12-2011

State-Based Techniques For Designing, Verifying And Debugging Message Passing Systems

Rajaa Khaled Alqudah
Syracuse University

ABSTRACT

Message passing systems support the applications of concurrent events, where independent or semi-independent events occur simultaneously in a nondeterministic fashion. The nature of independence, random interactions and concurrency made the code development of such applications complicated and error-prone. Conventional code development environments or IDEs, such as Microsoft Visual Studio, provide little programming support in this regard. Furthermore, ensuring the correctness of a message passing system is a challenge. Typically, it is important to guarantee that a system meets its desired specifications along its construction process. Model checking is one of the techniques used in software verification which has proven to be effective in discovering hidden design and implementation errors. The required advanced knowledge of formal methods and temporal languages is one of the impediments in adopting model checking by software developers. To integrate model checking environments and conventional IDEs, this dissertation proposes a multi-phase development framework that facilitates designing, verifying, implementing and debugging state-based message passing systems. The techniques and design principles of the proposed framework focus on improving and easing the software development experience. In the first phase, a two-level design methodology is proposed through using abstract high-level communication blocks and hierarchical state-behavioral descriptions that were developed in this research. In the second phase, a new method based on choosing from a pre-determined set of patterns in concurrent communication properties is proposed to facilitate collecting the essential specifications of the system where the atomic propositions are linked with the system design. A complex property can be attained by hierarchically nesting some of these patterns. A procedure to automatically generate formal models in a model checker (MC) language is proposed. Once the model that contains both the design and the properties of the system are generated, a model checker is used to verify the correctness of the proposed system and ensure its compliance with specifications. To help in locating the source of an undesired specification, if any, a procedure to map a counter example generated by the MC to the original design is presented. In the third phase, a skeleton code of the design specification is generated in a general programming language such as Microsoft C#, Java, etc. moreover, the ability to debug the generated code using a conventional IDE while tracing the debugging process back to the original design was established. Finally, a graphical software tool that supports the proposed framework is developed where SPIN MC is used as a verifier. The tool was used to develop and verify several case studies. The proposed framework and the developed software tool can be considered a key solution for message passing systems design and verification.

STATE-BASED TECHNIQUES FOR DESIGNING, VERIFYING AND DEBUGGING MESSAGE PASSING SYSTEMS

by
Rajaa Khaled Alqudah

B.S., Yarmouk University, 2003
M.S., Northeastern University, 2007

Dissertation
Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Engineering

Syracuse University

December 2011

Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

The success of this research endeavor has been due to the help and support of many professors, friends, and family members. First, I wish to extend my gratitude to my dissertation advisor, Dr. C. Y. Roger Chen, for giving me the opportunity to study under his guidance and for his mentoring contributions towards my growth as a graduate student, researcher, and as a person, in addition to his encouragement, valuable comments and suggestions that helped shape this research work. I would like to also thank Dr. James Fawcett and Dr. Dan Pease who provided valuable insights and comments throughout this research. I am also thankful to the dissertation committee members for their help and time.

I am thankful to my friends who have contributed a great deal towards my success and completion of this dissertation. I am forever grateful to my parents for their sacrifices that paved the way towards the successful completion of my education. Finally, I would like to thank my husband Emad and my lovely daughter Jude for their standing by my side throughout this time and for their long patience.

Chapter 1

Introduction

Message Passing Systems (MPS), such as multi-thread or multi-process based programming environments, support the applications of concurrent events, where independent or semi-independent events occur simultaneously in a nondeterministic fashion. Communication protocols, such as the Transmission Control Protocol (TCP), are typical examples of such applications where a set of rules govern the syntax, semantics and synchronization of communication between nodes via a network. Typically, this communication happens to achieve a reliable transfer of data from one node to another. The design, verification, development and debugging of Message Passing Systems (MPS) tend to be a difficult task and subject to errors that are not common in single-process applications. Most of these errors come from the non-deterministic nature of events during the execution. Deadlocks and race conditions are examples of these errors just to mention few. Conventional code development environments or IDE, such as Visual Studio, provide little programming support or assistance in this regard. A large body of research has focused on techniques that can help verifying and validating software systems, examples can be found in [1–5].

Software testing is a common technique in which experiments are conducted on the actual system before deployment to assure its compliance with requirements. For testing to be effective it requires checking all the possible inputs and states; this is a costly and time-consuming process that makes it impractical in most cases. Moreover, even a heavily tested system may have errors that hardly appear or are impossible to replicate. Limitations in time and resources encouraged industry to find different methods to verify their systems other than dealing with the actual system.

Formal verification is a technique in which the correctness of the system is proven or disproven using mathematical and formal methods [6]. One approach of formal verification is model checking. Model checking is a technique to automatically test if a model of a system meets a given specification by exhaustively exploring all the possible executable paths. Using model checking in the verification process requires building a model of the system in some formal language accepted by the MC in addition to writing the requirement specifications as temporal logic formulas. Building a model and writing temporal properties require knowledge in the model checker programming language and a strong mathematical background. These obstacles prevent using the model checking technique widely in the software development process.

From the other hand, debugging is a process conducted to discover bugs that may reside in a software system by allowing the programmer to monitor the execution of the system, pause it, set some breakpoints and check the values in memory or even change it at runtime. This technique uses stop-the-world [7] approach which may be useful for debugging single process applications but not sufficient for multi-process applications. To our best knowledge, there is no debugger tool that helps developers in debugging distributed systems by providing a high-level picture of the current state of the application execution, visualizing all the interactions between processes, and reasoning about processes. Several attempts exist in the literature to facilitate using model checking techniques in software verification and to fill the existing gap between existing model checkers and the development environments.

There is a need for a comprehensive framework to help software developers design, verify, implement and debug their systems without the need to learn all the mathematical details of formal verification. To address this problem, a multiple phase approach for developing, verifying and debugging state-based message passing systems is proposed in this dissertation. The target system is described using a two-level design approach: abstract communication blocks level and hierarchical state-behavioral specification level. A procedure to automatically generate formal models in a MC language is proposed. Moreover, a new method based on choosing from a pre-determined set of patterns in concurrent communication properties is proposed to facilitate collecting the essential specifications of the system. Once the model and the properties of the system are generated, a model checker is used to verify the correctness of the proposed system and ensure its compliance with specifications. In case of any inconsistencies found, the error source will be automatically localized using an algorithm that was developed for this purpose. A skeleton code of the system that includes the correct design specifications is generated in a general programming language i.e. Microsoft C#, Java, etc., moreover, the ability to debug the generated code using a conventional IDE, such as Visual Studio, while tracing the debugging process back to the original design is established.

The proposed approach was successfully implemented in the form of a tool support that was developed in Microsoft C# where SPIN model checker was used as the verification engine. Several case studies are presented to show this frame work effectiveness. The following section gives a background about message passing systems, software verification and model checking.

1.1 Background

1.1.1 Message Passing Systems

Message Passing is a model of communication in a distributed system where messages are sent from one node to another. In general, the purposes of this communication can be one of the following: resource sharing, concurrency, scalability, transparency, openness or fault-tolerance [8]. A message is a collection of data objects that can be managed by a process and delivered to the destination. Usually, the communication between processes is governed by rules that should be followed to ensure the success of the communication. These rules are called communication protocols.

1.1.2 Software Verification

As software systems are being widely involved in everyday life, the need for reliable systems is critical and the failure is unacceptable. The process for checking if a software system meets its specification is called software verification. Many techniques can be used to verify the correctness of a software system. Simulation, testing, deductive verification and model checking are among these techniques. While simulation is a technique in which experiments are conducted on an abstraction or model of the system, testing, on the other hand, is a technique in which experiments are conducted on the actual system before the deployment. Verifying the correctness of a system using simulation or testing requires checking all of its possible inputs and states which is time-consuming and impractical in most cases. Moreover, even a heavily tested system can have errors that hardly appear and are impossible to replicate. When an error is detected, it is often difficult to spot since, in most cases, the sequence of events leading to this error cannot be reconstructed.

Deductive verification involves using axioms and proof rules to verify the correctness of a system. This method, which can be used for infinite and finite state systems, is not fully automatic and needs to be done by experts in logical reasoning since a lot of user interaction and guidance for the theorem proving tool is required. Moreover, using this technique is a time-consuming process which may last from days to months to prove a single protocol or circuit. Consequently, the use of this method is limited to highly sensitive systems such as security protocols.

The most widely used verification technique is Model checking [6] which is an automated method in which the system to be verified is modeled as a finite state machine and the specifications to be proven are written as temporal logic properties.

1.1.3 Model Checking

Given a finite-state model of a system and a logical property, model checkers check whether this property holds for that model or not by exploring all the possible executions paths of the state machine and checking if the property being tested holds. In case the property fails, a counterexample is generated in the form of a sequence of states.

The logical properties of a system can be classified into two major categories, safety and liveness properties. The safety property specifies what should not ever happen or what should always happen. In model checking, a counterexample of safety property is a list of states where the last state contradicts the property. Conversely, the liveness property specifies what should eventually happen. A counterexample is a loop that does not contain a state with a specified property [6].

One restriction of using model checking is the number of states in the system to be verified. Since the model checker uses an exhaustive search of the state space of the system and the number of states can grow exponentially with respect to the number of values, the state space of a system can be very large, or even infinite. Thus it is impossible to explore the entire state space with the limited resources of time and memory. This problem is called "state explosion problem".

In many cases, model checker tools are able to find errors that cannot be found by simulation. This is due to two major reasons; first, model checkers examine all possible executions of the system while simulators examine a relatively small part of these executions. Second, writing the temporal properties in formal language can help the designer in clarifying ambiguities in the system's specifications.

To use model checker, the behavior of the target system should be presented by some formal structure. Usually, the behavior of the system is represented by a Kripke Structure proposed in [9]. Kripke structure is a graph such that its nodes are the states of the system and its edges are the state transitions. Formally, a Kripke structure M is a 4-tuple *M=(S, I, R, L)* where S is a finite set of states, $I \subseteq S$ is a set of the initial states, $R \subseteq S \times S$ is a transition relation where $\forall s \in S, \exists s' \in S$ such that $(s, s') \in R$ and $L : S \rightarrow 2^{AP}$ is a labeling function that maps each node to a set of properties that hold in the corresponding state where AP is a set of atomic propositions, i.e., boolean expressions over variables, constants or predicate symbols. A path in the structure M from a state s is an infinite sequence of states $\pi = s_0 s_1 s_2 ...$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

As an example, Figure 1.1 gives the pseudo code of two processes competing at some shared resource. Only one process at a time can use this resource. This problem is known as mutual exclusion [10]. Each process has a program counter where 1 represents the critical section. Any process can access the shared resource only if this process is in its critical section. For this system, there are four states *S = 00, 01, 10, 11*. Let us assume that the starting state is where both processes are in position 0, i.e., *I = 00*. The transitions between states are specified by two rules, the next state will be the initial state unless the current state is already the initial state and the initial state can transit to either state 01 or 10. Thus *R=(00,01), (00,10),(10,00),(01,00),(11,00)*. Figure 1.2 gives a graphical representation of the Kripke structure of this system. The sequence *00,01,00,10,00,01,...*is an initialized path such that each process takes its turn to enter its critical section and use the shared resource. This system satisfies the property of safety since at most one process can be in its critical section at a time and nothing bad will happen.

```
            process A                                    process B
               forever                                      forever
A.pc =0           wait for B.pc =0          B.pc =0            wait for A.pc =0
A.pc =1           access shared resource    B.pc =1            access shared resource
               end forever                                  end forever
            end process                                  end process
```

Figure 1.1 Model Checking Example

The properties of a system should be formulated as temporal logic expressions. Linear Temporal Logic (LTL) is a specification language that is used for describing sequences of transitions between states in a system. LTL is an extension of classical logic which inherits Boolean variables and Boolean operators (negation $\sim$, conjunction $\wedge$, disjunction $\vee$ and implication $\rightarrow$) from propositional logic. In addition, LTL has the following temporal

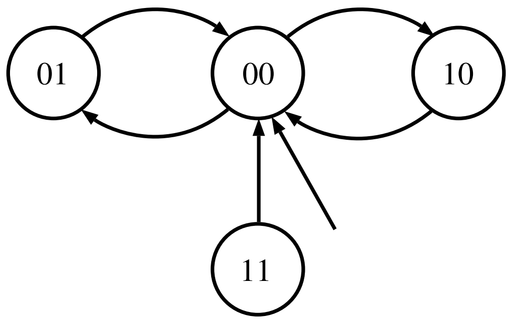


Figure 1.2 A Kripke structure for two processes that preserve mutual exclusion [10]

operators: **X** for next time, **F** for eventually or in future, **G** for globally and **U** for until. These operators describe properties of a path through a structure. The formula **X**p requires that the property p holds in the next state of the path. Figure 1.3(a) and 1.3(b) show a path for which **X**p does and does not hold respectively. In both cases each state is labeled with the properties that hold in it. The formula **F**p holds along a path if p holds at some state on the path. **F**p holds in both paths in Figure 1.3 since, in both cases, the first state already holds p. The formula **G**p holds along a path if p holds in all the states of the path. Clearly, the path in Figure 1.3(a) satisfies **G**p while the path in Figure 1.3(b) does not, since p does not hold in the second state. The formula p**U**g holds if there is a state on the path where q holds and, at every preceding state on the path, p holds. Figure 1.4(a) and (b) show a path in which p**U**g does and does not hold respectively.

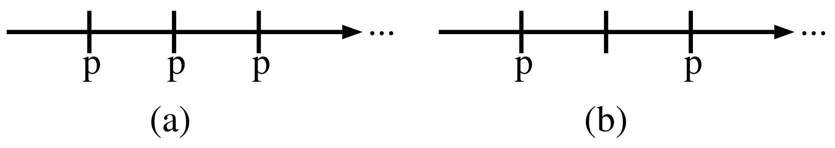


Figure 1.3 Next Operator

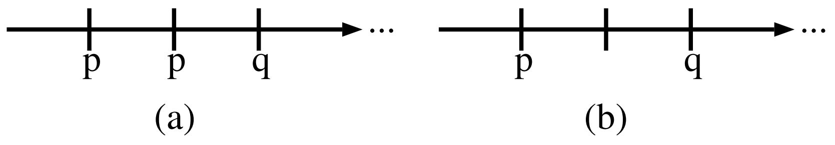


Figure 1.4 Until Operator

Going back to the example in Figure 1.1, the global operator **G** can be specified as $G \sim (c_1, c_2)$, where c_i labels the states where process i is in its critical section. Literally, this can be translated to English as follows: it is not possible for both c_1 and c_2 to be true at the same state. For the same example, the property which says that the first process will eventually reach its critical section can be formalized using the eventually operator as **F**. This property does not hold since the system may loop between the initial state *00* and the state *10* and never reach *01*.

A Kripke structure M satisfies a temporal formula f, if and only if all the initialized paths of the structure satisfy the formula. Formally, $M \models f$, iff $\pi \models d$ for all the initialized paths π of M. The relation $\models$ is defined recursively as follows:

$\pi \models p \quad iff \quad \pi \in L(\pi(0))$

$$\pi \models \mathbf{X}f \quad iff \quad \pi_1 \models f$$

$$\pi \models \mathbf{G}f \quad iff \quad \pi_i \models f \; for \; all \; i \geq 0$$

$$\pi \models \mathbf{F}f \quad iff \quad \pi_i \models f \; for \; some \; i \geq 0$$

$$\pi \models f\mathbf{U}q \quad iff \quad \pi_i \models q \; for \; some \; i \geq 0 \; and \; \pi_i \models f \; for \; all \; 0 \leq j \leq i$$

Using model checkers in verifying software systems can catch a range of logical and functional design errors, especially errors related to concurrency and multi-threading issues such as:

- Deadlock and starvation
- Race conditions
- Locking and priority problems
- Resource allocation errors
- Violation of system bounds (memory, stack ..)
- Redundancy

In order to use model checker on the verification process, two things are required from the user. The first is to learn how to build logic models i.e., how to extract a model from the software code to verify. The second is to learn how to use abstraction to solve the problem of space explosion. The third is to learn how to formalize requirements in LTL. Examples of available model checkers are Blast [11] Chess [12] and SPIN [13].

1.2 Motivations and Contributions

This research was motivated by the fact that a gap exists between model checking environments and conventional integrated development environments (IDEs). This gap prevents a simultaneous, efficient and effective benefit from both of these environments in the software development process. Figure 1.5 shows a schematic representation of this gap. On one hand, model checking is an efficient technique to ensure the correctness of a system before deployment. Using model checking requires building a model of the system in some formal language accepted by the MC in addition to writing the requirement specifications as temporal logic formulas. Building a model and writing temporal properties require knowledge in the model checker specification language and a strong mathematical background. These are obstacles for the majority of developers which in turn prevent model checking techniques from being widely used in the software development process. On the other hand, IDEs provide a debugging ability which enables the user to monitor and pause the execution of a system, set breakpoints and check the values in memory at runtime. Even though, debuggers are useful in single process applications, the nature of independence, random interaction and concurrency made the debugging of multi-process applications not a sufficient practice. Current debuggers are not able to provide a high-level picture of the current state of a multi-process application, visualize the interactions between processes or link the implementation code to the original design.

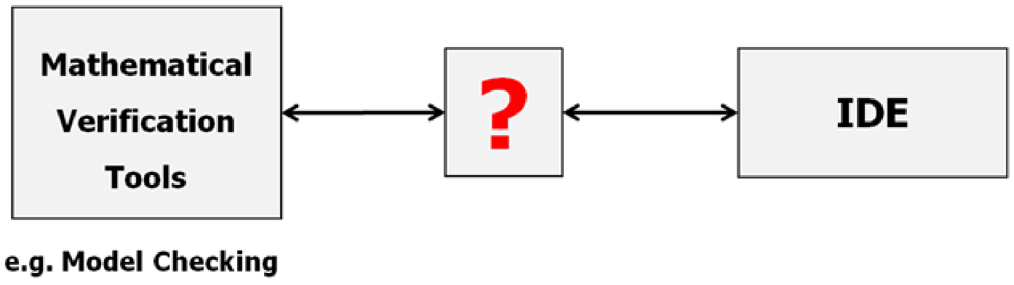


Figure 1.5 Motivation

In this research, the need for a comprehensive engineering development environment to help software developers design, verify, implement and debug their systems without the need to learn all the mathematical details of formal verification was realized. To address this problem, a multiple phase approach for developing and verifying message passing systems is proposed. The target system is described using a two-level design approach: an abstract communication blocks level and a hierarchical state-behavioral specification level. A procedure to automatically generate formal models in a MC language is proposed. On the other hand, a new method based on choosing from a predetermined set of patterns in concurrent communication properties is proposed to facilitate collecting the essential specifications of the system.

Once the model and the properties of the system are generated, a model checker is used to verify the correctness of the proposed system and ensure its compliance with specifications. In case of any inconsistencies found, the error source will be automatically localized using an algorithm that was developed for this purpose. A skeleton code of the system that includes the correct design specifications is generated in a general programming language i.e. Microsoft C#, Java, etc. Finally, the ability to debug the code using a conventional IDE while tracing the debugging process back to the original design was established. Figure 1.6 shows a schematic diagram of the proposed framework. A graphical software tool that supports the proposed framework is developed where SPIN MC is used as a verifier. The tool was used to develop and verify several case studies. The proposed framework and the developed software tool can be considered as a key solution for message passing systems design, verification and debugging.

The main contributions of this research can be summarized in the following points:

- A comprehensive framework to help software developers design, verify, implement and debug their systems without the need to learn all the mathematical details of formal verification.

- A multi-level graphical approach to design state-based message passing systems.

- A Hierarchical Designed-for-Verification extended finite state machine (HDV-EFSM) to assist in designing complex systems with too many states.

- A set of hierarchical event-based and time-based patterns for concurrent properties

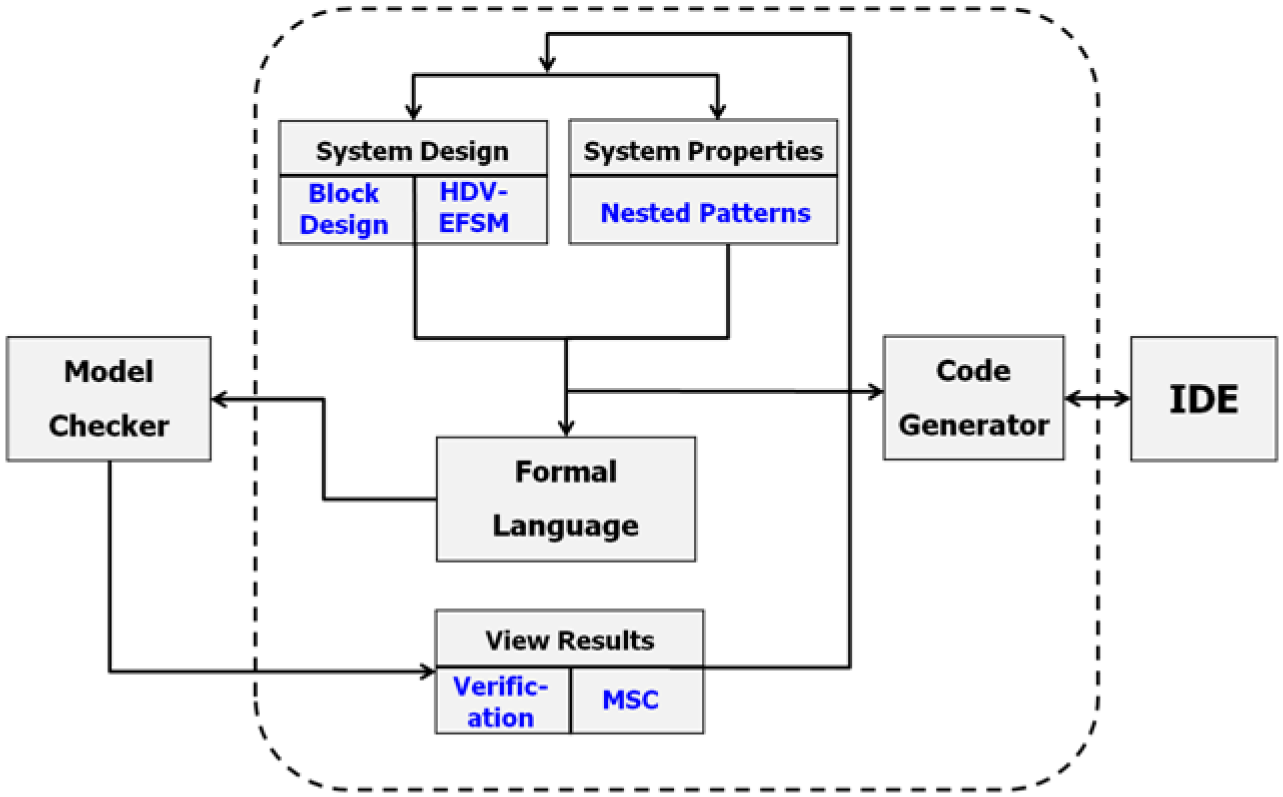


Figure 1.6 Framework Architecture

is proposed along with a mapping between these patterns to linear temporal logic (LTL).

- A tool environment to support the proposed approach where SPIN model checker was used as a verification engine along with a formal approach for translating the design system to PROMELA language (the input of SPIN) and to a general programming language e.g., C# or Java.

- A procedure to integrate an IDE to the developed tool during the debugging mode and switch between the graphical system design and its code implementation.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows. A related literature review of software verification, model checking and debugging multi-process applications is given in Chapter 2. Chapter 3 presents in details the proposed approach to the design, verification and debugging of message passing communication systems. Chapter 4 introduces the software tool developed based on the proposed approach. The tool was implemented in Microsoft Visual C# and includes the design, verification, implementation and debug phases. Case studies will be presented in Chapter 5. Finally, the dissertation conclusions and future work are discussed in Chapter 6.

Chapter 2

Review of Literature

Different approaches have been proposed in literature to model, verify and debug software systems. This chapter presents a state-of-the-art survey of these topics.

2.1 Model Checking in Software Verification

Model checking is a technique used to verify the correctness of systems. Different approaches have been proposed in literature to model check software systems. These approaches differ mainly in the following three aspects: the system modeling technique, the formalization of the system properties and the used model checker engine. The following subsections discuss these classifications.

2.1.1 Modeling System Specifications

Using a model checker to verify the correctness of a system requires building a model of the system in a formal language accepted by the model checker. Different approaches have been proposed to generate such a model. The following subsections discuss these approaches.

2.1.1.1 Hand-Written Formal Modeling

This approach requires writing a formal model in the model checker specification language by hand [14–16]. This approach is error-prone and needs advanced knowledge of the MC language.

2.1.1.2 Extraction from Code

This approach is based on extracting a verifiable model from the implementation of the system. This can be done either manually as in [17–20] or automatically as in [21, 22]. The manual extraction is time-consuming and needs advanced knowledge in the modeling languages. On the other hand, the automatic extraction is a more efficient technique, but has the problem of ensuring that the extracted abstract model is a correct mapping to the actual system. Most of these techniques use a level of abstraction based on the properties to be verified such that a model that captures only the details relevant to that property needs to be extracted. AX, stands for Automatic eXtractor [22], extracts models from C programs at an abstraction level defined by the user. Java PathFinder [21], on the other hand, translates programs written in Java to models in PROMELA which can be verified using SPIN. Other examples can be found in [23, 24].

2.1.1.3 High-Level Modeling

This approach uses higher level diagram-based or text-based architecture specifications languages to describe the system then converts the result to the language of the model checker. Shi and He [25] use Petri-Net to describe the system behavior. Wright and Garlen [26] use Communication Sequential Processes (CSP). Even though, these languages are rich in describing the behavior of a concurrent system, using these languages requires advanced knowledge in the languages syntax and semantics. CHARMY [27] , Bose [28] and vUML [29] used standard unified modeling language (UML) class and sequence diagrams to describe the specifications of a system. Byun [30] presented pattern-based development methodology to design such systems.

2.1.1.4 Direct Code Model Checking

Other studies proposed model checkers that can work directly with the implementation code without requiring building a model. CMC [31], for example, is a model checker that works directly with unmodified C or C++ implementation without requiring any intermediate steps. Other tools are SLAM [32], JPF [33] and Verisoft [34].

2.1.2 Systems Properties

To verify the correctness of a system using model checking its properties should be expressed in temporal languages that are accepted by the MC such as Linear Temporal Language (LTL), Computational Tree Logic (CTL), CLT* and mu-calculus. LTL is the most popular one which is used in CBabel [16], vUML [29] and CHARMY [27]. It is well known that writing and understanding properties in these languages is not an easy task since a strong background in these formal languages is required [35]. On the other hand, some studies exist in literature that proposed visual graphical notations to build these properties. Ramakrishna et al. [36] used Real-Time Future Interval Logic (RTFIL) diagrams. Damm and Hare [37] used Live Sequence Chart (LSC). Activity diagram is used by Bose [28]. Smith et al. [38] proposed Timeline Editor which is a visual specification formalism to place a series of events on a timeline. Figure 2.1 shows a time line for a call waiting requirements.

2.1.3 Model Checking Tools

Different model checkers are presented in literature and used to model check software systems. Examples of available model checkers are Blast [11] Chess [12] and SPIN [13]. In this review SPIN model checkers will be discussed. SPIN [13] is one of the most powerful model checkers that can be used to verify the correctness of concurrent systems and can check either hand-built or mechanically generated models. SPIN differs from other model checkers in offering a number of options to speed up the model checking process and save

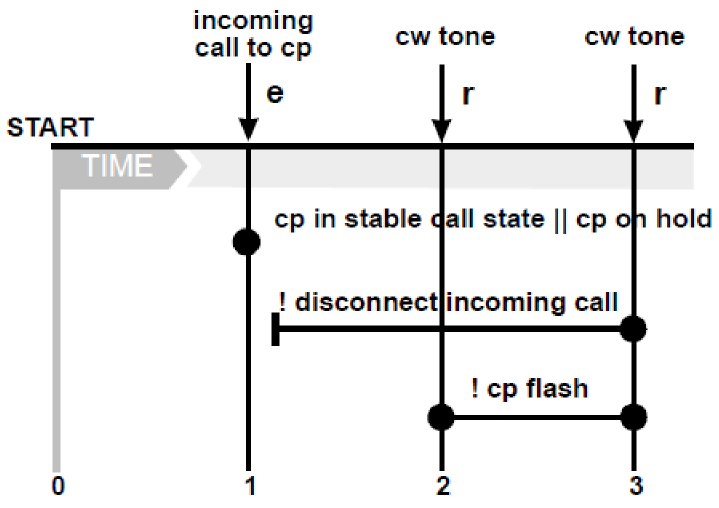

incoming call to cp
cw tone
cw tone
e
r
r
START
TIME
cp in stable call state || cp on hold
! disconnect incoming call
! cp flash
0
1
2
3

Figure 2.1 Time line for a Call waiting requirements [38]

memory. Moreover, SPIN supports asynchronous modeling which is a main property of message passing systems. Let L(S) be a set of possible behavior of a system S and L(p) is a set of the valid or desirable behaviors of the system, model checking tries to prove that $L(S) \subseteq L(p)$ i.e. everything possible is also valid. On the other hand, SPIN's verification engine tries to prove that $L(S) \cap L(\sim p) = \phi$. From Figure 2.2, if m is an empty set then S satisfies p, on the other hand if m is not an empty set then there are one or more counter examples that shows how S violates p.

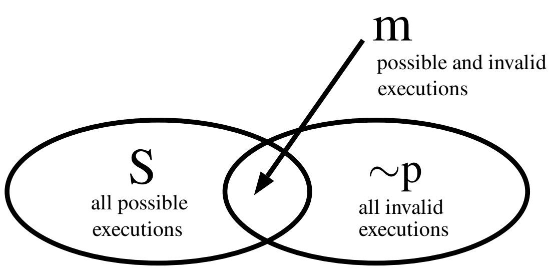


Figure 2.2 SPIN MC Principle

Systems to be verified using SPIN are described in PROMELA (*Process Meta Language*), which supports the modeling of asynchronous distributed algorithms. Properties to be verified are expressed as Linear Temporal Logic (LTL) formulas. The major target of PROMELA is to be a system description language, not an implementation language. This language focuses in the modeling of process synchronization rather than mathematical computations and on the description of concurrent software systems rather than hardware circuits. The syntax of PROMELA is similar to C language but simplified; no pointers, no real datatypes such as float or real and no functions. No notion of time and clock exist in PROMELA.

In PROMELA, system components are modeled as processes which communicate via channels or through shared memory represented as global variables. Given a PROMELA model as input, SPIN generates a C program which conducts system verification by scanning the state space using a depth-first search (DFS) algorithm. Two types of properties can be checked in this stage, first; liveness properties such as non-progress cycles and eventual reception of messages, second; safety properties such as invalid end states, assertions, unspecified message receptions and absence of deadlock. Other specific properties can be checked by writing Linear Temporal Logic (LTL) formulas. SPIN converts an LTL expression to a never claim which is just another process that is executed in lock-step where a step from the never claim process executes after each step from the system. When an error is found, SPIN saves the trace of steps led to the invalid state.

Figure 2.3 shows an overall architecture of SPIN. The inputs for SPIN are the specification model written in PROMELA and the properties to be verified written in LTL. SPIN will generate a verifier in C language called pan.c. This file should be complied and run.

In addition to model checking, SPIN can also operate as a simulator, following one possible execution path through the system and presenting the resulting execution trace to the user. All Spin software is written in ANSI standard C, and is portable across all versions of Unix, Linux, cygwin, Plan9, Inferno, Solaris, Mac, and Windows [13]. SPIN uses different approaches to cope with the state-space expulsion problem. These approaches include state-vector compression, bit-state hashing and partial order reduction.

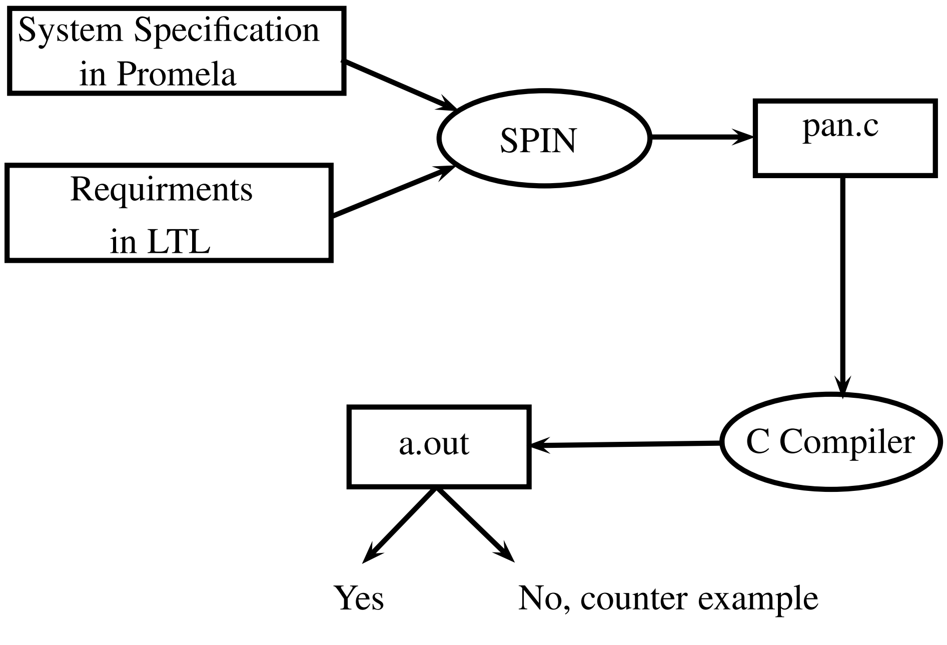


Figure 2.3 SPIN Architecture

2.2 Debugging multi process applications

The problems of testing software were first mentioned few decades ago; since that time the complexity of software, the available memory and computer speed have been increased but software is basically still developed and tested in the same way. It is not possible to predictably produce reliable software.

Testing is a common method to validate software systems. The process starts by unit test which is the isolation of a single module or process in the system and verify its correctness. Once this test passed, a system integration test is done with a group of unit tests to build the whole system step-by-step. For sequential software systems, this method can be used even it is very time-consuming. In distributed software systems, concurrency problems may prevent proper execution of the system while making testing a hard task. Concurrency can cause race conditions, starvation, deadlock or delay problems.

Debugging is a process of discovering any bugs that may reside in a software system to make it behaves as expected. Debuggers are tools which help the programmer to monitor the execution of the system, pause it, set some breakpoints and check the values in memory or even change it at runtime. This technique uses stop-the-world [7] approach which may be useful for debugging single process applications but not sufficient for distributed applications. Several methods for helping the developers in debugging distributed software systems are proposed in literature. The known methods for debugging parallel applications can be divided into three main categories [39]: Automated verification of correctness, Comparative debugging and Interactive debugging.

The first category is based on automated correctness verification where a specific conditions of the system are checked while the application is executing. These conditions may include the correctness of calls of libraries that control the parallel execution and message exchange, the correct synchronization of processes and threads when shared data are used, and so on. This kind of debugging is done fully automated; the user is not involved in any stage of this process. Types of errors that can be detected using this debugging technique are: Parallelism specifications, deadlocks, incorrect parameters, incorrect access of shared data, use of uninitialized variables and other errors. Examples of debuggers of this type are: Intel Message Checker [40], MPI debugger in the DVM system [41] , HP Visual Threads [42],and Intel Thread Checker [43].

The second category is called Comparative debuggers where a comparison of several runs of the same program is conducted by analyzing the traces collected in the course of the execution. Example of this type of debuggers is DVM system [44].

The third category is called Interactive debuggers where an interaction between the user and the debugger is allowed in any stage of the execution of the application by setting break points, step-by-step execution, testing and displaying values of particular variables. Examples of this type of debuggers are: TotalView [45], TDB [46], P2D2 [47], DDBG [48] and Allinea DDT [49].

This review will focus on the third type of debuggers, the Interactive debuggers for distributed programming. Allinea DDT is fully integrated with Microsoft Visual Studio which gives full control of parallel and multi-threaded applications as well as MPI applications. It is able to debug programs written in C, C++, C-Sharp, Python, FORTRAN and all .Net languages [49]. TotalView is a dynamic source code and memory debugger for C, C++ and FORTRAN applications written for Linux, UNIX and Mac OSX platforms. It can debug one or many processes and/or threads with complete control over program execution. TotalView does not support C-Sharp programs and Windows Operating System [45]. While both TotalView and DDT debuggers provide the developer with detailed information about the running processes in the application, no one provides a high-level picture of the current state of the application execution.

A debugger for flow graph based Dynamic Parallel Scheduled (DPS) parallelization framework is presented in [50]. The flow graph is used to provide both a high level and a detailed picture of the current state of the application execution. The parallel nature of DPS application is presented as an acyclic directed graph that specifies the dependences between messages and computations, where vertices represent serial computations and edges represent messages transfers. A later study by the same authors of [50] extends the concept of debugging DPS applications to debug MPI applications [51]. They designed a debugger that displays a message-passing graph of the execution of an MPI application which provides a higher level view of its communication patterns. This debugger provides the following features for the user: hiding/unhiding part of the processes, attaching regular debugger to a process, changing the contents of a message at run time, reordering the messages to check for race conditions, displaying the contents of message queues at any point and setting conditional breakpoints.

Figure 2.4 describes the system proposed in [51]. The Debugger is a standalone program which the parallel application should connect before startup. Each application has a hock with the debugger, which sends and receives from the debugger. The communications between the debugger and the processes are as follows:

- Each process will notify the debugger upon startup
- For every message a process sends, the process will first notify the debugger by sending the message contents and then waits for an acknowledgment
- When the debugger acknowledges the process, the process sends the message to the destination.
- When the process wants to do any operation i.e., processing the message, it will first notify the debugger and waits for acknowledgment

Microsoft Visual Studio 2008 includes a debugger with multi-threaded debugging support. It provides a "Threads" window which lists all of the current threads in the system. From this window the developer can freeze or resume a thread, switch the current active thread, examine thread state and examine the state of local variables within the thread. This win-

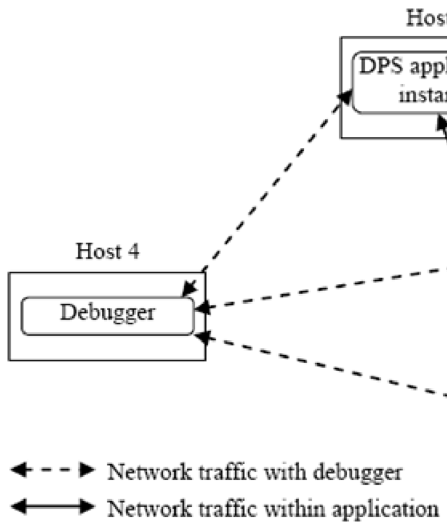


Figure 2.4 Parallel Debugger Architecture [51]

dow acts as the command center for examining and controlling the different threads in an application. Visual Studio 2010 supports the debugging of parallel applications by being able to present the state of the application anytime during the execution in the debugging mode across the different parallel execution units. It has task and thread windows to show the status of the processes and/or threads at any points in addition to stack view window which will show graphically the execution path of individual tasks [52].

HP proposed Causeway [53], a debugger for message-passing distributed systems. Their approach debugs the system by generating data traces that can be processed offline i.e. post-mortem debugging. This way of debugging generates large amount of tracing data which required effective searching techniques to find a bug. In [54] a live distributed debugger is proposed for debugging stream applications. It allows the user to limit the amount of tracing data depending in which part of the system is under investigation. The results are presented as visual graphical models. These debugging tools generate data traces while the system is running and help the user displays these data in different formats. These tools do not provide any help for the developer in reasoning about the cause and the solution of any residence bug.

Chapter 3

A Framework for Message Passing Systems Development

Verifying that a software system satisfies its required properties is important. This chapter presents in details an approach to formally verify and debug message passing communication systems using model checking.

3.1 Overview

As mentioned before, in order to verify the correctness of a system using model checking, a model of the system should be built in some language accepted by the model checker and the specifications to be verified should be given as temporal logic formulas. Building a model and writing temporal properties require advance knowledge in these languages and strong mathematical background which are not always available among the majority of software developers. These obstacles prevent using model checking technique widely in the software development process [55].

This research aims to facilitate using model checking in software verification by proposing a multi-phase approach. These phases are: system design phase, properties specification phase, verification phase, code generation phase and debugging phase. The message passing system is described using a two-level design approach: abstract communication blocks level and hierarchical state-behavioral specification level. A procedure to automatically generate formal models in a model checker language is proposed. A new method based on choosing from a pre-determined set of patterns in concurrent communication properties is proposed to facilitate collecting the essential specifications of the system. Once the model and the properties of the system are generated, a model checker is used to verify the correctness of the proposed system and ensure its compliance with specifications. In case of any inconsistencies found, the error source will be automatically localized using an algorithm that was developed for this purpose. A skeleton code of the system that includes the correct design specifications is generated in a general programming language i.e. Microsoft C#, Java, etc. Moreover, the ability to debug the code using a conventional IDE while tracing the debugging process back to the original design was established. Figure 3.1 illustrates the proposed framework. The following subsections describe these phases in details.

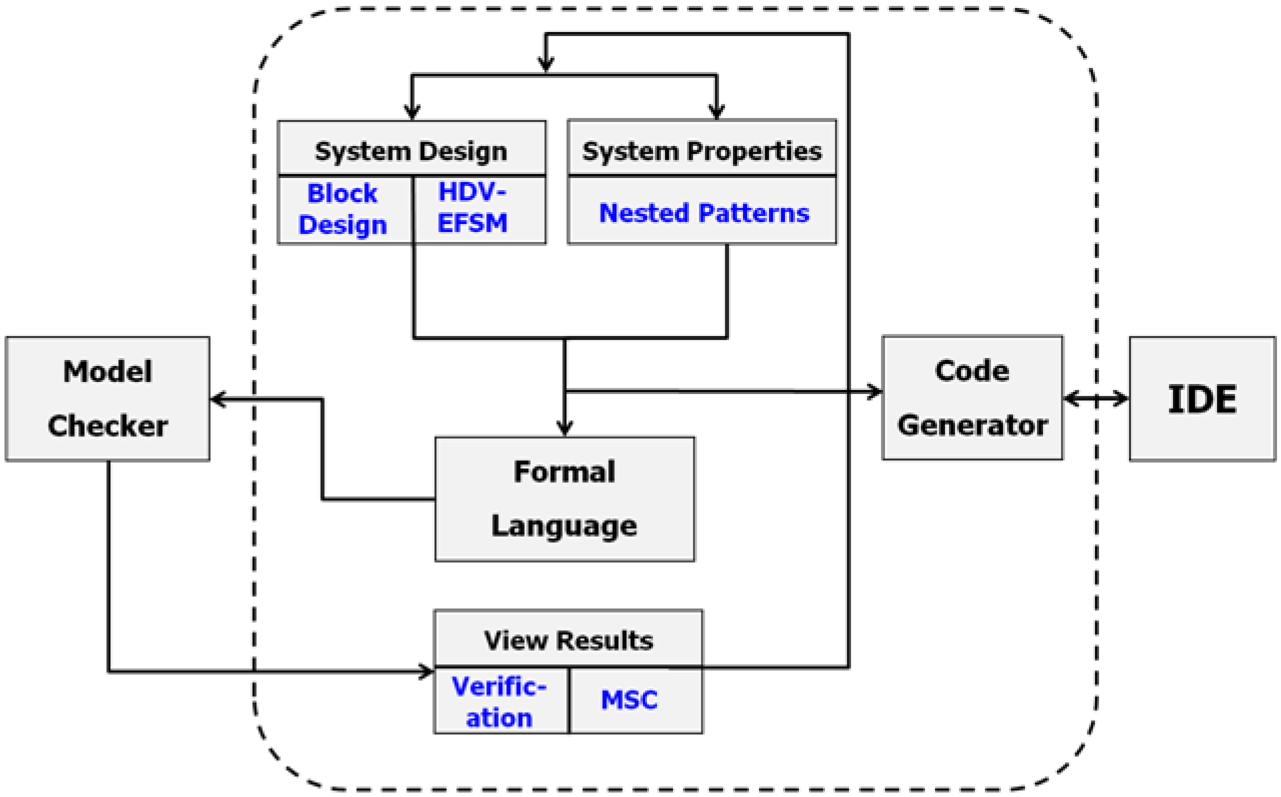


Figure 3.1 Framework Summary

3.2 System Design

A formalism to model the specifications of a Message Passing System (MPS) structure and interactions is required. A two-level design approach is proposed where the design of the system is decomposed into high-level block description and state-behavioral description. An MPS-Block Diagram was proposed to provide a high-level abstract architecture of the system in the form of blocks and communication channels between these blocks. In the state-behavioral description, the internal behavior of the high-level blocks is described using a modified finite state machine diagram. To easily explain the proposed approach, a simple example will be discussed and used as a sample use case through this chapter.

Example: Simple Reliable Transfer Protocol

A simple reliable communication protocol is a basic protocol to transfer data on a lossy lower layer. Messages are sent from node *A* to node *B* such that when *B* receives a message from *A*, it replies with an acknowledgment to ask for the next message, i.e. each message should be acknowledged before next message can be sent. Here, it was assumed that the lower layer may loose messages randomly but not corrupt the contents. Each message from *A* to *B* contains a data part and a one-bit sequence number, i.e., a value that is either 0 or 1. Messages from *B* to *A* contain an acknowledgment bit (0 or 1). Node *A* starts by sending a message with sequence number *x*, *A* now will wait to get a response from *B*. when *B* receives the message, it compares the message's sequence number with the expecting sequence number (0 for the case of first message). If they are equal, *B* acknowledges by sending ACK message to *A* with *x* sequence number, accepts the message and complements the expected sequence number. But if they are different, *B* sends ACK message to *A* with x sequence number. When *A* gets an ACK from *B*, two possibilities can happen:

- If the sequence number of the ACK message and the original sent number are equal, *A* complements the sequence number and continues sending the next data frame (if any).

- If the sequence number of the ACK message and the original sent number are not equal, *A* will resend the same message again.

To avoid infinite waiting for an acknowledgment from *B*, node *A* starts a timer each time it sends a message to *B* such that if this timer expires before an ACK is received, *A* will resend the message. Node *A* will try to send a message frame *N* times. If all of these trials fail, *A* will quit. The following two subsections provide details about the proposed design phase and how to apply it on this example.

3.2.1 High-Level Block Design

An MPS-Block Diagram was proposed to decompose the large MPS into several subsystems in order to reduce the design complexity. Each subsystem may be treated as a high-level black box where all the internal details are ignored at this stage and only the interactions between the subsystems are considered. The high-level block description of a system can contain:

- High-Level Blocks: the architectural building elements of the system
- Communication Channels: the interaction paths between the high-level blocks
- Messages: the type of interactions between any two blocks via a communication channel

Figure 3.2 shows the modeling of the simple reliable transfer protocol descried above using the proposed high-level block design. There are four blocks and six communication

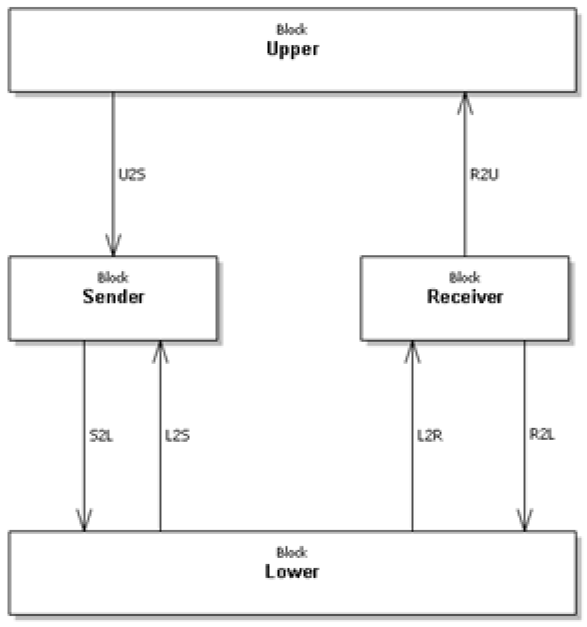


Figure 3.2 High-Level Block Design of a simple reliable transfer protocol

channels. The main blocks needed to model this system are:

1. Upper Block: which generates and consumes the date messages
2. Sender Block: which accepts messages from the Upper block and sends these messages to the lossy lower layer block
3. Receiver Block: which accepts messages from the lower layer and sends these messages to the Upper layer block
4. Lower Block: which simulates the lossy communication layer between the sender and the receiver

The communication channels and their corresponding messages are:

1. UTOS: this channel connects the upper block to the sender block. The message that goes through this channel is *send(byte d)* where *d* is the data byte to send.

2. STOL: this channel connects the sender block to the lower block. The message that goes through this channel is *send2L(byte d, bit a)* where *d* is the byte to send and *a* is the sequence bit.

3. LTOS: this channel connects the lower block to the sender block. The message that goes through this channel is *ack2S(bit b)* where *b* is the acknowledgment bit.

4. LTOR: this channel connects the lower block to the receiver block. The message that goes through this channel is *send2R(byte s, bit a)* where *s* is the received data byte and *a* is the received sequence bit.

5. RTOL: this channel connects the receiver block to the lower block. The message that goes through this channel is *ack2L(bit b)* where *d* is the acknowledgment bit.

6. RTOU: this channel connects the receiver block to the upper block. The message that goes through this channel is *receive(byte d)* where *d* is the received byte.

3.2.2 State-Behavioral Description

Once an abstract general architecture of the system is built as a group of MPS-level blocks with communications channels between these blocks, the internal behavioral of each block should be described. This behavioral can be described as a finite state machine (FSM) [56]. Although FSM is useful in systems specification description, this diagram cannot hold local variables in the state which is necessary in most message passing systems, especially with increased complexity of such systems. Ellsberger et al. [57] proposed a Communicating Extended Finite State Machine (CEFSM) which is a finite state machine that is extended with variables and actions.

In order to make the FSM a suitable presentation of the system behavior in the proposed framework, a number of parameters should be added to the state machine. In the verification process, it is important to determine the end state of a machine since this state is considered the only valid termination point of all the execution paths otherwise an error should be reported. To overcome this, an extension to the CEFSM is proposed in this research to facilitate the verification process. In the modified CEFSM, the end state and the required assertions were added. The new machine is called Designed for Verification CEFSM (DV-CEFSM). Follows is the formal definition of the DV-CEFSM:

Definition: An DV-CEFSM is a 7-tuple$(S, s_0, M, f, V, s_{end}, a)$ where:

- S is a set of states

- s_0 is the initial state
- s_{end} is the end state (to be used in the verification phase)
- M is a set of all input and output messages and their parameters
- V is a set of local variables and their initial values
- f is a state transition relation
- a is a list of assertions, if any (to be used in the verification phase)

Logical and arithmetic operators are used to operate in the local variables and the message parameters. A new definition of a state transition is given here which includes a number of added parameters required for verification and debugging purposes. A State Transition f is defined as follows:

Definition: A State Transition f is an nine-tuple$(s_s, s_e, i, p, a, o, d, t, b)$ where:

- s_s is the starting state of the transition
- s_e is the ending state of the transition
- i is the input message (if any) that will initiate the transition
- p is a condition (if any) than governs the final state of the transition

- a is a set of actions
- o is a set of output messages
- d is a boolean variable to indicate if the output messages are deterministic or non-deterministic
- t is a boolean variable to indicate if this operation is atomic or not
- b is a boolean variable to indicate if there is a breakpoint at this transition or not

The system goes from one state to another in response to an incoming message. If a system is in specific state $s \in S$ and receives a message $m \in M$. Depending on the contents of this message (by testing some conditions if any), the system performs some actions, emits output messages and then it may move to the next state in the state transition relation f. In some cases, the destination state of a transition is not only governed by the input message and its contents but also by the result of some mathematical or logical operations in a local variable of the state itself or a global variable of the machine. As MPS tends to be more complex and harder to model as FSM in practice, hierarchical DV-CEFSM is introduced in Section 3.2.4.

Figure 3.3 gives an example of a DV-CEFSM where $S = \{s_0, s_1, s_2\}$, the initial state is s_0, the end state is s_2, the set of messages is $M = \{m_0(byte\ x), m_1(byte\ x, bit\ a), m_2(bit\ b), m_3\}$ and c is a local variable with initial value of zero.

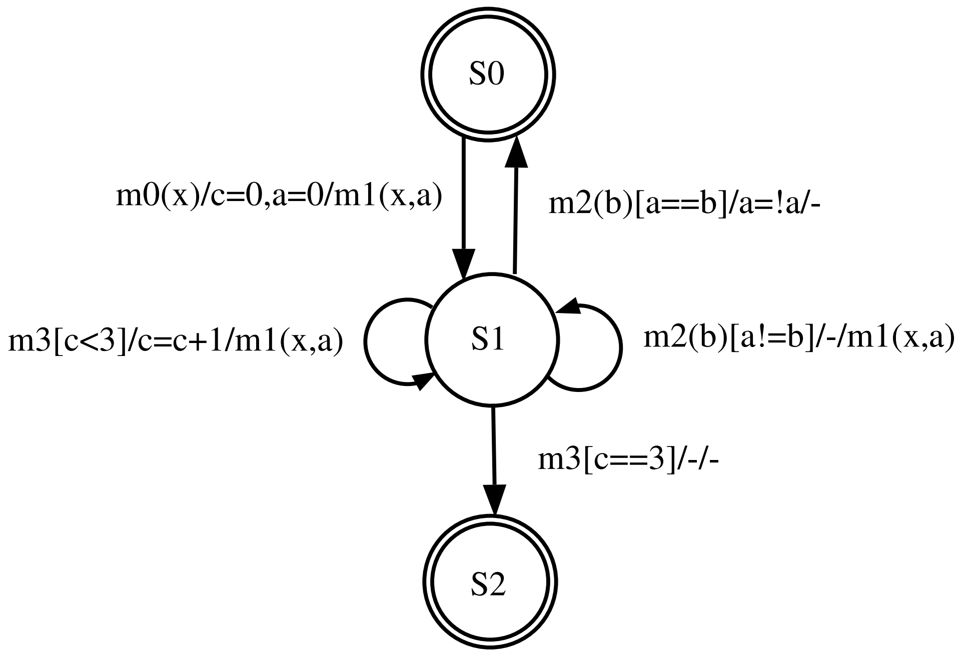


Figure 3.3 DV-CEFSM Example

Based on the above two-level approach to design a MPS, a formal definition of a state based MPS is given as follows:

Definition:A State-Based Message Passing System (SB-MPS) is a three-tuple (B, Ch, G) where:

- B is a set of all the high-level blocks in the system and their DV-CEFSM
- Ch is a set of all the communication channels in the system and their messages
- G is a set of global variables of all the blocks in the system and their initial values

The state behavioral diagram of the four blocks of the simple reliable transfer protocol discussed above is shown in Figure 3.4.

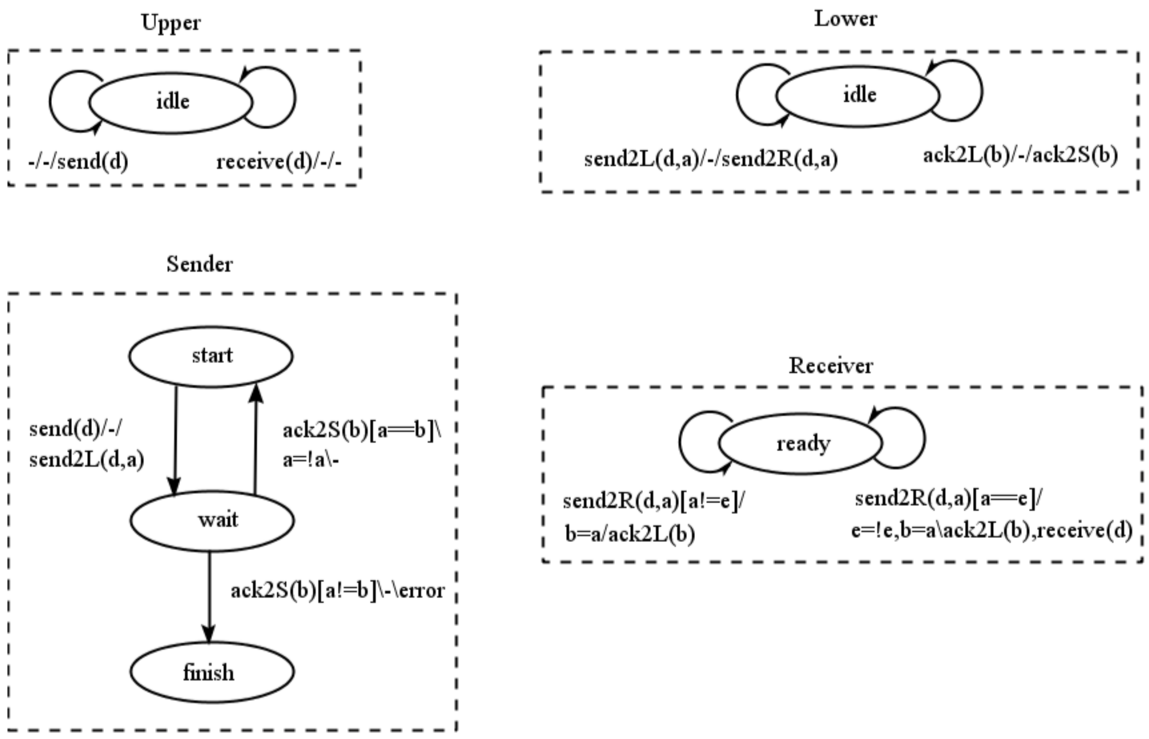


Figure 3.4 The state behavioral design of the four blocks in the Simple Reliable Transfer Protocol

3.2.3 Comparing to Petri Nets

Petri-Net, first designed in [58] is a formal representation of parallel and distributed system with mathematical semantics that is used in formal verification. There are many parallel computation applications where Petri Net has been successfully used. The basic components of a Petri Net model are places, transitions, arcs and tokens. Places and transitions are basic nodes where an arc may exists only from a place to a transition or from a transition to a place. A place can contain zero or more tokens that may be moved or "fired" to other place.

Colored Petri Net (CPN) [59] is an extended Petri net where a token can hold data of a given type or color in a place. Formally, a Colored Petri Net model is defined as follows [59]:

Definition: A Colored Petri Net (CPN) is an eight-tuple $(P, T, A, \Sigma, V, C, G, E)$ where:

- P is a set of places
- T is a set of transitions where $P \cap T = \phi$
- $A \subseteq P \times T \cup T \times P$ is a set of directed arcs
- Σ is a set of non empty types or color sets
- V is a set of typed variables
- C is a color set function assigning a color set (or a type) to each place.
- G is a guard function assigning a guard to each transition
- E is an arc expression function assigning an arc expression to each arc

Figure 3.5 shows the model of the simple reliable transfer protocol discussed above in Petri Net. As shown in the figure, it is hard to draw a clear model for such a simple protocol. Moreover, as the target system becomes larger, the final Petri Net model becomes more complicated and hard to visualize and understand [60]. On the other hand, the proposed modeling approach decomposes the modeling process into two stages (block design and behavioral design) which makes it relatively easier compared to Petri Net.

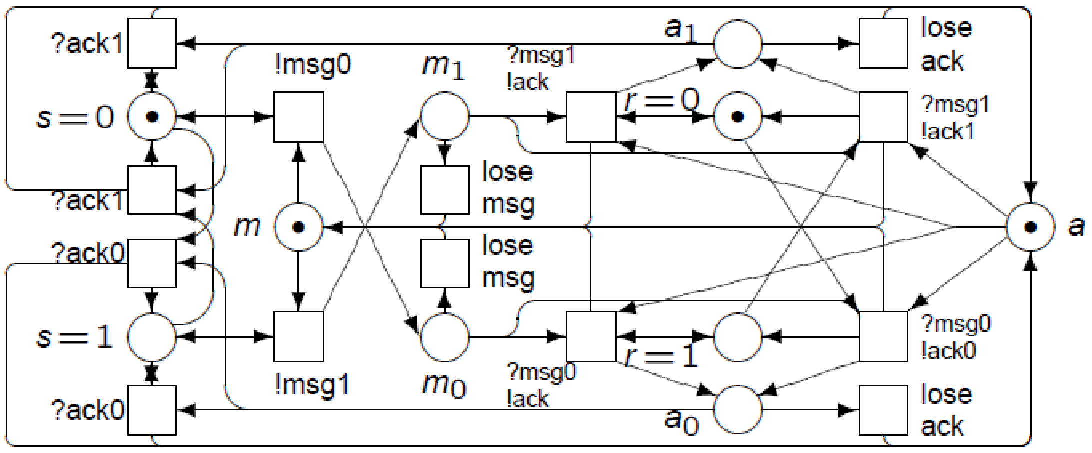


Figure 3.5 Simple Reliable Transfer Protocol in Petri Net

3.2.4 Hierarchical Designed-For-Verification CEFSM

In practice, modeling the behavioral of a system as a finite state machine tends to become harder as the number of states in the FSM increases. This also impedes the reading and understanding of the result FSM. To model complex systems as FSM, several studies proposed using hierarchical FSM [61]. In this dissertation, a hierarchical or nested DV-CEFSM is proposed to facilitate the process of top-down decomposing of complex message passing systems.

A Hierarchical DV-CEFSM has the following components:

- A set of Simple States *S*

- A set of Complex States C
- An initial simple state s_0
- An end simple state s_{end}
- A set of all input and output messages and their parameters
- A set of local variables and their initial values
- A list of assertions, if any
- A simple state transition relation
- A complex transition relation

A Complex state recursively contains simple states and other complex states in addition to the following two pseudo states:

- *Enter* State: This state acts as an entrance to the state diagram of the complex state. All the incoming transitions to the state diagram should go through this pseudo state. This state directs an incoming transition into its final destination.
- *Exit* State: This state acts as an exit gate to the state diagram such that all the outgoing transition from the complex state should pass through this state.

A simple state s_i within a complex state C_i can communicate with other outside states only through the *Enter* or *Exit* states. In other words, there is only one entrance state and one exit state to the complex state.

A simple transition connects two simple states and a complex transition connects a simple state to a complex state, a complex state to a complex state or a complex state to a simple state. A Complex State Transition differs from a simple transition in the definition of the source and destination of the transition. The source of a complex transition should be defined as the name of the complex state and the corresponding internal simple state.

An example of a hierarchical DV-CEFSM is shown in Figure 3.6. In this example S_0 and S_3 are simple states. C_1 is a complex state which contains the simple states S_1 and S_2. t_2 and t_4 are simple transitions that connects two simple states. t_1 and t_5 are complex transitions such that t_1 connects the simple state S_0 to the complex state C_1 and t_5 connects C_1 to S_3. The transitions t_3 and t_6 are pseudo transitions that connects the pseudo states. The source of the transition t_1 is defined as S_0 while the destination of t_1 is defined as $C_1 : S_1$. Note here that in hierarchical state diagram only one state is active at a time.

In some cases, it may be required to convert a hierarchical state machine to a one without any complex states. This process is called Flatten Process since it expands any complex state to a set of simple states. To explain this process, let H is a hierarchical DV-CEFSM and its corresponding flat DV-CEFSM D. D is generated using the following rules:

- For each simple state in H add a state to D

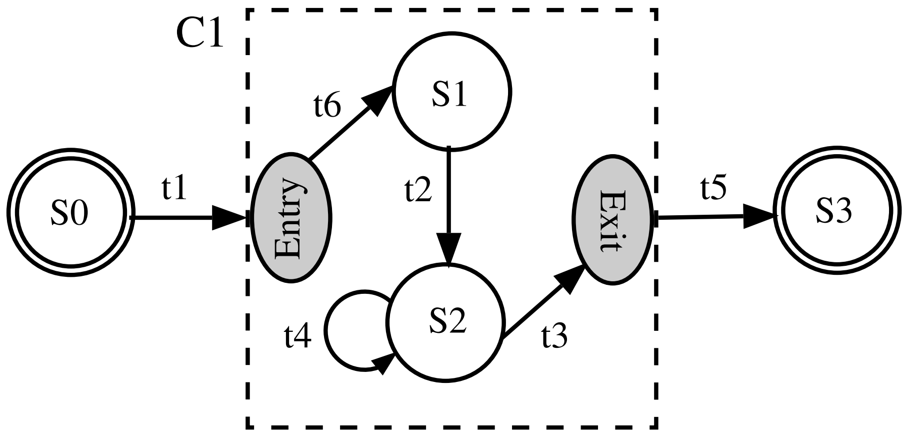


Figure 3.6 An Example of Hierarchical CEFSM

- For any complex state in H, recursively add all of its states to D
- For each simple transition in H add a transition in D between the same corresponding states
- The initial and end states in H are mapped to the initial and end states in D
- For each complex transition in H that connects $C_i : S_j$ to $C_k : S_l$, recursively convert the complex transition to a regular transition in D that connects the state S_j to state S_l after expending C_i and C_k.

3.3 Concurrent Property Specifications

Once the design of the system is described, the required properties should be specified. These properties usually expressed in a temporal formal language. Generally, the properties of a system can be classified either as Liveness properties or Safety Properties [13]. A

Liveness property indicates that something good should eventually happen while a safety property indicates that something bad should never happen. In the proposed framework, a simplified pattern-based approach to construct these properties is used. The following subsections describe this approach.

3.3.1 Pattern based properties

The pattern approach is based on the similarities noticed in system's requirements such that a specification pattern can be defined as a general description of a recurring required sequence of events in a finite state machine. This method has the advantage of providing a pre-caned solution for the properties modeling and can effectively and efficiently simplify the MPS verification process. Patterns in finite state machine property specifications were first introduced in [62].

In this work, patterns in property specifications are studied and classified based on the type of the incident that raises the pattern. Patterns in property specifications can be classified as event-based or time-based patterns. While event-based patterns describe the occurrence of an event in relation to other events, time-based patterns describe the occurrence of an event in relation to an absolute of relative time. The following subsections describe these patterns.

3.3.1.1 Event-Based Patterns

Event-based patterns describe the occurrence of an event in relation to other events. A previous work by Dwyer et al. [62] to identify the different property specification patterns was adapted in this research. An extension of these patterns in addition to new patterns was proposed and identified in this work. Each pattern holds valid for a certain range which specifies to what extend this pattern should hold. The following ranges where proposed in [62]: *Global* (throughout the entire execution path), *Before R* (up to the occurrence of another event *R*), *After Q* (after an event *Q* occurred) and *Between Q and R* (the part of the execution after the occurrence of *Q* and before the occurrence of *R*). Figure 3.7 shows the event-based ranges. The following subsections list these patterns. In the proposed framework and its resulted tool support, a mapping between these patterns along with the different scopes to LTL was also performed.

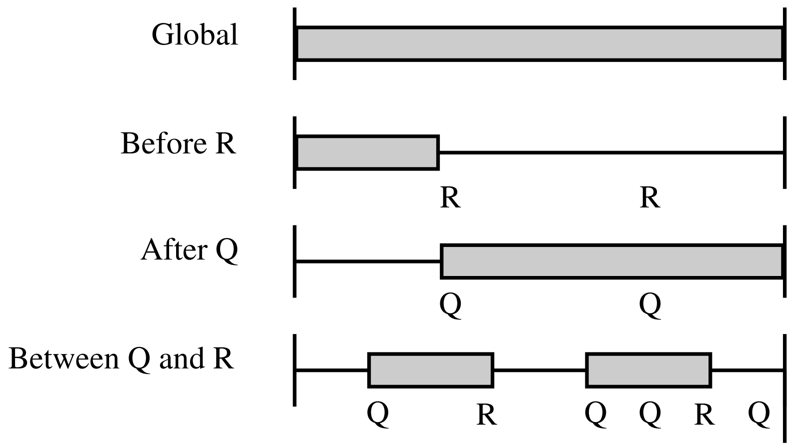


Figure 3.7 Event-Based Specification Pattern Ranges [62]

3.3.1.1.1 Never Pattern

Intent To describe that a given event *P* **should not** happen within a specified scope condition.

LTL Mapping Expressing this pattern as a Linear Temporal Logic is done using the *! (not)* operation. Table 3.1 gives the formulas of this pattern in different scopes.

Example and Known Uses An example of using this pattern is in the well known Mutual Exclusion algorithm [62]. This algorithm is used to avoid using a common resource (e.g. critical section, global variable, ...) simultaneously. Let *P* is the event that two processes p_1 and p_2 are in their critical sections, the property can be written in LTL as $[\,](!P)$.

Table 3.1 Never Pattern LTL Mapping

Scope Condition	**LTL Expression**
Globally	[] (!P)
Before R	<> R -> (!P U R)
After Q	[](Q -> [](!P))
Between Q and R	[]((Q & !R & <>R) -> (!P U R))

3.3.1.1.2 Invariant Pattern

Intent To describe that a given event *P* **should always** happen within a specified scope condition.

LTL Mapping Table 3.2 gives the formulas of this pattern in different scopes.

Example and Known Uses For the Mutual Exclusion algorithm, let P is the event that process p_1 and p_2 are not using a shared resource at the same time, then this property can be written as $[\,]P$.

Table 3.2 Invariant Pattern LTL Mapping

Scope Condition	LTL Expression
Globally	[] (P)
Before R	<> R -> (P U R)
After Q	[](Q -> [](P))
Between Q and R	[]((Q & !R & <>R) -> (P U R))

3.3.1.1.3 Existence Pattern

Intent To describe that a given event P should happen **at least once** within a specified scope condition.

LTL Mapping Table 3.3 gives the formulas of this pattern in different scopes.

Example and Known Uses An example of the existence pattern can be found in the situation where a process eventually reaches it is final state and terminates. This can be modeled as $<>P$ where P is the event that the process is in its end state s_e.

Table 3.3 Existence Pattern LTL Mapping

Scope Condition	**LTL Expression**
Globally	<>(P)
Before R	!R U ((P & !R) | [] !R)
After Q	[](!Q) | <>(Q & <>P)
Between Q and R	[](Q & !R -> (!R U((P & !R) | [] !R)))

3.3.1.1.4 At-Most-K Existence Pattern

Intent At-Most-K Existence *P,K* describes that a given event *P* should happen at most *K* times within a specified scope condition.

LTL Mapping Table 3.4 gives the formulas of this pattern in different scopes where *k*=2.

3.3.1.1.5 Exactly-K Existence Pattern

Intent Exactly-K Existence *P,K* describes that a given event *P* should happen exactly *K* times within a specified scope condition.

LTL Mapping Table 3.5 shows the LTL formulas for Exactly-K Existence pattern. The formula for K=N is built recursively from sub formulas. Table 3.6 gives the formulas of this pattern in different scopes where K=2.

Table 3.4 At-Most-K Existence Pattern LTL Mapping where *K*=2

Scope Condition	**LTL Expression**
Globally	(!P U((P U((!P U((P U ([]!P | []P)) [] !P)) | [] P))) |[] !P)
Before R	<>R -> ((!P & !R) U (R | ((P & !R) U (R | ((!P & !R) U (R | ((P & !R) U (R | (!P U R)))))))))
After Q	<>Q -> (!Q U (Q & (!P U((P U((!P U((P U ([]!P | []P)) [] !P)) | [] P))) |[] !P)))
Between Q and R	[]((Q & <>R) -> ((!P & !R) U (R | ((P & !R) U (R | ((!P & !R) U (R | ((P & !R) U (R | (!P U R))))))))))

3.3.1.1.6 At-Least-K Existence Pattern

Intent At-Least-K Existence *P,K* describes that a given event *P* should happen at least *K* times within a specified scope condition.

LTL Mapping Table 3.7 gives the formulas of this pattern in different scopes where k=2.

3.3.1.1.7 Sub-Set Existence Pattern

Intent To describe that a given sub set of n events *P1, P2, P3, .., Pn* should happen within a specified scope condition. These events can occur in any order.

LTL Mapping Table 3.8 gives the formulas of this pattern 2 out of 3 events P1,P2, P3 should happen.

Table 3.5 Recursive Exactly-K Existence Pattern for Global Scope

Function	**LTL Expression**
Exactly_1(P)	<>P & [] (P -> X ([]!P))
Exactly_2(P)	<>P & (!P U (P & X (Exactly_1(P))))
Exactly_N(P)	<>P & (!P U (P & X (Exactly_N-1(P))))

Table 3.6 Exactly-K Existence Pattern LTL Mapping where K=2

Scope Condition	**LTL Expression**
Globally	<>P & (!P U (P & X<>P & [] (P -> X ([]!P))))
Before R	<>R ->(<>(P&!R) & ((!P &!R) U ((P&!R) & X<>(P&!R) & [] ((P&!R) -> X ([]!P)))))
After Q	<>Q -> (!Q U(Q & (<>P & (!P U (P & X<>P & [] (P -> X ([]!P)))))))
Between Q and R	[]((Q & <>R) ->(<>(P&!R) & ((!P &!R) U ((P&!R) & X<>(P&!R) & [] ((P&!R) -> X ([]!P))))))

3.3.1.1.8 Response Pattern

Intent This pattern describes the situation where a given event *P* should happen after a given event *S* within a specified scope condition.

LTL Mapping Table 3.9 gives the formulas of this pattern in different scopes.

Table 3.7 At-Least-K Existence Pattern LTL Mapping

Scope Condition	**LTL Expression**
Globally	P U <>P
Before R	<>R ->((P& !R) U <>(P & !R))
After Q	<>Q -> (!Q U (Q & (P U <> P)))
Between Q and R	[](Q->((P& !R) U <>(P & !R)))

Table 3.8 Sub-Set Existence Pattern LTL Mapping

Scope Condition	**LTL Expression**
Globally	((<>P1 & <>P2) | (<>P1 & <>P2) | (<>P2 & <>P3))
Before R	<>R -> ((<>P1 & <>P2) | (<>P1 & <>P2) | (<>P2 & <>P3)) U R)
After Q	[](Q -> [](<>P1 & <>P2) | (<>P1 & <>P2) | (<>P2 & <>P3)))
Between Q and R	[]((Q & !R & <>R) -> (((<>P1 & <>P2) | (<>P1 & <>P2) | (<>P2 & <>P3)) U R))

3.3.1.1.9 Precedence Pattern

Intent This pattern describes the situation where a given event *P* should happen as a pre-condition for a second event *S* to happen within a specified scope.

LTL Mapping Table 3.10 gives the formulas of this pattern in different scopes.

Table 3.9 Response Pattern LTL Mapping

Scope Condition	**LTL Expression**
Globally	[](P -> <>S)
Before R	<>R -> (P -> (!R U (S & !R))) U R
After Q	[](Q -> [](P -> <>S))
Between Q and R	[]((Q & !R & <>R) -> (P -> (!R U (S & !R))) U R)

Table 3.10 Precedence Pattern LTL Mapping

Scope Condition	**LTL Expression**
Globally	!P U (S | [] !P)
Before R	<>R -> (!P U (S | R))
After Q	[]!Q | <>(Q & (!P U (S | [] !P)))
Between Q and R	[]((Q & !R & <>R) -> (!P U (S | R)))

3.3.1.1.10 Sequence of Events Pattern

Intent To describe a situation where two or more events *(P1, P2, P3 ,.., Pn)* should happen within a scope condition in the same given order. This pattern can be considered as a general version of the response pattern but with two or more events.

LTL Mapping Table 3.11 gives the formulas of this pattern where n = 3 i.e. events P1, P2, P2 should occur in sequence.

Table 3.11 Sequence of Events Pattern LTL Mapping

Scope Condition	**LTL Expression**
Globally	[](<>P1 & (P1-><>P2) & (P2-><>P3))
Before R	<>R ->([](<>(P1 & !R) & (P1-><>(P2& !R)) & (P2-><>(P3 & !R))))
After Q	<>Q ->(!Q U (Q & ([](<>P1 & (P1-><>P2) & (P2-><>P3)))))
Between Q and R	[]((Q & <> R) ->([](<>(P1 & !R) & (P1-><>(P2& !R)) & (P2-><>(P3 & !R)))))

3.3.1.1.11 Set of Events Pattern

Intent To describe that a given set of n events *P1, P2, P3, .., Pn* should happen within a specified scope condition. These events can occur in any order.

LTL Mapping Table 3.12 gives the formulas of this pattern where n = 3 i.e. events P1, P2, P2 should occur.

Table 3.12 Set of Events Pattern LTL Mapping

Scope Condition	**LTL Expression**
Globally	(<>P1 & <>P2 & <>P3)
Before R	<>R -> ((P1 & P2 & P3) U R)
After Q	[](Q -> [](P1 & P2 & P3))
Between Q and R	[]((Q & !R & <>R) -> ((P1 & P2 & P3) U R))

3.3.1.2 Timer Specification Patterns

Time is an important factor in describing message passing systems. Usually, these systems are time dependent in nature where events occur based on time. Taking this into consideration, time-based patterns in MPS were identified in this research. Konard and Cheng [63] proposed a list of real-time specification patterns that work with ranges in Figure 3.7. An extension of these patterns in addition to new patterns was proposed and identified in this work in different ranges. The proposed operation ranges of these patterns depend on either a relative time u or an absolute time t. Relative time ranges can be described as follows: Exactly after u time units, anytime after u time units, anytime within u time units and anytime between the range u_1 and u_2 time units. Figure 3.8 shows these ranges. On the other hand, absolute time ranges include: Any future time, anytime no later than t, anytime after t, exactly at time t and anytime between t_1 and t_2. Figure 3.9 shows these ranges.. The identified time-based patterns are listed in the next subsections.

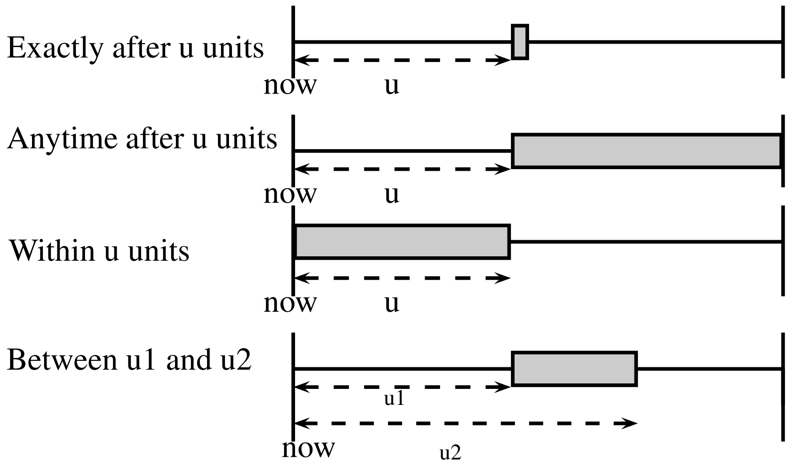


Figure 3.8 Relative time Pattern Ranges

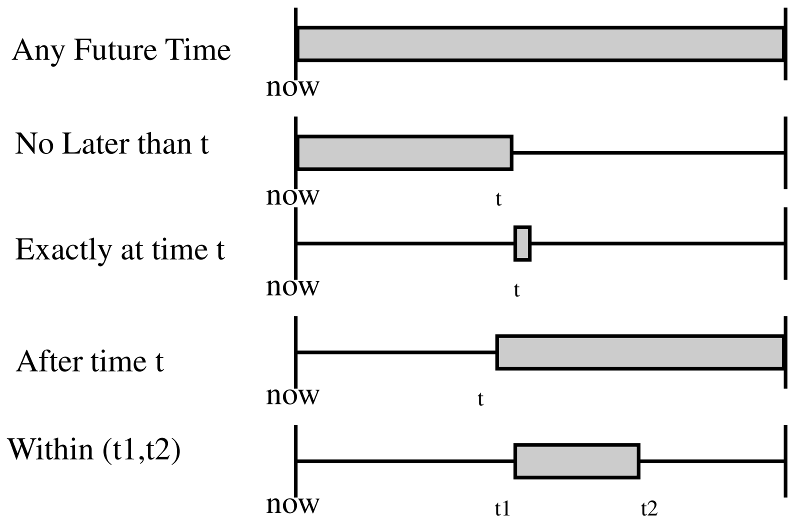


Figure 3.9 Absolute time Pattern Ranges

3.3.1.2.1 Duration Existence

The intent of this pattern is to describe that an event should happen and continue to be valid (at least or at most) for a specific period of time units *t*.

3.3.1.2.2 Cyclic Existence

This pattern describes the situation where an event should happen "every" specific period of time units *t*.

3.3.1.2.3 Timed Response

This pattern describes the situation where an event *P* must be followed by an event *Q* within a specified time frame *t* i.e. *Q* occurs within *t* time units after *P*.

3.3.1.2.4 Timed Existence

Indicate that an event *P* exists within a specified period of time. This applies to all the variations of the Existence Pattern (bounded, subset, etc).

3.3.1.2.5 Response Duration

This pattern describes the situation where an event *P* must be followed by an event *Q* and *Q* should hold valid for a specific period of *t* time units .

3.3.1.3 Hierarchical Properties

Systems complexity is increasing from day to day. In order to facilitate modeling more complex system properties, a hierarchical approach is proposed such that a hierarchical property is built up from a set of simple properties or propositions using boolean operators or property patterns. Formally a hierarchical property is built as follows:

Property -> Hierarchical_Expression

Hierarchical_Expression -> (Expression_Join) Expression

Expression_Join -> and | or

Expression -> Atomic_Proposition | Expression_Pattern

Expression_Pattern -> Unary_Pattern | Binary_Pattern | N-ary_Pattern

Unary_Pattern -> Never | Invariant | Existence

Binary_Pattern -> Response | Precedence

N-ary_Pattern -> Sequence of Events | Set of Events | Sub Set of Events

3.3.1.4 Extracted Properties

To facilitate properties generation even more, a procedure to extract properties out of the design patterns was proposed. In this procedure, LTL expressions will be automatically extracted from the state machines at either the state level or the transition level. The extracted properties can be used as atomic expressions in building more advanced properties. Follows is a description of the proposed procedure at both state and transition levels.

State level: the state level can be divided to initial state and not-initial state levels:

- **Initial State:** For the initial state s_0 in a finite state machine, the following properties are extracted:
  - If s_0 has only one outgoing transition, then the input message *in_msg* of this transition should be received eventually to initiate the state machine, i.e.

    $<> (in\_msg)$

  - If s_0 has n outgoing transitions (with or without predicates), then one of the input messages of these transitions should be eventually received. i.e.

    $<> (\bigvee_{i=0}^{n}(in\_msg\,[\wedge predicate]))$

  - If s_0 has n outgoing transitions (with or without predicates), then always one of these transitions should occur. i.e.

$\Box(\bigvee_{i=0}^{n}((in\_msg\,[\wedge predicate]) \rightarrow <> (\bigwedge_{j=0}^{m} out\_msg_{i,j}))$

- **Not-Initial State:** For any state s_i other than the initial state, the following properties are extracted:

  – If a state s_i has one incoming transition and n output transitions, then the occurrence of the incoming transition should be eventually followed by one of the outgoing transitions. i.e.

  $\Box(incoming\_in\_msg \rightarrow (\bigvee_{i=0}^{n}(<> (outgoing\_in\_msg_i\,[\wedge outgoing\_predicate_i] \wedge$
  $(\bigwedge_{j=0}^{m} outgoing\_out\_msg_{i,j}))))$

  – If the state has one or more outgoing transitions, then each transition input message should be followed by its corresponding output. i.e.

  $\Box(\bigwedge_{i=0}^{n}((outgoing\_in\_msg\,[\wedge outgoing\_predicate]) \rightarrow$
  $<> (\bigwedge_{j=0}^{m} outgoing\_out\_msg_{i,j})))$

  – If the state has one or more incoming and outgoing transitions, then the input messages of the incoming transitions should imply the firing of any output messages of the outgoing transitions. i.e.

  $\Box(\bigvee_{i=0}^{n}(incoming\_in\_msg_i\,[\wedge incoming\_pred_i]) \rightarrow$

$<> (\bigvee_{i=0}^{n}(\bigwedge_{j=0}^{m}(outgoing\_out\_msg_{i,j})))$

– If the state s_i has one incoming transition and more than one outgoing transitions, then the input message of the incoming transition should be followed by one of the input messages of the outgoing transitions (with predicate if any).

Transition level: For any transition t in a state machine, the receiving of the input message in_msg while the predicate is true (if any) should always imply the firing of all the n output messages out_msg_i , i.e. $\Box((in\_msg\,[\wedge predicate]) \rightarrow <> (\bigwedge_{i=0}^{n} out\_msg_i))$

Applying the above rules to the simple reliable transfer protocol discussed in Section 3.2, the following eleven properties are extracted:

- $\Box((send2R(s,a) \wedge [a == e]) \rightarrow (<> (ack2L(b) \wedge receive(s))))$, i.e., Always any occurrence of send2R(s,a) AND [a==e] implies ack2L(b) AND receive(s).

- $\Box(send2R(s,a) \wedge [a! = e]) \rightarrow <> ack2L(b)$, i.e, Always any occurrence of send2R(s,a) where [a!=e] implies ack2L(b).

- $\Box(send2R(s,a) \wedge [a == e]) \rightarrow (<> (ack2L(b) \wedge receive(s)) \vee ((send2R(s,a) \wedge [a! = e]) \rightarrow (<> ack2L(b))))$

- $\Box((send2R(s,a) \vee send2R(s,a)) \rightarrow (<> (((send2R(s,a) \wedge [a == e]) \wedge (!(send2R(s,a) \wedge [a! = e]))) \vee ((!(send2R(s,a) \wedge [a == e])) \wedge (send2R(s,a) \wedge [a! = e])))))$

- $\Box(send(d) \rightarrow <> send2L(d,a))$, i.e., Always any occurrence of send(d) implies send2L(d,a).
- $\Box(ack2S(b) \rightarrow <> send(d))$, i.e., Always any occurrence of ack2S(b) implies send(d).
- $\Box(send(d) \rightarrow (<> (((ack2S(b) \wedge [a == b]) \wedge (!(ack2S(b) \wedge [a! = b]))) \vee ((!(ack2S(b) \wedge [a == b])) \wedge (ack2S(b) \wedge [a! = b])))))$.
- $\Box(ack2L(b) \rightarrow <> ack2S(b))$, i.e., Always any occurrence of ack2L(b) implies ack2S(b)
- $\Box(send2L(d,a) \rightarrow <> send2R(s,a))$, i.e., Always any occurrence of send2L(d,a) implies send2R(s,a)
- $\Box((ack2L(b) \rightarrow (<> ack2S(b))) \vee (send2L(d,a) \rightarrow <> send2R(s,a)))$
- $\Box((ack2L(b) \vee send2L(d,a)) \rightarrow (<> ((ack2L(b) \wedge (!send2L(d,a))) \vee ((!ack2L(b)) \wedge send2L(d,a)))))$

3.4 Model Generation and Verification

As stated earlier, in order to use model checking to verify the correctness of a system, a model of the system should be built in a formal language accepted by the model checker. As mentioned above, while model checking can be useful in validating systems, it is not a common practice [55]. Building models is not easy; it can take more time to write the model than it did to write the system's code. Furthermore, it is easy to miss errors when checking the model rather than checking the code itself. To overcome these challenges, an algorithm was designed to automatically generate a model that accurately represents the system (design and properties specification) in a language accepted by the model checker.

Model generation processes depends on the used model checker since each model checker has it is own modeling language. Section 4.3 describes how to build a model in PROMELA language which is the formal language of SPIN model checker. Once the model is generated, the properties to be checked should be added to the model. After generating the model along with the properties, the model is validated using the model checker. The system properties such as the absence of deadlocks, non- progress cycles and un-executable code are also checked.

3.5 Code Generation

When the properties of the system are verified, a skeleton code of the design specifications can be generated in a general programming language. The method described in Section 3.2 to design a system facilitates this process such that each block in the structural design can be mapped to a unique node process while communication paths are mapped to network sockets which enable data transfer between nodes. The state machine of each block in the structural design is converted to the body of the node process. Different approaches are existing in literature to implement finite state machines. Two of these techniques will be discussed here.

Nested Switch-Case: The most common technique is by using nested switch-case. Even though this approach is easy to write, the complexity of the code increases rapidly as the size of the state machine increases. The following listing gives the generated code of the Simple Reliable Transfer Protocol discussed in Section 3.2.

Listing 3.1 Nested-Switch Case

```
public class Sender {
enum State {wait_data, wait_ack, finished };
enum Event {put,data_req,ack_ind};
private State s;
public Sender () {  s = wait_data;}
public boolean connect (...) {...}
public void accept() {...}
// define sockets
void Transition (Event e) {
  switch (s)
  { case wait_data:
                      switch (e) {
                           case put:
                                // check the predicate
                                // conduct the actions list
                               // send any output messages to the
                                  next block
                                s= wait_ack;
                                break;}
```

```
                    break;
                    case wait_ack:
                         switch (e) {
                              case ack_ind:
                                   // check the predicate
                                   // conduct the actions list
                                  // send any output messages to the
                                     next block
                                   s=finished;
                                   break;
                              case timer_expire:
                                   // check the predicate
                                   // conduct the actions list
                                  // send any output messages to the
                                     next block
                                   s=wait_ack;
                                   break; }
                    break;
                       case finished:
                       // any actions required
                       break;
     } } }
```

State Design pattern: State design pattern is one of the behavioral patterns discussed in [64]. This pattern can be used to implement a finite state machine since an object can change its state based on some events. Figure 3.10 shows the structure of state pattern. Follows is the State Design Pattern code which is adapted in this tool for the Simple Reliable Transfer Protocol Sender process.

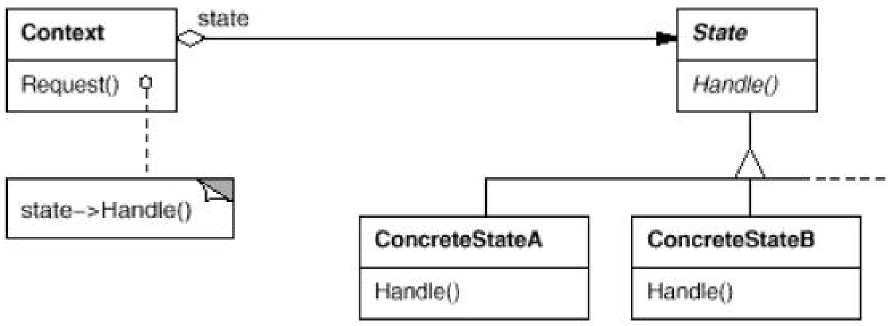


Figure 3.10 State Design Pattern Structure [64]

Listing 3.2 State Design Pattern generated code

```
abstract class abstractState {
  public virtual void PUT(Sender s, byte d){ }
  public virtual void DATAREQ (Sender s, byte d, bool a){ }
  public virtual void ACKIND (Sender s,bool b){ }
  public void GOTO(Sender s, abstractState des)
  {   s.ChangeState(des);  }
};


public class WaitData: abstractState {
  public override void PUT(Sender s,byte d) {
    //start the timer
    GOTO(s, new WaitAck()); } }

public class WaitAck: abstractState {
  public override void ACKIND(Sender s,bool b){
    if ( a!=b ){
      //start the timer
      GOTO(s, new WaitAck());}
     if (a == b){
      a = !a;
      // reset the timer
      GOTO(s, new WaitData());}  }}

public class Finished: abstractState{...}

public class Sender{
  abstractState state;
  public Sender() {  state = new WaitAck(); }
  public void PUT(byte d) { state.PUT(this,d); }
  public void DATAREQ (byte d, bool a) { state.DATAREQ(this, d, a); }
  public void ACKIND(bool b) { state.ACKIND(this, b); }
  public void ChangeState(abstractStateSender s) { state = s;}
}
```

3.6 Debugging

Debugging is a process of discovering any bugs that may reside in a software system to make it behaves as expected. Different approaches are used to simplify debugging software systems which can be based either on a comparison of different executions of the system or interactions with the user. In Comparative debuggers a comparison of several runs of the same program is conducted by analyzing the traces collected in the course of the execution. Interactive debuggers provide an interaction between the user and the debugger in any stage of the execution of the application by setting break points, step-by-step execution and displaying values of particular variables. These type of debuggers are not able to provide a high-level picture of the current state of the multi-process application, visualize the interactions between processes or link the implementation code to the original design.

In this framework a step forward in debugging multi-process applications is proposed. Starting from the design phase, a break point can be inserted at any transition in the behavioral description of the system. These break points are reflected in the code generation phase by adding a breakpoint in the corresponding code. During the execution of the code at the debugging mode, if a process stops at one of the breakpoints, the ability to link this break point to its corresponding source in the design is provided. This integration of the state behavioral diagram and the debuggers allow the developer to get a high level picture of the system while debugging and trace the interactions between the processes. More details and examples about this will be shown in Chapter 4.

3.7 Summary

A multi-phase approach for developing, verifying and debugging message passing systems is described in this chapter. The target system is described using a two-level design approach: abstract communication blocks level and hierarchical state-behavioral specification level. A procedure to automatically generate formal models in a MC language is proposed. On the other hand, a new method based on choosing from a pre-determined set of patterns in concurrent communication properties is proposed to facilitate collecting the essential specifications of the system. Once the model and the properties of the system are generated, a model checker is used to verify the correctness of the proposed system and ensure its compliance with specifications. In case of any inconsistencies found, the error source will be automatically localized using an algorithm that was developed for this purpose. A skeleton code of the system that includes the correct design specifications is generated in a general programming language i.e. Microsoft C#, Java, etc., moreover, the ability to debug the code using a conventional IDE while tracing the debugging process back to the original design was established.

In order to use the proposed approach to design and verify a system, the communication between the components of the system, i.e. processes or threads, should only be done through passing messages not through shared memory. Moreover, the behavior of the system components should be described as a finite state machine.

Chapter 4

MPS Verification and Debugging Tool

This chapter introduces a software tool developed to support the proposed framework. The tool was implemented in Microsoft Visual C# and includes the design, verification, implementation and debugging phases. A Java version is also under development. The proposed framework is general such that any model checker can be applied as a verification engine. In the developed tool, the model checker of choice is SPIN [13]. Figure 4.1 shows a schematic representation that summarizes the structure of the proposed tool.

The ideal tool for verifying multi-process systems should be easy to use such that it does not require previous knowledge of mathematical details of formal verification and model checking but on the other hand it should be powerful and flexible in finding undesired properties of the system. The tool parts are described in the following sections.

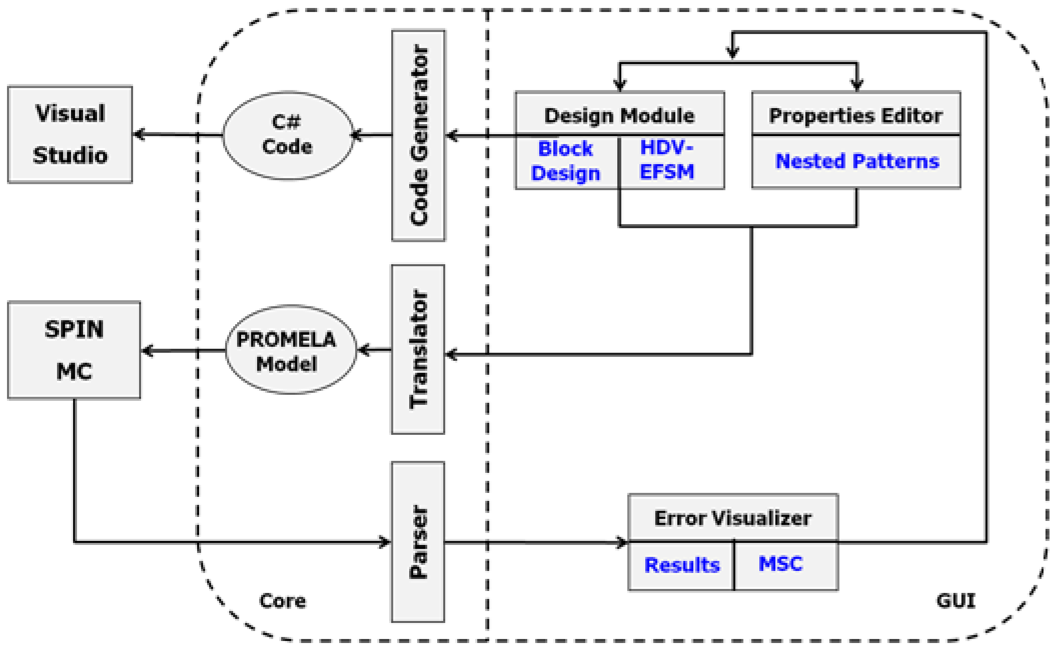


Figure 4.1 Proposed tool

4.1 Model Designer

The tool has a main window which allows the user to start building the high-level block and state-behavioral design of the system. Figure 4.2 shows the window to edit the high-level block model. The diagram shows the structural design of the simple reliable transfer protocol described before. To the top, the tools to add blocks and communication channels between these blocks are given. To the right, a display of the systems properties that will be added later is shown (Section 4.2)

Each block should have a unique name and can be connected to one or more blocks via communication channels. Each channel should have a name and a list of messages with

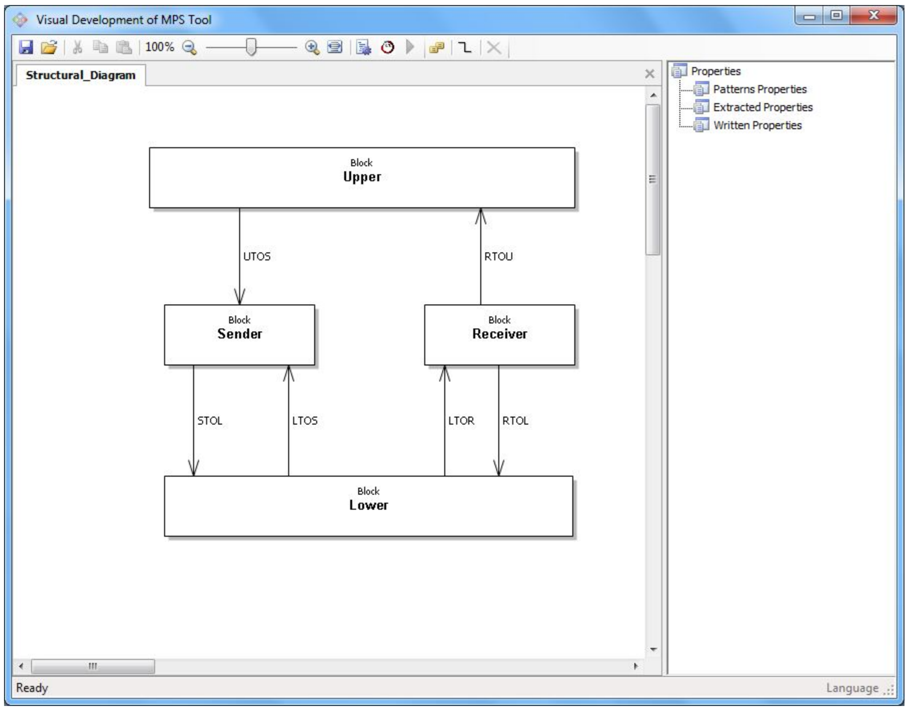


Figure 4.2 High-Level Block diagram of the simple reliable transfer protocol

parameters that this channel can pass. These massages can be added using the form in Figure 4.3.

Each block has a state behavioral model which can be displayed by a double click on the block shape. Figure 4.4 shows the behavioral model of block *Sender*. To the top, the tools to add states and state transitions are placed. In some cases, local variables should be added to the state diagram to complete the required functionality. For example, in the simple Reliable Transfer Protocol, a counter is added to the sender state diagram to keep track of the number of the conducted trials for sending a data bucket. Figure 4.5 shows the form used to add a local variable.

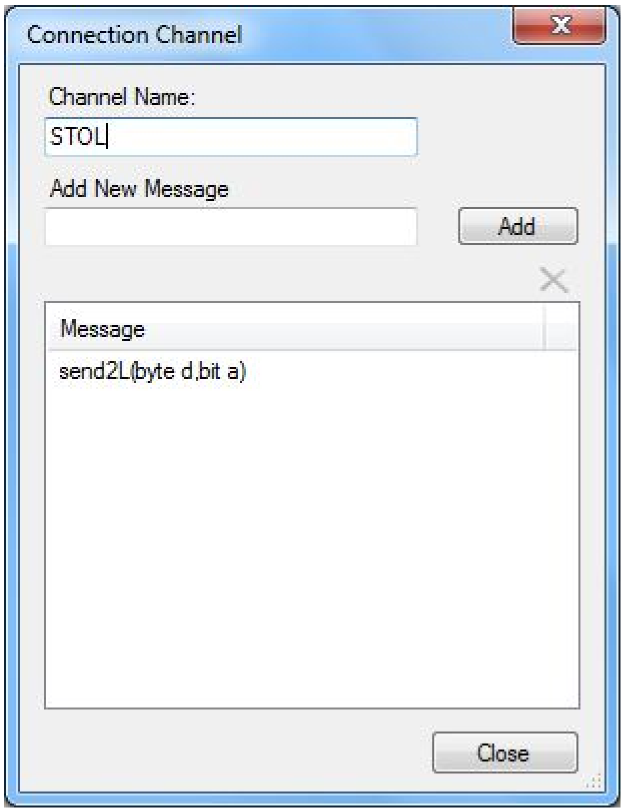


Figure 4.3 Channel Editing Form

Each state should have a unique name and can have one or more state transitions. Each transition can have input event, predicate, list of actions and list of output events. Figure 4.6 shows the form used to update a state transition.

End state: while designing the state-behavior of the system, the user should specify the control state that can be accepted as a valid termination point. In the verification phase, if the system terminates at a state that is not labeled as an end state, an error will be reported.

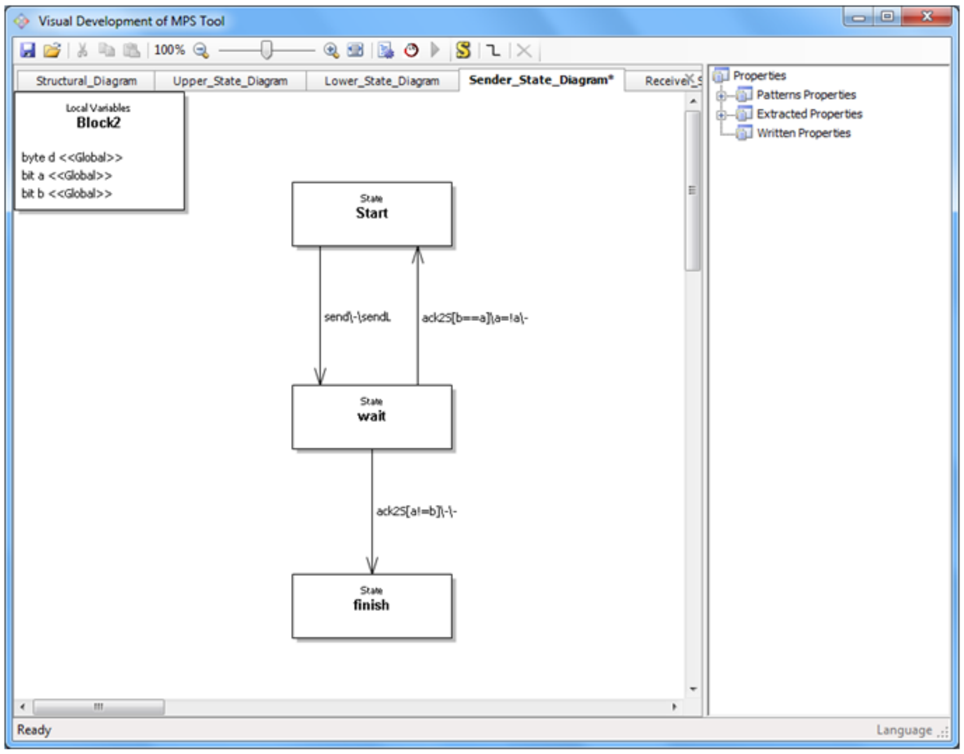
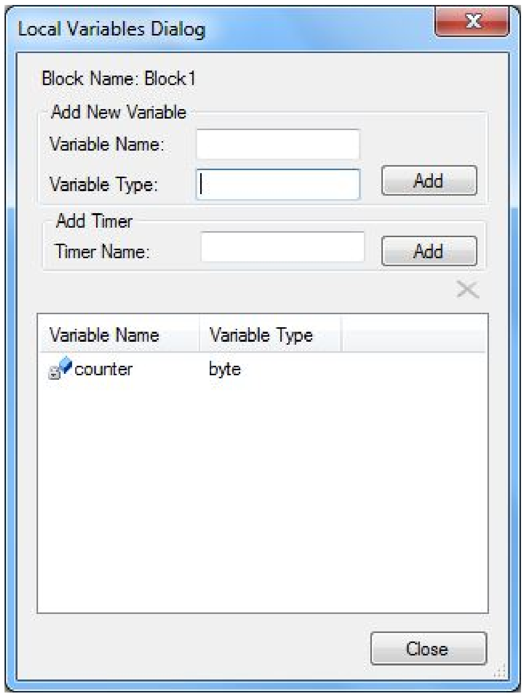


Figure 4.4 State-Behavioral diagram for Sender Block

4.2 Property Editor

Another part of the tool is the Property Editor which allows the user to edit the specification properties of the system. As stated earlier, the temporal logic used by SPIN is Linear Temporal Logic(LTL). Writing correct LTL expressions is not an easy task [35, 65]. Two modes in writing user defined properties are introduced in this tool: Not-Expert mode and Expert mode. In the not-Expert mode the hierarchical pattern-based technique discussed in Chapter 3 is used to help in writing the LTL properties. Figurer 4.7 shows the form to write an LTL using patterns. The user can choose the required pattern and scope form a drop-down list. For each chosen pattern and scope, an LTL formula will be displayed to



Variable Name	Variable Type
counter	byte

Figure 4.5 Form to add local variables

the user in addition to the required atomic proposition that should be filled. These atomic propositions can be chosen from other drop-down lists of all the basic events in the behavioral models and automatic extracted properties as in Section 3.3.1.4. Based on the built property, the tool will give help in the meaning of the selected property.

The hierarchical property is built up from a set of simple properties, a boolean operator and a specification pattern. These hierarchical properties can be built using the form in Figure 4.8.

The Expert mode allows the user to write the property in LTL from scratch. This requires a background in temporal logic. A syntax check is conducted in the final formula to report

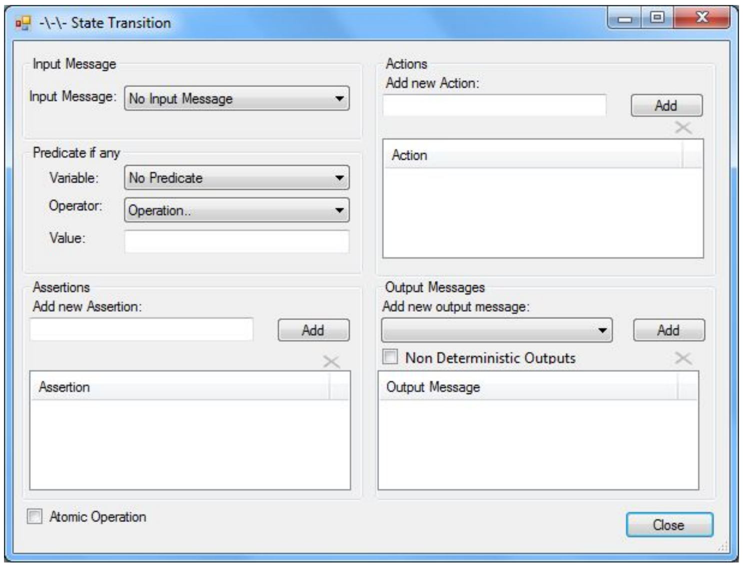


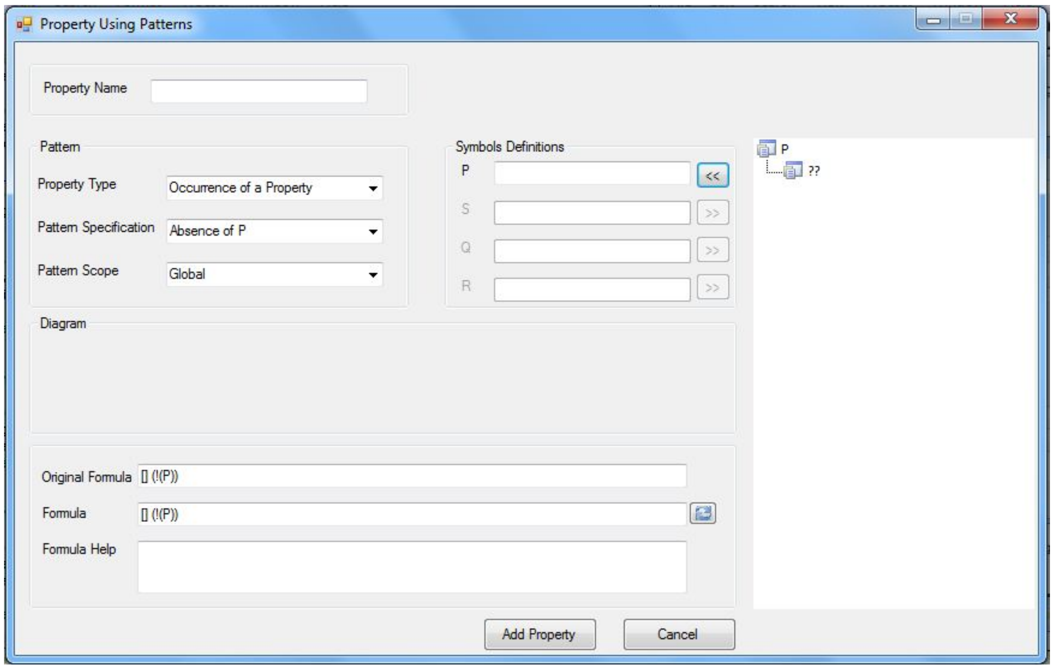
Figure 4.6 State transition editing form

any invalid token or any error in parsing the formula.

4.3 Model Generation

Once the design of the system and its properties are determined, the tool will generate a PROMELA model that will be given to SPIN for verification. Using the method described in chapter 3 to design a message passing system, PROMELA code can be generated automatically from the design since the semantics of the modeling approach are closely related to the semantics of PROMELA language. Each high-level block is translated to a process in PROMELA such that the process describes the behavior of the block. The mapping from the block and state-behavioral design of the system to PROMELA is done using the following rules:

- High-Level Block Design



Property Using Patterns

Property Name	

Pattern

Property Type	Occurrence of a Property
Pattern Specification	Absence of P
Pattern Scope	Global

Symbols Definitions

P	
S	
Q	
R	

Diagram

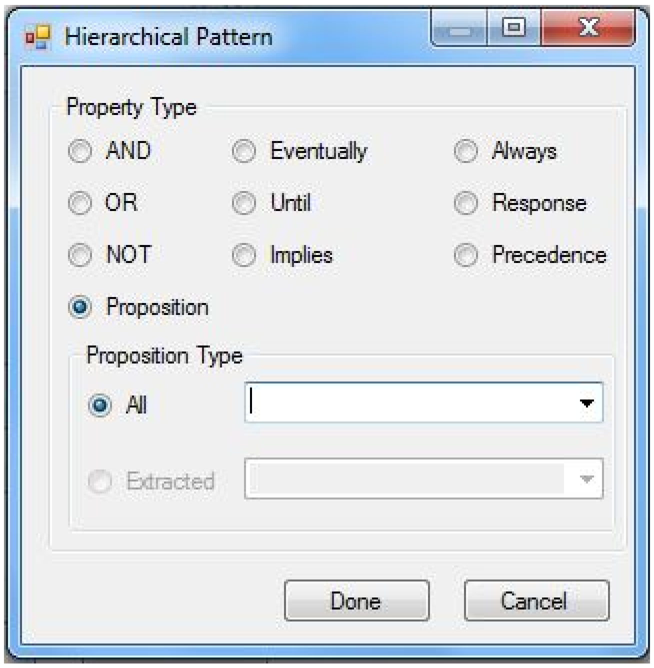
Original Formula	[] (!(P))
Formula	[] (!(P))
Formula Help	

Figure 4.7 Property Using Patterns Form

- Block → PROMELA Process
- Block Communication Path → PROMELA Channel
- Message → constant in PROMELA

• State-Behavioral Design

- Block DV-CEFSM → PROMELA Process Body
- State → Label
- Message Exchange → PROMELA send(?) and receive(!)



Property Type
AND Eventually Always
OR Until Response
NOT Implies Precedence
[✓ Proposition]

Proposition Type
[✓ All]
Extracted

Figure 4.8 Form to build the property hierarchically

- State Transition $\rightarrow$ goto statement
- Predicate $\rightarrow$ statement guard
- Message Parameters $\rightarrow$ process local variables

4.3.1 Translation Steps

The formal approach of translating the design of the system to PROMELA made it possible to implement a module to generate PROMELA code automatically. Let S is the presentation of the target system and PModel is the PROMELA model of the generation, the following are the steps to generate PModel from S:

Step 1: If any block of the system has a hierarchical CEFSM, use the method described in Section 3.2.4 to flatten the machine.

Step 2: Globally, define a symbolic name for each message in all the communication channels at the system S using the *mtype* keyword. For example, the following statement declare symbolic names for the messages msg1(), mg2(...), mg3(...)

```
mtype={msg1,msg2,msg3}
```

Step 3: For each communication channel in S, define a PROMELA channel that can store messages of type *mtype* and the communication channel messages parameters type. For example, if there is a channel *chan1* between *Block1* and *Block2* that can pass the messages {*msg1(byte d), msg2 (bit a)*}, this channel will be defined as

```
chan chan1 = [0] of {mtype, byte, bit}
```

Step 4: For each block B in the System S, add the process *proctype active B() {...}* to the PModel, where B is the name of the block. For example, let the S contains the blocks Block1, Block2, Block3, then the PModel should contain the following statements:

```
proctype active Block1() {...}

proctype active Block2() {...}

proctype active Block3() {...}
```

In PROMELA, the keyword *active* is used to indicate that a process is required to run immediately once the PROMELA program starts.

Step 5: For each process p, define the parameters of the corresponding block. These parameters include the arguments of the incoming and outgoing messages of the block and any local variables added by the user during the design phase.

Step 6: For each state in the block state machine, write the name of the state as a label in PModel. In PROMELA the syntax of the label is *Label_Name:*. The **end** prefix should be added to the beginning of the initial and end states labels.

Step 7: For each state in the block state machine, group the outgoing transitions of this state based on the incoming message in each transition as follows:

- One outgoing transition without predicate : in this case write the following

  ```
  channelName?incoming_message ->
  assertions // if any
  list of actions // if any
  list of outgoing messages with their corresponding channels i.e.
  channelName!outcoming_message // if any
  ```

  In Promela, the symbol "?" means receive a message through a specific channel while the symbol "!" means send a message through a specific channel. If this operation is atomic ,i.e. t=true, the *atomic {..}* keyword is added around the transition code.

- More than one outgoing transitions: if a state has more than one outgoing transitions, group these transitions based on the incoming message. The corresponding PROMELA code is:

  ```
  if
  :: group1
  :: group2
  ...
  fi;
  ```

  Here $group_i$ presents the outgoing transitions with the same incoming message and different predicates, the following is the code of $group_i$ where the symbol *::* is used to indicate the nondeterminacy of events :

  ```
          channelName?incoming_message ->
          if
               :: predicate1
                  assertions // if any
                  list of actions // if any
                  list of outgoing messages with their corresponding
                  channels i.e.  channelName!outcoming_message // if any
               :: predicate2
                  assertions // if any
                   list of actions // if any
  ```

```
                list of outgoing messages with their corresponding
               channels i.e.  channelName!outcoming_message // if any
          ...
      fi;
```

4.3.2 Timers in PROMELA

Timer is an important component in modeling most of Message Passing systems. In MPS, a message may be received later than expected or not received at all. A timer is defined as a component that is assigned a specific value such that the timer will generate a special expiration signal when that value exceeded [57]. There are two operations related to the timer,*set(v)* and *reset()*. While, the *set(v)* assigns the value v to the timer and starts the timer, the *reset()* operation stops the timer. Timer is used in the example of the simple reliable transfer protocol that was discussed in Section 3.2. In that example, to avoid infinite waiting for an acknowledgment *ACK* from the receiver *B*, the sender *A* starts a timer each time it sends a message *d* to *B* such that if this timer expires before an *ACK* is received, *A* will resend the message. Node *A* will try to send a message frame *N* times.

Timer should be translated to PROMELA in order to use SPIN in the verification process. PROMELA language does not support timers and timers operations directly [13]. PROMELA has a "timeout" statement which is used to test for a deadlock in the system and has no relations to timer expiration. An abstraction of a timer is proposed in [66] where a timer is modeled as a boolean variable. In that abstraction, setting the timer is conducted by assigning true value to the variable, while resetting the timer is conducted by assigning its value to false. The generation of the expiration signal is abstracted by testing the value of the timer. Having a true value at the time of testing means that the timer was set before and it is expired at this moment. The following PROMELA code shows the modeling of a timer *T*:

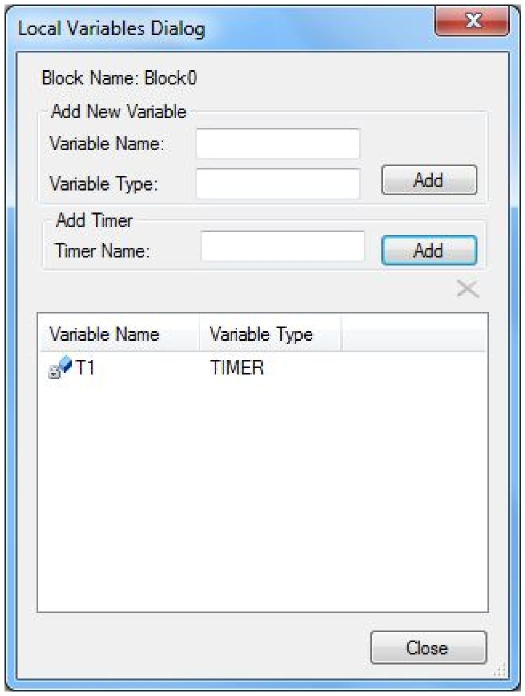
```
bool T = false;  /*Declare the timer*/

T = true; /* see the timer to value v i.e. set(v)*/

T = false; /*reset the timer i.e. reset()*/

(T == true) /*test timer expiration*/
```

The above example can be modeled in PROMELA as follows, assuming that A and B are connected through channel ch:

```
 /*the Sender A*/

   bool T = false;

   ch!d;

   T = true;

   if

     :: ch?ack;

           T = false;

     :: (T == true) ->

      code to handle the timer expiration
```

```
  fi;


/* the receiver B */

   ch?data;

   if

     :: ch!ack;

     :: skip;

  fi;
```

In the developed tool, dealing with timers as discussed above is hidden from the user. Figure 4.9 shows the form that is used to add timers to the design.



Block Name: Block0

Variable Name:

Variable Type:

Timer Name:

Variable Name	Variable Type
T1	TIMER

Figure 4.9 Form to add timers

4.3.3 LTL Properties in PROMELA

In order to verify the properties of the system using SPIN, these properties should be expressed in LTL as described in Section 2.1.3. These LTL expressions should be converted to a never claim in PROMELA which is just another process that is executed in lock-step where a step from the never claim process executes after each step from the system. For example, going back to the Simple Reliable Transfer protocol described in Section 3.2, let P is the property that the sender block eventually received the message *send* form the upper block and this message should be followed by sending the message $send2L$ to the lower block. This property can be expressed in LTL as $(<> p \wedge [\,](p \rightarrow <> q))$ where q is the event that *send* is received and q is the event that $send2L$ is sent. The corresponding never claim for this property is generated using SPIN with the *-f* option for the negation of the property. Follows is an example of a never process code:

```
never  {    /* !(<>p && [](p-><>q)) */
T0_init:
        if
        :: (! ((p))) -> goto accept_S2
        :: (! ((q)) && (p)) -> goto accept_S8
        :: (1) -> goto T0_S5
        fi;
accept_S2:
        if
        :: (! ((p))) -> goto accept_S2
        fi;
accept_S8:
        if
        :: (! ((q))) -> goto accept_S8
        fi;
T0_S5:
        if
        :: (! ((q)) && (p)) -> goto accept_S8
        :: (1) -> goto T0_S5
        fi;
}
```

In PROMELA model, in order to define *p* and *q* in the above example, remote reference labels [13] are introduced where a label is added right after the occurrence of the corresponding event and the reachability of this label is checked. For example, for the above property *P* in the Simple Reliable Transfer protocol, two labels are added to PROMELA code in the Sender process. The first label *label_1* is added right after the statement of receiving the *send* message through the *UTOS* channel while the second label *label_2* is added after sending the message *send2L* through the *STOL* channel. The following code shows these labels:

```
       if
       :: UTOS?send(d) ->
label_1:          /* added label for verification  */
       /* do some operations */
       STOL!send2L(d,a);
label_2:           /* added label for verification */
       goto wait;
       fi;
```

The following statements are adding to the beginning of the PROMELA code to define *p* and *q* where *Sender* is the name of the sender process.

```
       # define p Sender@label_1
       # define q Sender@label_2
```

In the developed tool, the above process of generating the never claim, adding the verification labels and adding the parameters definitions are done automatically without any interaction from the user. Once the user chooses a property to verify, the LTL expression is converted to never claim and added to the PROMELA model along with the required definitions.

4.3.4 Correctness of the PROMELA Model Generation

To justify the correctness of the translation steps discussed in Section 4.3.1 to generate a PROMELA model from the design of a system, the completeness of the translation and the consistency between the two models are discussed in this section.

4.3.4.1 Completeness

The translation process from the description of a system to a PROMELA model is complete if there is a corresponding mapping for all the elements in the design to PROMELA language.

Lemma 1. Let *S = (B, Ch, G)* is the description of the target system, and *PModel* is the translated PROMELA mode where *B* is a set of all the high-level blocks in the system *S*, *Ch* is a set of all the communication channels in the system and their messages and *G* is a set of global variables of all the blocks in the system and their initial values. The translation is complete if the elements *B, Ch, G* are correctly mapped to PROMELA.

Proof. Steps (2-4) in Section 4.3.1 cover all the elements in S, i.e., $\forall ch \in Ch$ a channel is created in step 3 using *chan* keyword, $\forall b \in B$ a process *proctype b()* is created in Step 4 and for each message in *S* a symbolic name is globally defined using the *mtype* keyword in Step 2.

Lemma 2. Given a block *b* in *S* and its corresponding process *proctype b( )* in *PModel*, the translation of b is complete if all the elements of the DV-CEFSM of b are properly mapped to PROMELA in *PModel*.

Proof. The DV-CEFSM of a block b is defined as $(S, s_0, M, f, V, s_{end}, a)$. Steps (5-7) in Section 4.3.1 cover all the elements in DV-CEFSM. In Step 5, the parameters M of the DV-CEFSM are added to PModel. In Step 6, each state name is converted to a label in PModel in addition to the **end** prefix added to the beginning of the initial state s_0. In Step 7, the parameters $\{s_s, s_e, i, p, a, o, d, t, b\}$ of each outgoing transition of a state are mapped to PROMELA such that an incoming message i is translated to receive (!), a predicate p is translated to a statement guard and outgoing messages o are translated to send (?). If this operation is atomic ,i.e. t=true, the *atomic {..}* keyword is added around the transition code.

Lemma 3. Let S is the description of the target system and *PModel* is the translated PROMELA model. The translation is complete if $\forall b \in B$, there exists a process *proctype b( ) {...}* in *PModel* such that, *b( )* covers the syntactic and static semantics of b.

Proof. From Lemma 2, the translation rules can be applied to all the blocks in S. Therefor, *PModel* is a complete translation of S.

4.3.4.2 Consistency

The description of a system S is consistence with its corresponding PROMELA model PModel if every execution path in S has a corresponding path in PModel.

Lemma 4. Given a System S and a Promela model PModel, the two models are consistence if the initial behavior of both models are the same.

Proof. For each block b in S, the only initial executable instruction in the corresponding PModel process is one of the following: (i) if the initial state of the block has no incoming input message then the sending of the output message is executable, (ii) if the initial state of the black has an incoming input message then the block has to waits for receiving of this message.

4.4 Verification

Once the user is done building the system and its required properties, SPIN can be used to model check the system. Assuming that SPIN is already installed in the machine, the user should generate PROMELA code using a button in the top bar and specify the path to save the created file. Verifying the model can be done using a "verify" button which will run SPIN model checker on the created PROMELA file. To run SPIN as a verifier, the following three commands should be given to SPIN:

```
spin -a  PROMELA_file_Name
gcc -o pan [Compile-time options] pan.c
pan [Run-time options]
```

In the developed tool, two types of properties can be verified. The first type is a group of predefined built-in properties in SPIN such as checking deadlock existence. The second type is user defined properties which depend on the application. The next two subsections explain these types in more details.

4.4.1 User specific LTL Properties

To check a specific user defined property, the user can right click on that property in the properties panel and choose verify. The result of the verification will be displayed in a message box. The compile-time options used with this property are *-w -D_POSIX_SOURCE -DMEMLIM=1024 -DXUSAFE -DNOFAIR* while the run-time options are *-v -X -m10000 -w19 -a -n -c1*.

4.4.2 Built-in Properties

Following is a list of predefined properties that can be checked using SPIN along with the compile-time and run-time options required to conduct a specific property:

- **Assertion Violation:** While design a system the user can specify some assertions to be checked during the verification phase. If any assertion is violated SPIN reports this problem. The compile-time options used with this property are *-w -D_POSIX_SOURCE -DMEMLIM=1024 -DSAFETY -DNOCLAIM -DXUSAFE -DNOFAIR* while the run-time options are *-v -X -m10000 -w19 -E -n -c1*

- **Dead lock (invalid end state):** If a PROMELA process blocked at a state that its label does not start with the 'end' prefix, SPIN reports an invalid end state problem which leads to a deadlock. The compile-time options used with this property are

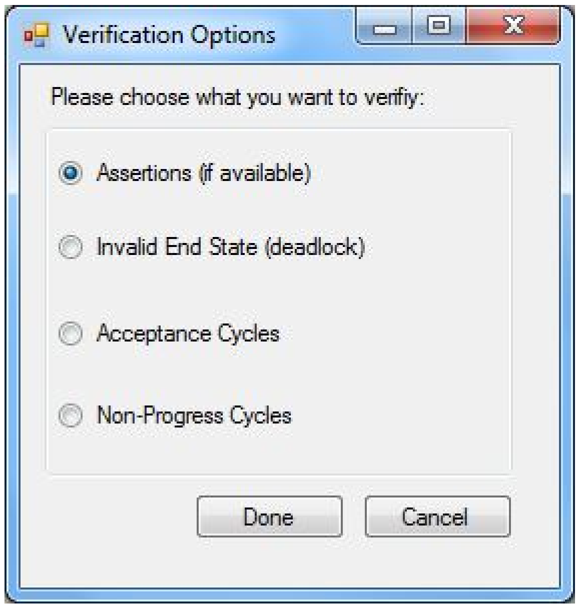
-w -D_POSIX_SOURCE -DMEMLIM=1024 -DSAFETY -DNOCLAIM -DXUSAFE -DNOFAIR while the run-time options are *-v -X -m10000 -w19 -A -n -c1*.

- **Acceptance Cycle:** During the verification, is SPIN finds a cycle that infinitely often visits a state which is labeled as acceptance state, an acceptance cycle problem will be reported. The compile-time options used with this property are *-w -D_POSIX_SOURCE -DMEMLIM=1024 -DNOCLAIM -DXUSAFE -DNOFAIR* while the run-time options are *-v -X -m10000 -w19 -a -n -c1*.

- **Non-Progress Cycle:** During the verification, if SPIN finds a cycle that does not infinitely often visit a state which is labeled as progress state, an Non-Progress cycle problem will be reported. The compile-time options used with this property are *-w -D_POSIX_SOURCE -DMEMLIM=1024 -DNP -DNOCLAIM -DXUSAFE -DNOFAIR* while the run-time options are *-v -X -m10000 -w19 -l -n -c1*.

Figure 4.10 shows the form used to choose the required property to check. Using the developed tool, running SPIN verifier is conducted in the background and does not need any user interaction. Once the user chooses a property to check, the tool generates the required commands along with the appropriate parameters and passes these commands to SPIN. The verification output of SPIN looks like the following:

```
(Spin Version x.x )

         + Partial Order Reduction
```



Verification Options

Please choose what you want to verifiy:

[✓ Assertions (if available)]

Invalid End State (deadlock)

Acceptance Cycles

Non-Progress Cycles

Figure 4.10 Verification Options

```
Full statespace search for:

        never-claim                - (none specified)

        assertion violations       +

        acceptance   cycles        - (not selected)

        invalid endstates       +

State-vector x byte, depth reached x, errors: x

        x states, stored

        x states, matched

        x transitions (= stored+matched)

        x atomic steps

        x memory usage (Mbyte)
```

This output is parsed and a message box will appear to show the results of verification which can be one of the following: Syntax Error in the model, Acceptance Cycle, Assertion Violation, Invalid End State (deadlock), Unknown Verification Error or no errors.

4.5 Error Localization

During the verification process, if the model does not satisfy a specific property then SPIN creates a trial file that contains a counter example that leads to the problem. In most cases, this trail file is hard to explore and understand. The following code is a segment of a trail file generated by SPIN for a dead lock property in the Simple Reliable Transfer Protocol discussed in Section 3.2.

```
4: proc  2 (Receiver) SRTP.pml:102  Sent ack2L,0 -> queue 1 (RTOL)
5: proc  3 (Lower)    SRTP.pml:122  Recv ack2L,0 <- queue 1 (RTOL)
7: proc  2 (Receiver) SRTP.pml:103  Sent receive,0 -> queue 2 (RTOU)
8: proc  0 (Upper)    SRTP.pml:21   Recv receive,0 <- queue 2 (RTOU)
...
```

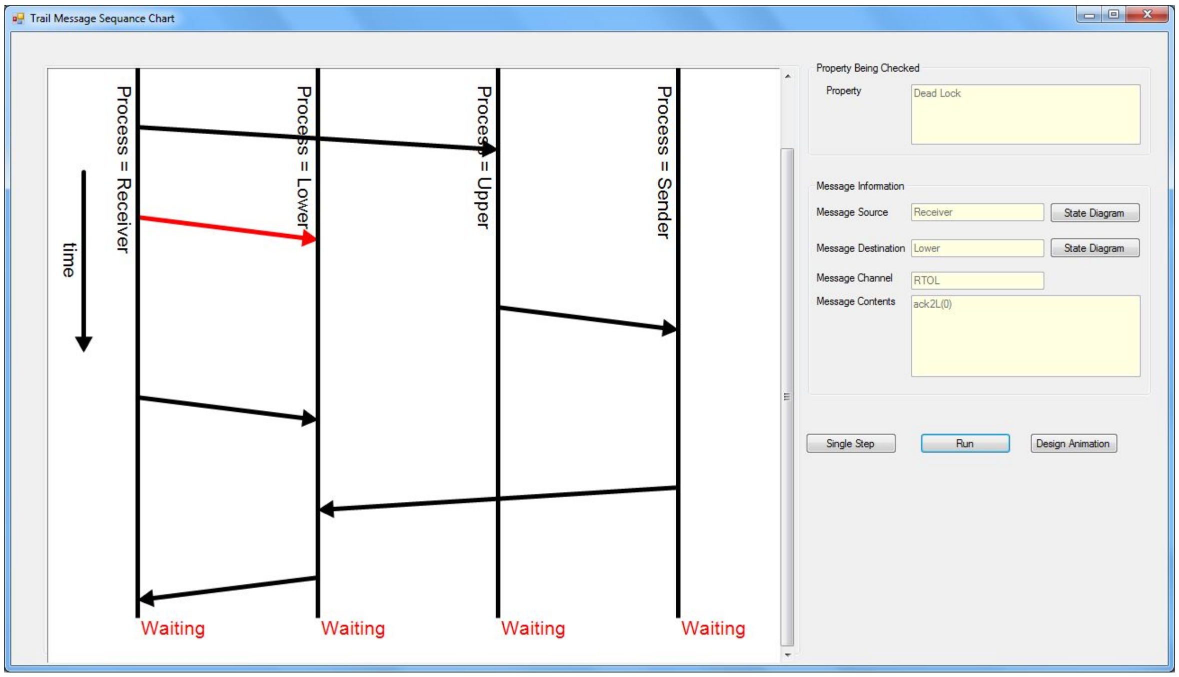
Each statement in the above code include process id, process name, the name of the trail file, the number of the line, the received or sent message name, the parameters of the message and the channel name.

Different graphical tools have been developed to facilitate the studying of trial files. iSpin [13] is a tool developed by the creator of SPIN to be used as a graphical interface to SPIN.

iSpin, earlier called xSpin, is written in tcl/tk and convert the trail file to Message Sequence Chart (MSC) to display messages in the trail scenario. St2msc [67] is similar to iSpin but with additional functionalities such as merging processes in the MSC to easily understand the chart.

In this research a step forward was taken to help in interpreting the counter example which is created by SPIN. A Message Sequence Chart (MSC) is generated with a user interaction ability that allows the user to choose a message in the MSC to show its contents and to map this message to the original design diagram. This module is called the error localization module. The error localization module consists of a parser, user interface and a connector. The parser reads the trail file and identifies the processes and the messages types. The user interface graphically draws the message sequence chart and the connector connects the MSC to the original behavioral design (state machines). Moreover, an animation of the counter example in the original behavioral design is also implemented. This animation allows the user to go from one transition to another in the counter example. Figure 4.11 shows the trail MSC generated for a deadlock in the Simple Reliable Transfer Protocol discussed in Section 3.2.

The figure shows four processes with messages interactions between these processes that lead to a deadlock. Choosing any one of these messages will display the source, destination, channel name and the contents of the message. Drawing the MSC can be done either step by step or at once.



Property Being Checked

Property: Dead Lock

Message Information

Message Source: Receiver

Message Destination: Lower

Message Channel: RTOL

Message Contents: ack2L(0)

Figure 4.11 The Trail Message Sequence Chart

To help in locating the source of the problem, a further step is conducted by linking the message sequence chart of the counter example to the original design. Choosing a message in the MSC allows the user to display the same message in either its source or its destination state diagram such that the corresponding state transition will be displayed in red color. Figure 4.12 shows an example where the transition that contains the *send(byte d)* message is marked. Moreover, an animation of the counter example can be run in the behavioral design.

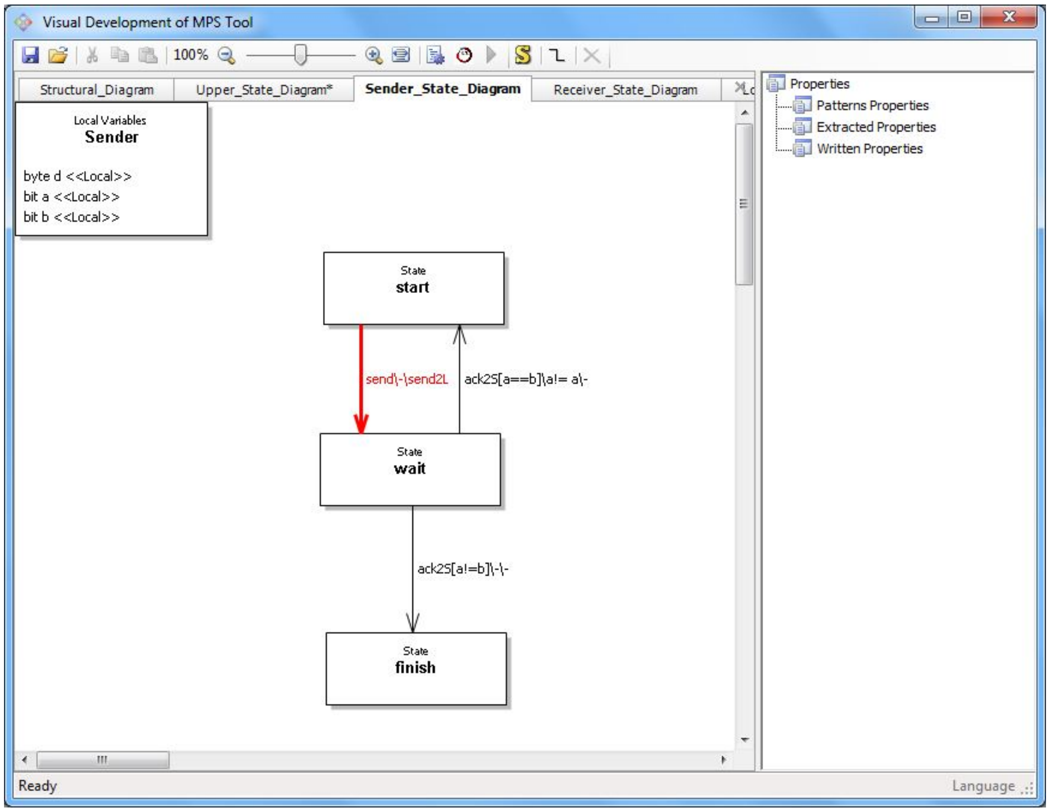


Figure 4.12 Linking the MSC to the original design

4.6 Code Generation

In the developed tool, a module to generate C# code for the system design is built. This module uses the state design pattern discussed in Section 3.5. A single file will be generated for each block in the system. If a transition in the design has a break point, the corresponding generated code of this transition will be proceeded by the following statement to allow a break during the execution in the debugging mode.

```
#if DEBUG
        System.Diagnostics.Debugger.Break();
#endif
```

4.7 Debugging

Conventional IDE debuggers give the developer the ability to monitor the execution of a software system and set break points at specific points in the code to pause the execution at these points. These debuggers are useful in debugging single process application but provide no special help for systems with more than one process. In the developed tool a further step is taken to help the developer benefit from the proposed verification approach through the debuggers. Figure 4.13 shows an example of a transition with a break point such that when a system pauses at the corresponding code of the transition, the code will be linked to the break point in the diagram. In this way the user can benefit from the conventional debugger, e.g. Visual studio debugger, in monitoring the values of the local variables of the process and benefit from the tool in linking the code to the original design in order to give a high level picture of the system execution.

4.8 Summary

This chapter presented a graphical software tool that supports the proposed approach in Chapter 3. The tool, which was developed in Microsoft C#, includes the following phases: system design, description of the properties, model generation, verification, code generation, error localization and debugging. SPIN model checker used as the verification engine.

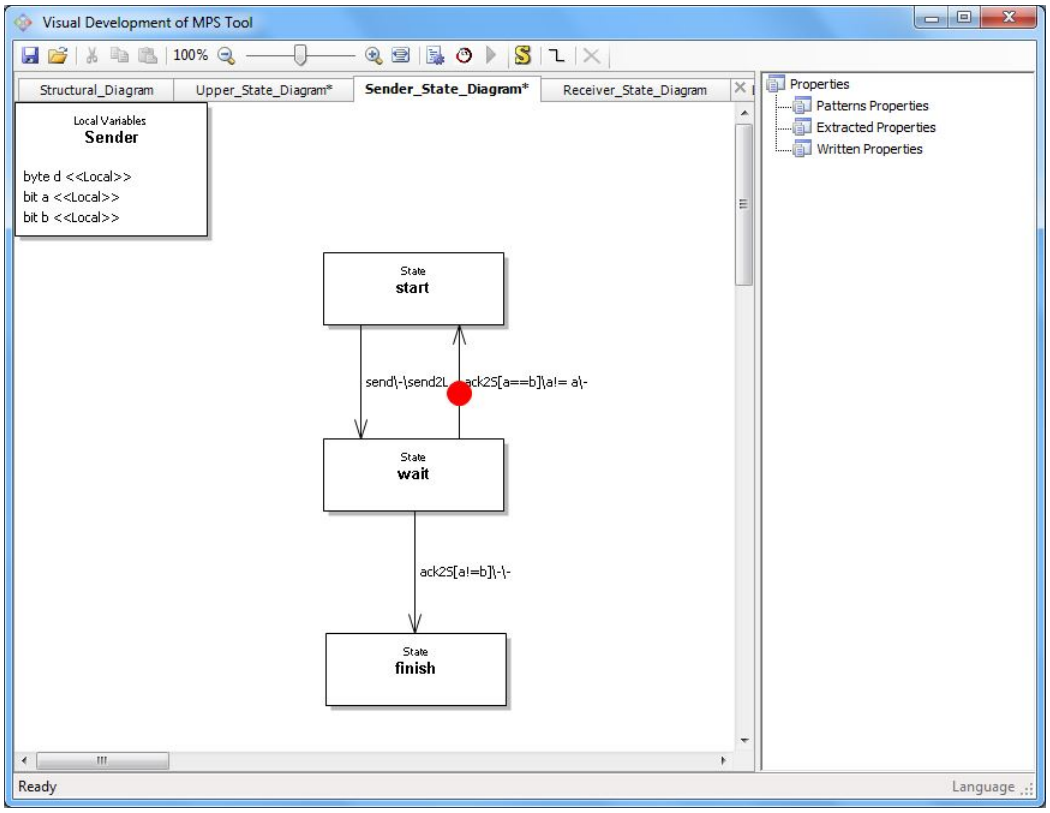


Figure 4.13 Inserting a break point to a state transition in the design

Chapter 5

Case Study

As a case study, the connection establishment phase of the Transmission Control Protocol (TCP) will be verified using the proposed approach.

5.1 Transmission Control Protocol (TCP)

Transmission Control Protocol (TCP) is the most broadly used transport protocol in the Internet. It guarantees a reliable data transfer between machines such that data will be delivered in order without duplication or loss. TCP was described using message sequence diagram and finite state machine [68]. Many modifications and improvements have been conducted in the protocol over years [69–75]. TCP has a connection management protocol for establishing and terminating the connections along with data transfer protocol for reliable data transfer. This research will focus on the modeling and verification of the connection management protocol in order to verify its correctness.

5.1.1 TCP connection management

In TCP, before data can be sent between any two machines, a connection between these machines needs to be set up by exchanging special messages. TCP connection is full duplex i.e. data can flow in both direction concurrently and independently. After finishing sending data TCP machines close the connection and release resources. The procedures of establishing and closing the connection are called TCP connection management.

Connection Establishment

Assume a process running in one machine A wants to start a connection with a process running in another machine B. A's application process should first inform the A's TCP that it wants to start a connection with a process in machine B. The TCP in machine A proceeds to establish a TCP connection with the TCP in machine B following these steps [76]:

1. A's TCP sends a special initial TCP segment to B's TCP. This segment has the SYN bit of the flag segment header set to 1 and has a random initial sequence number A_ISN.

2. Assuming B's TCP receives the segment from A, B allocates the required TCP buffers and variables to the connection and sends a TCP segment to A which has the SYN bit set to 1, the acknowledgment field set to A_ISN+1 and a random se-

lected initial sequence number for B i.e. B_ISN. This segment is called SYNACK segment.

3. Upon receiving the SYNACK segment, A also allocates buffers and variables for the connection and then sends a segment to B that has the SYN bit set to 0, sequence number equals A_ISN+1 and acknowledgment number equals B_ISN+1.

Once these three steps are conducted, the connection is established between the two machines and they can send segments containing data to each other. Since three segments are sent between the two machines to establish the connection, this process is often referred to as three-way handshaking [76]. Figure 5.1 shows this process. This way of establishing the connection is called client-server connection, since one machine initiates the connection(client) and the other responses (server). Figure 5.2 and Figure 5.3 show the state diagram of TCP Server and Client respectively,

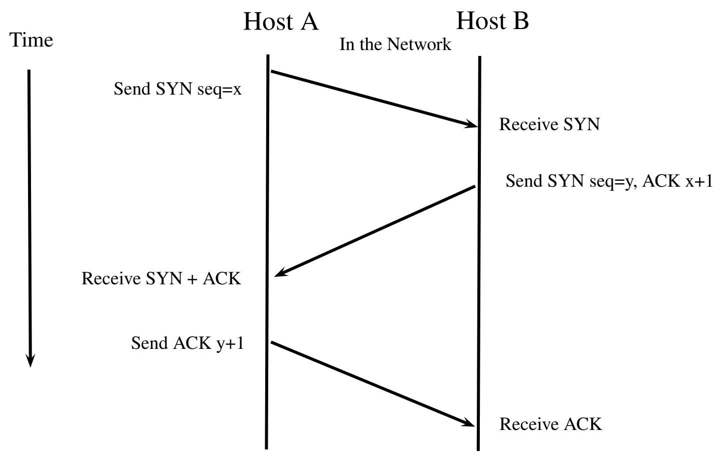


Figure 5.1 TCP three-way handshaking [76]

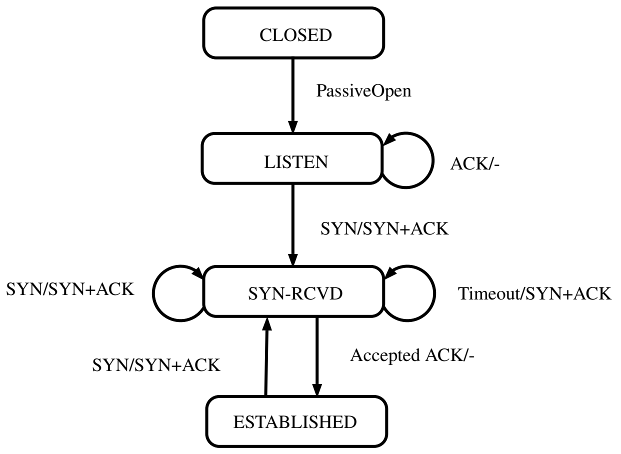


Figure 5.2 State diagram of TCP Server

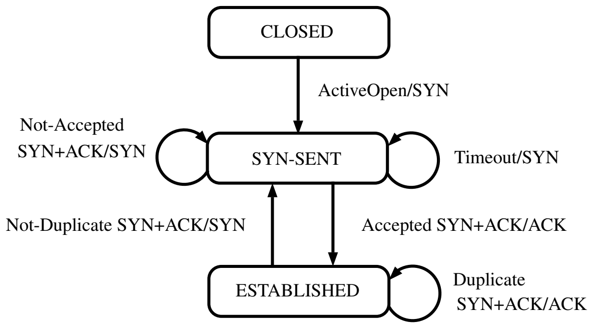


Figure 5.3 State diagram of TCP Client

TCP allows both sides to start a connection simultaneously. In this case each machine sends a SYN segment to the other side and enters SYN_SENT state. Upon receiving the SYN message, each machine responds with SYNACK message and goes to SYN_RCVD state. After receiving SYNACK from the other machine, each side will go to ESTABLISHED indicating the connection is now set up. Figure 5.4 shows this connection.

During the life of a TCP connection, the TCP protocol running in each machine makes

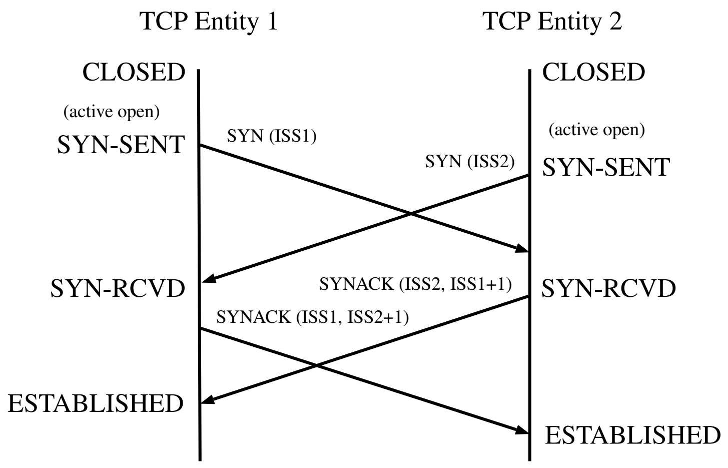


Figure 5.4 Simultaneous TCP three-way handshaking [76]

transitions between several TCP states. Figure 5.5 shows the state machine of the TCP. Please not that this diagram is only a summary of the TCP transitions and is not the total specification of the TCP [68].

5.1.2 Modeling TCP

In this case study, only the client-server connection approach of the TCP three-way handshaking will be considered. A model of the TCP will be built using the structural and behavioral design described in Section 3.2.

5.1.2.1 TCP Structural Design

TCP has two peer TCP entities communicating via a lossy channel and interacting with their application processes. Client and Server structural blocks are required to communicate with each other. To simulate the lossy channel, a Lower block is required which receives from Client and sends to Server and vice versa. An Upper block is required to

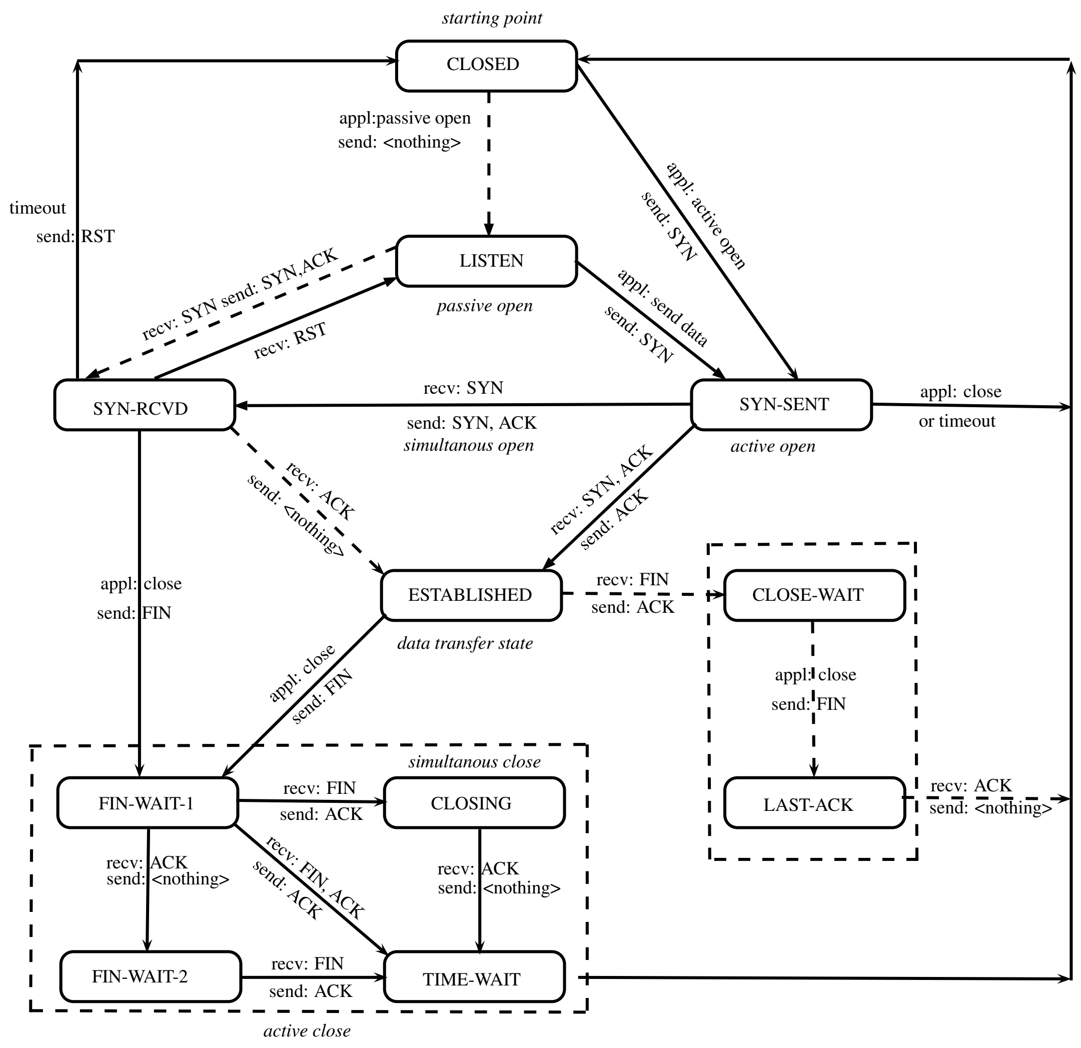


Figure 5.5 Finite State Diagram of TCP [76]

generate the user calls to both Client and Server. Figure 5.6 illustrates the system.

Using the tool, each one of these blocks are added to the structural diagram, and channels between these blocks are added to specify the type of connection messages that can be sent between any two blocks. For example, in both the channels *Upper2Client* and *Upper2Server*, the messages that can be sent are: *ActiveOpen, PassiveOpen* and *Close*. While

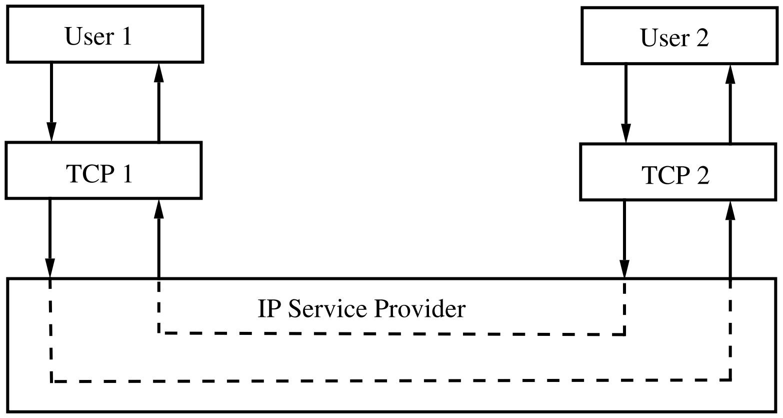


Figure 5.6 Structural Diagram of TCP

in the channels *Client2Lower, Server2Lower, Lower2Client* and *Lower2Server*, the message *TCP (bit SYN, bit ACK, byte seqnum, byte acknum)* can be sent. Figure 5.7 shows the structural diagram built using the tool.

5.1.2.2 TCP Behavioral Design

After the Structural design is done, the behavior of each block was described as DV-CEFSM (Section 3.2.2). Figure 5.8 shows the state diagram of the Client block of the TCP 3-way handshaking protocol built using the tool.

5.1.3 PROMELA Model Generation

The PROMELA model of TCP is generated automatically using the tool. The PROMELA model has four processes (one for each block). The full model code is given in appendix A.

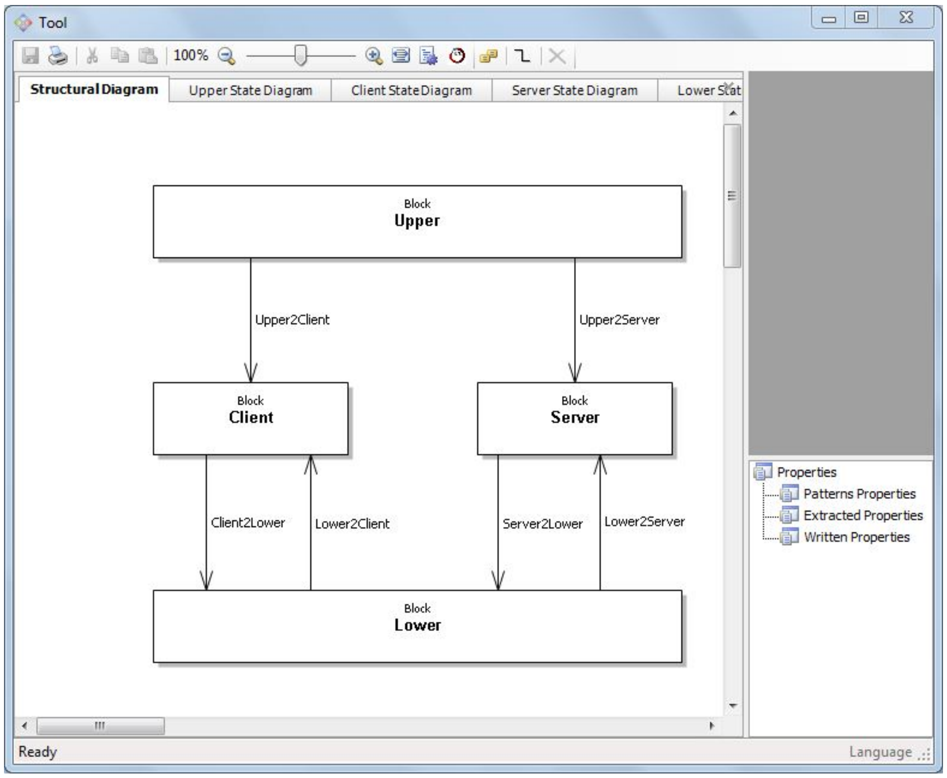


Figure 5.7 Building Structural Diagram of TCP

5.1.4 Verification

As SPIN is built in the tool, it can be used to validate the TCP model. TCP has some correctness requirements that should be checked in the verification process. Each one of these properties is described below along with its pattern and LTL formalization as discussed in chapter 3:

1. Deadlock Free: the model should be free of deadlock. This property is verified using by SPIN by checking that all PROMELA processes terminate at an *end* state.

2. No unexecutable code

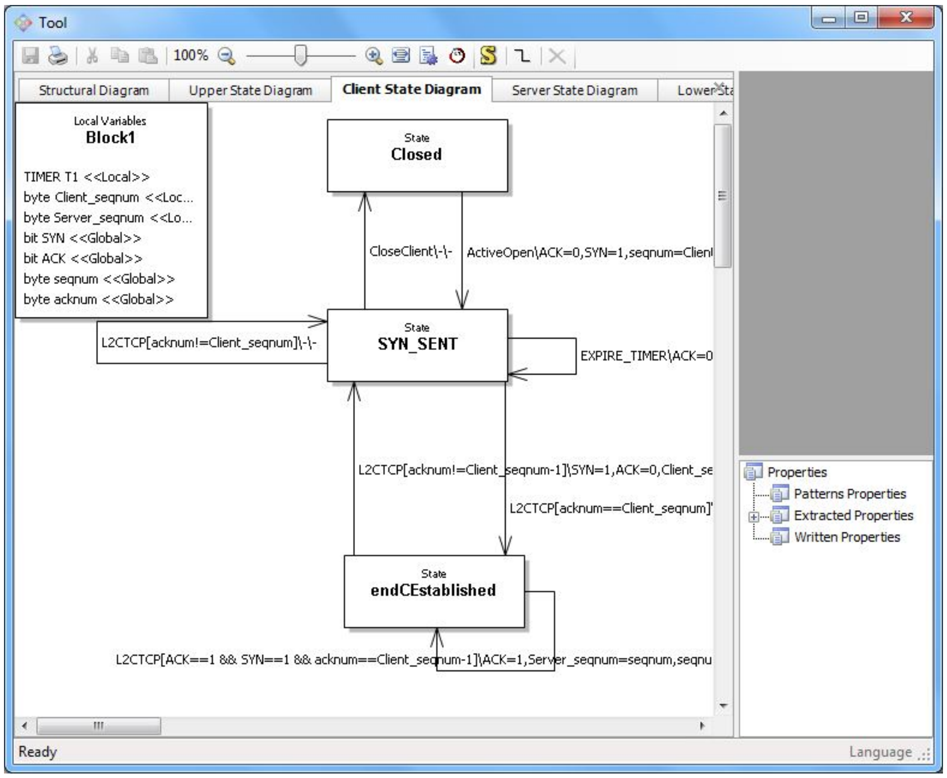


Figure 5.8 Behavioral Diagram of TCP Client

3. The Client should eventually receive ActiveOpen request, checking this property is important since it is required to start the three-way handshaking process. This property can be formalized using the global existence pattern (Section 3.3.1.1.3) and the LTL expression for this is : $<>(P)$ where P represents the reception of message ActiveOpen.

4. The Server should eventually receive PassiveOpen request. Same as previous property, checking this property is important since it is required to start the three-way handshaking.The LTL expression for this is : $<>(P)$ where P represents the reception of message PassiveOpen.

5. The connection should eventually be established. This means that Client and Server should eventually enter the ESTABLISHED state. The LTL expression for this is : $<>(P \wedge Q)$ where P represents Client in ESTABLISHED state and Q represents Server in the ESTABLISHED state.

The TCP model as built in Section 5.1.2 in free of deadlocks and cycles and all the model code is executed. Both property 4 and 5 are proven to be true. Property 6 (the establishment of the connection) is not proven to be true. SPIN generates an error trace which shows that the SYN message sent from the Client is lost at the lower layer repeatedly without any progress. One solution to solve this problem is to limit the number of trials of resending the messages. So, when a message is lost consecutively for more than a specific number , the sender gives up and the communication process stops.

Chapter 6

Conclusions and Future Work

This dissertation has addressed the problem of designing, verifying and debugging state based message passing systems (MPS). It is well recognized that the development of these types of applications is an error-prone process. The target system is modeled using an abstract communication blocks and hierarchical Designed for-Verification extended finite state machine (HDV-EFSM) that was developed in this research. A procedure to automatically generate formal models in a MC language is proposed. To simplify writing the required properties of the system, a new method based on choosing from a pre-determined set of patterns in concurrent communication properties is proposed. Once the model and the properties of the system are generated, a model checker is used to verify the correctness of the proposed system and ensure its compliance with specifications. In case of any inconsistencies found, the error source will be automatically localized using an algorithm that was developed for this purpose. A skeleton code of the system that includes the correct design specifications is generated in a general programming language i.e. Microsoft C#, Java, etc., moreover, the ability to debug the code using a conventional IDE while tracing the debugging process back to the original design was established. The framework was applied in the form of a software tool and successfully implemented in several case studies. The proposed framework and the developed software tool can be considered as a key solution for message passing systems design and verification. Using this research results, the techniques and design principles of the proposed framework contribute strongly toward a better development experience through providing an easy, cost effective, and comprehensive method that filled the existing gap between MC and IDE and will enable efficient and effective MPS design, verification and debugging than before.

6.1 Summary of Contributions

The main contributions of this research can be summarized in the following:

- A comprehensive framework to help in design, verify, implement and debug message passing systems.
- A multi-level graphical approach to design state-based message passing systems.
- A Hierarchical Designed-for-Verification extended finite state machine (HDV-EFSM) to assist in designing complex systems with too many states.
- A set of hierarchical event-based and time-based patterns, identified in this research

and gathered from literature, for concurrent properties is proposed along with a mapping between these patterns to linear temporal logic (LTL).

- A tool environment to support the proposed approach where SPIN model checker was used as a verification engine along with a formal approach for translating the design system to PROMELA language (the input of SPIN) and to a general programming language e.g., C# or Java.
- A procedure to integrate an IDE to the developed tool during the debugging mode and switch between the graphical system design and its code implementation.

6.2 Future Work

As the size and complexity of software systems increase, a need for tools to help in design and verify such systems increases. This research proposed a framework in this direction. Following are some ideas to extend this work.

- Incorporate other model checkers, such as Blast [11] and Chess [12] to the developed tool.
- Make any changes in the generated code reflected to the corresponding graphical design.
- Extend available Development Environments (IDEs), e.g., Visual Studio and Eclipse, by incorporating the developed tool as add-in.
- Apply the proposed framework to other case studies.

Bibliography

[1] M. Yabandeh, N. Kenzevic, D. Kostic, and V. Kuncak, “Predicting and Preventing Inconsistencies in Deployed Distributed Systems,” In *ACM Transactions on Computer Systems*, 28 (2010).

[2] P. Camara, M. Gallardo, P. Merino, and D. Sanan, “Checking the reliability of socket based communication software,” In *International Journal of Software Tools for Technology Transfer*, 11 (2009).

[3] A. Sobeih, M. Viswanatha, D. Marinov, and J. C. Hou, “Finding Bugs in Network Protocols Using Simulation Code and Protocol-Specific Heuristics,” In *Lecture Notes in computer Science*, (2005).

[4] D. Engler and M. Musuvathi, “Static analysis versus Software model checking in finding bugs,” In *Lecture Notes in computer Science*, (2004).

[5] W. Zhou, O. Sokolsky, B. Thau, and I. Lee, “DMaC: Distributed Monitoring and

Checking," In *Lecture Notes in computer Science*, (2009).

[6] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking* (MIT Press, Cambridge, MA, 1999).

[7] "Virtual debugger for stream applications," .

[8] E. M. Clarke, O. Grumberg, and D. Peled, *Distributed network systems: from concepts to implementation* (MIT Press, Cambridge, MA, 2004).

[9] K. S., "emantical Analysis of Modal Logic," Zeitschrift fÃijr Mathe Mtische Logik und Grundlagen der Mathematik **9,** 67–96 (1963).

[10] A. Bier, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," Advances in Computers 58 (2003).

[11] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST," International Journal of Software Tools Technology Transfer (2007).

[12] M. Musuvathi, S. Qadeer, and T. Ball, "CHESS: A Systematic Testing Tool for Concurrent Software," Microsoft Research Technical Report MSR-TR-2007-149 (2007).

[13] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual* (Addison-Wesley, 2004).

[14] M. CALDER and A. MILLER, "Analysing a basic call protocol using promela/xspin,"

In *Proceedings of Fourth SPIN Workshop*, (Paris, France, 1998).

[15] T. GARCIA-FANJUL and J. CORRALES, “Formal verification and simulation of the netbill protocol using spin,” In *Proceedings of Fourth SPIN Workshop*, (Paris, France, 1998).

[16] C. Braga and A. Sztajnberg, “Towards a rewriting semantics for a software architecture description language,” In *Electron. Notes Theor. Comput*, **95,** 148–168 (2003).

[17] C. T., “Modeling and Verification of a Multiprocessor Realtime OS Kernel,” In *In International Conference on Formal Description Techniques*, (1994).

[18] D. G. and J. J., “Modeling and verification of the RUBIS Kernel with Spin,” In *In International Spin Workshop*, (1995).

[19] P. R., P. D., T. K., and H. G., “Process sleep and wakeup on shared-memory multiprocessors,” In *EurOpen Conference*, (1991).

[20] T. P., T. J., M. J., L. J., C. A., and B. G., “Formal methods: A practical tool for OS implementors,” In *Hot Topics in Operating Systems*, (1997).

[21] K. Havelund, “Java PathFinder, "A Translator from Java to Promela",” Theoretical and Practical Aspects of SPIN Model Checking (1999).

[22] G. J. Holzmann, “Logic Verification of ANSI-C Code with Spin,” SPIN **1885,** 131–

147 (2000).

[23] G. J. Holzmann and M. Smith, “Software model checking. Extracting verification models from source code,” In *Proc. PSTV/FORTE99*, (Dordrecht, 1999).

[24] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, S. Robby, and H. Zheng, “Bandera: extracting finite state models from java source code,” In *Proceeding of the 22nd International Conference on Software Engineering*, ACM Press (London, 2000).

[25] T. SHI and X. HE, “A methodology for dependability and performability analysis in SAM,” In *DSN*, pp. 679–688 (2003).

[26] R. ALLEN and D. GARLAN, “A formal basis for architectural connection,” ACM Transactions Software Engineering Methodology 6 (1997).

[27] P. Inverardi, H. Muccini, and P. Pelliccione, “Automated check of architectural models consistency using spin,” In *In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pp. 346–349 (2001).

[28] P. Bose, “Automated translation of uml models of architectures for verification and simulation using spin,” In *In: Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pp. 102–109 (1999).

[29] J. Lilius and I. Paltor, “vUML: A Tool for Verifying UML Models,” In *Proceedings of the 14th IEEE international conference on Automated software engineering*, (1999).

[30] Y. Byun, “A Tool support for Design and Validation of Communication Protocol using State Transition Diagrams,” In *International Conference on information Technology*, (2007).

[31] M. Musuvathi, D. Park, A. Choou, D. Engler, and D. Dill, “CMC: a pragmatic approach to model checking real code,” In *In: Proceedings of the 5th symposium on Operating systems design and implementation*, ACM pp. 75–88 (New York, 2002).

[32] T. Ball, B. Cook, V. Levin, and S. Rajamani, “SLAM and static driver verifier: technology transfer of formal methods inside microsoft,” In *IFM*, (Springer, Heidelberg, 2004).

[33] W. Visser, K. Havelund, G. Brat, and S. Park, “Model checking programs,” In *Proceeding of the 15th International Conference on Automated Software Engineering*, pp. 3–12 (2000).

[34] P. Godefroid, “Model checking for programming languages using verisoft,” In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, (1997).

[35] G. Holzmann, “The logic of bugs,” In: FSE Foundations of Software Engineering. (2002).

[36] Y. S. Ramakrishna, P. MelliarSmith, L. E. Moser, L. K. Dillon, and G. Kutty, “Really

visual temporal reasoning," In *in 14th IEEE Real Time Systems Symposium*, pp. 262–273 (1993).

[37] W. Damm and D. Hare, "Lscs: Breathing life into message sequence charts," In *Formal Methods in System Design*, **19,** 45–80 (2001).

[38] M. H. Smith, G. J. mann, and K. Etessami, "Events and constraints: A graphical editor for capturing logic requirements of programs," In *in Fifth IEEE International Symposium on Requirements Engineering*, (2001).

[39] A. B. Bugerya, "Interactive Debugging of Parallel Programs: Distributed Scheme of Interacting Components," Programming and Computer Software **34,** 154–159 (2008).

[40] V. Samofalov, V. Kryukov, B. Kuhn, S. Zheltov, A. Konovalov, and J. DeSouza, "Automated correctness Analysis of MPI Programs with Intel Message Checker," In *Proc. Int. Conf. ParCo*, **33,** 901–907 (2005).

[41] V. F. Aleksakhin, K. Efimkin, V. Ilyakov, V. A. Kryukov, M. Kuleshova, and Y. L. Sazanov, "Tools for Debugging MPI Programs in the DVM System," In *Research Service on the Internet, Trudy Vserossiiskoi Nauchnoi Konferentsii*, pp. 113–115 (Moscow, 2005).

[42] "HP Visual Threads," http://www1.hp.com/products/software/visualthreads .

[43] "Intel Thread Checker," http://www.intel.com/cd/software/products/asmo-na/eng/

threading/286406.htm .

[44] "DVM System," http://www.keldysh.ru/dvm .

[45] "TotalView," http://www.etnus.com/TotalView .

[46] M. Kovalenko, "Interactive Debugging of MPI Programs Using the Parallel TDB Debugger," In *Research Service on the Internet, Trudy Vserossiiskoi Nauchnoi Konferentsii*, pp. 115–119 (Moscow, 2005).

[47] R. Hood, "The p2d2 Project: Building a Portable Distributed Debugger," In *Proc. of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pp. 127–136 (Philadelphia, 1996).

[48] "DDBG," http://doi.acm.org/10.1145/238020.238058 .

[49] "Allinea DDT," http://www.allinea.com .

[50] A. Al-Shabibi, S. Gerlach, R. Hersch, and B. Schaeli, "A Debugger for Flow Graph Based Parallel Applications," In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, (2007).

[51] B. Schaeli, A. Al-Shabibi, and R. Hersch, "Visual Debugging of MPI Applications," In *15th European PVM/MPI User Group Meeting (EuroPVM/MPI)*, (Ireland, 2008).

[52] Microsoft, "Microsoft Visual Studio 2010 Overview," .

[53] T. Stanley, T. Close, and M. S. Miller, “Causeway: A message-oriented distributed debugger,” Technical report, HPL-2009-78, HP Laboratories (2009).

[54] W. D. Pauw, M. LeÅčia, B. Gedik, H. Andrade, A. Frenkiel, M. Pfeifer, and D. Sow, “Visual Debugging for Stream Processing Applications,” Lecture Notes in Computer Science (2010).

[55] D. Rosenblum, “Formal methods and testing: Why the state-of-the-art is not the state-of-the-practice,” In *ACM SIGSOFT Software Engineering*, **21,** 7–15 (1996).

[56] D. Lee and M. Yannakakis, “Principles and methods of testing finite state machines-A survey,” In *In Proceedings of the IEEE*, pp. 1090–1123 (1996).

[57] J. Ellsberger, D. Hogrefe, and A. Sarma, *SDL: Formal Object-Oriented Language for Communication Systems* (Prentice-Hall, New York, 1997).

[58] C. A. Petri, “Kommunikation mit Automaten,” New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377 1 (1966).

[59] K. Jensen and L. M. Kristensen, *Coloured Petri Nets, Modeling and Validation of Concurrent Systems* (Springer Verlag Monograph, 2008).

[60] G. J. Holzmann, *Design and Validation of Computer Protocols* (Prentice Hall, 1991).

[61] R. Alur and M. Yannakakis, “Model checking of hierarchical state machines,” In *Pro-*

ceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, (1998).

[62] M. Dwyer, G. Avrunin, and J. Corbett, "Property specification patterns for finite-state verification," In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pp. 7–15 (1998).

[63] S. Konrad and B.Cheng, "Real-time Specification Patterns," In *Proceeding of the 27th International Conference of Software Engineering*, (2005).

[64] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, Reading MA, 1995).

[65] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in property specifications for finite-state verification," In *Proceedings of the 21st international conference on Software engineering*, (1999).

[66] B. D., D. D., H. L., and S. N., "Model checking SDL with Spin," In *Tools and Algorithms for Construction and Analysis of Systems*, (Springer, Germany, 2000).

[67] T. Kovse, B. Vlaovic, A. Vreze, and Z. Brezocnik, "Spin Trail to Message Sequence Chart Conversion Tool," In *The 10th International Conference on Telecommunications*, (2009).

[68] J. Postel, "Transmission Control Protocol. RFC 793," IETF (1981).

[69] J. Postel, “Requirements for Internet Host Communication Layers. RFC 1122,” IETF (1989).

[70] S. Floyd and T. Henderson, “The New Reno Modification to TCP’s Fast Recovery Algorithm. RFC 2582,” IETF (1999).

[71] M. Allman, V. Paxson, and W. Stevens, “TCP Congestion Control. Request for Comments 2581,” IETF (1999).

[72] V. Paxson, “Known TCP Implementation Problems. RFC 2525,” IETF (1999).

[73] V. Jacobson and R. Braden, “TCP Extensions for Long-Delay Paths. RFC 1072,” IETF (1988).

[74] V. Jacobson, R. Braden, and L. Zhang, “TCP Extension for High Speed Paths. RFC 1185,” IETF (1990).

[75] V. Jacobson, R. Braden, and D. Borman, “TCP Extensions for High Performance. RFC 1323,” IETF (1992).

[76] J. Kurose and K. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet* (Addison-Wesley, 2005).

Appendix A

PROMELA Code

```
#define BUFSIZE 0

#define TIMER bool

mtype = {ActiveOpen, CloseClient, PassiveOpen, CloseServer, C2LTCP, L2CTCP, S

chan Upper2Client = [BUFSIZE] of {mtype};
chan Upper2Server = [BUFSIZE] of {mtype};
chan Client2Lower = [BUFSIZE] of {mtype, bit, bit, byte, byte};
chan Lower2Client = [BUFSIZE] of {mtype, bit, bit, byte, byte};
chan Server2Lower = [BUFSIZE] of {mtype, bit, bit, byte, byte};
chan Lower2Server = [BUFSIZE] of {mtype, bit, bit, byte, byte};

active proctype Upper ()
{

        Start:
        end:

                        if
                                ::Upper2Client!ActiveOpen;
                                ::Upper2Server!PassiveOpen;
                                ::Upper2Client!CloseClient;
                                ::Upper2Server!CloseServer;
                        fi;
label_0:
label_1:
                                goto Start;


}

active proctype Client ()
{
        TIMER T1=0;
        byte Client_seqnum=0;
        byte Server_seqnum=0;
        bit SYN;
        bit ACK;
        byte seqnum;
        byte acknum;

        Closed:
        end:
                Upper2Client?ActiveOpen ->
label_2:
                                ACK=0;
                                SYN=1;
                                seqnum=Client_seqnum;
                                T1=1;
                                Client2Lower!C2LTCP(SYN,ACK,seqnum,acknum);
label_3:
                                goto SYN_SENT;

        SYN_SENT:
                if
                ::Upper2Client?CloseClient ->
label_4:
label_5:
                                goto Closed;

                ::T1->
label_6:
                                ACK=0;
                                SYN=1;
                                seqnum=Client_seqnum;
                                T1=0;
                                Client2Lower!C2LTCP(SYN,ACK,seqnum,acknum);
label_7:
                                goto SYN_SENT;


                ::Lower2Client?L2CTCP(SYN,ACK,seqnum,acknum) ->
                        if
                        :: acknum==Client_seqnum+1 && SYN==1 && ACK==1 ->
label_8:
                                ACK=1;
                                SYN=0;
                                Server_seqnum=seqnum;
                                Client_seqnum=Client_seqnum+1;
                                seqnum=Client_seqnum;
                                acknum=Server_seqnum+1;
                                Client2Lower!C2LTCP(SYN,ACK,seqnum,acknum);
label_9:
                                goto endCEstablished;

                        :: acknum!=Client_seqnum ->
label_10:
label_11:
                                goto SYN_SENT;

                        fi;

                fi;

        endCEstablished:
                Lower2Client?L2CTCP(SYN,ACK,seqnum,acknum) ->
                        if
                        :: ACK==1 && SYN==1 && acknum==Client_seqnum-1 ->
label_12:
                                ACK=1;
                                Server_seqnum=seqnum;
                                seqnum=Client_seqnum;
                                acknum=Server_seqnum+1;
                                Client2Lower!C2LTCP(SYN,ACK,seqnum,acknum);
label_13:
                                goto endCEstablished;

                        :: acknum!=Client_seqnum-1 ->
label_14:
                                SYN=1;
                                ACK=0;
                                Client_seqnum = Client_seqnum-1;
                                acknum=seqnum;
                                seqnum=Client_seqnum;
                                Client2Lower!C2LTCP(SYN,ACK,seqnum,acknum);
label_15:
                                goto SYN_SENT;

                        fi;

}

active proctype Server ()
{
        TIMER T2=0;
        byte Client_seqnum=0;
        byte Server_seqnum=0;
        bit SYN;
        bit ACK;
        byte seqnum;
        byte acknum;

        Closed:
        end:
                Upper2Server?PassiveOpen ->
label_16:
label_17:
                                goto Listen;


        Listen:
                Lower2Server?L2STCP(SYN,ACK,seqnum,acknum) ->
                        if
                        :: ACK==1 ->
label_18:
label_19:
                                goto Listen;

                        :: SYN==1 ->
label_20:
                                ACK=1;
                                Client_seqnum=seqnum;
                                seqnum=Server_seqnum;
                                acknum=Client_seqnum+1;
                                T2=1;
                                Server2Lower!S2LTCP(SYN,ACK,seqnum,acknum);
label_21:
                                goto SYN_RCVD;

                        fi;

        SYN_RCVD:
                if
                ::T2->
label_22:
                                SYN=1;
                                ACK=1;
                                seqnum=Server_seqnum;
                                acknum=Client_seqnum+1;
                                Server2Lower!S2LTCP(SYN,ACK,seqnum,acknum);
label_23:
                                goto SYN_RCVD;


                ::Lower2Server?L2STCP(SYN,ACK,seqnum,acknum) ->
                        if
                        :: SYN==1 && seqnum==Client_seqnum ->
label_24:
                                ACK=1;
                                seqnum=Server_seqnum;
                                acknum=Client_seqnum+1;
                                Server2Lower!S2LTCP(SYN,ACK,seqnum,acknum);
label_25:
                                goto SYN_RCVD;

                        :: SYN==1 && seqnum!=Client_seqnum ->
label_26:
                                ACK=1;
                                Client_seqnum=seqnum;
                                seqnum=Server_seqnum;
                                acknum=Client_seqnum+1;
                                Server2Lower!S2LTCP(SYN,ACK,seqnum,acknum);
label_27:
                                goto SYN_RCVD;

                        :: ACK==1 && acknum==Server_seqnum+1 ->
label_28:
label_29:
```

```
                                goto endSEstablished;

                        fi;

                fi;

        endSEstablished:
                skip;

}

active proctype Lower ()
{
        bit SYN;
        bit ACK;
        byte seqnum;
        byte acknum;

        Start:
        end:
                if
                ::Client2Lower?C2LTCP(SYN,ACK,seqnum,acknum) ->
label_30:
                        if
                                ::skip;
                                ::Lower2Server!L2STCP(SYN,ACK,seqnum,acknum);
                        fi;
label_31:
                                goto Start;


                ::Server2Lower?S2LTCP(SYN,ACK,seqnum,acknum) ->
label_32:
                        if
                                ::skip;
                                ::Lower2Client!L2CTCP(SYN,ACK,seqnum,acknum);
                        fi;
label_33:
                                goto Start;

                fi;
}
```

VITA

NAME OF AUTHOR: Rajaa Alqudah

PLACE OF BIRTH: Ayn Janna, Jordan

DATE OF BIRTH: August 6, 1980

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

Syracuse University, Syracuse, NY

Northeastern University, Boston, MA

Yarmouk University, Irbid, Jordan

DEGREES AWARDED:

Master of Science in Computer Engineering, 2007, Northeastern University

Bachelor of Science in Computer Engineering, 2003, Yarmouk University

PROFESSIONAL EXPERIENCE:

Teaching Assistant, Department of Electrical Engineering and Computer Science (EECS), Syracuse University, 2008 - 2011

Lab Engineer, Department of Computer Engineering, Yarmouk University, 2003-2005