Reconfigurable Embedded Control Systems:

Applications for Flexibility and Agility

Mohamed Khalgui
Xidian University, China

Hans-Michael Hanisch
Martin Luther University, Germany

Information Science REFERENCE

INFORMATION SCIENCE REFERENCE

Hershey · New York

Director of Editorial Content: Kristin Klinger
Director of Book Publications: Julia Mosemann
Acquisitions Editor: Lindsay Johnston
Development Editor: Joel Gamon
Publishing Assistant: Casey Conapitski
Typesetter: Milan Vracarich Jr.
Production Editor: Jamie Snavely
Cover Design: Lisa Tosheff

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Reconfigurable embedded control systems : applications for flexibility and agility / Mohamed Khalgui and Hans-Michael Hanisch, editors.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-60960-086-0 (hardcover) -- ISBN 978-1-60960-088-4 (ebook) 1. Programmable controllers. 2. Embedded computer systems. 3. Digital control systems. I. Khalgui, Mohamed. II. Hanisch, Hans-Michael.
TJ223.P76R43 2011
629.8'95--dc22

2010042276

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 13
Formal Analysis of Real-Time Systems

Osman Hasan
National University of Science and Technology (NUST), Pakistan

Sofiène Tahar
Concordia University, Canada

ABSTRACT

Real-time systems usually involve a subtle interaction of a number of distributed components and have a high degree of parallelism, which makes their performance analysis quite complex. Thus, traditional techniques, such as simulation, or state-based formal methods usually fail to produce reasonable results. The main limitation of these approaches may be overcome by conducting the performance analysis of real-time systems using higher-order-logic theorem proving. This chapter is mainly oriented towards this emerging trend and it provides the details about analyzing both functional and performance related properties of real-time systems using a higher-order-logic theorem prover (HOL). For illustration purposes, the Stop-and-Wait protocol, which is a classical example of real-time systems, has been considered as a case-study.

INTRODUCTION

Real-time systems can be characterized as systems for which the correctness of an operation is dependant not only on its functional correctness but also on the time taken. Some commonly used real-time system applications include embedded systems, digital circuits with uncertain delays, communication protocols and dynamic reconfigurable systems.

DOI: 10.4018/978-1-60960-086-0.ch013

Until the last decade, real-time systems were analyzed using traditional techniques, such as paper-and-pencil proof methods or simulation. The paper-and-pencil based proof techniques usually have some risk of an erroneous analysis due to the human-error factor. Similarly, accuracy of analysis cannot be guaranteed in computer simulation as well since the fundamental idea in this approach is to approximately answer a query by analyzing a large number of samples. These inaccuracy limitations of paper-and-pencil proof methods and simulation techniques may lead to

disastrous consequences in today's world, where real-time systems are extensively being used in safety critical and extremely sensitive applications such as medicine, military and transportation. In fact, some unfortunate incidents have already happened in this regard. One of the well-known incidents is the loss, in December 1999, of the Mars Polar Lander; a $165 million NASA spacecraft launched to survey Martian conditions. The Mars Polar Lander is believed to be lost mainly because of its engine shutdown while it was still 40 meters above the Mars surface. The engine shutdown happened due to the vibrations caused by the deployment of the Lander's legs, i.e., a probabilistic behavior that gave false indication that spacecraft had landed. Some other such incidents related to inaccurate or inadequate analysis of real-time systems include the loss of $125 million Mars Climate Orbiter in 1998 and the performance degradation of Microsoft's IIS indexing service DLL due to a buffer overflow problem caused by the "Code Red" worm in 2001, which resulted in a loss of over $2 billion to the company. A more recent incident is the faulty operation of the fly-by-wire primary flight control real-time software of a Boeing 777, operated by the Malaysia Airlines, in August 2005, which could have resulted in the loss of 177 passenger lives if the pilot had not manually taken over the autopilot program in time. All these incidents happened because the erroneous conditions were not caught during the analysis phase, due to the imprecise nature of the analysis techniques, and thus bugs appeared in the original product. Therefore, techniques like paper-and-pencil proof methods and simulation should not be relied upon for the analysis of real-time systems especially when they are used in safety or financial critical domains.

Formal methods are capable of conducting precise system analysis and thus overcome the above mentioned limitations. The main principle behind formal analysis of a system is to construct a computer based mathematical model of the given system and formally verify, within a computer, that this model meets rigorous specifications of intended behavior. A number of elegant approaches for the formal analysis of real-time systems can be found in the open literature using state-based or theorem proving techniques (e.g., Alur, 1992; Cardell-Oliver, 1992; Amnell, 2001; Beyer, 2003; Kwiatkowska, 2002; Bucci, 2005; Kwiatkowska, 2007; Hasan, 2009). However, some of these existing formal verification tools are only capable of specifying and verifying hard deadlines, i.e., properties where a late response is considered to be incorrect. Whereas, in the case of performance analysis of real-time systems, soft deadlines, i.e., properties that provide the quality of service in terms of probabilistic quantities or averages, play a vital role. Also, the above mentioned state-based approaches are limited by reduced expressive power of their automata based or Petri net based specification formalism. On the other hand, the higher-order-logic theorem proving based technique [Hasan, 2009] tends to overcome the above mentioned limitations of existing formal real-time system analysis techniques.

The main principle behind the higher-order logic theorem proving based approach is to leverage upon the high expressiveness of higher-order logic to formally specify and reason about the temporal properties and random behaviors of the present age complex real-time systems. This approach is primarily based upon the previous work reported for the functional verification of hard real-time systems [Cardell-Oliver, 1992], the formalization of random variables [Hurd, 2002] and the verification of expectation properties for discrete random variables [Hasan, 2008]. The idea is to formally specify the given real-time system as a logical conjunction of higher-order-logic predicates [Cadell-Oliver, 1992], whereas each one of these predicates defines an autonomous component or process of the given real-time system, while representing the unpredictable or random elements in the system as formalized random variables [Hurd, 2002]. The functional correctness and the performance related proper-

ties for various parameters for this formal model can then be verified using an interactive theorem prover with the help of the useful theorems already proved in [Cadell-Oliver, 1992; Hurd, 2002; Hasan, 2008]. Since the analysis is conducted within the core of a mechanical theorem prover, there would be no question about the soundness or the precision of the results. Also, there is no equivalence verification required between the models used for functional verification and performance evaluation as the same formal model is used for both of these analyses in this approach.

The main focus of this chapter is on this theorem proving based real-time system analysis approach. In order to illustrate the utilization and practical effectiveness of the presented approach, the chapter includes the functional verification and performance analysis of a variant of the Stop-and-Wait protocol [Widjaja, 2004], which is a classical example of a real-time system. The Stop-and-Wait protocol utilizes the principles of error detection and retransmission and is a fundamental mechanism for reliable communication between computers. Indeed, it is one of the most important parts of the Internet's Transmission Control Protocol (TCP). The main motivation behind selecting the Stop-and-Wait protocol as a case study is its widespread popularity in the literature regarding real-time system analysis methodologies. The Stop-and-Wait protocol and some of its closely related variants have been checked formally for functional verification using theorem proving [Cadell-Oliver, 1992], state-based formal approaches (e.g. [Gallasch, 2006]) and a combination of both techniques (e.g. [Havelund, 1996]) and their performance has been analyzed using a number of innovative state-based formal or semi-formal techniques (e.g. [Wells, 2002]). In all of these previous works, only one aspect, i.e., either functional correctness or performance was analyzed. However, this chapter utilizes a single formal model of the Stop-and-Wait protocol and presents the analysis of its functional correctness and performance by leveraging upon the expressiveness of higher-order logic.

The chapter is organized as follows. Section 2 provides some preliminaries including an overview of higher-order logic theorem proving and Stop-and-Wait protocol. The higher-order-logic theorem proving based technique for the analysis of real-time systems is outlined in Section 3. Next in Section 4, we present a higher-order-logic specification of the Stop-and-Wait protocol and formally verify its functional and performance related properties using a theorem prover in Sections 5 and 6, respectively. Finally, Section 7 will conclude the chapter.

PRELIMINARIES

In this section, we provide an overview of higher-order-logic theorem proving and the HOL theorem prover that will be used in the rest of this chapter. The intent is to provide a brief introduction to these topics along with some notation that is going to be used later.

Higher-Order-Logic Theorem Proving

Theorem proving [Gordon, 1989] is a widely used formal verification technique. The system that needs to be analyzed is mathematically modeled in an appropriate logic and the properties of interest are verified using computer based formal tools. The use of formal logics as a modeling medium makes theorem proving a very flexible verification technique as it is possible to formally verify any system that can be described mathematically. The core of theorem provers usually consists of some well-known axioms and primitive inference rules. Soundness is assured as every new theorem must be created from these basic axioms and primitive inference rules or any other already proved theorems or inference rules.

The verification effort of a theorem varies from trivial to complex depending on the under-

lying logic [Harrison, 2009]. For instance, first-order logic is restricted to propositional calculus and terms (constants, function names and free variables) and is semi-decidable. A number of sound and complete first-order logic automated reasoners are available that enable completely automated proofs. More expressive logics, such as higher-order logic, can be used to model a wider range of problems than first-order logic, but theorem proving for these logics cannot be fully automated and thus involves user interaction to guide the proof tools. For performance and probabilistic analysis, we need to formalize (mathematically model) random variables as functions and formalize characteristics of random variables, such as probability distribution functions and expectation, etc., by quantifying over random variable functions. Henceforth, first-order logic does not support such formalization and we need to use higher-order logic to formalize probabilistic analysis.

HOL Theorem Prover

In this chapter, we use the HOL theorem prover [Gordon, 1993] to conduct all the real-time system performance analysis related formalization and verification. HOL is an interactive theorem prover developed by Mike Gordon at the University of Cambridge for conducting proofs in higher-order logic. It utilizes the simple type theory of Church [Church, 1940] along with Hindley-Milner polymorphism [Milner, 1977] to implement higher-order logic. HOL has been successfully used as a verification framework for both software and hardware as well as a platform for the formalization of pure mathematics.

Secure Theorem Proving

In order to ensure secure theorem proving, the logic in the HOL system is represented in the strongly-typed functional programming language ML [Paulson, 1996]. An ML abstract data type is used to represent higher-order-logic theorems and the only way to interact with the theorem prover is by executing ML procedures that operate on values of these data types. The HOL core consists of only 5 basic axioms and 8 primitive inference rules, which are implemented as ML functions. Soundness is assured as every new theorem must be verified by applying these basic axioms and primitive inference rules or any other previously verified theorems/inference rules.

Terms

There are four types of HOL terms: constants, variables, function applications, and lambda-terms (denoted function abstractions). Polymorphism, types containing type variables, is a special feature of higher-order logic and is thus supported by HOL. Semantically, types denote sets and terms denote members of these sets. Formulas, sequences, axioms, and theorems are represented by using terms of Boolean types.

Theories

A HOL theory is a collection of valid HOL types, constants, axioms and theorems and is usually stored as a file in computers. Users can reload a HOL theory in the HOL system and utilize the corresponding definitions and theorems right away. The concept of HOL theory allows us to build upon existing results in an efficient way without going through the tedious process of regenerating these results using the basic axioms and primitive inference rules.

HOL theories are organized in a hierarchical fashion. Any theory may inherit types, definitions and theorems from other theories. Various mathematical concepts have been formalized and saved as HOL theories by the users. These theories are available to a user when he/she first starts a HOL session. The HOL theories of Booleans, lists, sets, positive integers, real numbers, measure and probability are some of the frequently used theories in

Figure 1. HOL symbols and functions

HOL Symbol	Standard Symbol	Meaning
$\wedge$	*and*	Logical *and*
$\vee$	*or*	Logical *or*
$\neg$	*not*	Logical *negation*
::	*cons*	Adds a new element to a list
++	*append*	Joins two lists together
`hd L`	*head*	Head element of list L
`tl L`	*tail*	Tail of list L
`el n L`	*element*	n^{th} element of list L
`mem a L`	*member*	True if a is a member of list L
`length L`	*length*	Length of list L
(a, b)	a x b	A pair of two elements
`fst`	fst (a, b) = a	First component of a pair
`snd`	snd (a, b) = b	Second component of a pair
λx.t	$\lambda x.t$	Function that maps x to $t(x)$
{x|P(x)}	$\{\lambda x.P(x)\}$	Set of all x such that $P(x)$
`num`	$\{0, 1, 2, \ldots\}$	Positive Integers data type
`real`	All Real numbers	Real data type
`suc n`	$n + 1$	Successor of a *num*

analyzing the performance of real-time systems. In fact, one of the primary motivations of selecting the HOL theorem prover for this work was to benefit from these built-in mathematical theories.

HOL Symbols

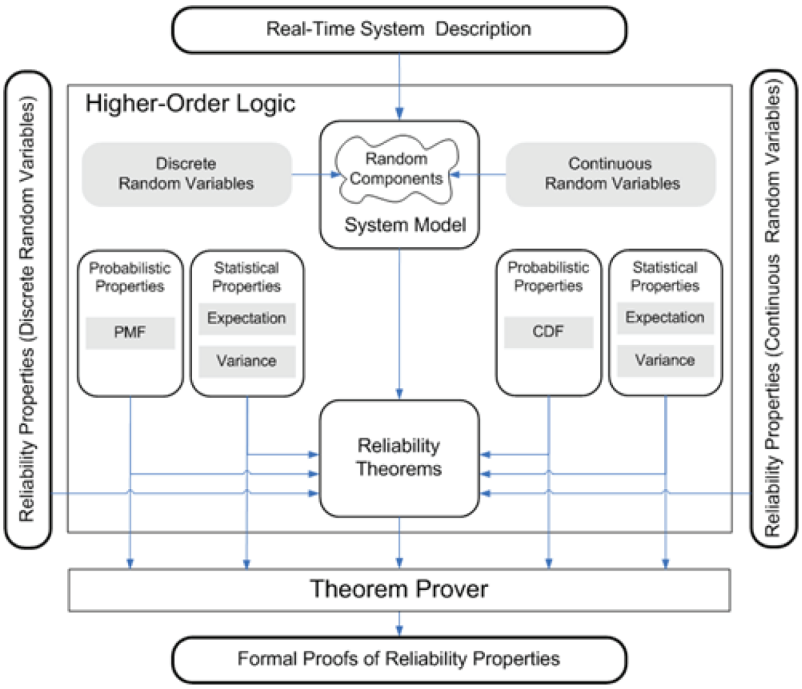
Figure 1 provides the mathematical interpretations of some frequently used HOL symbols and functions in this chapter.

PROBABILISTIC THEOREM PROVING BASED METHODOLOGY

A real-time system and its environment may be viewed as a bunch of concurrent, communicating processes that are autonomous, i.e., they can communicate asynchronously. The behavior of these processes over time may be specified by higher-order-logic predicates on positive integers [Cadell-Oliver, 1992]. These positive integers represent the ticks of a clock counting physical time in any appropriate units, e.g., nanoseconds. The granularity of the clock's tick is believed to be chosen in such a way that it is sufficiently fine to detect properties of interest. The behavior of a real-time system can now be formally specified by combining the corresponding process specifications (higher-order-logic predicates) using logical conjunction. In a similar way, additional constraints for the real-time system such as initial conditions or any assumptions, if required to ensure the correct behavior of the model, can also be defined as predicates and combined with its formal specification using logical conjunctions.

The performance analysis of real-time systems is primarily based on probability theory concepts. A hypothetical model of a theorem proving based real-time system performance analysis framework is given in Figure 2, with some of its most fundamental components depicted with shaded boxes. Like all traditional analysis problems, the starting point of performance analysis is also a system description and some intended system properties and the goal is to check if the given system satisfies these properties. For simplicity, we have divided system properties into two categories, i.e., system properties related to discrete

Figure 2. Theorem proving based real-time system performance analysis framework



random variables and system properties related to continuous random variables.

The first step in the methodology illustrated in Figure 2 is to construct a model of the given real-time system in higher-order logic. For this purpose, we model the real-time system as a logical conjunction of processes as illustrated above while modeling the random components of the system by random variables. The foremost requirement for this step is the availability of infrastructures that allow us to formalize all kinds of discrete and continuous random variables as higher-order-logic functions, which in turn can be used to represent the random components of the given system in its higher-order-logic model. The second step is to utilize the formal model of the system to express system properties as higher-order-logic theorems. The prerequisite for this step is the ability to express probabilistic and statistical properties related to both discrete and continuous random variables in higher-order-logic. All probabilistic properties of discrete and continuous random variables can be expressed in terms of their *Probability Mass Functions* (PMFs) and *Cumulative Distribution Functions* (CDFs), respectively. Similarly, most of the commonly used statistical properties can be expressed in terms of the expectation and variance characteristics of the corresponding random variable. Thus, we require the formalization of mathematical definitions of PMF, CDF, expectation and variance for both discrete and continuous random variables in order to be able to express the given system's performance characteristics as higher-order-logic theorems. The third and the final step for conducting performance analysis of a real-time system in a theorem prover is to formally verify the higher-order-logic theorems developed in the previous step using a theorem prover. For this verification, it would be quite handy to have access to a library of some pre-verified theorems corresponding to some commonly used properties regarding probability distribution functions, expectation and variance. Since, we can build upon

such a library of theorems and thus speed up the verification process. The formalization details regarding the above mentioned steps are briefly described now.

Discrete Random Variables and the PMF

A random variable is called discrete if its range, i.e., the set of values that it can attain, is finite or at most countably infinite. Discrete random variables can be completely characterized by their PMFs that return the probability that a random variable X is equal to some value x, i.e., $Pr(X = x)$. Discrete random variables are quite frequently used to model randomness in performance analysis. For example, the Bernoulli random variable is widely used to model the fault occurrence in a component and the Binomial random variable may be used to represent the number of faulty components in a lot.

Discrete random variables can be formalized in higher-order-logic as deterministic functions with access to an infinite Boolean sequence B^{∞}; an infinite source of random bits with data type (*natural* → *bool*) [Hurd, 2002]. These deterministic functions make random choices based on the result of popping the top most bit in the infinite Boolean sequence and may pop as many random bits as they need for their computation. When the functions terminate, they return the result along with the remaining portion of the infinite Boolean sequence to be used by other functions. Thus, a random variable that takes a parameter of type α and ranges over values of type β can be represented by the function

$$F: \alpha \rightarrow B^{\infty} \rightarrow \beta \times B^{\infty}$$

For example, a Bernoulli(½) random variable that returns 1 or 0 with probability ½ can be modeled as

$$bit = \lambda s.\ (\text{if } shd\ s \text{ then } 1 \text{ else } 0,\ stl\ s)$$

where the variable *s* represents the infinite Boolean sequence and the functions *shd* and *stl* are the sequence equivalents of the list operations *'head'* and *'tail'*. A function of the form $\lambda x.\ t(x)$ represents a lambda abstraction function that maps x to $t(x)$. The function *bit* accepts the infinite Boolean sequence and returns a pair with the first element equal to either 0 or 1 and the second element equal to the unused portion of the infinite Boolean sequence.

The higher-order-logic formalization of the probability theory [Hurd, 2002] also consists of a probability function P from sets of infinite Boolean sequences to *real* numbers between 0 and 1. The domain of P is the set E of events of the probability. Both P and E are defined using the Caratheodory's Extension theorem, which ensures that E is a σ-algebra: closed under complements and countable unions. The formalized P and E can be used to formally verify all the basic axioms of probability. Similarly, they can also be used to prove probabilistic properties for random variables. For example, we can formally verify the following probabilistic property for the function *bit*, defined above,

$$P\ \{s \mid fst\ (bit\ s) = 1\} = ½$$

where $\{x|C(x)\}$ represents a set of all elements x that satisfy the condition C, and the function *fst* selects the first component of a pair.

The above mentioned infrastructure can be utilized to formalize most of the commonly used discrete random variables and verify their corresponding PMF relations. In this chapter, we will utilize the models for Bernoulli and Geometric random variables formalized as higher-order-logic functions *prob_bernoulli* and *prob_geom* and verified using the following PMF relations in [Hurd, 2002] and [Hasan, 2008], respectively.

Theorem 1:

$$\forall p.\ 0 \leq p \wedge p \leq 1 \Rightarrow (P\ \{s \mid fst\ (prob\_bernoulli\ p\ s)\} = p)$$

Theorem 2:
$\forall$ n p. $0 \leq p \wedge p \leq 1 \Rightarrow$ (P {s | fst (prob_geom p s) = (n + 1)} = p(1 - p)n)

The Geometric random variable returns the number of Bernoulli trials needed to get one success and thus cannot return 0. This is why we have *(n+1)* in Theorem 2, where *n* is a positive integer {0,1,2,3 …}. Similarly, the probability *p* in Theorem 2 represents the probability of success and thus needs to be greater than 0 for this theorem to be true as has been specified in the precondition.

Continuous Random Variables and the CDF

A random variable is called continuous if it ranges over a continuous set of numbers that contains all real numbers between two limits. Continuous random variables can be completely characterized by their CDFs that return the probability that a random variable X is exactly less than or equal to some value *x*, i.e., $Pr(X \leq x)$. Examples of continuous random variables include measuring *T*, the arrival time of a data packet at a web server $(S_T = \{t \mid 0 \leq t < \infty\})$ and measuring *V*, the voltage across a resistor $(S_V = \{ v \mid -\infty < v < \infty \})$.

The sampling algorithms for continuous random variables are non-terminating and hence require a different formalization approach than discrete random variables, for which the sampling algorithms are either guaranteed to terminate or satisfy probabilistic termination, meaning that the probability that the algorithm terminates is 1. One approach to address this issue is to utilize the concept of the nonuniform random number generation, which is the process of obtaining arbitrary continuous random numbers using a Standard Uniform random number generator. The main advantage of this approach is that we only need to formalize the Standard Uniform random variable from scratch and use it to model other continuous random variables by formalizing the corresponding nonuniform random number generation method.

Based on the above approach, a methodology for the formalization of all continuous random variables for which the inverse of the CDF can be represented in a closed mathematical form is presented in [Hasan, 2007}. The first step in this methodology is the formalization of the Standard Uniform random variable, which can be done by using the formalization approach for discrete random variables and the formalization of the mathematical concept of limit of a *real* sequence [Harrison, 1998]. The formalization details are outlined in [Hasan, 2007].

The second step in the methodology for the formalization of continuous probability distributions is the formalization of the CDF and the verification of its classical properties. This is followed by the formal specification of the mathematical concept of the inverse function of a CDF. This definition along with the formalization of the Standard Uniform random variable and the CDF properties, can be used to formally verify the correctness of the Inverse Transform Method (ITM). The ITM is a well known nonuniform random generation technique for generating nonuniform random variables for continuous probability distributions for which the inverse of the CDF can be represented in a closed mathematical form. Formally, it can be verified for a random variable *X* with CDF *F* using the Standard Uniform random variable *U* as follows

$$\Pr (F^{-1}(U) \leq x) = F(x)$$

The formalized Standard Uniform random variable can now be used to formally specify any continuous random variable for which the inverse of the CDF can be expressed in a closed mathematical form as $X=F^{-1}(U)$. Whereas, the CDF of this formally specified continuous random variable, *X*, can be verified using simple arithmetic reasoning and the formal proof of the ITM. Based on this approach, Exponential, Uniform, Rayleigh and

Triangular random variables have been formalized and their CDF relations have been verified [Hasan, 2007].

Statistical Properties for Discrete Random Variables

In probabilistic analysis, statistical characteristics play a major role in decision making as they tend to summarize the probability distribution characteristics of a random variable in a single number. Due to their widespread interest, the computation of statistical characteristics has now become one of the core components of every contemporary probabilistic analysis framework.

The expectation for a function of a discrete random variable, which attains values in the positive integers only, is formally defined as follows.

$$\forall X.\ expec\ X = suminf\ (\lambda n.\ n\ P\ \{s \mid fst\ (X\ s) = n\})$$

where the mathematical notions of the probability function *P* and random variable *X* have been inherited from [Hurd, 2002], as presented in the previous section. The function *suminf* represents the HOL formalization of the infinite summation of a *real* sequence [Harrison, 1998]. The function *expec* accepts the random variable *X* with data type $B^{\infty} \rightarrow natural \times B^{\infty}$ and returns a *real* number. The above definition can be used to verify the average values of most of the commonly used discrete random variables [Hasan, 2008]. For example, the average value of the Geometric random variable can be verified as the following theorem.

Theorem 3:

$\forall\ p.\ 0 \leq p \wedge p \leq 1 \Rightarrow (expec\ (\lambda s.\ prob\_geom\ p\ s) = 1/p)$

In order to verify the correctness of the formal definition of expectation and facilitate reasoning about expectation properties in probabilistic systems, many widely used expectation properties have been formally verified in the HOL theorem prover [Hasan, 2009a]. Namely being the linearity of expectation, Markov and Chebyshev's inequalities, variance and linearity of variance. In this chapter, we utilize the following linearity property out of this rich library of formally verified expectation properties.

Theorem 4:

$\forall\ a\ b\ X.\ expec\ (\lambda s.\ (a\ (fst\ (X\ s) + b, snd\ (X\ s))) = a((expec\ X) + b)$

Statistical Properties for Continuous Random Variables

The expectation of a continuous random variable has been formally defined in [Hasan, 2009b] using the Lebesgue integral, which has strong relationship with the measure theory fundamentals [Galambos, 1995]. This definition is general enough to cater for both discrete and continuous random variables and is thus far more superior than the commonly used Rieman integral based definition that is only applicable to continuous random variables with well-defined PDF. Though, the main limitation of the Lebesgue integral based definition is the complex reasoning process involved for verifying expectation properties. This limitation has been tackled in [Hasan, 2009b] and the main idea is to verify two relatively simplified expressions for expectation by building on top of the Lebesgue integral based definition. The first expression is for the case when the given continuous random variable is bounded in the positive interval *[a,b]* and the second one is for an unbounded random variable. Both of these expressions are verified using the fundamentals from measure and Lebesgue integral theories but once verified, they can be utilized to verify expectation properties of any continuous random variable without applying these complex underlying concepts. The usage of these expressions is illustrated by verifying the expected values of Uniform, Triangular and Exponential random variables [Hasan, 2009b].

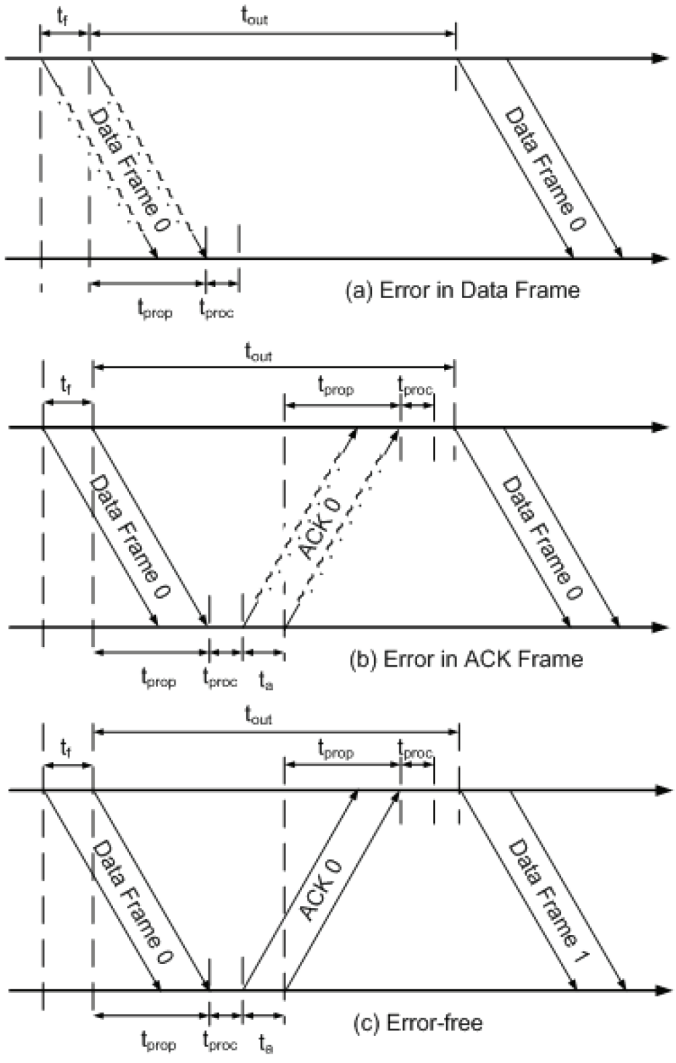
STOP-AND-WAIT PROTOCOL

This section provides a brief introduction to the Stop-and-Wait protocol [Widjaja, 2004], which will be used as case study for the formal analysis framework presented in the previous section. The Stop-and-Wait is a basic Automatic Repeat Request (ARQ) protocol that ensures reliable data transfers across noisy channels. In a Stop-and-Wait system, both sending and receiving stations have error detection capabilities. The operation is illustrated in Figure 3 using the following notation.

- t_f: Data message transmission time
- t_a: ACK message transmission time
- t_{prop}: One-way signal propagation delay between transmitter and receiver
- t_{proc}: Processing time required for error detection in the received message at both transmitter and receiver ends
- t_{out}: Timeout period

The transmitter sends a data message to the receiver and spends t_f time units in doing so. Then, it stops and waits to receive an acknowl-

Figure 3. Stop-and-Wait operation



edgement (ACK) of reception of that message from the receiver. If no ACK is received within a given time out, t_{out}, period, the data message is resent by the transmitter and once again it stops and starts waiting for the ACK (Figure 3.a). If an ACK is received within the given t_{out} period then the transmitter checks the received message for errors during the next t_{proc} time units. If errors are detected then the ACK is ignored and the data message is resent by the transmitter after t_{out} expires and once again the transmitter stops and waits for the ACK (Figure 3.b). Thus, the main idea is that the transmitter keeps on retransmitting the same data message, after a pre-defined time-out period, t_{out}, until and unless it receives a corresponding error-free ACK message from the receiver. When an error-free ACK message is finally received then the transmitter transmits the next data message in its queue (Figure 3.c).

The receiver is always waiting to receive data messages. When a new message arrives, the receiver checks it for errors during the next t_{proc} time units. If errors are detected then the data message is ignored and the receiver continues to be in the wait state (Figure 3.a), otherwise it initiates the transmission of an ACK message, which takes t_a time units (Figure 3.b,c).

Under the above mentioned conditions, the ACK message cannot be received before $t_{prop} + t_{proc} + t_a + t_{prop} + t_{proc}$ units of time after sending out a data message. It is, therefore, necessary to set $t_{out} \geq 2(t_{prop} + t_{proc}) + t_a$, i.e., the retransmission must not be allowed to start before the expected arrival time of the ACK is lapsed, for reliable communication between transmitter and receiver.

ARQ allows the transmitting station to transmit a specific number, usually termed as *sending window*, of messages before receiving an ACK frame and the receiving station to receive and store a specific number, usually termed as *receiving window*, of error-free messages even if they arrive out-of-sequence. Generally, both the *sending window* and the *receiving window* are assigned the same value, which is termed as the *window size* of the ARQ protocol. The *window size* for the Stop-and-Wait protocol is 1, as can be observed from its transmitter and receiver behavior descriptions given above.

In order to distinguish between new messages and duplicates of previous messages at the receiver or transmitter, a sequence number is included in the header of both data and ACK messages. It has been shown that, for correct ARQ operation, the number of distinct sequence numbers must be at least equal to twice the *window size*. Thus, the simplest and the most commonly used version of the Stop-and-Wait protocol uses two distinct sequence numbers (0 and 1) and is known as the Alternating Bit Protocol (ABP). The transmitter keeps track of the sequence number of the last data message it had sent, its associated timer and the message itself in case a retransmission is required. Whereas, the receiver keeps track of the sequence number of the next data message that it is expecting to receive. Thus, if an out-of-sequence data message arrives at the receiver, it ignores it and responds with the ACK for the data message that it is expecting to receive. On the other hand, when an in-sequence data message arrives at the receiver, it updates its sequence number by performing a modulo-2 addition with the number 1, i.e., 0 is updated to 1 and 1 is updated to 0. Similarly, if an out-of-sequence ACK message appears at the transmitter, it ignores it and retains the sequence number of the last data message it had sent. Whereas, in the case of the reception of an in-sequence ACK message, the sequence number at the transmitter is also updated by performing a modulo-2 addition by 1, which becomes the sequence number of the next data message as well. More details about sequence numbering in the Stop-and-Wait protocol can be found in [Widjaja, 2004].

The most widely used performance metric for the Stop-and-Wait protocol is the time required for the transmitter to send a single data message and know that it has been successfully received at the receiver. In the case of error-free or noiseless

channels, which do not reorder or loose messages (Figure 3.c), the message transmission delay is given by

$$t_f + t_{prop} + t_{proc} + t_a + t_{prop} + t_{proc} \quad (1)$$

On the other hand, in the presence of noise, every damaged or lost message (data or ACK) will cause a retransmission from the transmitter and thus wastes $t_f + t_{out}$ units of time (Figure 3a,b). Whereas, the final successful transmission will take the amount of time equal to the one given by Equation (1). In order to obtain more concise information about this delay, we consider the probability, p, of a message transmission being in error. This allows us to model the number of retransmissions in the Stop-and-Wait protocol in terms of a Geometric random variable, which returns the number of trials required to achieve the first success, with success probability *1-p*. Therefore, the delay of the Stop-and-Wait protocol can be mathematically expressed as

$$(t_f + t_{out})\,(G_{(1-p)}\text{-}1) + t_f + t_{prop} + t_{proc} + t_a + t_{prop} + t_{proc} \quad (2)$$

where G_x denotes a Geometric random variable with success probability x. The above representation allows us to express the average delay of a single data message in a Stop-and-Wait protocol using the average or mean value of Geometric random variables as follows

$$(t_f + t_{out})p/(1\text{-}p) + t_f + t_{prop} + t_{proc} + t_a + t_{prop} + t_{proc} \quad (3)$$

The main scope of the rest of the chapter is to formally specify the Stop-and-wait protocol, described in this section, as a real-time system and mechanically verify its functional correctness and average message delay relation, given in Equation (3), using the methodology described in the previous section.

FORMAL SPECIFICATION OF THE STOP-AND-WAIT PROTOCOL

Based on the formal probabilistic analysis methodology presented earlier, we formally specify the Stop-and-Wait protocol described in the previous section as a combination of six processes, as shown in Figure 2. The protocol mainly consists of three major modules, i.e., the sender or the transmitter, the receiver and the communication channel. Each one of these modules can be subdivided into two processes as both the sender and the receiver transmit messages and receive them and the channel between the sender and receiver consists of two logical channels: one carrying data messages from the sender to the receiver and one carrying ACK messages in the opposite direction.

Next we present the data type definitions of the six predicates, corresponding to each one of the processes in Figure 4, and finally the formal specification of the Stop-and-Wait protocol, which also includes the predicates for assumptions and initial conditions. We include the timing information associated with every action in these predicates so that the corresponding model can be utilized to reason about the message delay characteristic of the Stop-and-Wait protocol.

Type Definitions

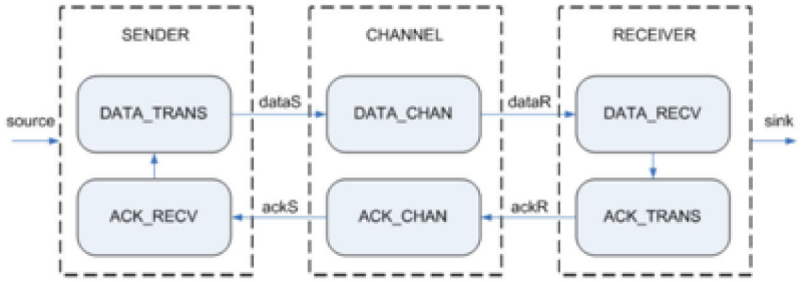
The input to the Stop-and-Wait protocol, *source*, is basically a list of data messages that can be modeled in HOL by a list of **data* elements

```
source: *data list
```

where **data list* represents any concrete HOL data type such as a record, a character, an integer or an n-bit word. The output of the protocol, *sink*, is also a list of data messages that grows with time as new data messages are delivered to the receiver. It can be modeled in HOL as follows

```
sink: time → *data list
```

Figure 4. Logical structure of an ARQ protocol



where *time* is assigned the HOL data-type for natural number and represents physical time in this case. This kind of variable, which is time dependant, is termed as a *history* in this chapter. The arrows in Figure 4 between processes represent information that is shared between the sender, channel and receiver. Data messages are transmitted from the sender to the receiver (*dataS, dataR*) and ACK messages are transmitted in the opposite direction (*ackR, ackS*). These messages are transmitted across the Stop-and-Wait protocol in a form of a packet, which can be modeled in HOL as a pair containing a sequence number and a message element

```
packet: natural x *data
```

where a *natural number* is used here for the sequence number and the **data* represents the message. Since we are dealing with an unreliable channel, the output of a channel may or may not be a packet. In order to model the no-packet case in HOL, a data-type *non_packet* is defined, which has only one value, i.e., *one*. Every message can either be of type *packet* or of type *non_packet*.

```
message: packet + non_packet
```

Data Transmission

The process *DATA_TRANS* in Figure 4 characterizes the data transmission behavior of the Stop-and-Wait protocol and the corresponding predicate is defined as follows in Box 1.

The variables *ws* and *sn* represent the *window size* and the number of distinct sequence numbers available for the protocol, respectively. By using these variables in our definitions, instead of their corresponding fixed values of 1 and 2 for the case of the Stop-and-Wait protocol, we attain two benefits. Firstly, it makes our definitions more generic as they can now be used, with minor updates, to formally model the corresponding processes of other ARQ protocols, such as Go-Back-N and Selective-Repeat [Garcia, 2004], as well. Secondly, this allows us to establish a logical implication between our definitions for the six processes (Figure 4) to the corresponding definitions for the Sliding Window protocol, given in [Cadell-Oliver, 1992]. This relationship can be used to inherit the functional correctness theorem, verified for the Sliding Window protocol in [Cadell-Oliver, 1992], for our Stop-and-Wait protocol model and thus saves us a considerable amount of verification time and effort. More details on this are given in the next section. It is important to note that in order to model the correct behavior for the Stop-and-Wait protocol; we will assign the values of 1 and 2 to the variables *ws* and *sn*, respectively, in an assumption that is used in all of the theorems that we verify for the Stop-and-Wait protocol.

The history *datas* represents the data messages transmitted by the sender at any particular time.

Box 1.

```
∀ ws sn dataS s rem i ackS tout tf dtout dtf.
     DATA_TRANS_STOP_WAIT ws sn dataS s rem i ackS tout tf dtout dtf =
       ∀ t. (if ~NULL (tli (i t) (rem t)) ∧ i t < ws then
         (if dtf t = 0 then
           (i (t + 1) = i t + 1) ∧ (dtout (t + 1) = tout - 1) ∧
           (dtf (t + 1) = tf) ∧
           (dataS t =
             new_packet (mod_n_add (s t) (i t) sn) (hdi (i t) (rem t)))
           else
             (i (t + 1) = i t) ∧ (dtout (t + 1) = tout) ∧
             (dtf (t + 1) = dtf (t - 1) ∧ (dataS t = set_non_packet))
       else
         (dtf (t + 1) = tf) ∧ (dataS t = set_non_packet)) ∧
         (if (dtout t = 1) ∨
           good_packet (ackS t) ∧
           mod_n_sub (label (ackS t)) (s t) sn < ws
         then
           (i (t + 1) = i t - 1) ∧ (dtout (t + 1) = tout)
         else
           (i (t + 1) = i t) ∧ (dtout (t + 1) = dtout t - 1))
```

The history *s* represents, modulo *sn*, the sequence number of the first unacknowledged data message. Data remaining to be sent at any time is represented by the history *rem* that has type *time*→**data list*. whereas, the history *i*: *time*→*natural* is used to identify the number of data messages, at any particular time, that have been transmitted by the sender but are still unacknowledged by the receiver. the history *acks* represents the ack messages received by the sender at any particular time. the variables *tout* and *tf* hold the values for the t_{out} and t_f delays, respectively, defined in section 2, and histories *dtout* and *dtf* keep track of the timers associated with these delays.

The HOL functions *tli* and *hdi*, in the above definition, accept two arguments, i.e., a list *l* and a positive integer *n*, and return the tail of the list *l* starting from its n^{th} element and the n^{th} element of the list *l*, respectively. Whereas the functions *new_packet* and *set_non_packet* declare a message of type *packet* (using its two arguments) and *non_packet*, respectively. The function *label* returns the sequence number of a *packet* and the predicate *good_packet* checks the message type of its argument and returns *False* if it is *non_packet* and *True* otherwise. The functions *mod_n_add* and *mod_n_sub* return the modulo-*n*, where *n* is their third argument, addition or subtraction results of their first two arguments, respectively.

The definition of *DATA_TRANS_STOP_WAIT* should be read as follows. At all times *t*, check for the transmission conditions, i.e., there is data available to be transmitted ~*NULL* (*tli* (*i t*) (*rem t*))) and the number of unacknowledged messages is less than the *window size* (*i t* < *ws*). If the transmission conditions are satisfied, then wait for the next t_f time units, i.e., decrement the timer *dtf* by one at every increment of the time until it reaches 0 and during this time maintain the values of histories *I* and *dtout* while holding

the transmission of a new packet to the channel. Once t_f time units have elapsed, i.e., the contents of *dtf* timer become 0, then instantly transmit the *(i t)*th message in the window *hdi (i t) (rem t))* using the sequence number *mod_n_add* (*s t*) (*i t*) *sn*) and increment the value of the history *i* by 1, activate the timer *dtout*, associated with the t_{out} delay, by decrementing its value by 1 and initialize the timer *dtf*, associated with the t_f delay, to its default value of t_f, in the next increment of time *t*. On the other hand, for all times *t* for which one of the transmission conditions is not satisfied, no message is transmitted (*set_non_packet*) and the initial value of the *dtf* timer is maintained. The values of *i* and *dtout*, under the no transmission conditions, depend on the event if the timer *dtout* reaches 1 or an ACK message *(good_packet* (*ackS t*)) is received for a data message that has been sent and not yet acknowledged, i.e., if the difference between the label of (*ackS t*) and the sender's sequence number is less than *ws* (*mod_n_sub (label (ackS t)) (s t) sn* < *ws*). If this event happens, then the timer *dtout* is initialized to its default value *tout* and the value of *i* is decremented by 1 in the next increment of time *t*. Otherwise, we remain in the wait state until the timer *dtout* expires or a valid ACK is received, while maintaining the value of *i* and decrementing the timer *dtout* by one at every increment of the time *t*.

Data Reception

The process *DATA_RECV* in Figure 4 characterizes the data reception behavior, at the receiver end, of the Stop-and-Wait protocol and the corresponding predicate is defined as follows in Box 2, where the history *dataR* represents the data messages received by the receiver at any particular time. The history *r* represents, modulo *sn*, the sequence number of the data messages that the receiver is expecting to receive. The function *data* returns the data portion of a *packet* and ++ is the symbol for the list *cons* function in HOL.

The definition of *DATA_RECV_STOP_WAIT* should be read as follows. At all times *t*, if (*dataR t*) is not a *non_packet*, i.e., (*good_packet (dataR t)*), and the sequence field of the packet (*dataR t*) is equal to the next number to be output to the sink (*label (dataR t) = r t*), then the data part of the packet is appended to the *sink* list and *r* is updated to the sequence number of the next message expected, i.e., *(r (t + 1) = mod_n_add (r t) 1 sn)*. Otherwise if a valid data packet is not received then the output list *sink* and *r* retain their old values.

We have intentionally assigned a fixed value of 1 to the processing delay (t_p), which specifies the time required for processing an incoming message at the receiver end, in order to simplify the understandability of the proofs presented in the next two sections. If required, the processing

Box 2.

```
∀ sn dataR sink r.
     DATA_RECV_STOP_WAIT sn dataR sink r =
     ∀ t. (if good_packet (dataR t) ∧ (label (dataR t) = r t) then
          (sink (t + 1) = sink t ++ [ data (dataR t) ]) ∧
          (r (t + 1) = mod_n_add (r t) 1 sn)
        else
          (sink (t + 1) = sink t) ∧ (r (t + 1) = r t))
```

delay can be made a variable quantity by using a similar approach that we used for t_{out} and t_f delays in the predicate *DATA_TRANS_STOP_WAIT*.

ACK Transmission

The process *ACK_TRANS* in Figure 4 characterizes the ACK transmission behavior of the Stop-and-Wait protocol and the corresponding predicate is defined as follows in Box 3.

The history *ackR* represents the ACK messages transmitted by the sender. The history *ackty* represents the data part of the ACK message that could be used to specify properties of protocols, such as negative acknowledgements: a type of acknowledgement message which enables the sender to retransmit messages efficiently. The variable *ack_msg* represents a constant data field that is sent along with every ACK message by the receiving station, as in the Stop-and-Wait protocol the ACK messages do not convey any other information except the reception of a data message. The variable *ta* holds the value for the t_a delay, defined in Section 2 and the history *dta* keeps track of the timer associated with this delay. Whereas, the history *rec_flag* keeps track of the reception of a data message at the receiver until a corresponding ACK message is sent.

The definition of *ACK_TRANS_STOP_WAIT* should be read as follows. At all times *t*, the history *ackty* is assigned the value of the default ACK message for the Stop-and-Wait protocol, i.e., *ack_msg*. For all times *t*, if an in-sequence data message arrives at the receiver *~(r t = r (t - 1))*, then instantly transmit an ACK message if the contents of the timer *dta* are 0, otherwise do not issue an ACK and retain the information of receiving a valid data in the *rec_flag* while activating

Box 3.

```
∀ sn ackR r ackty ack_msg ta dta rec_flag.
     ACK_TRANS_STOP_WAIT sn ackR r ackty ack_msg ta dta rec_flag =
     ∀ t. (ackty t = ack_msg) ∧
       (if ~(r t = r (t - 1)) then
         (if dta t = 0 then
           (ackR t = new_packet (mod_n_sub (r t) 1 sn) (ackty t)) ∧
           (dta (t + 1) = ta) ∧ (rec_flag (t + 1) = F)
         else
           (ackR t = set_non_packet) ∧ (dta (t + 1) = dta t - 1) ∧
           (rec_flag (t + 1) = T))
       else
         (if rec_flag t then
           (if dta t = 0 then
             (ackR t = new_packet (mod_n_sub (r t) 1 sn) (ackty t)) ∧
             (dta (t + 1) = ta) ∧ (rec_flag (t + 1) = F)
           else
             (ackR t = set_non_packet) ∧
             (dta (t + 1) = dta t - 1) ∧ (rec_flag (t + 1) = T))
         else
           ackR t = set_non_packet) ∧ (dta (t + 1) = ta) ∧
           (rec_flag (t + 1) = F)))
```

the timer associated with t_a by decrementing its value by 1. On the other hand, for all times *t* for which no in-sequence data message arrives at the receiver, check if there exists a valid data message that has successfully arrived at the receiver but has not been acknowledged so far (*rec_flag t*). If that is the case, then if the timer associated with the delay t_a has expired (*dta t =0*) then instantly issue the respective ACK message while initializing histories *dta* and *rec_flag* to their default values of *ta* and *False*, respectively. Otherwise wait for the *dta* timer to expire while holding the ACK transmission and the value of history *rec_flag* and decrementing the value of the timer *dta* by 1. On the other hand, if there is no valid data arrival or no pending ACK transmission, then the receiver is not allowed to transmit an ACK message and it assigns the histories *dta* and *rec_flag* to their default values of *ta* and *False*, respectively.

ACK Reception

The process *ACK_RECV* in Figure 4 characterizes the ACK reception behavior, at the sending station, of the Stop-and-Wait protocol and the corresponding predicate is defined as follows in Box 4.

The sender checks the label of every ACK message it receives to find out if it is one of the messages that has been sent and not yet acknowledged, i.e., if the modulo-*sn* difference between the sequence number of (*ackS t*) and the sender's sequence number is less than *ws*, i.e., (*mod_n_sub (label (ackS t)) (s t) sn < ws*). If this is the case, then the sender slides the window up by updating the sender's history (*s t*) to be the first message not known to be accepted: (*mod_n_add (label (ackS t)) 1 sn*) and by updating (*rem t*), the list of data remaining to be sent. Otherwise, both histories *s* and *rem* retain their previous values. Just like the receiver, we again assigned a fixed value of 1 to the processing delay (t_p).

Communication Channel

The processes *DATA_CHAN* and ACK_*CHAN* in Figure 4 characterize the communication channel connecting the sender and receiver in the Stop-and-Wait, respectively. In this chapter, we are dealing with a channel that has a fixed propagation delay (t_{prop}). We present two definitions for the communication channel for the Stop-and-Wait protocol; the first one models the channel that is noiseless and the second one models a noisy channel, which may lose packets. The noiseless channel predicate is defined as follows in Box 5, where the histories *in*, *out* and *d* represent the input message, output message and the propagation delay for the channel at a particular time, respectively. The variable *tprop* represents the fixed value of channel delay (*d t*) for all *t*. According to the above definition, the output from a channel at time *t* is a copy of the channel's input at time (*t - tprop*).

Box 4.

```
∀ ws sn ackS rem s.
     ACK_RECV_STOP_WAIT ws sn ackS rem s =
     ∀ t. (if good_packet (ackS t) ∧
          mod_n_sub (label (ackS t)) (s t) sn < ws
       then
         (s (t + 1) = mod_n_add (label (ackS t)) 1 sn) ∧
         (rem (t + 1) = tli (mod_n_sub (s (t + 1)) (s t) sn) (rem t))
       else
         (s (t + 1) = s t) ∧ (rem (t + 1) = rem t))
```

Box 5.

```
∀ in out d tprop.
     NOISELESS_CHANNEL_STOP_WAIT in out d tprop =
     ∀ t. (if t < tprop then
         out t = set_non_packet
       else
         out t = in (t - d t)) ∧ 0 < tprop ∧ (d t = tprop)
```

Next, we define a predicate that models a noisy channel that looses a message with probability *p*. (Box 6)

In Box 6, we utilized the formal definition of the Bernoulli(*p*) random variable to model the noise effect. The variable *p* represents the probability of channel error or getting a *True* from the Bernoulli random variable and the history *bseqt* keeps track of the remaining portion of the infinite Boolean sequence, explained in Section 3, after every call of the Bernoulli random variable. According to the above definition, a valid packet that arrives at input of the channel appears at the output of the channel after *tprop* time units with probability *1-p*.

Stop-and-Wait Protocol

We first define some constraints that are required to ensure the correct behavior of our Stop-and-Wait protocol specification, before giving the actual formalization of the protocol.

INITIAL CONDITIONS

In case of the formal specification of real-time systems in HOL, we need to assign appropriate values to the history variables as initial conditions. We use following initial conditions for the Stop-and-Wait protocol (Box 7).

Box 6.

```
∀ in out d tprop p bseqt.
     NOISY_CHANNEL_STOP_WAIT in out d tprop p bseqt =
     ∀ t. (if t < tprop then
         (out t = set_non_packet) ∧ (bseqt (t + 1) = bseqt t)
       else
         (if good_packet (in (t - d t)) then
           (if ~fst (prob_bernoulli p (bseqt t)) then
             (out t = in (t - d t)) ∧
             (bseqt (t + 1) = snd (prob_bernoulli p (bseqt t)))
           else
             (out t = set_non_packet) ∧
             (bseqt (t + 1) = snd (prob_bernoulli p (bseqt t))))
         else
           (out t = set_non_packet) ∧ (bseqt (t + 1) = bseqt t))) ∧
           0 < tprop ∧ (d t = tprop)
```

Box 7.

```
∀ source rem s sink r i ackR dtout dtf dta tout tf ta rec_flag bseqt bseq.
INIT_STOP_WAIT source rem s sink r I ackR dtout dtf dta tout tf ta rec_flag
bseqt bseq = (rem 0 = source) ∧ (s 0 = 0) ∧ (sink 0 [ ]) ∧ (r 0 = 0) ∧
     (i 0 = 0) ∧ (dtout 0 = tout) ∧ (rec_flag 0 = F) ∧ (ackR 0 = set_non_pack-
et) ∧
     (dtf 0 = tf) ∧ (dta 0 = ta) ∧ (bseqt 0 = bseq)
```

ASSUMPTIONS

Liveness or Timeliness: While verifying a system, which allows nondeterministic or probabilistic choice between actions, we often need to include additional constraints to make sure that events of interest do occur. This has been done by including a *timeliness* constraint in the specification of the Stop-and-Wait protocol: if the sender's state has not changed over an interval of *maxP* time units, then the sender assumes that the receiver or the channel has crashed and aborts the protocol. A predicate *ABORT* is defined that is *True* only when the protocol aborts and *False* otherwise. Now, the predicate *ABORT* characterizes which *abort* histories satisfy this constraint. (Box 8)

A protocol is said to be live if it is never aborted. This kind of liveness is *assumed* using the following constraint

```
LIVE_ASSUMPTION abort = ∀ t. ~(abort t)
```

Window Size and Sequence Numbers: As has been mentioned before, instead of using their exact values of 1 and 2, we use the variables *ws* and *sn* to represent the *window size* and distinct sequence numbers for the Stop-and-Wait protocol in the above predicates. This has been done, in order to be able to establish logical implications between the predicates defined in this chapter and the corresponding predicates for the Sliding Window protocol, defined in [Cadell-Oliver_92]. Now, we assign the exact values to these variables in an assumption predicate as follows

```
∀ ws sn. WSSN_ASSUM_STOP_WAIT ws sn =
(ws = 1) ∧ (sn = 2)
```

The Stop-and-Wait protocol can now be formalized as the logical conjunction of the predicates defined in the preceding sections. We present two specifications corresponding to noiseless or ideal and noisy channel conditions. (Box 9)

The higher-order-logic predicate *STOP_WAIT_NOISELESS* formally specifies the behavior of the Stop-and-Wait protocol under ideal or noiseless conditions as the corresponding predicate for the channel has been used for both data and ACK channels. It is also important to note here that we do not initialize the history *bseqt* in the predicate *INIT_STOP_WAIT* as there is no need to use the infinite Boolean sequence in this case. Next, we utilize the noisy channel predicate

Box 8.

```
∀ abort maxP rem.
     ABORT abort maxP rem =
     ∀ t. abort t = (rem t = rem (t - maxP)) ∧ maxP ≤ t ∧ ~NULL (rem t)
```

Box 9.

```
∀ source sink rem s i r ws sn ackty maxP abort dataS dataR ackS ackR d
     tprop dtout dtf dta tf ack_msg ta tout rec_flag.
STOP_WAIT_NOISELESS source sink rem s i r ws sn ackty maxP abort dataS
       dataR ackS ackR d tprop dtout dtf dta tf ack_msg ta tout rec_flag =
     INIT_STOP_WAIT source rem s sink r I ackR dtout dtf dta tout tf ta rec_
flag ∧
     DATA_TRANS_STOP_WAIT ws sn dataS s rem i ackS tout tf dtout dtf ∧
     NOISELESS_CHANNEL_STOP_WAIT dataS dataR d tprop ∧
     DATA_RECV_STOP_WAIT sn dataR sink r ∧
     ACK_TRANS_STOP_WAIT sn ackR r ackty ack_msg ta dta rec_flag ∧
     NOISELESS_CHANNEL_STOP_WAIT ackR ackS d tprop ∧
     ACK_RECV_STOP_WAIT ws sn ackS rem s ∧
     ABORT abort maxP rem ∧ WSSN_ASSUM_STOP_WAIT ws sn
```

to formally specify the Stop-and-Wait protocol with a noisy channel as follows in Box 10.

In the above definition, the data channel has been made noisy while a noiseless channel is used for the ACK messages. This has been done on purpose in order to reduce the length of the performance analysis proof by avoiding some redundancy. On the other hand, this decision does not affect the illustration of the idea behind the performance analysis of the Stop-and-Wait protocol under noisy conditions as we present the complete handling of a noisy channel in one direction. The analysis can be extended to both noisy channels by choosing noisy channel predicates for both channels and then handling the ACK channel in a similar way as the noisy data channel is handled in this chapter.

FUNCTIONAL VERIFICATION OF THE STOP-AND-WAIT PROTOCOL

The job of an ARQ protocol is to ensure reliable transfer of a stream of data from the sender to the receiver. This functional requirement can be formally specified as follows [Cadell-Oliver, 1992].

```
∀ source sink.
     REQ source sink =
     (∃ t. sink t = source) ∧ ∀ n.
is_prefix (sink t) (sink (t + n))
```

where the predicate *is_prefix* is *True* if its first list argument is a prefix of its second list argument. According to the predicate *REQ*, an ARQ protocol satisfies its functional requirements only if there exists a time at which the *sink* list becomes equal to the original *source* list, i.e., a time when the data at the sender is transferred, as is, to the receiver, and the history *sink* is prefix closed.

In order to verify the functional correctness of our specification of the Stop-and-Wait protocol, we now define the predicates for the Stop-and-Wait protocol in such a way that they logically imply the corresponding predicates used for the formal specification of the Sliding Window protocol presented in [Cadell-Oliver, 1992]. This relationship allows us to inherit the functional correctness theorem verified for the specification of the Sliding Window protocol for our Stop-and-Wait protocol specification.

For illustration purposes, consider the example of the data transmission predicate. It has been

Box 10.

```
∀ source sink rem s i r ws sn ackty maxP abort dataS dataR ackS ackR d
     tprop dtout dtf dta tf ack_msg ta tout rec_flag bseqt bseq p.
     STOP_WAIT_NOISY source sink rem s i r ws sn ackty maxP abort dataS
       dataR ackS ackR d tprop dtout dtf dta tf ack_msg ta tout
       rec_flag bseqt bseq =
     INIT_STOP_WAIT source rem s sink r i
         ackR dtout dtf dta tout tf ta rec_flag bseqt bseq ∧
     DATA_TRANS_STOP_WAIT ws sn dataS s rem i ackS tout tf dtout dtf ∧ NOISY_
CHANNEL_STOP_WAIT in out d tprop p bseqt ∧
     DATA_RECV_STOP_WAIT sn dataR sink r ∧
     ACK_TRANS_STOP_WAIT sn ackR r ackty ack_msg ta dta rec_flag ∧
     NOISELESS_CHANNEL_STOP_WAIT ackR ackS d tprop ∧ ACK_RECV_STOP_WAIT ws sn
ackS rem s ∧
     ABORT abort maxP rem ∧
     WSSN_ASSUM_STOP_WAIT ws sn
```

defined in [Cadell-Oliver, 1992] for the Sliding Window protocol as follows in Box 11.

It can be easily observed, and we verified it in HOL using Boolean algebra properties, that the predicate DATA_TRANS_STOP_WAIT, given in the previous section, logically implies the above predicate

```
∀ ws ns dataS s rem i ackS tout tf
dtout dtf.
     DATA_TRANS_STOP_WAIT ws ns dataS
s rem i ackS tout tf dtout dtf ⇒
     DATA_TRANS_SW ws ns dataS s rem
i
```

In a similar way, we were able to prove logical implications between all the predicates used in the formal specification of the Sliding Window protocol and the corresponding predicates used for the formal specification of the Stop-and-Wait protocol (see Boxes 12 and 13). These relationships allowed us to formally verify the functional correctness of both of the formal specifications of the Stop-and-Wait protocol, given in the previous section, in HOL.

It is important to note that the generic specification of the Sliding Window Protocol in [Cadell-Oliver, 1992] is quite general and does not include many details, such as the precise conditions under which the messages are transmitted or acknowl-

Box 11.

```
∀ ws sn dataS s rem i.
     DATA_TRANS_SW ws sn dataS s rem i =
     ∀ t. (if ~NULL (tli (i t) (rem t)) ∧ i t < ws then
         (dataS t = new_packet (mod_n_add (s t) (i t) sn) (hdi (i t) (rem t))) ∨
         (dataS t = set_non_packet)
       else
         dataS t = set_non_packet)
```

Box 12. Theorem 5

```
∀ source sink rem s i r ws sn ackty maxP abort dataS dataR ackS
       ackR d tprop dtout dtf dta tf ack_msg ta tout rec_flag.
STOP_WAIT_NOISELESS source sink rem s i r ws sn ackty maxP abort dataS
     dataR ackS ackR d tprop dtout dtf dta tf ack_msg ta tout rec_flag ∧
LIVE_ASSUMPTION abort ⇒ REQ source sink
```

edged and the delays (t_{out}, t_f, t_a, etc.) associated with different operations. Therefore, it cannot be used for reasoning about message delays and thus performance related properties, as such. On the other hand, the formal specification of the Stop-and-Wait protocol, given in this chapter, is more specific and provides a detailed description of the protocol including the timing behavior associated with different operations.

Another major point that we would like to mention here is that in order to establish the logical implication between the two protocol models, we had to introduce some additional generality in our formal definitions, such as the usage of variables *ws* and *sn* instead of their exact values of 1 and 2, respectively.

Even though, such generalizations are not required for the functional description of the Stop-and-Wait protocol, they do not harm us in any way. They lead to a much faster functional verification, as has been illustrated in this section. On the other hand, they do not affect the formal reasoning related to the performance issues, since the exact values for these variables are assigned in an assumption (*WSSN_ASSUM_STOP_WAIT*) that is a part of our Stop-and-Wait protocol specification, which is used for conducting the performance analysis as well.

PERFORMANCE ANALYSIS OF THE STOP-AND-WAIT PROTOCOL

In this section, we present the verification of the message delay relations for the Stop-and-Wait protocol, given in Equations 1 and 3, for noiseless and noisy channels, respectively. The verification is based on the two formal specifications of the Stop-and-Wait protocol, STOP*_WAIT_NOISELESS* and *STOP_WAIT_NOISY*, given earlier in the chapter.

Noiseless Channel Conditions

The first and the foremost step in verifying the message delay characteristic for the Stop-and-Wait protocol is to formally specify it. Informally speaking, the message delay refers to the time required for the transmitter to send a single data message and know that it has been successfully

Box 13. Theorem 6

```
∀ source sink rem s i r ws sn ackty maxP abort dataS dataR ackS
       ackR d tprop dtout dtf dta tf ack_msg ta tout rec_flag
         bseqt bseq p.
STOP_WAIT_NOISY source sink rem s i r ws sn ackty maxP abort dataS
     dataR ackS ackR d tprop dtout dtf dta tf ack_msg ta tout
     rec_flag bseqt bseq ∧
LIVE_ASSUMPTION abort ⇒ REQ source sink
```

received at the receiver. We specify this in higher-order-logic as follows

```
∀ rem source.
     DELAY_STOP_WAIT_NOISELESS rem
source =
     @ t. (rem t = TL source) ∧ (rem
(t - 1) = source)
```

where *TL* refers to the *tail* function for *lists* and *@x.t* refers to the Hilbert choice operator in HOL that represents the value of *x* such that *t* is *True*. Thus, the above specification returns the time *t* at which the *rem* list, which represents the data remaining to be sent at any time *t*, is reduced by one element from its initially assigned value of the *source* list. Indeed it is precisely equal to the message delay of the first data element in the *source* list.

Based on the above definition of the message delay and the delays associated with the formal specification of the Stop-and-Wait protocol (*STOP_WAIT_NOISELESS*), Equation 1 can be formally expressed in HOL as follows in Box 14.

It is important to note here that the processing delay, t_p, has been assigned a value of 1 in our model, as explained in the previous section. The two assumptions that we have added to Theorem 7 ensure that the source list is not an empty list, i.e., *~(NULL source)*, otherwise no data transfer takes place, and the time out period *tout* is always greater than or equal to its lower bound. Rewriting the proof goal of Theorem 7 with the formal specification of the Stop-and-Wait protocol delay and removing the Hilbert Choice operator we get the following expression

```
(∃x. (rem x = TL source) ∧ (rem (x -
1) = source)) ∧
     ∀ x. (rem x = TL source) ∧ (rem
(x - 1) = source) ⇒
       (x = tf + tprop + 1 + ta +
tprop + 1
```

The above subgoal is a logical conjunction of two Boolean expressions and it can be proved to be *True* only if there exists a time *x* for which the conditions (*rem x = TL source*) and (*rem (x - 1) = source*) are *True* and the value of any variable *x* that satisfies these conditions is unique and is equal to *tf + tprop + 1 + ta + tprop + 1*.

We proceed with the proof of this subgoal by assuming the following expression

Lemma 1:

```
(rem (tf + tprop + 1 + ta + tprop +
1) = TL source) ∧
(rem ((tf + tprop + 1 + ta + tprop +
1) - 1) = source))
```

to be *True*, which we will prove later, under the given constraints for the Stop-and-Wait protocol. Lemma 1 leads us to prove the first Boolean expression in our subgoal as now we know an *x = (tf + tprop + 1 + ta + tprop + 1)* for which the

Box 14. Theorem 7

```
∀ source sink rem s i r ws sn ackty maxP abort dataS dataR ackS
     ackR d tprop dtout dtf dta tf ack_msg ta tout rec_flag.
STOP_WAIT_NOISELESS source sink rem s i r ws sn ackty maxP abort
dataS dataR ackS ackR d tprop dtout dtf dta tf ack_msg ta tout rec_flag ∧
~(NULL source) ∧ (tprop + 1 + ta + tprop + 1 ≤ tout) ⇒
     (DELAY_STOP_WAIT_NOISELESS rem source =
     tf + tprop + 1 + ta + tprop + 1)
```

given conditions are *True*. We verify the second Boolean expression in the subgoal by first proving the monotonically decreasing characteristic of the history *rem* under the given constraints of the Stop-and-Wait protocol, i.e.,

```
∀ a b. a < b ⇒ ∃c. c ++ rem b = rem a
```

where ++ represents the list *cons* function in HOL. Now, if there exists an *x*, that satisfies the conditions (*rem x = TL source*) and (*rem (x - 1) = source*), then it may be equal to, less than or greater than (*tf + tprop + 1 + ta + tprop + 1*). For the latter two cases, we reach a contradiction in the assumption list, based on the monotonically decreasing characteristic of the history *rem*, whereas, the case when *x = (tf + tprop + 1 + ta + tprop +1)* verifies our subgoal of interest, which concludes the proof of Theorem 7 under the assumption of Lemma 1.

Lemma 1 can now be proved in HOL using the definitions of the predicates in the formal specification of the Stop-and-Wait protocol under noiseless channels. The corresponding HOL proof step sequence is summarized in Figure 5.

Noisy Channel Conditions

The message delay, under noisy channel conditions, refers to the time required for the transmitter to send a single data message and know that it has been successfully received at the receiver. Though the delay, in this case, is a random quantity since its value is non-deterministic and depends on the outcomes of a sequence of Bernoulli trials, which are used to model the channel noise as can be seen in the definition of the predicate *NOISY_CHANNEL_STOP_WAIT*. Therefore, the message delay of the Stop-and-Wait protocol under noisy channel conditions needs to be formally specified as a random variable as follows in Box 15, where

Figure 5. HOL Proof Sequence for Lemma 1

Number	Formally Verified Statements
1	$\forall t.t \leq tf \Rightarrow (i(t) = 0)$
2	$\forall t.t < tf \Rightarrow (dataS\ t = set\_non\_packet)$
3	$\forall t.t < tf + tprop \Rightarrow (dataR\ t = set\_non\_packet)$
4	$\forall t.t < tf + tprop + 1 \Rightarrow (sink\ t = []) \wedge (r\ t = 0)$
5	$\forall t.t \leq tf + tprop + 1 \Rightarrow (rec\_flag\ t = F) \wedge (dta\ t = ta)$
6	$\forall t.t < tf + tprop + 1 + ta \Rightarrow (ackR\ t = set\_non\_packet)$
7	$\forall t.t < tf + tprop + 1 + ta + tprop \Rightarrow (ackS\ t = set\_non\_packet)$
8	$\forall t.t < tf + tprop + 1 + ta + tprop + 1 \Rightarrow (s\ t = 0) \wedge (rem\ t = source)$
9	$(dataS\ tf = new\_packet\ 0\ (HD\ source)) \wedge (i(tf + 1) = 1)$
10	$\forall t.tf < t \wedge t < tf + tprop + 1 + ta + tprop + 1 \Rightarrow (tout + tf - t \leq dtout\ t)$
11	$\forall t.tf < t \wedge t < tf + tprop + 1 + ta + tprop + 1 \Rightarrow$ $(i\ t = 1) \wedge (dataS\ t = set\_non\_packet)$
12	$dataR(tf + tprop) = new\_packet\ 0\ (HD\ source)$
13	$\forall t.tf + tprop < t \wedge t < tf + tprop + 1 + ta + tprop + 1 \Rightarrow$ $(dataR\ t = set\_non\_packet)$
14	$\forall t.tf + tprop + 1 \leq t \wedge t < tf + tprop + 1 + ta + tprop + 1 \Rightarrow (r\ t = 1)$
15	$\forall t.tf + tprop + 1 \leq t \wedge t < tf + tprop + 1 + ta \Rightarrow$ $(rec\_flag(t + 1)) \wedge (dta(t + 1) = ta - (t - (tf + tprop)))$
16	$ackR(tf + tprop + 1 + ta) = new\_packet\ 0\ ack\_msg$
17	$rem(tf + tprop + 1 + ta + tprop + 1) = TL\ source$

Box 15.

```
∀ rem source bseqt.
     DELAY_STOP_WAIT_NOISY rem source bseqt =
     ((@t. (rem t = TL source) ∧ (rem (t - 1) = source)),
     bseqt @t. (rem t = TL source) ∧ (rem (t - 1) = source))
```

history *bseqt t* represents the unused portion of the infinite Boolean sequence, explained in Section 2, after performing the required number of Bernoulli trials at any given time *t*. The above specification returns a pair with the first element equal to the time *t* that satisfies the two conditions (*rem t = TL source*) and (*rem (t - 1) = source*), and thus represents the random message delay of the first data element in the *source* list, and the second element equal to the unused portion of the infinite Boolean sequence at this time instant *t*.

As a first step towards the verification of the average value of the random delay specified in *DELAY_STOP_WAIT_NOISY*, we establish its relationship with the infamous Geometric random variable, which basically returns the number of trials to attain the first success in an infinite sequence of Bernoulli trials. This way, we can benefit from existing HOL theorems related to the average characteristic of Geometric random variable, such as Theorem 2, for the verification of the average value of the message delay of a Stop-and-Wait protocol. This relationship, given in Equation 2 can be expressed in HOL using the formal specification of the Stop-and-Wait protocol *STOP_WAIT_NOISY* and the Geometric random variable *prob_geom* [Hasan, 2007], as follows in Box 16, where *p* represents the probability of channel error, i.e., getting a *True* from the Bernoulli random variable. The first argument of the function *prob_geom* [Hasan, 2007] represents the probability of success for the corresponding sequence of the Bernoulli trials, which, in the case of our definition of the noisy channel, is equal to the probability of getting a *False* from a Bernoulli trial. The above theorem is proved under the assumption that the value of the probability *p* always falls in the interval *[0,1)*. It is not allowed to attain the value 1, in order to avoid the case when the channel always rejects incoming packets and thus leads to no data transfers. The assumption, *LIVE_ASSUMPTION abort* ensures liveness as has been explained in Section 2. The other assumptions used in the above theorem are

Box 16. Theorem 8

```
∀ source sink rem s i r ws sn ackty maxP abort dataS dataR ackS ackR
       d tprop dtout dtf dta tf ack_msg ta tout rec_flag bseqt bseq p.
STOP_WAIT_NOISY source sink rem s i r ws sn ackty maxP abort dataS
       dataR ackS ackR d tprop dtout dtf dta tf ack_msg ta tout
       rec_flag bseqt bseq ∧
LIVE_ASSUMPTION abort ∧
0 ≤ p ∧ p < 1 ∧ ~NULL source ∧
tprop + 1 + ta + tprop + 1 ≤ tout ⇒
     (DELAY_STOP_WAIT_NOISY rem source bseqt =
         ((tf + tout) * (fst (prob_geom (1 - p) bseq) - 1) + tf +
       tprop + 1 + ta + tprop + 1, snd (prob_geom (1 - p) bseq)))
```

Box 17.

```
(∀ p bseq.
     BERNOULLI_TRIAL_F_IND 0 p bseq = ~fst (prob_bernoulli p bseq)) ∧
   ∀ n p bseq. BERNOULLI_TRIAL_F_IND (SUC n) p bseq =
     fst (prob_bernoulli p bseq) ∧
     BERNOULLI_TRIAL_F_IND n p (snd (prob_bernoulli p bseq))

(∀ p bseq. NTH_BERNOULLI_TRIAL_SND 0 p bseq = bseq) ∧
   ∀ n p bseq. NTH_BERNOULLI_TRIAL_SND (SUC n) p bseq =
     snd (prob_bernoulli p (NTH_BERNOULLI_TRIAL_SND n p bseq))
```

similar to the ones used for the verification of Theorem 7.

We proceed with the verification of Theorem 8 in HOL by first defining the following two recursive functions (see Box 17).

The first function, *BERNOULLI_TRIAL_F_IND* returns *True* if and only if its first argument, say *n*, represents the positive integer index of a trial, in a sequence of independent Bernoulli trials, that returns a *False* while all Bernoulli trials with lower index values than *n* have returned a *True*. The second function *NTH_BERNOULLI_TRIAL_SND* returns the value of the *snd* element of the n^{th} Bernoulli trial in a sequence of independent Bernoulli trials, where *n* is the first argument of the function *NTH_BERNOULLI_TRIAL_SND*. In other words, it basically returns the unused infinite Boolean sequence after *n* independent Bernoulli trials have been performed using the given infinite Boolean sequence.

Under the given assumptions of Theorem 8, it can be shown that a data message available at the *source* list does finally make through the noisy channel at some time. This can be verified in HOL, for the top element of the *source* list, by proving that there exists some *n* for which the function *BERNOULLI_TRIAL_F_IND* returns a *True*

```
∃ n. BERNOULLI_TRIAL_F_IND n p bseq
```

for the given values of p and bseq. If a positive integer n exists that satisfies the above condition, then it can be verified in HOL that the Geometric random variable, which returns the number of trials to attain the first success in an independent sequence of Bernoulli(p) trials, with success probability equal to (1 - p) can be formally expressed as follows

```
∀ n p s.
     0 ≤ p ∧ p < 1 ∧ BERNOULLI_
TRIAL_F_IND n p s ⇒
     (prob_geometric_p (1 - p) s =
     (n + 1,NTH_BERNOULLI_TRIAL_SND
(n + 1) p s))
```

The HOL proof is based on the formal definition of the function *prob_geom* and the underlying probability theory principles, presented in [Hurd, 2002].

Based on the above results, the proof goal of Theorem 8 can be simplified using the definition of *DELAY_STOP_WAIT_NOISY* and removing the Hilbert choice operator as follows

```
(∃ x. (rem x = TL source) ∧ (rem (x -
1) = source)) ∧
     ∀ x.
       (rem x = TL source) ∧ (rem (x
- 1) = source) ⇒
       (x = (tf + tout) * n + tf +
```

Box 18. Lemma 3

```
∀ bseq v.
INIT_STOP_WAIT_GEN source rem s sink r i
     ackR dtout dtf dta tout tf ta rec_flag bseqt bseq v ∧
     BERNOULLI_TRIAL_F_IND n p bseq ⇒
     (rem (v + (tf + tout) * n + tf + tprop + 1 + ta + tprop + 1 - 1) = source) ∧
       (rem (v + (tf + tout) * n + tf + tprop + 1 + ta + tprop + 1) = TL
source)∧
       (bseqt (v + (tf + tout) * n + tf + tprop + 1 + ta + tprop + 1) =
NTH_BERNOULLI_TRIAL_SND (n + 1) p bseq)
```

```
tprop + 1 + ta + tprop + 1) ∧
       (bseqt x = NTH_BERNOULLI_TRI-
AL_SND (n + 1) p bseq)
```

The above subgoal is quite similar to the one that we got after simplifying the proof goal of Theorem 7. Therefore, we follow the same proof approach and assume the following expression

Lemma 2:

```
(rem ((tf + tout) * n + tf + tprop +
1 + ta + tprop + 1 - 1) = source) ∧
(rem ((tf + tout) * n + tf + tprop +
1 + ta + tprop + 1) = TL source) ∧
(bseqt ((tf + tout) * n + tf + tprop
+ 1 + ta + tprop + 1) =
     NTH_BERNOULLI_TRIAL_SND (n + 1)
p bseq)
```

to be *True*, which we will prove later, under the given assumptions of Theorem 8. Lemma 2 leads us to prove the first Boolean expression in the subgoal as now we know an *x = ((tf + tout) * n + tf + tprop + 1 + ta + tprop + 1)* for which the given conditions (*rem x = TL source*) and (*rem (x - 1) = source*) are *True*. The second Boolean expression in the subgoal can now be proved using Lemma 2 along with the monotonically decreasing characteristic of the history *rem* in a similar way as we handled the counterpart while verifying Theorem 7.

The next step is to prove Lemma 2 under the assumptions given in the assumption list of Theorem 8. We proceed in this direction by verifying a more generalized lemma (Box 18), under the assumptions of Theorem 8, for which Lemma 2 is a special case when v=0.

The first assumption in Lemma 3, i.e., the predicate INIT_STOP_WAIT_GEN, provides the status of the histories used in the predicate STOP_WAIT_NOISY at time *v* and is defined as shown in Box 19.

It can be proved to be a logical implication of the predicate *INIT_STOP_WAIT*, which is included in the definition of *STOP_WAIT_NOISY* and is thus present in the assumption list of Theorem 8, for the case when v = 0.

Whereas, the second assumption of Lemma 3, i.e., *BERNOULLI_TRIAL_F_IND n p bseq* has already been shown to be a consequence of the assumptions of Theorem 8. Thus, Lemma 2 can be proved as a special case of Lemma 3 when the positive integer variable *v* is assigned a value of 0. Now, in order to complete the formal proof of Theorem 8 in HOL, we need to verify Lemma 3. We proceed with this proof by applying induction on the positive integer variable *n*. For the base case, i.e., *n = 0*, we get the following subgoal after some basic arithmetic simplification and using the function definitions of *BERNOULLI_TRIAL_F_IND* and *NTH_BERNOULLI_TRIAL_SND*.

Box 19.

```
∀ source rem s sink r i ackR dtout dtf dta tout tf ta rec_flag bseqt bseq v.
     INIT_STOP_WAIT_GEN source rem s sink r i ackR dtout dtf dta tout tf
       ta rec_flag bseqt bseq v =
     (i v = 0) ∧ (dtout v = tout) ∧ (dtf v = tf) ∧ (dta v = ta) ∧
     (bseqt v = bseq) ∧
     (∀ t.
       t ≤ v ⇒
       (rem t = source) ∧ (s t = 0) ∧ (sink t = [ ]) ∧ (r t = 0) ∧
       (rec_flag t = F) ∧ (ackR t = set_non_packet)) ∧
     ∀ t. v - (tout - 1) ≤ t ∧ t < v ⇒ (i t = 1))
```

```
INIT_STOP_WAIT_GEN source rem s sink
r i ackR dtout dtf dta tout tf ta
rec_flag bseqt bseq v ∧
~fst (prob_bernoulli p bseq) ⇒
     (rem (v + tf + tprop + 1 + ta +
tprop + 1 - 1) = source) ∧
     (rem (v + tf + tprop + 1 + ta +
tprop + 1) = TL source) ∧
     (bseqt (v + tf + tprop + 1 + ta
+ tprop + 1) =
       snd (prob_bernoulli p bseq))
```

The assumption *~fst (prob_bernoulli p bseq)* ensures that the noisy data channel allows reliable transmission of the first data message in the first trial. Thus, the base case of Lemma 3 becomes similar to the case of a noiseless data channel, as far as the transmission of the first data element of the *source* list is concerned. Therefore, its proof can be handled in a similar way as the proof of Lemma 1, presented in the last section, as the only difference between the two is the fact that now the initial conditions are defined for an arbitrary positive integer *v* instead of *0*. The HOL proof step sequence is summarized in Figure 6. These proofs are based on the INIT_*STOP_WAIT_GEN* and the predicates corresponding to the six processes, given in Figure 4, for the Stop-and-Wait protocol under a noisy data channel.

In the step case for Lemma 3, we get the following subgoal after some simplifications using the function definitions of *BERNOULLI_TRIAL_F_IND* and *NTH_BERNOULLI_TRIAL_SND* (Box 20), which needs to be proved under the assumption list of Theorem 8 along with the statement of Lemma 3.

The above subgoal can be proved in a very straightforward manner by specializing Lemma 3 for the case when *bseq* and *v* are equal to *snd (prob_bernoulli p bseq)* and (*v* + *tf* + *tout*), respectively, if the given initial conditions in the predicate *INIT_STOP_WAIT_GEN* hold for *snd (prob_bernoulli p bseq)* and *(v + tf + tout)*, i.e.,

Lemma 4:

```
INIT_STOP_WAIT_GEN source rem s sink
r i ackR dtout dtf dta
tout tf ta rec_flag bseqt (snd (prob_
bernoulli p bseq)) (v + tf + tout)
```

under the assumptions of Theorem 8 and the step case of Lemma 3. In order to prove Lemma 4 we need to formally verify the behavior of the histories, used in the predicate INIT_STOP_WAIT_GEN, at various points in the interval [0, v + tf + tout]. Therefore, we again use the same approach that we used to prove Lemma 1 and the base case of Lemma 3, i.e., to verify the value of

Figure 6. HOL Proof Sequence for the base Case of Lemma 3

Number	Formally Verified Statements
1	$\forall t.v \leq t \wedge t \leq v + tf \Rightarrow (i(t) = 0)$
2	$\forall t.v - (tout - 1) \leq t \wedge t < v + tf \Rightarrow (dataS\ t = set\_non\_packet)$
3	$\forall t.v \leq t \wedge t < v + tf + tprop \Rightarrow (dataR\ t = set\_non\_packet)$
4	$\forall t.v \leq t \wedge t \leq v + tf + tprop \Rightarrow (bseqt\ t = bs)$
5	$\forall t.v \leq t \wedge t < v + tf + tprop + 1 \Rightarrow (sink\ t = []) \wedge (r\ t = 0)$
6	$\forall t.v \leq t \wedge t \leq v + tf + tprop + 1 \Rightarrow (rec\_flag\ t = F) \wedge (dta\ t = ta)$
7	$\forall t.t < v + tf + tprop + 1 + ta \Rightarrow (ackR\ t = set\_non\_packet)$
8	$\forall t.v \leq t \wedge t < v + tf + tprop + 1 + ta + tprop \Rightarrow (ackS\ t = set\_non\_packet)$
9	$\forall t.v \leq t \wedge t < v + tf + tprop + 1 + ta + tprop + 1 \Rightarrow (s\ t = 0) \wedge (rem\ t = source)$
10	$(i(v + tf + 1) = 1) \wedge (dataS(v + tf) = new\_packet\ 0\ (HD\ source))$
11	$\forall t.v \leq t \wedge t \leq v + tf \Rightarrow (dtout\ t = tout)$
12	$\forall t.v + tf < t \wedge t < v + tf + tprop + 1 + ta + tprop + 1 \Rightarrow$ $v + tf + tout - t \leq dtout\ t$
13	$\forall t.v + tf + 1 \leq t \wedge t \leq v + tf + tprop + 1 + ta + tprop \Rightarrow$ $(i\ t = 1) \wedge (dataS\ t = set\_non\_packet)$
14	$dataR(v + tf + tprop) = new\_packet\ 0\ (HD\ source)$
15	$\forall t.v + tf + tprop < t \wedge t < v + tf + tprop + 1 + ta + tprop \Rightarrow$ $(dataR\ t = set\_non\_packet)$
16	$(r(v + tf + tprop + 1) = 1) \wedge (dta(v + tf + tprop + 1) = ta)$
17	$\forall t.v + tf + tprop + 1 < t \wedge t < v + tf + tprop + 1 + ta + tprop \Rightarrow (r\ t = 1)$
18	$\forall t.v + tf + tprop + 1 \leq t \wedge t < v + tf + tprop + 1 + ta \Rightarrow$ $(rec\_flag(t + 1)) \wedge (dta(t + 1) = v + ta - (t - (tf + tprop)))$
19	$ackR(v + tf + tprop + 1 + ta) = new\_packet\ 0\ ack\_msg$
20	$ackS(v + tf + tprop + 1 + ta + tprop) = new\_packet0ack\_msg$
21	$rem(v + tf + tprop + 1 + ta + tprop + 1) = TL\ source$
22	$\forall t.v + tf + tprop < t \wedge t < v + tf + tprop + 1 + ta + tprop + 1 + tprop \Rightarrow$ $(bseqt\ t = snd(prob\_bernoulli\ p\ bseq))$

Box 20.

```
INIT_STOP_WAIT_GEN source rem s sink r i ackR dtout dtf dta tout tf ta rec_flag
bseqt bseq v ∧
     fst (prob_bernoulli p bseq) ∧
     NTH_BERNOULLI_TRIAL_F n p (snd (prob_bernoulli p bseq)) ⇒
     (rem (v + (tf + tout) * (n + 1) + tf + tprop + 1 + ta + tprop + 1 - 1) =
source) ∧
     (rem (v + (tf + tout) * (n + 1) + tf + tprop + 1 + ta + tprop + 1) = TL
source) ∧
     (bseqt (v + (tf + tout) * (n + 1) + tf + tprop + 1 + ta + tprop + 1) =
       NTH_BERNOULLI_TRIAL_SND (n + 1) p (snd (prob_bernoulli p bseq))
```

these histories using the initial conditions and the definitions of the predicates used for the formal specification of the Stop-and-Wait protocol. In fact, the first 11 proof lines, given in Figure 6, for the base case of Lemma 3 can be used as they are for the proof of Lemma 4 as well, since a message transmission cannot complete before v + tf + tprop + 1 + ta + tprop + 1 time units are lapsed and the first data message is issued at time v + tf in both cases. Hereafter, contrary to the base case of Lemma 3, where one of the assumptions assured the reliable transmission of the first data message, in the case of Lemma 4 we have the assumption fst (prob_bernoulli p bseq) that forces the channel to lose the first data message. Thus, the sender keeps on waiting for a valid ACK until the timer associated with the tout delay expires and this is how the initial state at time v is maintained until the time v + tf + tout. We were able to verify this result, and thus Lemma 4, using the first 11 proof lines, given in Figure 6, followed by the proof sequence given in Figure 7. The proof of Lemma 4 concludes the proof of Lemma 3, which in turn leads to the proof of Theorem 8 as well.

Now, we are in the position of verifying the average message delay relation, given in Equation 3, for the Stop-and-Wait protocol under noisy channels. The corresponding theorem can be expressed in HOL as follows in Box 21.

The above proof goal can be reduced to the following subgoal using Theorems 4 and 8 and some arithmetic simplification

```
∀ p. 0 < p ∧ p ≤ 1 ⇒
   (expec
   (\.s. (fst (prob_geom p s) - 1,
     snd (prob_geom p s))) = (1 - p)
/ p)
```

which we were able to verify in HOL, using the formalization of the expectation theory and the Geometric random variable prob_geom, given in

Figure 7. HOL Proof Sequence for Lemma 4

Number	Formally Verified Statements
1	$\forall t.v + tf < t \wedge t < v + tf + tout \Rightarrow v + tf + tout - t \leq dtout\ t$
2	$\forall t.v + tf < t \wedge t < v + tf + tout \Rightarrow (i\ t = 1)$
3	$\forall t.v + tf < t \wedge t < v + tf + tout \Rightarrow (dataS\ t = set\_non\_packet)$
4	$\forall t.v \leq t \wedge t < v + tf + tout + tprop \Rightarrow (dataR\ t = set\_non\_packet)$
5	$\forall t.t < v + tf + tout + tprop + 1 \Rightarrow (sink\ t = []) \wedge (r\ t = 0)$
6	$\forall t.v \leq t \wedge t < v + tf + tout + tprop + 1 \Rightarrow (rec\_flag\ t = F) \wedge (dta\ t = ta)$
7	$\forall t.t < v + tf + tout + tprop + 1 \Rightarrow (rec\_flag\ t = F)$
8	$\forall t.t < v + tf + tout + tprop + 1 \Rightarrow (ackR\ t = set\_non\_packet)$
9	$\forall t.v \leq t \wedge t < v + tf + tout + tprop + 1 \Rightarrow (ackS\ t = set\_non\_packet)$
10	$\forall t.t < v + tf + tout + tprop + 1 \Rightarrow (s\ t = 0) \wedge (rem\ t = source)$
11	$\forall t.v + tf < t \wedge t \leq v + tf + tout \Rightarrow (dtf\ t = tf)$
12	$\forall t.v + tf + tprop < t \wedge t < v + tf + tout + tprop \Rightarrow$ $(bseqt\ t = snd(prob\_bernoulli\ p\ bseq))$
13	$\forall t.v + tf < t \wedge t < v + tf + tout \Rightarrow (dtout\ t = v + tf + tout - t)$
14	$dtout(v + tf + tout) = tout$
15	$i(v + tf + tout) = 0$

Box 21. Theorem 9

```
∀ source sink rem s i r ws sn ackty maxP abort dataS dataR
ackS ackR d tprop dtout dtf dta tf ack_msg ta tout rec_flag bseqt bseq p.
(∀ bseq. STOP_WAIT_NOISY source sink rem s i r ws sn ackty maxP abort
     dataS dataR ackS ackR d tprop dtout dtf dta tf ack_msg ta tout
       rec_flag bseqt bseq) ∧
     (LIVE_ASSUMPTION abort) ∧
     (0 ≤ p ∧ p < 1) ∧ (~NULL source) ∧
     tprop + 1 + ta + tprop + 1 ≤ tout ⇒
       (expec (DELAY_STOP_WAIT_NOISY rem source bseqt) =
         ((tf + tout) * p/(1-p) + (tf + tprop + 1 + ta + tprop + 1)))
```

[Hasan, 2007], and the probability theory principles, formalized in [Hurd, 2002].

Theorem 9 specifies the average message delay relation of a Stop-and-Wait protocol in terms of individual delays of the various autonomous processes, which are the basic building blocks of the protocol. Thus, it allows us to tweak various parameters of the protocol to optimize its performance for any given conditions. It is important to note here that the result of Theorem 9 is not new and the performance analysis of Stop-and-Wait protocols, based on Equation 3, existed since the early days of their introduction, however, using theoretical paper-and-pencil proof techniques. On the other hand, to the best of our knowledge, this is the first time that such a relation has been mechanically verified without any loss in accuracy or precision of the results. It therefore provides a superior approach to both paper-and-pencil proofs and simulation based performance analysis techniques.

CONCLUSION

In this chapter, we presented a higher-order-logic theorem prover based approach for the functional verification and performance analysis of real-time systems. A real-time system and its environment can be formalized as a logical conjunction of higher-order-logic predicates on positive integers, whereas the positive integers represent the ticks of a clock counting physical time in any appropriate units. Higher-order-logic has been successfully used for the formalization of a significant amount of probability theories. This feature allows us to use random variables in our model to represent the random and unpredictable elements of a real-time system and its environment. The functional and performance related properties, such as average characteristics, of a real-time system can now be formally verified, using this model, in a higher-order-logic theorem prover. Due to the inherent soundness of the theorem-proving based analysis, the presented approach ensures accurate and precise results and thus can prove to be quite useful for the performance and reliability optimization of safety critical and highly sensitive real-time system application domains, such as medicine, military or space travel. Similarly, unlike other commonly used state-based formal techniques, which are severely affected by the state-space explosion problem, the presented approach is capable of handling any real-time system that can be expressed in a closed mathematical form due to the high expressive nature of higher-order-logic. Also, there is no equivalence verification required between the models used for functional verification and performance evaluation as the

same formal model is used for both of these analysis in the approach presented in this chapter.

In order to illustrate the practical effectiveness of theorem proving in the domain of analyzing real-time systems, we have utilized it in this chapter to conduct the functional verification and performance analysis of a Stop-and-Wait protocol using the HOL theorem prover. A higher-order-logic specification for the Stop-and-Wait protocol is presented, with the noise effect modeled as a random variable. We also outlined the major steps in the verification of performance related theorems. The most significant result is the verification of the classical average message delay relation for the Stop-and-Wait protocol in HOL. To the best of our knowledge, formal verification of the average message delay relation for the Stop-and-Wait protocol cannot be handled by any other formal technique. Because of the fact that the Stop-and-Wait protocol bears most of the essential characteristics of the present day real-time systems, these results clearly demonstrate the usefulness of the proposed performance analysis approach.

The main limitation of the higher-order-logic theorem proving based performance analysis approach is the associated significant user interaction, i.e., the user needs to guide the proof tools manually since we are dealing with higher-order-logic. In the analysis of the Stop-and-Wait protocol, presented in this chapter, we tried to minimize the effect of this inherent limitation by taking a number of decisions, such as, building upon existing HOL theories, whenever possible, and choosing the discrete time domain for the analysis, which allows us to use the powerful induction technique for verification and thus minimize the proof effort considerably. The formalization and verification presented in this paper translated to approximately 6000 lines of HOL code and we had to spend about 300 man-hours on this project. Because of the interactive nature of the analysis, the proposed approach should not be viewed as an alternative to methods such as simulation and model-checking for the performance analysis of real-time systems but rather as a complementary technique, which can prove to be very useful when precision of the results is of prime importance.

REFERENCES

Alur, R. (1992). *Techniques for Automatic Verification of Real-Time Systems*. PhD Thesis, Stanford University, Stanford, CA.

Amnell, T., Behrmann, G., Bengtsson, J., D'Argenio, P., David, A., Fehnker, A., et al. (2001). Uppaal - Now, Next, and Future. In Cassez, F., Jard, C., Rozoy, B., Ryan, M.D. (Eds.), *Modeling and Verification of Parallel Processes,* (LNCS Vol. 2067, pp. 99-124). Berlin: Springer.

Beyer, D., Lewerentz, C., & Noack, A. (2003). Rabbit: A Tool for BDD-based Verification of Real-Time Systems. In W.A. Hunt, Jr. & F. Somenzi (Eds.), *Computer Aided Verification,* (LNCS. Vol. 2725, pp. 122-125), Boulder, CO. Berlin: Springer.

Bucci, G., Sassoli, L., & Vicario, E. (2005). Correctness Verification and Performance Analysis of Real-Time Systems Using Stochastic Preemptive Time Petri Nets. *Transactions on Software Engineering*, *31*(11), 913–927. doi:10.1109/TSE.2005.122

Cardell-Oliver, R. (1992). *The Formal Verification of Hard Real-time Systems*. PhD Thesis, University of Cambridge, Cambridge, UK.

Church, A. (1940). A Formulation of the Simple Theory of Types. *J. of Symbolic Logic*, *5*, 56–68. doi:10.2307/2266170

Galambos, J. (1995). *Advanced Probability Theory*. New York: Marcel Dekker, Inc.

Gallasch, G., & Billington, J. (2006). A Parametric State Space for the Analysis of the Infinite Class of Stop-and-Wait Protocols. In *Model Checking Software*, (LNCS 3925, pp. 201-218). Berlin: Springer.

Garcia, A. L., & Widjaja, I. (2004). *Communication Networks: Fundamental Concepts and Key Architectures*. New York: McGraw-Hill.

Gordon, M. J. C & Melham T.F. (1993). *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge, UK: Cambridge University Press.

Gordon, M. J. C. (1989). *Mechanizing Programming Logics in Higher-order Logic*. *Current Trends in Hardware Verification and Automated Theorem Proving* (pp. 387–439). New York: Springer.

Harrison, J. (1998). *Theorem proving with Real Numbers*. Berlin: Springer.

Harrison, J. (2009). *Handbook of Practical Logic and Automated Reasoning*. Cambridge, UK: Cambridge University Press. doi:10.1017/CBO9780511576430

Hasan, O., Abbasi, N., Akbarpour, B., Tahar, S., & Akbarpour, R. (2009b). Formal Reasoning about Expectation Properties for Continuous Random Variables. In A. Cavalcanti & D. Dams (Eds.), *Formal Methods*, (LNCS 5850, pp. 435-450). Berlin: Springer.

Hasan, O., & Tahar, S. (2007). Formalization of Continuous Probability Distributions. In F. Pfenning (Ed.), *Automated Deduction*, (LNCS Vol. 4603, pp. 2-18). Berlin: Springer.

Hasan, O., & Tahar, S. (2008). Using Theorem Proving to Verify Expectation and Variance for Discrete Random Variables. *Journal of Automated Reasoning*, *41*(3-4), 295–323. doi:10.1007/s10817-008-9113-6

Hasan, O., & Tahar, S. (2009). Performance Analysis and Functional Verification of the Stop-and-Wait Protocol in HOL. *Journal of Automated Reasoning*, *42*(1), 1–33. doi:10.1007/s10817-008-9105-6

Hasan, O., & Tahar, S. (2009a). Formal Verification of Tail Distribution Bounds in the HOL Theorem Prover. *Mathematical Methods in the Applied Sciences*, *32*(4), 480–504. doi:10.1002/mma.1055

Havelund, K., & Shankar, N. (1996). Experiments in Theorem Proving and Model Checking for Protocol Verification. *Industrial Benefit and Advances in Formal Methods*, (LNCS 1051, pp. 662-681). Berlin: Springer.

Hurd, J. (2002). *Formal Verification of Probabilistic Algorithms*. PhD Thesis, University of Cambridge, Cambridge, UK.

Kwiatkowska, M., Norman, G., & Parker, D. (2007). Stochastic Model Checking. In M. Bernardo and J. Hillston (Eds.). *Formal Methods for Performance Evaluation,* Bertinoro, Italy, (LNCS, 4486, pp. 220-270). Berlin: Springer.

Kwiatkowska, M., Norman, G., Segala, R., & Sproston, J. (2002). Automatic Verification of Real-Time Systems with Discrete Probability Distributions. *Theoretical Computer Science*, *282*(1), 101–150. doi:10.1016/S0304-3975(01)00046-9

Milner, R. (1977). A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, *17*, 348–375. doi:10.1016/0022-0000(78)90014-4

Paulson, L. C. (1996). *ML for the Working Programmer*. Cambridge, UK: Cambridge University Press.

Wells, L. (2002). Performance Analysis Using Coloured Petri Nets. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems,* (pp. 217-222). Washington, DC: IEEE Computer Society.

ADDITIONALREADING

Billington, J., Gallasch, G., & Petrucci, L. (2005). Fast Verification of the Class of Stop-and-Wait Protocols Modelled by Coloured Petri Nets. *Nordic Journal of Computing*, *12*(3), 251–274.

Duflot, M., Fribourg, L., Herault, T., Lassaigne, R., Magniette, F., Messika, S., et al. (2004). Probabilistic Model Checking of the CSMA/CD Protocol using PRISM and APMC. *Workshop on Automated Verification of Critical Systems*, (pp.195-214). Elsevier Science.

Hasan, O. Abbasi & Tahar, S. (2009). Formal Probabilistic Analysis of Stuck-at Faults in Reconfigurable Memory Arrays; M. Leuschel and H. Wehrheim (Eds.), *Integrated Formal Methods, LNCS Vol.5423*, (pp. 277-291) Springer. Düsseldorf, Germany.

Hasan, O., & Tahar, S. (2007). Formalization of the Standard Uniform Random Variable. [Elsevier.]. *Theoretical Computer Science*, *382*(1), 71–83. doi:10.1016/j.tcs.2007.05.009

Hasan, O., & Tahar, S. (2007). Verification of Probabilistic Properties in the HOL Theorem Prover, J. Davies and J. Gibbons (Eds.), *Integrated Formal Methods, LNCS Vol. 4591*, (pp. 333-352). Springer. Oxford, UK.

Hasan, O., & Tahar, S. (2008). Performance Analysis of ARQ Protocols using a Theorem Prover, *International Symposium on Performance Analysis of Systems and Software*, (pp. 85-94). IEEE Computer Society. Austin, Texas, USA.

Hasan, O., & Tahar, S. (2009). Probabilistic Analysis of Wireless Systems using Theorem Proving. [Elsevier.]. *Electronic Notes in Theoretical Computer Science*, *242*(2), 43–58. doi:10.1016/j.entcs.2009.06.022

Suzuki, I. (1990). Formal Analysis of the Alternating Bit Protocol by Temporal Petri Nets. [IEEE.]. *Transactions on Software Engineering*, *16*(10), 1273–1281. doi:10.1109/32.60315