ and predicate $(c = 1)$ is followed by $(N-1)$ times $e$ and $O_2$.
- LTL: $\Box((e \land (c = 1)) \rightarrow \Diamond((e \land (c = N)) \land \Diamond O_2))$

Moreover, the event $e$ should not happen after the $N$-th occurrence.

- Property: The input event $e$ cannot happen more than $N$ times.
- LTL: $\Box(1 \leq c \leq N)$

3.2.3.6 Variant: timed repeated events

In this variant, we add timers to enforce timing constraints. This pattern can thus be considered as a combination of the patterns of repeated events and timer. Two timers $T_1$ and $T_2$ are used in the pattern. Timer $T_1$ is used for the individual occurrences of event $e$, while $T_2$ is used for all of the events together. Figure 3–25 shows the resulting CEFSM and its SDL implementation. At the state $S_1$, the CEFSM receives the event $e$ until it has all events needed. If the number of events is less than $N$, the machine increases the $c$ and sets the timer $T_1$ again. If the timer $T_1$ or $T_2$ expires, the machine moves to the corresponding state to handle this exceptional case. Note that either timer could be omitted but not both. Then only the total or single event timing constraints would be checked.
Property specification. This pattern is a combination of the pattern repeated events with the pattern timer. Therefore, we can deduce the properties of this pattern from their ones.

- Property: The initial input event $e_0$ must happen eventually.
- LTL: $\Diamond e_0$

Then, the event is followed by one of the three possible events.

- Property: The input event $e_0$ is followed by one of three input events $e$, $T_1$, and $T_2$.
- LTL: $\Box (e_0 \rightarrow \Diamond ((e \land (c = N)) \land \neg T_1 \land \neg T_2) \lor (\neg (e \land (c = N)) \land T_1 \land \neg T_2) \lor (\neg (e \land (c = N)) \land T_1 \land T_2))$

- Property: The subsequent input events are followed by each corresponding output event.
- LTL: $\Box(((e \land (c = N)) \rightarrow \Diamond O_3) \land (T_1 \rightarrow \Diamond O_4) \land (T_2 \rightarrow \Diamond O_5))$

As a simplified property of the above properties, we may check that the initial input event $e_0$ is followed by one of three output events $O_3$, $O_4$, and $O_5$.

### 3.2.3.7 Variant: timed retrial

In the previous patterns, the number of repetitions of an event before the system moved to a different state was known in advance. This variant considers the
case where the number of repetitions is not known in advance (and may be 0), but the maximum number is bounded. A typical scenario would be where the repeated event is the expiration of a timer set. For example, a message is transmitted and the timer is set. If an acknowledgment is not received before the timer expires, the message is retransmitted. The message will be transmitted a maximum of N times before the system moves to an error state. Figure 3–26 shows the solution where the repeated event is the expiration of the timer.

**Property Specification.** The pattern timed retry has a similar state transition diagram as the pattern *timed repeated events*. It needs the initial input $e_0$ which will be responded by either the target event $e$ or $N$-th expiration.

- Property: The input event $e_0$ has to occur eventually.
- LTL: $\Diamond e_0$

- Property: The input event $e_0$ is followed by either the target event $e$ or $N$ times timeout.
- LTL: $\Box (e_0 \rightarrow \Diamond ((e \land \neg(T \land (c = N))) \lor (\neg e \land (T \land (c = N)))))$

- Property: After $N$-th expiration, no timeout $T$ exists.
- LTL: $\Box(1 \leq c \leq N)$
3.2.3.8 Example

Figure 3–27 (a) describes a part of a call setup software. It receives a telephone number composed of nine digits from a caller. After receiving the event dial nine times, the CEFSM generates an output conn_req to request a call connection with a callee of the number. The example can be expanded with a timer T to avoid unlimited waiting. If a caller does not push a digit in a given time bound, the machine notifies the caller that time is over by giving a special beep. This case is implemented by adding a transition for the timer expiration. The following are the part of properties obtainable from the patterns used.

- Property: A call connection request needs nine digits.
- LTL: $\Box((\text{dial}(\text{digit}) \land c = 1) \rightarrow \Diamond((\text{dial}(\text{digit}) \land (c = 9)) \land \Diamond\text{conn}\_\text{req}))$
- Property: A call connection needs nine digits in the timing constraints T.
- LTL: $\Box((\text{dial}(\text{digit}) \land c = 1) \rightarrow \Diamond(((\text{dial}(\text{digit}) \land (c = 9)) \land \neg T) \lor (\neg(\text{dial}(\text{digit}) \land (c = 9)) \land T)))$

As an example of the pattern timed retry, Figure 3–27 (b) shows an FTP client that receives an FTP request from a user and try to connect to an FTP server. The following are the properties obtainable from the example.

- LTL: $\Diamond\text{FTP}\_\text{req}$
- LTL: $\Box(\text{FTP}\_\text{req} \rightarrow \Diamond((\text{connect}\_\text{OK} \land (T \land (c = 9)))) \lor (\neg\text{connect}\_\text{OK} \land (T \land (c = 9)))$)
3.2.3.9 See also

Pattern TimerControlledRepeat [70] repeats a message transmission to avoid message loss during data transfer. If a sender entity does not receive an expected acknowledgment in the given expiration time from a receiver entity, the message is repeated by the sender. This pattern is considered as an instantiation of the pattern timed retrial.

3.2.4 Message Transfer

3.2.4.1 Context

In a communication network, two communicating entities in the same layer want to interact to transfer messages through their lower layer.

3.2.4.2 Problem

How do two communicating peers interact in a layered protocol?

3.2.4.3 Forces

- Role of interaction: In the communication, one party initiates communication and another reacts to it. Thus, it is important to identify the role of entity before communication.

3.2.4.4 Solution

Conceptually, a layer provides a set of services to its users. Thus, the users consider the lower layer as a service provider [64, 73]. They communicate with the service provider through service access points (SAP). The service provider coordinates and manages communications between users by using four types of primitives, request, indication, response, and confirm, as shown in Figure 3-28. The user A initiates a message transfer with the peer B by invoking the primitive request to the lower layer. The peer of the initiator is informed by the lower layer using the primitive indication. The respondent replies to the indication by invoking the primitive response to the lower layer. The lower layer notifies the initiator of
the response from the peer using the primitive confirm. Depending on the protocol, the name of each primitive might be different.

Typically, there are two kinds of communication between peers such as confirmed transfer and unconfirmed transfer. If an initiator needs an acknowledgment from a peer, it uses a confirmed transfer with the four primitives. However if this is not the case, the initiator sends a message without expecting an acknowledgment. For example, a connection establishment always uses a confirmed transfer because a peer must agree to establish a connection with its initiator. A data transfer, on the other hand, uses either confirmed or unconfirmed transfer depending on protocol [73].

3.2.4.5 Variant: unconfirmed sender

The pattern unconfirmed sender given in Figure 3–29 (a) initiates a simple but unconfirmed message transfer. It does not care about the progress and result of message transfer once it has been initiated.
**Property specification.** If a system is designed with this pattern, it is necessary to check the following properties, but are not limited.

- Property: The input event e has to occur eventually and it must be followed by an occurrence of request.
- LTL: $\diamond e \land \square (e \rightarrow \diamond request)$
- Property: The events e and request occur alternatively.
- LTL: $\square (0 \leq n_{er} \leq 1)$, where $n_{er}$ initialized to zero is increased after e and decreased after request.

**Example.** An IP datagram sending at Internet Protocol (IP) uses an unreliable, best-effort, connectionless packet delivery system [74]. As a user’s point of view, the transfer of an e-mail is also the unconfirmed transfer of a message.

**See also.** Push Interaction Pattern [30], SendReceive [23]

3.2.4.6 **Variant: unconfirmed receiver**

The pattern unconfirmed receiver takes a message from a sending party in the form of indication as shown in Figure 3-29 (b). It does not reply to the sending party. It is important to consider a receiving party with a sending party simultaneously because a message transfer is typically performed in pairs. If the pattern unconfirmed receiver communicates with the pattern unconfirmed sender, the indication corresponds to the request of unconfirmed sender.

**Property specification.** Since this pattern has a similar STD such as the pattern unconfirmed sender, the properties specified at the pattern are also applicable here.

- Property: The input event indication has to occur eventually and it must be followed by $O_2$ if the output exists.
- LTL: $\diamond indication \land \square (indication \rightarrow \diamond O_2)$
- Property: The events indication and $O_2$ occur alternatively.
• LTL: $\square(0 \leq n_{io} \leq 1)$, where $n_{io}$ initialized to zero is increased after the occurrences of indication and decreased after $O_2$.

Moreover, it may be necessary to check the correspondence of indication to the request if both unconfirmed sender and unconfirmed receiver are used in pair.

• Property: The indication must always follow the request and each event has to happen alternatively.

• LTL: $\square((request \rightarrow \Diamond indication) \land (0 \leq n_{ri} \leq 1))$

**Example.** Reception of an IP datagram uses the unconfirmed receive. As a user’s point of view, the reception of an e-mail is also the unconfirmed receive.

**See also.** Push Interaction Pattern [30], SendReceive [23]

3.2.4.7 **Variant: confirmed sender**

To initiate a confirmed message transfer, the pattern confirmed sender starts a message transfer with a primitive request. Then, it waits for a confirmation from its peer. The primitive confirm would contain the result of request, for instance, successful transfer of request, request refusal due to resource insufficient, etc.

Figure 3–30 (a) depicts the typical behavior of the pattern where the transition labeled with request is sequentially merged with the transitions that have several confirmation messages.

![Diagram](image)

Figure 3–30: Pattern confirmed sender and its example on an SNMP client
**Property specification.**

- Property: The input event $e_1$ has to occur eventually and the request follows the event.
- LTL: $(\Diamond e_1 \land \Box (e_1 \to \Diamond \text{request}))$
- Property: The request is followed by one of confirmations.
- LTL: $\Box (\text{request} \to \Diamond ((\text{confirm}_1 \land \neg \text{confirm}_2 \land \cdots \land \neg \text{confirm}_n) \lor \cdots \lor$\n$\neg \text{confirm}_1 \land \neg \text{confirm}_2 \land \cdots \land \text{confirm}_n)))$

Moreover, it is also necessary to check the correspondence between the request and confirmation messages.

**Example.** A network management protocol allows a network manager to monitor and debug the nodes attached to the network. A client of the Simple Network Management Protocol (SNMP), a standard TCP/IP network management protocol, needs a value of a host to check the status of the machine [75]. Figure 3-30 (b) presents an abstract behavior of the client which asks a value of a variable in a system using the message GetRequest-PDU. Then, it receives the result in the message Response-PDU. If the value was accessible at the remote machine, value field is set to the corresponding value. Otherwise, error status field has a non-zero value which indicates that an error occurred during the request handling. Connection establishment request at two-way handshake, status query, and resource reservation request also use this pattern.

**See also.** Pull Interaction Pattern [30], BlockingRequestReply [23]

### 3.2.4.8 Variant: confirmed receiver

After receiving a request from its peer, a confirmed receiver processes the request and then replies with an appropriate response message. There are three common ways to handle the behavior. In Figure 3–31 (a), a receiver takes an indication from a peer. Then, it decides an appropriate response after performing action $A_1$. The response message includes the information that indicates the type
of response such as success, remote host down, illegal address, etc. Another case is to reply with one of possible response messages which holds a predicate after action $A_1$ as in Figure 3-31 (b). There are many different names for each response in this case. As a last case, a receiver waits for another event which indicates the result of request handling.

![Diagram](image)

Figure 3-31: Pattern confirmed receiver in three common cases

**Property specification.** Since the STDs given in Figure 3-31 have the same form as in basic CEFSM, predicate after action, and confirmed sender, we can infer the properties of this pattern from them. The following are the properties applicable to Figure 3-31 (a).

- Property: The input event indication has to occur eventually and it is followed by the occurrence of response.
- LTL: $\Diamond(indication \land \Box(indication \rightarrow \Diamond(response))$

- Property: The events indication and response occur alternatively.
- LTL: $\Box(0 \leq n_{ir} \leq 1)$ where $n_{ir}$ initialized to zero is increased after the occurrences of indication and decreased after response.

In the case of Figure 3-31 (b) and (c), we can also infer the following property.

- Property: The input event indication is followed by one of response messages.
- LTL: $\Box(indication \rightarrow \Diamond((response_1 \land \neg response_2 \land \cdots \land \neg response_n) \lor \cdots \lor (\neg response_1 \land \neg response_2 \land \cdots \land response_n)))$
**Example.** As a corresponding receiver of SNMP protocol which is given in the example of confirmed sender, a remote host receiving GetRequest-PDU processes the variable binding to produce a Response-PDU. If the binding fails, a Response-PDU is replied with an error status field set. Many servers in the client-server model use this pattern to handle a request from clients. For example, a time-of-day server returns the current time when it receives a time request from a client [74].

**See also.** Pull Interaction Pattern [30], BlockingRequestReply [23]

### 3.2.4.9 Variant: timed receiver and timed retry receiver

The pattern timed receiver waits for an intended message, indication, after setting a timer. If the timer expires before receiving the message, the pattern considers it as an error case. This pattern is a combination of an unconfirmed receiver with a timer. As a general case of this pattern, we can try to receive the intended event several times, which is called timed retry receiver. Figure 3–32 represents the patterns. Note that it is also possible to have patterns timed confirmed receiver and timed confirmed retry receiver for confirmed message reception in timing constraints.

**Property specification.** The following is a property applicable to the pattern timed receiver.
Property: The initial input event $e_0$ must happen eventually and it has to be followed by either an intended event indication or a timeout $T$.

LTL: $\Diamond e_0 \land \Box (e_0 \rightarrow \Diamond ((\text{indication} \land \neg T) \lor (\neg \text{indication} \land T)))$

In some cases, the indication could arrive after $T$ because of event delay. If a delayed delivery of an event is not allowed in a system, we can describe this property using an absence property.

Property: After a timeout $T$, no indication can occur.

LTL: $\Box (T \rightarrow \neg \Diamond \text{indication})$ or equivalently, $\Box (T \rightarrow \Box (\neg \Diamond \text{indication}))$

For the pattern timed retrial receiver, we have to check the number of occurrences of timeout in addition to the event existence and respondence.

Property: The input event $e_0$ is followed by either the event indication or $N$ times timeout.

LTL: $\Diamond e_0 \land \Box (e_0 \rightarrow \Diamond ((\text{indication} \land \neg (T \land (c = N))) \lor (\neg \text{indication} \land (T \land (c = N)))))$

Property: After $N$-th timeouts, no timeout $T$ exists.

LTL: $\Box (1 \leq c \leq N)$ or equivalently, $\Box ((T \land (c = N)) \rightarrow \Box \neg T)$

Example. In the ATM signaling protocol, an access switch waits for an ALERT message from a called party in 10 seconds. If it fails to receive the message, it assumes that there is some problem with the called party and proceeds to clear the call connection.

See also. Pattern Timer [76]

3.2.4.10 Variant: timed confirmed sender and timed re trial confirmed sender

In the pattern timed confirmed sender, a CEFSM requests a message transfer. Then it waits for a confirmation message from its peer within a certain time interval. If a timer expires before the reception, the sender considers it as an error or abnormal case. As a general case of this pattern, the sender retries the request
until it receives the confirmation or reaches the maximum number of repetition.

The timed retry confirmed sender retransmits the request at most N times.

Figure 3–33: Patterns timed confirmed sender and timed retry confirmed sender

**Property specification.**

- Property: The input event $e_1$ has to occur eventually and the request follows the event.
  - LTL: $(\Diamond e_1 \land \Box (e_1 \rightarrow \Diamond request))$

- Property: The request is always followed by one of the confirmation messages or timeout $T$.
  - LTL: $\Box (request \rightarrow \Diamond ((T \land \cdots \land \neg confirm_n) \lor \cdots \lor (\neg T \land \cdots \land confirm_n)))$

In the case of pattern timed retry confirmed sender, we also need to check the number of occurrences of timeout.

- Property: The request is followed by either one of confirmations or N times timeout.
  - LTL: $\Box (request \rightarrow \Diamond ((T \land (c = N)) \land \cdots \land \neg confirm_n) \lor \cdots \lor (\neg (T \land (c = N)) \land \cdots \land confirm_n))$

- Property: After N-th timer expiration, no timeout $T$ exists.
  - LTL: $\Box (1 \leq c \leq N)$ or equivalently, $\Box ((T \land (c = N)) \rightarrow \Box \neg T)$

**Example.** As an example of timed confirmed sender, we can consider ping command in UNIX which tests reachability of a host using the Internet
Control Message Protocol (ICMP) [74]. A host sends an ICMP echo request, ECHO_REQUEST, to a specified destination and sets a timer for it. The default value of timeout is 20 seconds. If the request is answered with the ECHO_REPLY within the given time, the sender prints a successful result. Otherwise, it indicates the failure of the request. TCP timer/retransmission and Bootstrap Protocol (BOOTP) retransmission also use this pattern [74].

See also. TimerControlledRepeat [23], PAR (Positive Acknowledgment with Retransmission) [73], Bounded Retransmission Protocol [46], Abortable Interaction Pattern [30].

This is the end of pattern message transfer. As mentioned before, the patterns presented in this section are not a complete set. They can be combined to obtain more complex and high-level patterns. For example, we can get a new pattern, confirmed sender-receiver, a combination of confirmed sender and confirmed receiver for negotiation of communication. At an initial state, it sends a request to a peer, and then wait for a confirmation such as confirmed sender. Then, for the confirmation message, it replies with an acknowledge message to guarantee that the confirmation is agreed upon. This behavior is well-known as a three-way handshake [74].
CHAPTER 4
MODEL CHECKING PATTERN-BASED DESIGN

4.1 Modeling Pattern-Based Design in PROMELA

In our development methodology, the pattern-based design is followed by the validation of design specification to find errors in the design phase. The formal validation using SPIN needs to construct a model for the design specification and validate the model against the properties to be satisfied in the design. Figure 4–1 shows an overview of SPIN model checking for a pattern-based system. After

![Diagram](image)

Figure 4–1: Model checking pattern-based system using SPIN

the high-level designing, we can construct the PROMELA model and obtain the correctness properties of the final system. In this section, we present a model construction mechanism from the system design description and issues arising in the construction.
A model that is consistent with the design is essential to the subsequent validation results because any violation in the model has to exactly reflect the same fault of the design. The correspondence between our patterns and PROMELA makes it simple to build a consistent model of the design. The patterns are composed of blocks, communication paths, messages, dynamic creation of block instance, states, transitions, predicates, repetition, action list, state merge, timers, etc. Most components of the patterns have direct counterparts in PROMELA. By converting each component into the corresponding PROMELA construct, we can build a model. At the design phase, we separate the modeling in two steps for architectural design and behavioral design.

4.1.1 Model Construction from Patterns

4.1.1.1 Modeling architectural design

The main task in this step is to construct an outline of a model from architectural components. A block is mapped to a process declaration in proctype and an instance of the block can be created dynamically in the run operator. A communication path between blocks is implemented in a PROMELA channel chan that carries messages and their parameters. All input and output messages are defined by symbolic constants in mtype. As an example of the model construction, we present a fragment of PROMELA code for the architecture of the nine-digit dialing as in Figure 4-2 (a). The block nine_digit_dial reads a telephone number composed of nine digits from an upper layer caller. Then, it requests a connection conn_req

![Diagram](image-url)
with a callee via the lower layer. In the modeling, attention has to be paid to the interface with environments. The block is in the middle of two adjacency blocks \textit{caller} and \textit{lower}. There are two communication paths \textit{from\_caller} and \textit{to\_lower} among them.

```c
#define BUFFSIZE 2  /* channel capability */

chan from_caller = [BUFFSIZE] of {mtype,DIGIT};
chan to_lower = [BUFFSIZE] of {mtype,DIGITS};
mtype = {dial,conn_req};

proctype nine_digit_dial() {}
```

More detailed issue on communication path will be given later.

\subsection*{4.1.1.2 Modeling behavioral design}

Recall that the CEFSM used in the design description is composed of a set of states and transitions. Each state is converted to a \texttt{label} in \texttt{Promela}. Moving to the next state is implemented in \texttt{goto} control transfer construct with a target label. A transition performs a set of actions and output generation for a coming input message. For the message exchange on a communication path, the send and receive statements are used. A decision point with predicates is converted to the selection construct \texttt{if} and each predicate is converted to a corresponding guard. The repeated event is mapped to the repetition construct \texttt{do}. The source merge is represented using \texttt{if} for the selection of a transition from merged transitions. Other merge patterns are implemented with \texttt{goto}.

In the pattern description, the action list is usually composed of assignment commands and arithmetic operations. The C-like \texttt{Promela} syntax supports the actions directly. Meanwhile, the high-level abstraction action described in plain English is commented in the \texttt{Promela} model so that the action is to be designed in later development phases. \texttt{Promela} provides \texttt{bit}, \texttt{bool}, \texttt{short}, \texttt{int}, and \texttt{unsigned} data types as well as \texttt{array} and \texttt{typedef} for array and user-defined data types. They are usually enough for the data in a pattern.
The following code shows the finalized model of Figure 4-2.

```c
#define BUFFSIZE 2 /* channel capability */
#define N 9 /* repetition number */
#define DIGIT byte
typedef DIGITS {DIGIT ch[9];}

chan from_caller = [BUFFSIZE] of {mtype,DIGIT};
chan to_lower = [BUFFSIZE] of {mtype,DIGITS};
mtype = {dial, conn_req};

proctype nine_digit_dial()
{
    int c = 1;
    DIGIT digit;
    DIGITS digits;

dialing:
    from_caller? dial(digit) ->
    if
        :: (c < N) ->
            digits.ch[c-1] = digit; /* store digit */
            c = c + 1;
            goto dialing;
        :: (c == N) ->
            digits.ch[c-1] = digit; /* store the last digit */
            to_lower! conn_req(digits);
            goto connecting;
    fi;
connecting: skip;
}
```

The modeling is straightforward from the pattern-based design. Currently, the modeling is performed manually. However, it is believed that the conversion could be performed automatically.

### 4.1.2 Communication Path and Channel Capacity

Channels defined by `chan` are used to transfer data from one process to another. A process transfers data to other processes through message channels. For example, `chan UNI2MAIN=[8] of {mtype,int,byte}` declares a channel named `UNI2MAIN` that is able to store up to eight messages consisting of `mtype`, `int`, and `byte` fields.
The channel capacity is one of the important issues in modeling. Communication via a channel is either synchronous (i.e., rendezvous) or asynchronous (i.e., buffered) depending on the channel capacity. When we specify a communication path in design phase, we did not care about the size of message queue. We assumed that it has an unlimited size. In a PROMELA model, however, a channel can store a finite number of messages. Furthermore, increasing the channel capacity could increase the state space dramatically. Table 4–1 shows the memory and CPU usages for the different buffer size of the PROMELA model described in Chapter 5.1. Note that the memory (1024M) reaches its limit at the small change of channel size. A typical approach we have used regarding the channel capacity is to check both synchronous and asynchronous communication all the times if it is possible. The results are quite different in many cases. We start with the channel size zero for synchronous communication and then increase it gradually. When the validation cannot be performed due to the state space explosion, we use other techniques such as state-vector compression and bit-state hashing.

When a process transfers a message, message fields specified in a send or receive statement must have the same number of fields and each field has to be compatible with data type as in channel declaration. One problem in implementing a communication path is that several messages with different definitions could be passed through a single communication channel. As a simple solution of this problem, a channel can be declared to be wide enough to carry all messages on the

<table>
<thead>
<tr>
<th>channel size</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>state vector (byte)</td>
<td>80</td>
<td>80</td>
<td>104</td>
<td>120</td>
<td>144</td>
</tr>
<tr>
<td>depth reached</td>
<td>345</td>
<td>8422</td>
<td>172580</td>
<td>1373142</td>
<td>4657542 (stopped)</td>
</tr>
<tr>
<td>states stored</td>
<td>54</td>
<td>24311</td>
<td>397351</td>
<td>2.89692e+06</td>
<td>5.75892e+06 (stopped)</td>
</tr>
<tr>
<td>total memory (M)</td>
<td>2.542</td>
<td>4.488</td>
<td>47.756</td>
<td>597.439</td>
<td>over the limit (1024 M)</td>
</tr>
<tr>
<td>real time (sec)</td>
<td>0.0</td>
<td>1.8</td>
<td>46.5</td>
<td>7:59.3</td>
<td>17:36.5 (stopped)</td>
</tr>
</tbody>
</table>
channel. Each channel thus has appropriate fields to store each message with its parameters [58].

4.1.3 Timer Handling

One challenging issue in the behavioral pattern is the modeling of a timer. In a communication system, a message may occur later than it is expected, or not happen at all because of a lost message. Thus, timing constraints are necessary to control the waiting time of an expected message. For this purpose, we introduce the timer and timer-related operations, set and reset.

A timer is a component of the pattern language to generate a timer expiration signal when a time value assigned to the timer has been exceeded [14]. It has two modes, active and inactive. Initially, a timer is inactive. The operation set(v,T) allocates the time value v to the timer T and makes the timer be in the active mode. Once a timer is set, the timer creates a timer expiration signal after passing the time value unless it has been cancelled by the reset operation. The operation reset(T) stops the timer T and changes the timer to the initial inactive mode.

Typically, if an expected message arrived before a timer expiration signal, the system resets the timer. Otherwise, the system receives the timer expiration signal and then sends an error notification for the lossy message or requests resubmission of the expected message. Note that the unit of a time value depends on the context of an application.

To validate the pattern-based design, it is necessary to translate the timer, timer-related operations, and timer expiration in a PROMELA model. In our modeling, we use an abstraction of a timer as introduced in Bosnacki et al. [55] where a timer is represented in a Boolean variable initialized to false. For the set operation, the timer variable is assigned to true to activate the timer. For the reset operation, it has false value to inactivate the timer. The arrival of a timer expiration signal is simulated by investigating the current value of the timer variable. If a timer has true value at the investigation, this means that the timer was activated sometime before. Thus, we assume that the timer expiration
signal has arrived at the time of investigation. This method abstracts out the real concrete value of a timer. The following shows the modeling of timer $T$ in *Promela*.

```
timer declaration: bool $T$ = false;
set(v,$T$): $T$ = true;
reset($T$): $T$ = false;
timer expiration: ($T$ == true)
```

As an example the timer model, let us consider a communication between two communicating entities. A sending entity transfers a message (DATA) to a receiving entity and waits for an acknowledgment (ACK) from it through the channel `comm`. The sender has a timer `ACKTimer` for the ACK message.

```
/** SENDER /**
bool ACKTimer = false; /* initially inactivated */
comm!DATA; /* send DATA */
ACKTimer = true; /* set operation */
if
  :: comm?ACK; /* expected message */
    ACKTimer = false; /* reset operation */
  :: (ACKTimer == true) -> /* timer expiration */
    code to handle the timer expiration ...
fi;

/** RECEIVER /**
comm?DATA; /* receive DATA */
if
  :: comm!ACK; /* reply with ACK */
  :: skip; /* message loss */
fi;
```

This method is simple and covers all potential situations regardless of the timer value. One problem with this method is the possibility of an unreceived message remaining in the channel. This happens due to either the prematured timer expiration, i.e. a timer is expired even though the message is delivered in time, or a delayed message. Therefore, it is necessary for a process to consume the remaining message. For instance with the above example, it is possible for the sender to execute the timer expiration part even though the receiver replied with the ACK. As a result, the sender has to consume or receive the message from the channel `comm` sometime later. If there are many timers in a system, it is hard to
consider all situations of unreceived messages. Moreover, the prematurely timer expiration is not common in a real situation.

As a second method to model the timer, we assume that the timer expiration happens only due to the message loss. In this situation, we simulate the timer expiration by investigating the status of the model as well as the timer variable. If a timer has true value and the model is in the deadlock state, we assume that the deadlock was caused by a message loss. Thus, the code to handle the timer expiration signal is executed to resolve the deadlock. For the checking of deadlock, we use timeout, a PROMELA predefined variable. It is true if no statement is executable in any active process. Otherwise it is false. Consequently, the timer expiration is simulated as follows:

\[
\text{timer expiration: } (T == \text{true}) \&\&(\text{timeout})
\]

Typically, the usage of timeout for timer expiration is enough in many cases [48].

One drawback of this approach is that a deadlock which has occurred for some other reason, not by a message loss, can not be checked properly [48]. It may be necessary for a model to reflect the message loss precisely. Thus, in a third method, we introduce a special message that indicates the message loss. A similar trick is found in D’Argenio et al. [46]. In this method, we do not have a timer variable anymore. Instead, a message for a message loss is used. Then, an entity transfers the message to simulate the message loss. As an example of this method, the DATA-ACK example presented above could be modeled using the message ACKLoss.

```c
/** SENDER **/
comm!DATA; /* send DATA */
if
  :: comm?ACK; /* expected message */
  ACKTimer = false; /* reset operation */
  :: comm?ACKLoss -> /* timer expiration */
    code to handle the timer expiration ...
fi;

/** RECEIVER **/
comm?DATA; /* receive DATA */
if
:: comm!ACK; /* reply with ACK */
:: skip -> comm!ACKLoss; /* indicate the message loss */
fi;

4.1.4 Environment Process

A model to be validated with SPIN should be a closed system. As a result, we have to provide any missing parts of a whole system as well as the validation model itself [56]. If we have already developed these parts, this would provide a complete set for validation. Otherwise, we have to build them. Frequently, the abstracted environment is much difficult than the model because it is not well-designed and a developer may not have enough information for the environment. Moreover, it is necessary to identify whether an error has occurred in the model or in the environment. In our case, we extract the expected behavior of the environment processes from the design specification to make the model. The following presents one possible scenario of the environment process for the example of nine-digit dial described in Figure 4-2.

```
1 proctype ENV ()
2 {
3  byte count;
4  count = 1;
5  DIGITS callee_num;
6
7  do
8    :: (count < N) -> from_caller!dial(count);
9      count = count + 1
10   :: (count == N) -> from_caller!dial(count);
11      to_lower?conn_req(callee_num);
12     break
13  od;
14 }
```

4.2 Property Specification

After SPIN has checked obvious initial design flaws such as system deadlocks and unreachable code and these problems have been corrected, the system model is verified against system requirements [44] specified using temporal logic. One important issue in this stage is to identify requirements or properties to be verified. If such requirements specification is not clear or does not exist, it is difficult or impossible to determine the properties to be checked. Thus, the identification of
the desired behavior is an important issue in the system model [48]. As a high-level design document, the system design description represented in STD helps to find correctness properties in a system. The visual specification makes it easier for a developer to determine the intended events and relationship among them. In this section, we propose a property specification technique from behavioral patterns.

4.2.1 Linear Temporal Logic

Temporal logic [37] is a logic for statements and reasoning that involve the notion of order in time. It is a useful formalism to specify properties of reactive and concurrent systems and provides a formal and succinct notation for desired system behavior over time. We use LTL formulae in SPIN to express properties of a model.

A linear temporal formula is constructed from propositions to which we apply temporal and boolean operations [77]. Every proposition \( p \), a statement that has either true or false boolean value, is an LTL formula. For example, \( DATA \) was sent by a client is a true proposition if a client sent the message \( DATA \) sometime before. Otherwise, it has false value. If the propositions \( p \) and \( q \) are temporal formulae, then so are \( \neg p \), \( p \lor q \), \( p \land q \), \( p \rightarrow q \) for logical connectives negation, disjunction, conjunction, and implication. In addition to the boolean operators, an LTL formula contains temporal operators \( \Box(\text{always}) \), \( 
abla(\text{eventually}) \), \( U(\text{until}) \), and \( W(\text{weak until}) \).

A PROMELA model is considered to be a state transition system. LTL formulae are used to formally state properties concerned with the executions of the model. For instance,

\[
\Box(DATA \rightarrow \nabla(ACK))
\]

means that whenever the event \( DATA \) has occurred, the event \( ACK \) has to follow it eventually. A model checking tool can automatically validate that the property is satisfied with a system.
It is worth noting that some experience is required to write temporal logic statements, which is one of the obstacles to the more widespread usage of model checking techniques [7]. Dwyer and his colleagues [78, 79] introduced a pattern system for property specification for finite-state validation. The pattern system describes commonly occurring properties in a finite-state system. Patterns are largely classified for the occurrence and order of states and events. They include absence, universality, existence, bounded existence, precedence, response, chain precedence, and chain response. By using the patterns, it is possible for developers to express properties more easily. It expedites the developers learning and writing more expressive specification.

4.2.2 Property Extraction from Communication Patterns

One important issue in the validation is to identify the requirements or the properties of a designed system. If such properties are not provided properly, it is not possible to determine whether the system meets the required behavior. As a high-level design document, the system design description provides the requirements of a system. They are particularly useful to identify the existence of an event and order of events. Each pattern has a property specification section to describe the properties required by the pattern.

Meanwhile, we need a method to check an event occurrence such as the message transfer and reception in the SPIN validation. For this purpose, we associate an event occurrence with a control location that comes immediately after the event occurrence statement [47]. A label is used to identify a control location in a process declaration. By checking the reachability of the label, we know that the event has really happened.

For example, to see the reception of a message ACK in a sending process, we introduce a label ACK_recv after the reception statement:

\[
\ldots
\]
comm?ACK ->

ACK_rcv: skip; /* instrument a control label */

... 

Note that a label in a Promela model always prefixes a statement and thereby uniquely identifies a control state corresponding to the labeled statement [41].

Then, remote reference in an LTL formula is used to see the arrival of the process control at the control label. The remote reference makes it possible to determine the current state of an active process from an LTL formula. The remote reference operator takes three arguments: the first is the name of a process, the second is an instantiation number of an executing process of that type, and the third argument is the name of a label that is defined within the process. The operator returns a non-zero value if and only if the process referred to is currently in the state marked by the label. For instance, suppose a variable s_pid has an instance number of the sending process. Then, the reference sender[s_pid]@ACK_rcv evaluates to true when the process instance is at the state referred by the label ACK_rcv. Using the remote reference, we can check that the input event ACK has eventually occurred.
5.1 Case Study: Alternate Bit Protocol

To show the practical usage of pattern-based development, we have performed several case studies. As a simple one, we demonstrate a variation of alternating bit protocol (ABP) \([44, 73]\) which provides a simple but reliable message transfer on a lossy lower layer. The following is the requirements specification of the protocol: A sending entity transmits a DATA to a receiving entity. When the receiving entity takes the DATA from the sending entity, it replies with an ACK so that the sender is able to transfer the next DATA. Thus, each DATA should be acknowledged before the next one is sent. The lower layer could randomly lose messages but not corrupt the content and order of them.

5.1.1 Structural Design

The pattern split protocol layer looks suitable for the architecture of ABP. In adapting the pattern, it is worth noting that it is not clear from the requirements whether the ABP layer has an upper layer, which is an optional block of the pattern, or not. If an upper layer exists, the upper layer would initiate a DATA transfer. Otherwise, the ABP layer is the uppermost layer and internally generates and consumes the DATA. For this example, we suppose that there is a user layer of ABP that uses the messages PUT and GET to initiate and consume the DATA. Figure 5-1 illustrates the architecture of the protocol. For the validation of protocol, it is necessary to get an outline of Promela model from the structural pattern. Since the model has to be a closed system, we need the environment processes for the upper and lower layers.
5.1.2 Behavioral Design

In this section, we describe the execution of the blocks presented at the structural design. Note that this section illustrates the process of using SPIN step by step.

The confirmation for a DATA transfer in the ABP layer makes the blocks sender and receiver to be described in the patterns confirmed sender and confirmed receiver. One immediate problem that one can meet from the design is that the sender can wait for an ACK forever if a message is lost at the lower layer. To avoid the unbounded waiting, a timer is added to the sender so that it can be implemented in the pattern timed confirmed sender. Figure 5–2 presents the initial behavior of the protocol. From the patterns, we can construct the behavior part of PROMELA model and capture several correctness requirements to be checked for the protocol. The following is the list of the properties:

**Property 1** The event PUT has to be received eventually and DATA_req follows it.
LTL: $(\Diamond p \land \Box (p \rightarrow \Diamond q))$ where $p$ and $q$ represent the reception of PUT and transfer of DATA_req.

**Property 2** The request DATA_req is always followed by either reception of

ACK_ind or timeout T.

LTL: $\Box (p \rightarrow \Diamond ((q \land \neg r) \lor (\neg q \land r)))$ where $p$, $q$, and $r$ denote the transfer of

DATA_req, reception of ACK_ind, and timeout of timer T, respectively.

**Property 3** Reception of DATA_ind has to occur eventually.

LTL: $\Diamond p$ where $p$ means the reception of DATA_ind.

**Property 4** If the DATA_ind is received, the indication is always followed by

sending of ACK_req and GET.

LTL: $\Box (p \rightarrow \Diamond (q \land \Diamond r))$ where $p$, $q$, and $r$ represent the reception of DATA_ind,

transfer of ACK_req and GET, respectively.

**Property 5** As a combination of the sender and receiver, the input message PUT is

followed by the output message GET.

LTL: $\Box (p \rightarrow \Diamond q)$ where $p$ and $q$ represent the transfer of PUT and reception of

GET.

**Property 6** The messages PUT and GET must occur alternatively.

LTL: $\Box (0 \leq (n_p - n_g) \leq 1)$ where $n_p$ and $n_g$ denote the numbers of

occurrences of PUT and GET.

From the Promela model and the properties mentioned above, we can perform

the validation of the initial design.

During the validation, there is one problem at the receiver when the model is

checked with Property 3, or the existence of DATA_ind. SPIN generates an error

trace which shows that the DATA_req sent from the sender is lost at the lower layer

repeatedly without any progress. As a solution for this problem, we can limit the

number of trials of message DATA_req by replacing the pattern timed confirmed

sender with the pattern timed retrial confirmed sender. Thus, when the DATA is
lost more than $N$ times consecutively, the sender gives up the communication. As a result, we have an updated Property 3 saying that reception of $\text{DATA\_ind}$ has to occur eventually. Otherwise, the sender gives up the communication after the $N$ times timeout. Property 2 is also changed that the request $\text{DATA\_req}$ is always followed by either reception of $\text{ACK\_ind}$ or $N$ times timeout.

Another problem occurs at the receiver when we check the correspondence of $\text{PUT}$ and $\text{GET}$ as in Property 6. Because of the possibility of $\text{ACK}$ loss at the lower layer, the receiver could take a duplicated DATA. The receiver, therefore, has to check the duplication of received DATA and ignore a duplicated one although it replies the $\text{ACK}$ again. The checking can be achieved by appending one sequential bit called alternate bit to the DATA and inspecting the bit. The behavior is specified by predicates to examine the alternate bit after receiving a DATA.

After modifying the model and properties for the problems, we can validate it again. In this time, the updated model passes all properties at the synchronous communication. Nevertheless, one more problem still remains when the layer uses asynchronous communication, i.e., $\text{BUFFSIZE}$ is greater than zero. If sender receives a timeout signal for a DATA, it assumes that either a previous DATA or an ACK for the DATA is lost at the lower channel. Thus, it sends the DATA again. But, this assumption is not always true because there is a possibility of message delay. Suppose the sender had transferred a DATA with setting the alternate bit zero. Then, it transferred the DATA again after it received a timeout signal. Next, the sender received an ACK for the first DATA which had been delayed. Since it received the ACK, the sender transmits the second DATA which happens to be lost at the lower channel. Meanwhile, the receiver sends another ACK for the duplicated DATA. Unfortunately, the sender considers this ACK as an acknowledgment for the second DATA which was accidentally lost. Hence, sender transfers third DATA in which the alternate bit has zero again. The receiver
ignores the message because the alternate bit is not what it expects. Thus, this situation results in consecutive loss of two DATA messages. Tracing of this error is quite long and hard to find, but SPIN makes it possible to find this kind of error.

To fix the problem, the sender has to wait for an ACK that contains the same bit as the alternate bit being sent. If the sender receives the wanted ACK, it flips the alternate bit and sends the next DATA. The checking for the control bit of ACK can also be achieved by introducing predicates to inspect the bit. The updated description of ABP behavior is shown in Figure 5–3 where d, a, e, b, c means data, alternate bit, expected bit, control bit, and counter, respectively.

![Diagram of ABP behavior]

**Figure 5–3: Final description of ABP behavior**

Validation for the updated model shows that the design is error-free for the correctness properties. We, therefore, can go to the next development phase. Appendix A presents the PROMELA model of Figure 5–1 and Figure 5–3. It also includes the properties checked at the final validation.

Meanwhile this case study shows the possibility of a new pattern construction from the current pattern language. Actually, the ABP is a common protocol at the communication systems. Therefore, we register the CEFSMs described in Figure 5–3 as new patterns with the names of *ABP sender* and *ABP receiver*. 
5.2 Case Study: ATM UNI Signaling Protocol

5.2.1 Protocol Overview

ATM signaling [65, 80, 81] is a connection-oriented protocol and dynamic behavior for connection setup and release is achieved by signaling, an exchange of control information in a communication network. It is broadly classified by two interfaces: User-Network Interface (UNI) and Network-Node Interface (NNI). UNI signaling specifies interactions between an ATM end system and an ATM access switch, while NNI is used between switches within an ATM network. Our case study is devoted to the development of UNI signaling, but the method can be similarly used for NNI signaling. There are many standards for ATM UNI signaling such as Q.2931 [82], Q.2971 [83], UNI 3.1 [84], and UNI 4.0 [85] from the ITU-T and the ATM-Forum. The standards define a set of capabilities, messages, and procedures for user connections. Stiller [86] surveyed commonalities and differences between the standards. Our case study is based on the ITU-T’s Q.2931.

Figure 5-4 depicts the ATM signaling protocol stack [81]. The stack consists of

![Diagram of the ATM signaling protocol stack]

Figure 5-4: Protocol stack for signaling in ATM control plane

the signaling layer 3 or signaling protocol layer, the signaling ATM adaptation layer (SAAL), the ATM layer, and the physical layer. The signaling layer 3 has different functions depending on the interface over which it is being used. The UNI signaling protocol layer establishes, maintains, and releases user connections for incoming
and outgoing calls. Either user applications or switch functions such as routing, call admission control and management can be located on top of the layer [65].

SAAL provides reliable transfer of signaling messages between instances of signaling layer 3. It adapts the protocol data unit of a signaling layer to ATM cells. The layer has several sublayers such as service specific coordination function (SSCF), service specific connection oriented protocol (SSCOP), and ATM adaptation layer (AAL). The SAAL functions are accessed by the signaling layer 3 through AAL service access points. SSCF supplies a mapping function between SSCOP and signaling layer 3. It simplifies the upper layer interface of SAAL with regard to SSCOP. There are two kinds of SSCF, an SSCF at UNI [66] and an SSCF at NNI [87], because the needs of signaling layer 3 are different at each interface. SSCOP [88] implements a reliable transport protocol which sends signaling messages using sequence numbers, acknowledgments, and retransmissions to ensure an error-free sequence delivery. The AAL sublayer is used by SSCOP to send and receive AAL messages to the ATM layer. Typically, this layer is implemented in hardware for performance reason. Both ATM layer and physical layer are also implemented in hardware. The SSCOP and SSCF layers are usually in software and are often referred to as SAAL [81].

Recall that an ATM connection has three phases: establishment, data transfer, and release. During the establishment phase, an end system negotiates connection characteristics with a network. The connection is cleared during the release phase after data is exchanged. Figure 5-5 shows a typical message flow for an ATM connection [65,81]. To establish a connection, a calling party sends out an SETUP message to an ATM network. The message contains lower layer parameters, called party address, quality of service requirements, and other information needed for the connection. The message is received at an access switch of an ATM network. After checking the availability of resources, the switch may acknowledge the reception
Figure 5-5: Message flow for connection establishment and release
of the message by responding with a CALL_PROC to indicate that the call is proceeding well. Then the SETUP message is forwarded to a called party across the network. Along the way, each network switch determines if it has enough resources to serve the connection. If there is a problem within the network or at the called party, the call is rejected by returning either a REL or a REL_COMP message to the calling party. When the SETUP message reaches the called party, it may answer with a CALL_PROC message to indicate that it is going to start the processing of the connection. The called party may send an ALERT message to imply an alerting has begun at the called party. Note that both CALL_PROC and ALERT messages are optional, though we assume that these messages exist in our case study. Whether to send a CALL_PROC in response to a SETUP is usually a configuration feature of the access switch. Sending an ALERT is a feature of the end system software [65]. When the called party is ready to connect, it sends a CONN message to the calling party backwards through the network. The message is answered with a CONN_ACK. At this point, a connection is established and data can be exchanged.

To release the connection, which can be initiated by a calling party or a called party, a REL message is sent to the other side. This message includes the reason of release. The message is acknowledged with a REL_COMP after releasing the resources allocated. The connection release of Figure 5-5 is initiated by the called party. In addition to the message sequence, the protocol manages timing constraints of messages. There are many timers such as T303, T310, T301, and T308 for confirmed acknowledgments. Typically, if a timer expires, the call to be connected is cleared.

It is important to note that each side of UNI provides a different function, and thus the signaling protocol is defined separately for an ATM end system and
an ATM access switch. In this case study, we consider only the switch side of the protocol. However, the behavior of an end system can be similarly obtained.

### 5.2.2 Structural Design

An ATM access switch has two distinct functions to deal with a calling party and a called party. This separation of functionality leads the switch to be designed in the pattern split dynamic handler as shown in Figure 5–6. The block contains three internal blocks: MAIN CC to administrate calls, ORG CC to handle outgoing calls or originating calls, and TERM CC to handle incoming calls or terminating calls. The pattern also needs the definition of communication paths and messages between the blocks.

A static instance MAIN CC creates instances of ORG CC and TERM CC depending on the messages received. When a calling party tries a new call, the SETUP request comes to the MAIN CC from the lower layer through the path UNI2MAIN. Upon receiving the message, MAIN CC creates an instance of ORG CC to handle the originating call. The signaling messages between a calling party and the instance flows through the communication paths UNI2ORG and ORG2UNI via the lower layer of the protocol. To connect the call to a called party, it is necessary for the instance to inform the called party of the arrival of a new
call. The internal message SETUP.int is used for this purpose. The message makes the MAIN_CC create an instance of TERM_CC to handle the terminating call.

As mentioned earlier, this case study deals with UNI, and thus we consider only a local call in one ATM switch. In other words, the instances of ORG_CC and TERM_CC exist in the same switch. If there are many switches in the network and the call flows through the switches as a long distance call, the instances would need different protocols such as NNI. Note that ORG2TERM and TERM2ORG are used for the internal messages such as ALERT.int, CONN.int and REL.int. These messages are represented in the dotted line in Figure 5-5. Note that the difference between normal messages and internal messages. The normal messages are transmitted through a lower layer of the signaling protocol, whereas the internal messages are exchanged among block instances.

5.2.3 Behavioral Design

5.2.3.1 Behavior of MAIN_CC

As shown in Figure 5-6, MAIN_CC takes the role of administrator of the pattern split dynamic handler for the messages SETUP, SETUP.int, and TERMINATE. It creates instances of ORG_CC and TERM_CC for the messages SETUP and SETUP.int. After finishing communication, each instance informs the MAIN_CC of the termination of execution using the message TERMINATE. The

![Figure 5-7: MAIN_CC as an administrator of split dynamic handler](attachment:image.png)

Figure 5-7: MAIN_CC as an administrator of split dynamic handler

behavior is given in Figure 5-7. The whole behavior of MAIN_CC is obtained by
merging the CEFSMs of Figure 5-7 through the pattern source merge at the state
\texttt{main\_wait}.

We can extract the properties of \texttt{MAIN\_CC} behavior from the CEFSMs. In the case of Figure 5-7 (a), the input event \texttt{SETUP} has to be followed by the transferring of \texttt{SETUP} to the newly created \texttt{ORG\_CC}. Furthermore, the two events have to occur alternatively. These properties are also applicable to the \texttt{SETUP\_int}. Then, \texttt{TERMINATE} must follow the corresponding \texttt{SETUP} or \texttt{SETUP\_int}. For example, if an \texttt{ORG\_CC} instance was created once, it has to cease to exist sometime later in any situations. The property could be expressed in LTE as \(\square(\texttt{SETUP} \rightarrow \Diamond(\texttt{TERMINATE})).\)

5.2.3.2 Outgoing Call Establishment

\textbf{Confirmed receiver for SETUP.} The block \texttt{ORG\_CC} handles an outgoing call from a calling party at the access switch. When it receives a \texttt{SETUP} message from a calling party, it checks the possibility of connection first. If the service is possible in the node, the block allocates the resources needed for the connection. Then, it acknowledges with a message \texttt{CALL\_PROC} to the calling party to indicate that the call is being processed properly. If the connection is failed, it replies with a \texttt{REL\_COMP} message to notify that the connection is not possible and being cleared. This behavior is well-suited for the pattern confirmed receiver as in Figure 5-8 (a) where the \texttt{SETUP} is confirmed with either \texttt{CALL\_PROC} or \texttt{REL\_COMP}. Note that the corresponding sender of the confirmed receiver exists at the calling party. Because the behavior of the calling party is not our concern, we do not present it here. However, it is possible to design the behavior with the pattern timed retrial confirmed sender similarly as the handling of \texttt{SETUP} at the \texttt{TERM\_CC} which is given in the incoming call establishment. In addition to the communication with a calling party, \texttt{ORG\_CC} has to inform the \texttt{MAIN\_CC} that there is an incoming call request so that it initiates a \texttt{TERM\_CC}. This notification
is performed using the SETUP.int. From the pattern used in the design, we are

Figure 5–8: Behavior of ORG_CC for SETUP, ALERT, and CONN

able to capture the properties to be checked in validation phase. The input event

SETUP has to be followed by either CALL_PROC or REL_COMP. Moreover, other events in the block cannot proceed SETUP because it is the first event to have happened in the block.

**Unconfirmed sender for ALERT.** After handling the message SETUP, the block ORG_CC waits for an internal message ALERT.int from a TERM_CC from which it knows that a called party is being alerted. Then, it transfers the message ALERT to the calling party. This behavior can be designed in unconfirmed sender. The corresponding calling party uses timed receiver to handle it. Figure 5–8 (b) represents the behavior. We can extract an existence and response property of ALERT from the pattern property section.

**Confirmed sender for CONN.** After the ALERT message, the called party notifies the connection establishment using the internal message CONN.int. For the event, ORG_CC transfers the message CONN to the calling party and then it waits for a response from the calling party. There are two possibilities, either CONN_ACK or REL, from the party. This behavior can be achieved using the pattern confirmed sender, whereas the calling party uses timed confirmed receiver to handle it. Figure 5–8 (c) shows the behavior. The detailed description
of connection release is given later. From the pattern, we know that CONN.int has
to occur eventually and CONN subsequently follows the event. Furthermore, the
CONN is responded by one of confirmation messages.

The whole behavior of ORG.CC is obtained using the pattern sequence merge
for the CEFSMs designed in Figure 5–8. The state null is the initial state of the
merged CEFSM. Accordingly, we need to check the properties for the final CEFSM
such as SETUP must occur to initiate the operation and it has to be followed by
the reception of REL.COMP, REL, or CONN.ACK.

5.2.3.3 incoming call establishment

Timed retry confirmed sender for SETUP. The connection establishment
for an incoming call is performed by the block TERM.CC. When the block
is notified of a new call trial by SETUP.int, it sends a SETUP message to the
destination called party and waits for an acknowledgment in the timing interval
set by the timer T303. The block tries at least twice for a response. If it receives
CALL_PROC as an indication of successful proceeding, it moves to the next step.
However, if a REL.COMP is responded due to the connect rejection, the block re-
sets the timer and clears the call. If the block fails to receive any acknowledgment
in two trials, it recognizes this situation as a failure of the request and clears the
connection. All behavior to handle the SETUP can be designed by the pattern
timed retry confirmed sender as shown in Figure 5–9. Meanwhile, the corre-
sponding called party uses the pattern confirmed receiver to handle the message,
which is similar to the case of SETUP at ORG.CC as in Figure 5–8 (a). From the
pattern, it is possible to capture the properties of TERM.CC such as the input
event SETUP.int has to be followed by one subsequent input among CALL_PROC,
REL.COMP, and T303 timeout. Note that T303 cannot happen more than twice.

Timed receiver for ALERT. As shown in Figure 5–5, an access switch
waits for an alerting signal ALERT in a timing interval set by T310 after receiving
Figure 5–9: Behavior of TERM_CC for SETUP message

the CALL_PROC. If the block fails to receive the expected event ALERT in the
given time interval, it releases the connection and clears the call towards both the
calling and called parties. The pattern timed receiver is a good candidate for the
behavior as shown in Figure 5–10 (a). Note that the initialization of the timer
must be done prior to the state incall_proceed. The initialization is performed
during the transition from call_present to incall_proceed because it is a unique
transition coming to the state.

Figure 5–10: Behavior of TERM_CC for ALERT and CONN

**Timed confirmed receiver for CONN.** The block TERM_CC turns
on the timer T301 to measure the reception of the message CONN after the
ALERT. When the called party is ready to connect the call, it sends a connection
messageCONNtothecallingside. Upon receiving the CONN, TERM_CC stops T301 and replies to the called party with CONN_ACK. Then, it forwards the internal message CONN.int to ORG_CC to indicate that the connection is accepted. If the block fails to receive the expected event CONN in the given time interval, it releases the connection and clears the call. This behavior can be implemented in the pattern timed confirmed receiver as given in Figure 5–10 (b). The corresponding called party uses timed confirmed sender to handle these messages. The block has the property that it has to response with CONN_ACK or T301 for the CONN message. To get an entire behavior of TERM_CC, the CEFSMs of Figure 5–9 and 5–10 should be composed with sequential merge as ORG_CC. As a result, the block has to take an input event SETUP.int and then cease to exist by timers or REL.COMP message. Otherwise, it must stay at the active state after handling CONN.

5.2.3.4 Call/Connection Release

For the connection release, each block of ORG_CC and TERM_CC has the same functionality. There are two cases in the connection release.

Confirmed receiver for local party REL. Upon receiving a REL message initiated from a local party, the access switch releases resource allocated for the call and replies with REL.COMP to the party. This is an instantiation of the pattern

![Diagram](image)

Figure 5–11: Call clearing for an ATM connection
confirmed receiver. Figure 5-11 (a) shows the handling of connection release at ORG_CC and TERM_CC. Subsequently, the instance informs the remote party that the call is released using the internal message REL.int. The remote party uses timed retry confirmed sender to handle it as presented below. From the design we can extract the property that REL is followed by REL.COMP and terminates after it.

**Timed retry confirmed sender for remote party REL.** Figure 5-11 (b) shows the situation where the event REL.int happens from the remote instance. The network side initiates call clearing by sending REL to the local party and starts timer T308. Then, it waits for REL.COMP from the party. If it receives the message, the network instance stops the timer and completes the connection. For the timer expiration, it can try again. For the secondary expiration, the network places the connection in a maintenance condition and terminates it [80]. This behavior is an instance of pattern timed retry confirmed sender. From the design, we can extract the property that the REL is followed by either REL.COMP or T308 at most twice.

### 5.2.4 Validation of Pattern-based Design

In this subsection, we present the model construction and validation of the ATM UNI signaling protocol design.

#### 5.2.4.1 Model Construction in Promela

By converting the architectural design of Figure 5-6 to the corresponding Promela constructs, we can obtain an outline of the validation model as follows:

```plaintext
#define BUFFSIZE 2  /* Channel size. Changeable. */

/* message declaration */
mtype = {SETUP, CALL_PROC, ALERT, CONN, CONN_ACK, REL, REL.COMP, TERMINATE, SETUP_int, ALERT_int, CONN_int, REL_int};

/* channel declaration */
chan UNI2MAIN = [BUFFSIZE] of {mtype, Para};
chan ORG2MAIN = [BUFFSIZE] of {mtype, Para};
chan MAIN2ORG = [BUFFSIZE] of {mtype, Para};
```
chan TERM2MAIN = [BUFFSIZE] of {mtype, Para};
chan MAIN2TERM = [BUFFSIZE] of {mtype, Para};
chan ORG2TERM = [BUFFSIZE] of {mtype, Para};
chan TERM2ORG = [BUFFSIZE] of {mtype, Para};
chan UNI2ORG = [BUFFSIZE] of {mtype, Para};
chan UNI2TERM = [BUFFSIZE] of {mtype, Para};
chan ORG2UNI = [BUFFSIZE] of {mtype, Para};
chan TERM2UNI = [BUFFSIZE] of {mtype, Para};

/* process declaration */
proctype MAIN_CC() {}
proctype ORG_CC() {}
proctype TERM_CC() {}
proctype Calling_Party() {} /* environment process */
proctype Called_Party() {} /* environment process */
init {} /* initialization process */

The special process init instantiates the initial processes of the model by using
the run operator. In this case study, the static instance of MAIN_CC is created
from the process. For the closeness of the model, it is necessary to include a calling
party and a called party for interaction with the ORG_CC and TERM_CC. Thus,
we have two environments processes, Calling_Party and Called_Party.

From the model outline, we can acquire a complete PROMELA model by
converting all CEFSMs drawn in design phase to the corresponding PROMELA
constructs. Appendix B presents the complete model of the design.

5.2.4.2 Result and Experience

Deadlock and unreached code. As a typical procedure of SPIN validation,
we first check the possibility of a deadlock and the existence of unreached code.
One deadlock situation happens when both parties initiate the release of connection
at the same time. For instance, when an instance of ORG_CC or TERM_CC
waits for a REL_COMP from its local end party, it could receive a REL instead of
REL_COMP. That is, at the state release_indication of Figure 5-11 (b), an instance
have requested a connection release to the end party upon receiving a REL.int from
a remote peer. However, the end party has also requested the connection release
at the same time. This situation is called a clear collision [65], and we have not
considered this situation at the design phase. We can solve it by introducing one
more transition \textit{REL/-/TERMINATE} from the state \textit{release\_indication}. Another clear collision exists when each instances receives a REL from its local party and then transfers it to the peer using REL\_int. The peer party can also send the REL\_int at the same time. This is solved by checking and consuming a pending REL\_int from a peer before and after sending its REL\_int.

Another situation of deadlock happens when both parties wait for a request of connection release from the other party forever. To fix the problem, we change the end parties of the model so that at least one of them begins the disconnection.

The third situation of deadlock occurs at the ORG\_CC block because it does not provide the handling of abnormal connection release request during the connection establishment. It is possible for an instance of TERM\_CC to notify the connection failure due to the expiration of timer T310 or T301. The reason for this deadlock is that we missed the handling of the connection failure at the design phase. SPIN pinpoints the potential deadlock exactly. The situation is fixed by adding a transition to handle REL\_int at the states \textit{outcall\_proceed} and \textit{call\_delivered}.

Identification of unreachable code is important in the system validation because the unreached code could have a potential fault. In our case, some PROMELA code has not reached because we have not fully simulated the message loss in the environment processes.

\textbf{Property validation.} After SPIN has fixed obvious design flaws such as system deadlock and unreachable code, the system model is validated against system properties. From the patterns used in the design, we can obtain several properties of the ATM UNI signaling protocol as follow.

\textbf{Property 1} The input message SETUP has to occur eventually, and it is followed by the transferring of SETUP to the newly created ORG\_CC.

LTL: $\Diamond SETUP \land \Box (SETUP \rightarrow \Diamond (SETUP \text{ to ORG\_CC}))$. 
Property 2 The internal event SETUP.int has to occur eventually, and it is followed by the transferring of SETUP.int to the newly created TERM_CC.

LTL: $\diamond SETUP\text{-}int \land \Box (SETUP\text{-}int \rightarrow \diamond (SETUP\text{-}int to TERM\text{-}CC))$.

Property 3 TERMINATE must follow the corresponding SETUP or SETUP.int.

For example, if an ORG_CC instance was created once, it has to cease to exist sometime later in any situation.

LTL: $\Box (SETUP \rightarrow \diamond (TERMINATE))$.

Property 4 The event SETUP has to occur eventually to initiate the behavior and be followed by either CALL_PROC or REL_COMP.

LTL: $\diamond SETUP \land \Box (SETUP \rightarrow \diamond ((CALL\_PROC \land \neg REL\_COMP) \lor (\neg CALL\_PROC \land REL\_COMP)))$.

Property 5 Because the SETUP is the first event to have happened in the block, other events cannot proceed it in this block.

LTL: $\Box (\neg e_o) \lor (\neg e_o U SETUP)$, where $e_o$ means all other events of the block except SETUP.

Property 6 The internal event ALERT.int has to exist and ALERT follows it.

LTL: $\diamond ALERT\text{-}int \land \Box (ALERT\text{-}int \rightarrow \diamond ALERT)$.

Property 7 CONN.int has to occur eventually and CONN subsequently follows the event.

LTL: $(\diamond CONN\_int \land \Box (CONN\_int \rightarrow \diamond CONN))$.

Property 8 CONN responds by one of the confirmation messages REL or CONN_ACK.

LTL: $\Box (CONN \rightarrow \diamond ((REL \land \neg CONN\_ACK) \lor (\neg REL \land CONN\_ACK)))$.

Property 9 SETUP has to be followed by reception of REL_COMP, REL, or CONN_ACK.
LTL: $\Box(SETUP \rightarrow \Diamond((REL\_COMP \land \neg REL \land \neg CONN\_ACK) \lor
(\neg REL\_COMP \land REL \land \neg CONN\_ACK) \lor (\neg REL\_COMP \land \neg REL \land
CONN\_ACK)))$.

**Property 10** The input event $SETUP\_int$ has to occur eventually and be
responded by SETUP.

LTL: $(\Diamond SETUP\_int \land \Box(SETUP\_int \rightarrow \Diamond SETUP))$.

**Property 11** SETUP is followed by one of confirmations or two times timeout of
T303.

LTL: $\Box(SETUP \rightarrow \Diamond(((T303 \land (c = 2)) \land \neg REL\_COMP \land \neg CALL\_PROC) \lor
(\neg(T303 \land (c = 2)) \land REL\_COMP \land \neg CALL\_PROC) \lor (\neg(T303 \land (c =
2)) \land \neg REL\_COMP \land CALL\_PROC)))$, where $c$ denotes the number of
timeout.

**Property 12** After second timeouts, no timeout T303 exists.

LTL: $\Box(c \leq 2)$, where $c$ denotes the number of timeout.

**Property 13** CALL\_PROC has to be followed by either the intended message
ALERT or a timeout T310.

LTL: $\Box(CALL\_PROC \rightarrow \Diamond((ALERT \land \neg T310) \lor (\neg ALERT \land T310)))$.

**Property 14** ALERT has to be followed by either the intended message CONN
and CONN\_ACK or a timeout T301.

LTL: $\Box(ALERT \rightarrow \Diamond((CONN \land \neg T301) \lor (\neg CONN \land T301)))$.

**Property 15** For the SETUP\_int, the block TERM\_CC cease to exist or stay at
the active state after handling CONN.

LTL: ADDME LTL HERE!

**Property 16** The input event REL has to occur eventually and it is followed by
the occurrence of REL\_COMP.

LTL: $\Diamond REL \land \Box(REL \rightarrow \Diamond REL\_COMP)$.
**Property 17** The input event \( REL.int \) has to occur eventually and be responded by \( REL \).

\[
\text{LTL: } (\Diamond REL.int \land \Box (REL.int \rightarrow \Diamond REL)).
\]

**Property 18** \( REL \) is followed by either \( REL_{COMP} \) or two times timeout of \( T308 \).

\[
\text{LTL: } \Box (REL \rightarrow \Diamond ((T308 \land (c = 2)) \land \neg REL_{COMP}) \lor (\neg(T308 \land (c = 2)) \land REL_{COMP}))), \text{ where } c \text{ denotes the number of timeout.}
\]

**Property 19** After second timeouts, no timeout \( T308 \) exists.

\[
\text{LTL: } \Box (c \leq 2), \text{ where } c \text{ denotes the number of timeout.}
\]

Most of the properties are satisfied in the validation. One problem is at the property 2 in which an existence of SETUP.int at the \( TERM_{CC} \) is checked. The reception can not be possible if \( ORG_{CC} \) rejects the SETUP request continuously. Thus, it is necessary to avoid this situation at the protocol design. It it important to note that the properties used in this validation phase are not only for the validation of the design, but also for later development phases. They can be used, for instance, when an integration testing of a whole system is performed including the environment blocks of the model. These are, therefore, a valuable document for the system maintenance.

**State explosion in multiple instances.** At the initial validation, we have validated the model with one pair of handlers. In other words, there were one calling party and one called party in the model. For the validation of protocol with multiple handlers, we increase the number of caller and callee instances. In this situation, we realized that it is necessary to identify the address of each instance for proper communication. Additionally, the validation has reached the default memory limitation (64M) immediately. Thus, we need to use compression and a supertrace technique as well as increase the memory.
CHAPTER 6
CONCLUSION

A pattern is a powerful artifact for design reuse. It describes a solution to a problem that recurs several times in similar situations. In this dissertation, we, first, suggested a communication pattern language for the high-level description of communication protocols. Our pattern language is composed of structural and behavioral patterns. One feature of the pattern description is to use an STD for a CEFSM which makes it easier to represent a design in a visual notation. Finite state machines are commonly used in protocol description and the STD can describe the behavior in states, events, and actions. Moreover, the STD helps to translate the design into our validation language PROMELA systematically. Another feature of the pattern language is that the patterns include the SDL implementation section which assists a developer with usage of the patterns in applications to be implemented in the language. Furthermore, our pattern language has a hierarchical and expandable pattern framework. High-level patterns can be constructed from elementary patterns. It is also possible to make patterns for other domains from the framework.

Second, we proposed a software development methodology for a communication protocol system emphasizing the high-level design of a system and the validation of that design using a model checking technique. When a system designer develops a protocol, patterns are used for the abstract description of the protocol. The designer is able to devise an architecture of the protocol system with structural patterns. Then, the interactions of messages among the architectural blocks are represented with behavioral patterns. The system design description
acquired at the design step is a good maintenance document for a protocol system. After the pattern-based design, a validation on the design is conducted on a Promela model of the design. The validation helps to find design faults in the early stages of the development.

Third, the validation using a model checker was described in detail. For the validation of a pattern-based design, it is necessary to construct a validation model and identify the system properties. Checking the correctness and consistency of a design is achieved by Spin and Promela. A Promela model can be obtained systematically from the patterns used in the design. Special attention is necessary for the communication paths, timer, and environment processes. One of our contributions is to present the properties in the property specification section in pattern description. As a result, a designer can ensure several correctness properties such as event occurrence, ordering among the events, correspondence or alternativeness of events during the design phase. Although the formal validation provides designers significant advantages, it has not been widely used in industry because of some practical barriers such as difficulty of formal descriptions of system properties. The patterns enable non-experts to obtain the correctness properties from the pattern description.

Fourth, to show the feasibility of our methodology, we have performed several case studies such as the alternate bit protocol and ATM UNI signaling protocol. These case studies provide a proof of concept for the usefulness of the pattern language and validation technique. We were also able to mine the designs to obtain additional high level patterns.

Although our methodology provides useful support for protocol development, there are several issues to be addressed in the future. A pattern language is not a closed set. Our pattern language could be enhanced by the addition of more high-level patterns. We can extend the language by either composing the patterns
or specializing the existing ones as well as developing new ones. The framework of the pattern language will be valid for further pattern development.

Tool support is necessary to make it possible to provide automatic creation, selection, and adaptation of patterns. The translation of patterns into PROMELA code is also essential. For the tool support, the concrete syntax and semantics for the pattern language need to be defined.

Our approach to model the timer and its operations provides an approximate solution to reflect elapsed time in the model. In fact, it is a limitation of the current SPIN version and not trivial to provide a concrete solution without changing of SPIN’s internals. As a complement to SPIN, we can investigate other tools such as UPPAAL and KRONOS for the design validation. As an another approach, it would be possible to represent our STD in a common representation IF [89]. Thus, we could use many tools that support the format.

In this research work, we presented a pattern language for the high-level description of communication protocols and suggested a validation mechanism for the description to find design error at the early development phase. The system design description will be a useful maintenance documentation after the development of a protocol because it can show the abstract architecture and behavior of the protocol in a few patterns. We also believe that the patterns are applicable to other communication areas. Communicating blocks and finite state machines are well-known means to describe these applications.
#define BUFFSIZE 1
#define N 3 /* number of iteration for timeout retrial */
define MAX 4 /* maximum sequence number for DATA */
define message byte
#define timer bool

/* message declarations */
mtype = {PUT, GET, DATA_req, DATA_ind, ACK_req, ACK_ind};

/* channel declaration */
chan SND2LO = [BUFFSIZE] of {mtype, byte, bit}; /* DATA_req */
chan L02RCV = [BUFFSIZE] of {mtype, byte, bit}; /* DATA_ind */
chan RCV2LO = [BUFFSIZE] of {mtype, bit}; /* ACK_req */
chan L02SN = [BUFFSIZE] of {mtype, bit}; /* ACK_ind */
chan UP2SN = [BUFFSIZE] of {mtype, byte}; /* PUT */
chan RCV2UP = [BUFFSIZE] of {mtype, byte}; /* GET */

/* PID of processes which are used for validation */
byte P_sender;
byte P_receiver;
byte P_UPPER;

proctype sender()
{
    message d; /* DATA field */
    bit a; /* alternate bit */
    bit b; /* control bit for ACK */
    timer T = false; /* timer. Initially deactivated. */
    byte c; /* counter for iteration */

    wait_data:
    end:
    if
        :: UP2SN?PUT(d) ->
    PUT_rcv: /* label for validation */
        T = true;
        c = 1;
        SND2LO!DATA_req(d,a);
        DATA_req_snt: /* label for validation */
        goto wait_ack;
    fi;

    wait_ack:
    if

:: LO2SND?ACK_ind(b) ->
  if
    :: a == b -> /* expected ACK */
  ACK_ind_rcv: /* label for validation */
    a = !a; /* toggle the alternate bit for next DATA*/
    T = false; /* timer reset() */
    goto wait_data;

    :: a != b ->
    T = true; /* timer set again */
    SND2LO!DATA_req(d,a);
    goto wait_ack;
  fi;

  :: T -> /* timer expiration */
  if
    :: c < N ->
    c = c + 1;
    T = true;
    SND2LO!DATA_req(d,a);
    goto wait_ack
  :: c == N ->
    T_rcv: /* N times timeout */
    printf ("ERROR: SO MANY DATA LOST\n");
  fi;
  fi;
} /* end of "sender()" */

proctype receiver()
{
  bit e = 0; /* expected bit */
  bit a; /* alternate bit */
  message d; /* DATA field */

  ready:
  end:
  LO2RCV?DATA_ind(d,a) ->
    if
      :: (a == e) ->
    DATA_ind_rcv: /* label for property validation */
      e = !e;
      RCV2LO!ACK_req(a);
    end
    ACK_req_snt: /* label for property validation */
      RCV2UP!GET(d);
    end
    GET_snt: /* label for property validation */
      goto ready;

    :: (a != e) -> /* duplicated DATA */
      RCV2LO!ACK_req(a);
      goto ready;
  fi;
} /* end of "receiver()" */

proctype UPPER()
{ message s; /* sending DATA */
message r; /* receiving DATA */
message e; /* expected DATA */

idle:
end:
do
:: atomic {
    UP2SND!PUT(s);
    PUT_snt: /* label for validation */
        s = (s+1) % MAX
}
:: atomic {
    RCV2UP?GET(r);
    GET_rcv: /* label for validation */
        assert (r == e); /* checking the alternativity
                          of Property 6 */
        e = (e+1) % MAX
}
od
} /* end of "UPPER()" */

proctype LOWER()
{
    message d;
    bit a;
    bit b;

idle:
end:
do
:: SND2LO?DATA_req(d,a) ->
    if
    :: LO2RCV!DATA_ind(d,a); /* proper DATA transfer */
    :: skip; /* lost DATA */
    fi
:: RCV2LO?ACK_req(b) ->
    if
    :: LO2SND!ACK_ind(b); /* proper ACK transfer */
    :: skip; /* lost ACK */
    fi
od
} /* end of "LOWER()" */

init{
    atomic{ P_sender  = run sender();
            P_receiver = run receiver();
            P_UPPER   = run UPPER();
            run LOWER(); }
}
The following enumerates the properties checked during the validation step for the Promela model.

**Property 1** The event PUT has to be received eventually and DATA_req follows it.
LTL: $(\Diamond p \land \square (p \rightarrow \Diamond q))$ where $p$ and $q$ represent the reception of PUT and transfer of DATA_req.

**Property 2** The request DATA_req is always followed by either reception of ACK_ind with $(a = b)$ or $N$ times timeout.
LTL: $\square (p \rightarrow \Diamond ((q \land \neg r) \lor (\neg q \land r)))$ where $p$, $q$, and $r$ denote the transfer of DATA_req, reception of ACK_ind with $(a = b)$, and $N$ times timeout of timer $T$, respectively.

**Property 3** Reception of DATA_ind has to occur eventually or $N$ times timeout should happen.
LTL: $\Diamond (p \lor q)$ where $p$ and $q$ means the reception of DATA_ind and $N$ times timeout.

**Property 4** If the DATA_ind is received and $a$ is equal to $e$ at that time, the indication is always followed by sending of ACK_req and GET.
LTL: $\square (p \rightarrow \Diamond (q \land \Diamond r))$ where $p$, $q$, and $r$ represent the reception of DATA_ind with $(a = e)$, transfer of ACK_req and GET, respectively.

**Property 5** As a combination of the sender and receiver, the input message PUT is followed by the output message GET.
LTL: $\square (p \rightarrow \Diamond q)$ where $p$ and $q$ represent the transfer of PUT and reception of GET.

**Property 6** The messages PUT and GET must occur alternatively.
LTL: $\square (0 \leq (n_p - n_g) \leq 1)$ where $n_p$ and $n_g$ denote the numbers of occurrences of PUT and GET.
#define BUFFSIZE 4 /* Size of message buffer */
#define timer bool

/* Symbols */
#define OK 1
#define NOK 0

/* message declarations */
mtype = {SETUP, CALL_PROC, ALERT, CONN, CONN_ACK, REL, REL_COMP,
         TERMINATE, SETUP_int, ALERT_int, CONN_int, REL_int};

/* Parameter repository. All parameters are included in the structure */
typedef Para {
    byte calling_addr; /* keep the calling party address */
    byte called_addr; /* keep the called party address */
    byte org_addr; /* keep the ORG_CC address */
    byte term_addr; /* keep the TERM_CC address */
}

/* Channel Declaration. */
chan UNI2MAIN = [BUFFSIZE] of {mtype, Para};
chan ORG2MAIN = [BUFFSIZE] of {mtype, Para};
chan MAIN2ORG = [BUFFSIZE] of {mtype, byte, Para};
chan TERM2MAIN = [BUFFSIZE] of {mtype, Para};
chan MAIN2TERM = [BUFFSIZE] of {mtype, byte, Para};
chan ORG2TERM = [BUFFSIZE] of {mtype, byte, Para};
chan TERM2ORG = [BUFFSIZE] of {mtype, byte, Para};
chan UNI2ORG = [BUFFSIZE] of {mtype, byte, Para};
chan UNI2TERM = [BUFFSIZE] of {mtype, byte, Para};
chan ORG2UNI = [BUFFSIZE] of {mtype, byte, Para};
chan TERM2UNI = [BUFFSIZE] of {mtype, byte, Para};

byte ADDR = 1; /* Used for a unique number generation for addressing */

/* Used for property validation. Keep the PID of each instance. */
byte P_MAIN_CC;
byte P_ORG_CC;
byte P_TERM_CC;

/* Main Call Control Block */
proctype MAIN_CC()
{
    Para para;
main_wait:
end_MAIN_CC:
do
:: UNI2MAIN?SETUP(para) ->
SETUP_rcv: /* label for property check */
d_step { /* distribute a unique address */
    para.org_addr = ADDR;
    ADDR = ADDR + 1;
}
P_ORG_CC = run ORG_CC(para.org_addr);
/** save instance id **/
MAIN2ORG!SETUP(para.org_addr, para);
:: ORG2MAIN?SETUP_int(para) ->
SETUP_int_rcv: /* label for property check */
d_step {
    para.term_addr = ADDR;
    ADDR = ADDR + 1;
}
P_TERM_CC = run TERM_CC(para.term_addr);
/** save instance id **/
MAIN2TERM!SETUP_int(para.term_addr, para);
:: ORG2MAIN?TERMINATE(para);
TERMINATE_rcv: /* label for property check */
/** delete instance id **/
skip;
:: TERM2MAIN?TERMINATE(para);
T_TERMINATE_rcv: /* label for property check */
/** delete instance id **/
skip;
od;
} /* end of "MAIN_CC()" */

/* Originating Call Control Block */
proctype ORG_CC(byte org_addr)
{
    Para para;
    bit status;
    byte c;
    timer T308 = false;
null:
    MAIN2ORG?SETUP(eval(org_addr), para);
O_SETUP_rcv:

    /*
     * status := "check availability"
     * random selection of status value in the model
     */
    if
    :: status = OK;}
:: status = NOK;
fi;

call_initiated:
if :: (status == NOK) ->
   ORG2UNI!REL_COMP(para.calling_addr, para);
O_REL_COMP_snt: skip; /* label for property check */
ORG2MAIN!TERMINATE(para);
O_TERM1_snt: skip; /* label for property check */
goto finished;
:: (status == OK) ->
/** resource allocation **/

/* CALL_PROC to the calling party */
ORG2UNI!CALL_PROC(para.calling_addr, para);
O_CALL_PROC_snt: skip; /* label for property check */

/* SETUP_int to MAIN_CC. */
ORG2MAIN!SETUP_int(para);
goto outcall_proceed;
fi;

outcall_proceed:
if :: /* ALERT_int from TERM_CC */
   TERM2ORG?ALERT_int(eval(org_addr), para);

/* ALERT to calling party */
ORG2UNI!ALERT(para.calling_addr, para);
O_ALERT_snt: skip; /* label for property check */
goto call_delivered;
:: /* connection release due to T310 */
   TERM2ORG?REL_int(eval(org_addr), para);
/** release resource **/
T308 = true;
c  = 1;

/* REL to calling party */
ORG2UNI!REL(para.calling_addr, para);
goto release_indication;
fi;

call_delivered:
if :: /* CONN_int from TERM_CC */
   TERM2ORG?CONN_int(eval(org_addr), para);

/* CONN to calling party */
ORG2UNI!CONN(para.calling_addr, para);
O_CONN_snt:
    goto call_delivered_prime;
:: /* connection release due to T301 */
   TERM2ORG?REL_int(eval(org_addr), para);
   /** release resource **/
   T308 = true;
   c = 1;

   /* REL to calling party */
   ORG2UNI!REL(para.calling_addr, para);
   goto release_indication;
fi;

call_delivered_prime:
  if
    :: /* connection release from calling party */
   UNI2ORG?REL(eval(org_addr), para);
   /** release resource **/
   /* REL_COMP to calling party */
   ORG2UNI!REL_COMP(para.calling_addr, para);

   /* REL_int to TERM_CC. Need to check a clear collision. */
   if
      :: TERM2ORG?REL_int(eval(org_addr), para);
      :: timeout -> ORG2TERM!REL_int(para.term_addr, para);
   fi;

   /* finishing */
   ORG2MAIN!TERMINATE(para);
  goto finished;
:: /* CONN_ACK from calling party */
   UNI2ORG?CONN_ACK(eval(org_addr), para);
O_CONN_ACK_rcv: skip; /* label for property check */
  goto call_active;
fi;

call_active:
  if
    :: /* REL from calling party */
   UNI2ORG?REL(eval(org_addr), para);
   /** release resource **/
   /* REL_COMP to calling party */
   ORG2UNI!REL_COMP(para.calling_addr, para);

   /* REL_int to TERM_CC. Need to check a clear collision. */
   if
      :: TERM2ORG?REL_int(eval(org_addr), para);
      :: timeout -> ORG2TERM!REL_int(para.term_addr, para);
   fi;
/* finishing */
ORG2MAIN!TERMINATE(para);

goto finished;
:: TERM2ORG?REL_int(eval(org_addr), para);
/** release resource **/
T308 = true;
c = 1;
/* REL to calling party */
ORG2UNI!REL(para.calling_addr, para);
goto release_indication;
fi;

release_indication:
if
:: /* T308 handling */
(T308 == true) && (timeout) ->
if
:: (c < 2) ->
c = c+1;
T308 = true; /* the timer set again. */
goto release_indication;
:: (c == 2) ->
ORG2MAIN!TERMINATE(para);
goto finished;
fi;
:: UNI2ORG?REL_COMP(eval(org_addr), para);
T308 = false;
ORG2MAIN!TERMINATE(para);
:: /* Handle the clear collision */
UNI2ORG?REL(eval(org_addr), para);
ORG2MAIN!TERMINATE(para);
goto finished;
fi;

finished: skip;
} /* end of "ORG_CC()" */

/* Terminating Call Control Block */
proctype TERM_CC(byte term_addr)
{
Para para;
byte c;
timer T303 = false;
timer T310 = false;
timer T308 = false;
timer T301 = false;
null:
    MAIN2TERM?SETUP_int(eval(term_addr), para);
T_SETUP_int_rcv: skip; /* for property check */

    /* create a corresponding called party here */
    d_step {
        para.called_addr = ADDR;
        ADDR = ADDR + 1;
    }
    run Called_Party(para.called_addr);
    
    T303 = true;
    c    = 1;

    /* SETUP to called party to initiate connection */
    TERM2UNI!SETUP(para.called_addr, para); /*
    goto call_present;

    call_present:
    if
        :: (T303 == true) && (timeout) ->
            if
                :: (c < 2) ->
                    c = c+1;
                    T303 = true;
                    goto call_present;
                :: (c == 2) ->
                    /** release resource **/
                    TERM2ORG!REL_int(para.org_addr, para);
                    TERM2MAIN!TERMINATE(para);
                    goto finished;
            fi;
        :: UNI2TERM?REL_COMP(eval(term_addr), para);
        T308 = false;
        /** release resource **/
        TERM2ORG!REL_int(para.org_addr, para);
        TERM2MAIN!TERMINATE(para);
        goto finished;
    :: UNI2TERM?CALL_PROC(eval(term_addr), para);
        /** resource allocation **/
        T303 = false;
        T310 = true;
        goto incall_proceed;
    fi;

    incall_proceed:
    if
        :: (T310 == true) && (timeout) ->
            /** release resource **/
T308 = true;
c = 1;

TERM2UNI!REL(para.called_addr, para);
TERM2ORG!REL_int(para.org_addr, para);

goto release_indication;

:: UNI2TERM?ALERT(eval(term_addr), para);

T310 = false;
T301 = true;

TERM2ORG!ALERT_int(para.org_addr, para);

goto call_received;
fi;

call_received:
  if
    if :: (T301 == true) && (timeout) ->
      /** release resource **/
      T308 = true;
c = 1;

      TERM2UNI!REL(para.called_addr, para);
      TERM2ORG!REL_int(para.org_addr, para);

      goto release_indication;

    :: UNI2TERM?CONN(eval(term_addr), para);
    T301 = false;

    TERM2UNI!CONN_ACK(para.called_addr, para);
    TERM2ORG!CONN_int(para.org_addr, para);

    goto call_active;
  fi;
fi;

call_active: /* Note: "active" is a keyword of Promela */
  if
    /* REL from called party */
    UNI2TERM?REL(eval(term_addr), para);

    /** release resource **/

    /* REL_COMP to called party */
    TERM2UNI!REL_COMP(para.called_addr, para);

    /* REL_int to ORG_CC. Need to check a clear collision */
    if :: ORG2TERM?REL_int(eval(term_addr), para);
    :: timeout -> TERM2ORG!REL_int(para.org_addr, para);
  fi;
/* finishing */
TERM2MAIN!TERMINATE(para);

goto finished;

:: ORG2TERM?REL_int(eval(term_addr), para);

/** release resource **/
T308 = true;
c = 1;

/*/ REL to called party */
TERM2UNI!REL(para.called_addr, para);

goto release_indication;
fi;

release_indication:
if
:: (T308 == true) && (timeout) ->
if
:: (c < 2) ->
c = c+1;
T308 = true;
goto release_indication;
:: (c == 2) ->
TERM2MAIN!TERMINATE(para);
goto finished;
fi;

:: UNI2TERM?REL_COMP(eval(term_addr), para);
T308 = false;
TERM2MAIN!TERMINATE(para);
goto finished;

:: /* Handle the clear collision */
UNI2TERM?REL(eval(term_addr), para);
TERM2MAIN!TERMINATE(para);
goto finished;
fi;

finished: skip;

} /* end of "TERM_CC()" */

/* an environment process for calling party */
proctype Calling_Party()
{
Para para;

/* assign the calling party address */
UNI2MAIN!SETUP(para);

if :: /* reject from ORG_CC because service is not possible now. */
    ORG2UNI!REL_COMP(eval(para.calling_addr), para);
    goto finished;

:: /* due to T303 timer expiration */
    ORG2UNI!REL(eval(para.calling_addr), para);
    if :: UNI2ORG!REL_COMP(para.org_addr, para);
       :: skip; /* no response for REL_COMP */
       fi;
    goto finished;

:: ORG2UNI!CALL_PROC(eval(para.calling_addr), para);
fi;

if ::/* due to T310 timer expiration */
    ORG2UNI!REL(eval(para.calling_addr), para);
    if :: UNI2ORG!REL_COMP(para.org_addr, para);
       :: skip; /* no response for REL_COMP */
       fi;
    goto finished;

:: ORG2UNI!ALERT(eval(para.calling_addr), para);
fi;

if ::/* due to T301 timer expiration */
    ORG2UNI!REL(eval(para.calling_addr), para);
    if :: UNI2ORG!REL_COMP(para.org_addr, para);
       :: skip; /* no response for REL_COMP */
       fi;
    goto finished;

:: ORG2UNI!CONN(eval(para.calling_addr), para);
fi;

if :: UNI2ORG!CONN_ACK(para.org_addr, para);
:: /* unsatisfy to the CONN */
    UNI2ORG!REL(para.org_addr, para);
    ORG2UNI!REL_COMP(eval(para.calling_addr), para);
    goto finished;
fi;
if
:: /* initiate the call clearing from this party */
UNI2ORG!REL(para.org_addr, para);

if
:: ORG2UNI?REL_COMP(eval(para.calling_addr), para);
:: /* in case of clear collision */
ORG2UNI?REL(eval(para.calling_addr), para);
fi;

:: /* get a connection release request from network */
skip;
ORG2UNI?REL(eval(para.calling_addr), para);
if
:: UNI2ORG!REL_COMP(para.org_addr, para);
:: skip; /* no response for REL_COMP */
fi;
fi;

finished: skip;

} /* end of "Calling_Party()" */

/* an environment process for called party */
proctype Called_Party(byte called_addr)
{
  Para para;

  TERM2UNI?SETUP(eval(called_addr), para);

  if
  :: skip; /* first loss of CALL_PROC */
  if
  :: skip; /* second loss of CALL_PROC */
goto finished;
  :: UNI2TERM!CALL_PROC(para.term_addr, para);
  :: UNI2TERM!REL_COMP(para.term_addr, para);
goto finished;
  fi;
  :: UNI2TERM!CALL_PROC(para.term_addr, para);
  :: UNI2TERM!REL_COMP(para.term_addr, para);
goto finished;
  fi;

  if
  :: UNI2TERM!ALERT(para.term_addr, para);
  :: skip; /* simulate no sending the ALERT */
  TERMIN?REL(eval(called_addr), para);
  if
  :: UNI2TERM!REL_COMP(para.org_addr, para);
  :: skip; /* no response for REL_COMP */
  fi;
goto finished;
fi;

if
:: UNI2TERM!CONN(para.term_addr, para);
TERM2UNI?CONN_ACK(eval(called_addr), para);
:: skip; /* simulate no sending the CONN */
TERM2UNI?REL(eval(called_addr), para);
if
:: UNI2TERM!REL_COMP(para.org_addr, para);
:: skip; /* no response for REL_COMP */
fi;
goto finished;
fi;

if
:: /* initiate the call clearing from this party */
UNI2TERM!REL(para.term_addr, para);

if
:: TERM2UNI?REL_COMP(eval(called_addr), para);
:: /* in case of clear collision */
TERM2UNI?REL(eval(called_addr), para);
fi;

:: /* get a call clearing from network */
/* skip; */ /*
 * commented to avoid deadlock which occurs
 * for each party to wait a release request
 * from other side
 */
TERM2UNI?REL(eval(called_addr), para);
if
:: UNI2TERM!REL_COMP(para.term_addr, para);
:: skip; /* no response for REL_COMP */
fi;
fi;

finished: skip;

} /* end of "Called_Party()" */

init{
atOMIC{
    P_MAIN_CC = run MAIN_CC();
    run Calling_Party(); /* start one connection */
    run Calling_Party(); /* start another connection */
}
}
REFERENCES


[78] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, “Property specification patterns for finite-state verification,” in *Proc. 2nd Workshop on Formal*


BIOGRAPHICAL SKETCH

Youngjoon Byun received his B.E. degree in computer engineering from Kyungpook National University, Taegu, Korea, in 1991 and his M.S. degree in computer science from the Korea Advanced Institute of Science and Technology, Taejon, Korea, in 1993. Between 1993 and 1998, he worked for the Electronics and Telecommunications Research Institute, Taejon, Korea, as a member of the research staff. His research areas include design patterns for communication systems, software analysis and verification, software development environments and tools, and real-time systems. He is a student member of the Association for Computing Machinery and the IEEE Computer Society.