Verification of the WAP Transaction Layer Using the Model Checker SPIN

Yu-Tong HE
Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada L8S 4K1

Abstract

This report presents a formal methodology of formalizing and verifying the Transaction Layer Protocol (WTP) design in the Wireless Application Protocol (WAP) architecture. Corresponding to the Class 2 Transaction Service (TR-Service) definition and the Protocol (TR-Protocol) design, two models at different abstraction levels are built with a finite state automaton (FSA) formalism. By using the model checker SPIN, we uncover defects in a latest approved version of the TR-Protocol design, which can lead to deadlock, channel buffer overflow and unfaithful refinement of the TR-Service definition. As an extended result, a set of safety, liveness and temporal properties is verified for the WTP to be operating in a more general environment which allows for loss and re-ordering messages.

Contents

1 Introduction

This report presents an application of model checking methodology to verify a communication protocol design. Due to the vastness of research topics in formal methods and protocol engineering, it is necessary for us to use this first section to locate our research work in the related realms.

1.1 Background

1.1.1 Formal Methods

In [8], Clarke and Wing defined *formal methods* as mathematically-based languages, techniques and tools for specifying and verifying hardware and/or software systems (henceforth *systems*). Formal methods, however, can be used in any phase of the system development process as described in the *waterfall model* [43]. Namely, they can

be applied in the initial analysis, through system design, implementation, integration (or testing) and maintenance.

To explain what can be done with formal methods, we adopt the terminology in [42] with the aid of Figure 1. In this figure, the inscriptions on the arrows show examples of formal methods *activities*. The contents of each ellipse represents *descriptions* of the system at different levels. We will elaborate on these terminology as follows, and we will be consistent to this interpretation throughout the report.

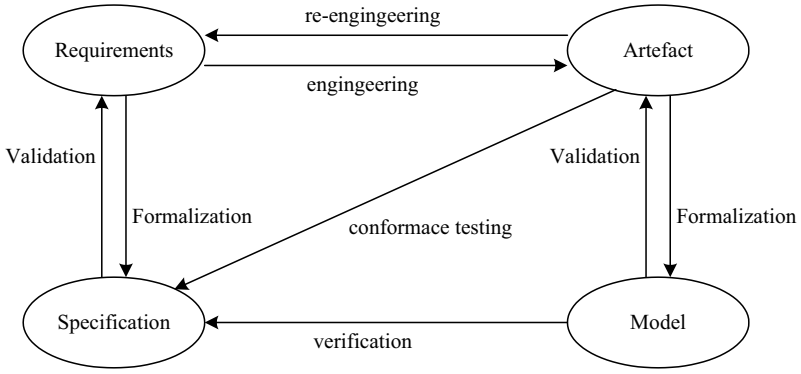


Figure 1: Formal methods activities

By *requirements*, we mean descriptions of intended behaviour and/or structure of the system. *Artefacts* are the designs, as well as the implementations, which realize the requirements. Requirements and artefacts are, in practice, *informal* descriptions of the system. *Engineering* is a process of deriving an artefact from requirements, while *re-engineering* is the reversed process. In this report, the research is restricted to the use of formal methods where both requirements and artefacts are given.

Formalization is the process of translating an informal description by using a formal language, which is mathematically defined by a formal syntax, associated semantics and satisfies relation [20]. Here, formalization is similar to what is called *formal specification* in [8], though, we use the term *specification* to refer to the resulting formal representation of the requirements. The term *model* refers to the formal representation of the artefact. Methods of formalization on which we will touch in this report include net theory (specifically, Coloured Petri Nets [33]), finite state automata (FSA) [29], process algebra (mainly CSP [21] and CCS [38]) and temporal logic [41].

Validation is used to establish whether the formalization is correct after abstractions and/or assumptions are made upon requirements or the artefact. In this report, validation is carried out by *manual* comparison of the formal and informal descriptions.

Verification is the task of establishing a formal relation between a model and specification by mathematical proof. Two distinguished approaches of verification are *model checking* and *theorem proving*. Roughly speaking, model checking is an automatic technique that relies on building a finite model of a system and checking whether a

property (expressed in the *specification*) holds by exploring the reachable state space of the model. In [8], model checking approaches are classified into two categories, based on whether the *specification* is expressed in temporal logic or given as an automaton. More related background will be introduced in Section 2. *Theorem proving* is the process of finding a proof of a property with an automated system, which formally defines a set of axioms and a set of inference rules.

Conformance testing is used to check if the functional behaviour of a given implementation is equivalent to the specification. The general method is to provide the implementation (viewed as a black box) with a series of input signals (called a *conformance test suite*) and observe the resulting output signals. The *implementation under test* passes the test only if all observed outputs match those prescribed by the formal specification. Formal methods for testing consist of methods for generating an efficient conformance test suite, applying the test suite to an implementation and evaluating the outcomes. Exhaustive tests of an artefact are possible only under a number of additional assumptions regarding the occurrence of errors, e.g. no implementation error can increase the number of reachable states [22]. Therefore, in most cases, only the presence of desirable behaviour can be tested for, not the absence of undesirable behaviour. This report does not deal with the topic of testing.

1.1.2 Protocol Engineering

This report focuses on an artefact which is a *protocol*. Protocols are sets of rules that govern the message exchanges of concurrent processes in distributed systems. Holzmann [22] proposed five essential elements of a protocol definition, which are:

1. The *service* to be provided by the protocol;
2. The *assumptions* about the environment in which the protocol is executed;
3. The *vocabulary* of messages used to implement the protocol;
4. The *format* of each message in the vocabulary;
5. The *procedure* rules guarding the consistency of message exchanges.

We will give examples of each element using a real world protocol design in a later section, though, our main interest is on the fifth item.

Protocol engineering [36] involves the application of formal methods to the design of protocols; it is a process of incrementally refining the user requirements until a target implementation is obtained. When the user requirements have been identified, the complex system for providing the distributed applications is usually organized into a layered (or hierarchical) structure, which enables easier design and maintenance of smaller subsystems. A well-known layered structure is the Reference Model for Open Systems Interconnection (OSI) [31]. The independent engineering process applied to each layer is: define the service to be provided by the layer; design a protocol that will

provide the service; and generate a target implementation from the service and protocol design.

We use Figure 2 to illustrate what formal activities can be carried out in the protocol engineering process of each layer. The artefact in Figure 1 is presented separately in Figure 2 as protocol design and implementation. Requirements are specific to service provided by the protocol. Implementation results in the target code. Related formal activities investigate the functional behaviour and evaluate the performance of the protocol, though, we focus mostly on the behavioural aspect.

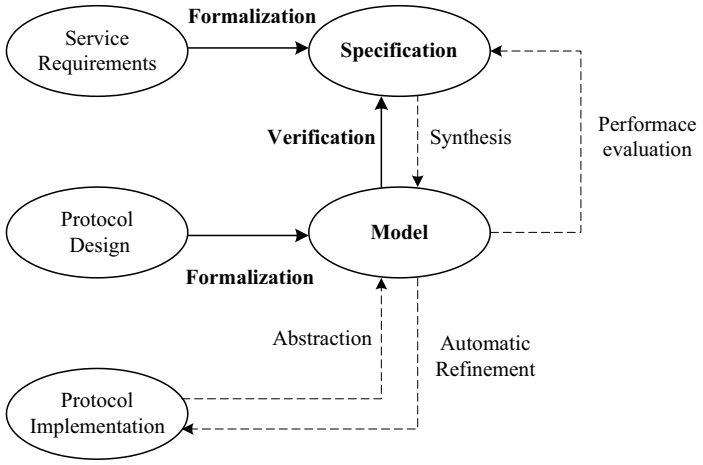


Figure 2: Protocol engineering activities

For a newly-built protocol, it is desirable that target code can be automatically generated from a protocol design, which conforms to the specified service while being error-free. One approach to ensure such protocol design is *synthesis* methodology [46], which aims "ideally" to derive from the service description a protocol design that can be proven correct by construction. However, we pay more interest to another alternative, i.e., to *verify* that an existing protocol provides the defined service by comparing the service languages generated from their formalized descriptions. This will be discussed more in the later part of the report.

On the other hand, for implementations in use which were not generated from a verified design, there has been growing demands of checking their conformance and potential behavioural errors to ensure safety. To develop a practical method for such a purpose, we need a much higher level of *abstraction*, which is a challenging task.

To be suitable for protocol engineering, related formal methods must possess several features:

- intuitive modelling of concepts inherent in communication protocols, most notably concurrency and non-determinism,

- the ability to specify systems at different levels of abstraction (e.g. service and protocol), and
- adequate support from computer tools.

1.2 Motivation

The idea of allowing people to communicate from almost anywhere and at anytime is fuelling the demand for wireless networking and mobile data service. The needs of wireless subscribers, however, are different from those of wire-line Internet users, due to the strong constraints in the mobile data service. Such constraints are characterized by the limited display and computation power of the wireless devices, as well as the limited bandwidth and high latency of the networks. The Wireless Application Protocol (WAP) [56] defines an architecture that aims to solve the transport and content problems of the constrained wireless environment, while providing a complete standard for wireless applications that includes a wireless counterpart of TCP/IP and a framework for telephony integration. The Transaction layer [60] in WAP defines a protocol for performing short request/response transactions between two entities. This Transaction Protocol is a component of browsing-type applications, where mobile users request information and receive a response from a server.

WAP is a *de-facto* global standard because in 2000 it became "a service with 8 million users, 10,000 applications and over 40 phone models" [56] in North America and Europe. Critical to the inter-operability of WAP between different networks and devices is ensuring that the communication protocols are unambiguous and functionally correct. So, the objective of this report is to verify the design of the WAP Transaction Layer (WTP) and determine whether the design is well-formed. By *well-formed* [22], we mean:

1. The protocol is not over-specified, i.e. it does not contain unreachable states.
2. The protocol is not under-specified, which may, for instance, cause unspecified receptions[1] during its execution.
3. The protocol is bounded, i.e. it cannot overflow given system limits.
4. The protocol is robust, which herein means it can recover from the errors caused by the protocol environment.
5. The protocol is functionally consistent, which means there is no improper termination (which includes deadlocks) and no bad cycles (which include livelocks and violations of temporal requirements) during its execution.

[1] An unspecified reception occurs if a message arrives when the receiver does not expect it or cannot respond to it.

6. The protocol is conformable, i.e., is a faithful refinement of the service defined.

The verification of WTP involves comparing the Transaction Protocol to the Transaction Service. Our approach is:

- to formalize the Transaction Service (TR-Service) and Transaction Protocol (TR-Protocol), from their narrative descriptions and state tables to formal system models;
- to formalize the above *well-formed* properties to classified formal specifications;
- to verify, through *model checking*, the system models against formal specifications, for a set of configurations determined by protocol parameters.

We choose the protocol modelling language PROMELA and the model checker SPIN for this verification task, because:

- They are formal methods that support the desirable features for protocol engineering;
- They have been used to model and analyze a range of protocols including Sliding-window Protocol [28, 52], Bounded Retransmission Protocol [10], Routing Information Protocol [39], X.509 Authentication Protocol [34]. (For more examples, we refer to [26, 49].)

1.3 Related Work

Protocol verification has been an active area, pushed by the development of formal method theories and automated software tools. As mentioned above, Stahl [52] proved the safety of the sliding-window protocol used in MASCARA (a special medium access control protocol designed for wireless ATM), with SPIN and PVS. He took advantage of the *data independency* property of the data-transmission protocol, to establish its correctness with the full window size of 16 (which can generate a prohibitive state space). However, the PROMELA model of the protocol was built directly from a textual description of the sliding-window mechanism, so that the modelling procedure sometimes seems more like programming with PROMELA (due to its closeness to ANSI-C) than formalizing with mathematical definitions. Verification of service refinement is limited, since a formal description of the service definition is seldom made, especially when the protocol model is abstracted from an implementation as was done in [12]. These problems commonly appear in other published work we are aware of.

Gordon's work [18] is one of the few publications which adopts a formal protocol engineering methodology. He has verified an old version of the WTP TR-Protocol design [59] against some requirements like: faithful refinement of TR-Service, successful termination, absence of livelocks and absence of unexpected dead transitions. He used

Coloured Petri Nets in the system modelling, but the graphical model is restricted to a protocol environment of *lossless* channels, which is not the general case in wireless applications. To obtain a correct verification result which is applicable to GSM SMS (Short Message Service in Global System for Mobile communication) networks, it required several hours for a single verification run. With our work, we will see it pays off to reformulate a *new* version of the WTP design [60] in PROMELA. Using the model checking tool SPIN, we can express more user requirements with greater flexibility, which enable us to find a new deadlock error not revealed before. We are also able to extend the verification results to more general protocol environments, using considerably less runtime.

1.4 Report Outline

The remainder of this report consists of the following sections and appendices:

Section 2 introduces some background on FSA models, which are the formal basis of our protocol modelling. The automata-theoretical approach of model checking is reviewed, as well as the functionality of the SPIN tool. The verification and simulation algorithms adopted by SPIN are summarized in Appendix A. In addition, a verification procedure is proposed in this section as the guideline in our report for verifying a protocol design with SPIN.

Section 3 presents the primary work of this report. It includes (1) modelling both TR-Service and TR-Protocol with formal automata definitions and PROMELA syntax, and (2) specifying correctness requirements on safety, liveness and temporal behaviours with Linear-time Temporal Logic (LTL) and Büchi automata. The state table representations of the TR-Protocol design are shown in Appendix B, as an origin from which our formal models are built. Appendix C lists the PROMELA code and the GNU *m4* programs that produce the TR-Service and TR-Protocol models to be checked in SPIN.

Section 4 continues to describe the major outcomes of our verification experiment, which include defects uncovered in the TR-Protocol design. For a revised design, the correct verification results are extended to a more general protocol environment. This increases developers' confidence in implementing the protocol model for use in practical networks. A comparison with the Coloured Petri Nets method also shows the benefits of using SPIN for model checking.

Section 5 concludes this report with a summary of the contributions of the research and proposes some future work.

2 Preliminaries and Notations

In the first part of this section, we review the fundamental concepts of model checking as a verification method. Those concepts include linear-time temporal logic, the

automata-theoretical approach and the techniques to manage complexity during verification. Section 2.2 mainly deals with the model checking tool SPIN and its modelling language PROMELA. Finally in Section 2.3, we will propose the procedure for verifying a model with regard to a set of requirement specifications using SPIN.

2.1 Model Checking

Generally speaking, a model checking problem is finding if a formula ψ is true in a model M. The model is generally a universe (over which variables may take values) together with a set of relations [48]. More specifically, a model can be formulated in an automaton (or an equivalent *temporal structure*), which represents the hardware/software system to be verified. The formula (built from variables, operators and relations), when used to specify some requirement or property, can be expressed in temporal logic.

One approach to solve the model checking problem is based on fixed-point computation, with respect to the temporal logical formula. Vardi and Wolper [54] showed that the temporal logic (specifically, Linear-time Temporal Logic) model checking problem could be recast in terms of automata, i.e., both the model and the formula are given as automata so that the model checking problem reduces to the automata-theoretic problem of proving language containment. In this section, we will give a brief introduction of the automata approach as the mathematical basis of our verification work. But first, we begin with the concept of temporal logic, as later we will use its notations to express the correctness requirements.

2.1.1 Linear-time Temporal Logic

A temporal logic deals with the notion of *paths* (i.e., successions of states in the model) which describes possible behaviours of the system. Its semantics is given with respect to a temporal structure (also called a *Kripke structure*).

Temporal Structure Suppose a system has a set of variables V, which range over a finite set D. A temporal structure defined for that system is a quintuple $M := (\mathcal{V}, S, S_0, R, L)$ consisting of:

1. a set of atomic propositions $\mathcal{V} = \{v = d \mid v \in V,\, d \in D\}$;
2. a finite set of *states* S, which is defined as $S : V \to D$ and the atomic proposition $v = d$ is true in state s if $s(v) = d$;
3. a set of *initial states* S_0, where $S_0 \subseteq S$;
4. a *transition relation* $R \subseteq S \times S$, where $\forall s \in S, \exists s' \in S, (s, s') \in R$, i.e., R is *total*; and
5. a *label function* $L : S \to 2^{\mathcal{V}}$, i.e., $L(s)$ contains all *atomic propositions* true in state s.

If R is a function, i.e., for every state there exists exactly one successor state, then we call M a *linear* structure.

Figure 3 shows an example of a temporal structure M. We have $V = \{p_1, p_2\}$, $D = \{0, 1\}$, $S = \{s_1, s_2\}$ where $s_1(p_1) = 1$, $s_1(p_2) = 0$, $s_2(p_1) = 0$, $s_2(p_2) = 1$, $S_0 = \{s_1\}$ indicated by a dashed arrow, $R = \{\langle s_1, s_2\rangle, \langle s_2, s_1\rangle, \langle s_1, s_1\rangle\}$, $L(s_1) = \{p_1 = 1, p_2 = 0\}, L(s_2) = \{p_1 = 0, p_2 = 1\}$.

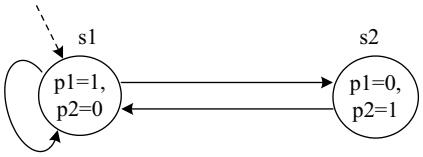


Figure 3: A Kripke structure M representing a model of $\mathcal{GF}(p_1 = 1)$

A *path* of a structure M is a succession of states starting at an initial state $s_0 \in S_0$, and $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. A path π is

- finite, if $\pi := s_0 s_1 ... s_i s_{i+1} ... s_n$ and $R(s_n) = s_n$, or
- infinite, if no such s_n exists.

With π^i $(i \geq 0)$, we denote a *suffix* of a path π starting at state s_i, i.e. $\pi^i = s_i s_{i+1} ....$

The notion of *computation* of M is the sequence $L(s_0)L(s_1)....$ Actually, the difference between path and computation is quite small if the labelling function is known.

For a non-deterministic system, there are two time models for the system behaviour. One is a *linear time* model, which assumes that for each time instance there exists exactly one successor time point, so time is modelled as a (linear) single path. The other is a *branching time* model, which allows several successors of each time instance so that time is modelled by branching, tree-like structures. Linear-time Temporal Logic (LTL) and Computation Tree Logic (CTL) are two prominent representatives to express system behaviours with the above two time models, respectively. We will focus mostly on LTL, by which specified behaviour may comprise a set of paths instead of being a computation tree as in branching time logic.

Linear-time Temporal Logic First we recall the basic operators for LTL in the form given in [14]. Besides boolean connectives (e.g., $\neg$ denoting negation, $\vee$ for disjunction, $\wedge$ for conjunction, $\Rightarrow$ for implication), LTL provides temporal operators to make statements about linear-time behaviour. Frequently used operators include: $\mathcal{G}$ (*always*), $\mathcal{F}$ (*future*), $\mathcal{X}$ (*next*) and binary connective $\mathcal{U}$ (*strong until*). The following syntax and semantics definition of LTL are given in terms of a minimal operator set, which includes only $\neg$, $\vee$, $\mathcal{X}$, $\mathcal{U}$.

The **syntax** of an LTL formula is given by the following rules:

1. If $\psi \in \mathcal{V}$, then ψ is a formula.

2. If ϕ and ψ are formulae, then so are $\neg\psi$, $\phi \vee \psi$, $\phi \mathcal{U} \psi$ and $\mathcal{X}\psi$.

Given the above basic operators and the notation $\top$ for $\psi \vee \neg\psi$, we can treat other temporal operators as syntactic abbreviations as follows:

$$\begin{aligned} \mathcal{F}\psi &\equiv \top \mathcal{U} \psi \\ \mathcal{G}\psi &\equiv \neg(\mathcal{F} \neg\psi) \end{aligned} \tag{1}$$

The **semantics** of an LTL formula are defined with respect to a linear structure M. Given a path $\pi := s_0 s_1 ...$ in M, we define the satisfaction relation $\models$ in the following way [54]:

1. $\pi \models \psi$ iff $\psi \in L(s_0)$, for $\psi \in \mathcal{V}$,

2. $\pi \models \neg\psi$ iff not $\pi \models \psi$,

3. $\pi \models \phi \vee \psi$ iff $\pi \models \psi$ or $\pi \models \phi$,

4. $\pi \models \mathcal{X}\psi$ iff $\pi^1 \models \psi$,

5. $\pi \models \phi \mathcal{U} \psi$ iff $(\exists k \geq 0)$ $\pi^k \models \psi$ and $(\forall j : 0 \leq j < k)$ $\pi^j \models \phi$

We may give the semantics of $\mathcal{F}\psi$ and $\mathcal{G}\psi$ more explicitly as:

1. $\pi \models \mathcal{F}\psi$ iff $(\exists k \geq 0)$ $\pi^k \models \psi$,

2. $\pi \models \mathcal{G}\psi$ iff $(\forall k \geq 0)$ $\pi^k \models \psi$.

Here we introduce another temporal operator $\mathcal{W}$ (*weak until*) which can be expressed as:

$$\phi \mathcal{W} \psi \equiv (\mathcal{G}\phi) \vee (\phi \mathcal{U} \psi) \equiv (\mathcal{F}\neg\phi) \Rightarrow (\phi \mathcal{U} \psi) \tag{2}$$

The semantics of $\phi \mathcal{W} \psi$ are a disjunction of $\mathcal{G}\phi$ and $\phi \mathcal{U} \psi$, which does not require that there exists a time instance at which ψ holds. But in such a case, ϕ must always be true.

The Model Checking Problem Given a temporal structure M, a state $s \in S$ and a temporal logic formula ψ, the model checking problem is to determine whether the following satisfaction relation holds:

$$M, s \models \psi. \tag{3}$$

If Formula (3) holds for an initial $s \in S_0$, such a structure M is called a *model* of ψ. To determine whether M is a model of an LTL formula ψ, we are checking if ψ is satisfied on *all* of the paths starting at s. Recalling Figure 3, we can see $M, s_1 \models \mathcal{G}\mathcal{F}(p_1 = 1)$, since $\forall\pi$ starting at s_1, $\forall i \geq 0, \exists k \geq i$ *s.t.* $\pi^k \models (p_1 = 1)$.

2.1.2 Finite State Automata

A finite automaton is a 5-tuple $A := (\Sigma, S, s_0, \delta, F)$, where

- Σ is a finite nonempty *alphabet*, with elements called *symbols*,
- S is a finite nonempty set of *states*,
- $s_0 \in S$ is an *initial* state,
- δ is a *transition function* $\delta : S \times \Sigma \to 2^S$,
- $F \subseteq S$ is the set of *final* states (also called *accepting* states).

An automaton is essentially an edge-labelled directed graph: the states are the nodes and the edges are labelled by symbols. Since $\delta(s, a)$ is the set of states that A can move into when it reads the symbol $a \in \Sigma$ in state s , then $t \in \delta(s, a)$ means there is an edge from s to t labelled with a. If a is never recognized at the state s, then $\delta(s, a) = \emptyset$.

The automaton A is *deterministic* if $|\delta(s, a)| \leq 1$ for all $s \in S$ and $a \in \Sigma$, in which case the transition function becomes $\delta : S \times \Sigma \to S \cup \emptyset$. But we are interested in the case when A is *non-deterministic* with $|\delta(s, a)| > 1$ for one or more states.

A *word* ω is a sequence of symbols. Let Σ^* be the set of words including the empty sequence ε, we define the extended transition function $\delta^* : S \times \Sigma^* \to 2^S$ as follows:

1. $\delta^*(s, \varepsilon) = \{s\}$.
2. $\delta^*(s_i, \omega_{i(i+1)}) = \delta(s_i, a_i)$, where $\omega_{i(i+1)} = a_i$ is a word of length 1.
3. Given two finite words $\omega_{ij} = a_i...a_j$ and $\omega_{i(j+1)} = \omega_{ij}a_{j+1} = a_i...a_{j+1}$ with arbitrary lengths, $(\forall j : j > i)\ \delta^*(s_i, \omega_{i(j+1)}) = \{\delta(s_k, a_{j+1}) \mid \forall s_k \in \delta^*(s_i, \omega_{ij})\}$.

The resulting sequence $r = s_0 s_1...s_n$ is called a *run* of the automaton A on the word $\omega_{0(n-1)}$. For a given word, a nondeterministic automaton can have many runs. The word ω is *accepted* by A if there exists a run with the last state $s_n \in F$. The finitary *language* of A, denoted as $\mathcal{L}(A)$, is the set of such accepted words, i.e., $\mathcal{L}(A) = \{\omega \in \Sigma^* \mid \delta^*(s_0, \omega) \cap F \neq \emptyset\}$.

A *Büchi automaton* [53] is one type of automata viewed on infinite words (i.e., $\omega = a_0 a_1...a_i...$). The run $r = s_0 s_1...s_i...$ of an automaton A is an infinite sequence. The infinite word ω is *accepted* by a Büchi automaton if there is a run with some accepting state that repeats in the run r infinitely often, i.e., $(\forall i \geq 0, \exists j > i)\ s_{j+1} \in \delta^*(s_i, \omega_{ij})$ and $s_{j+1} \in F$, where $\omega_{ij} = a_i...a_j$ is a subsequence of ω. The infinitary language of the Büchi automaton A, denoted $\mathcal{L}_\omega(A)$, is the set of such accepted words.

A Büchi automaton A is *nonempty* if $\mathcal{L}_\omega(A) \neq \emptyset$. The problem of determining whether the automaton is nonempty is equivalent to a graph reachability problem. This is justified by the fact that A is nonempty if and only if it has some state $s \in F$ that is

both reachable from the initial state and reachable from itself. In other words, the graph representing A has a cycle of nodes that contains the state s; so we call it an *accepting cycle*. Formally, suppose a run $r = s_0 s_1 ... s_{i+1} ... s_{j+1} ...$ of A is over some accepted word $\omega = a_0 a_1 ... a_i ... a_j ...$, then there exists some state $s \in F$ such that $s = s_{i+1} = s_{j+1}$, where $0 < i < j$. Thus s is connected to s_0 via $\delta^*(s_0, \omega_{0i})$ and is also connected to itself via $\delta^*(s_i, \omega_{ij})$. Conversely, given both connections, a word $\omega = a_0 a_1 ... (a_i ... a_j) ...$ (in which the sequence $(a_i ... a_j)$ repeats infinitely often) is accepted, since the state $s_{i+1} = s_{j+1} \in F$ appears infinitely often.

Product Automaton Suppose we have a finite number of automata $A_1 ... A_n$, we can build a product automaton A to model the concurrent state changes of those automata. More specifically, with $A_i = (\Sigma_i, S_i, s_{0_i}, \delta_i, F_i)$, we may have $A = (\Sigma, S, s_0, \delta, F)$ where

- $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$ is the union of each component's alphabet,
- $S = \prod_{i=1}^{n} S_i$ is the cartesian product of each component's state space,
- $s_0 = (s_{0_1}, ..., s_{0_n})$
- δ is defined by $(s'_1, ..., s'_n) \in \delta((s_1, ..., s_n), a)$ iff
  - $s'_i \in \delta_i(s_i, a)$ for each i such that $a \in \Sigma_i$ and
  - $s'_i = s_i$ for each i such that $a \notin \Sigma_i$,

  i.e., those automata having common symbols can move to next states together, whereas symbols exclusive to one automaton correspond to interleaved state changes.
- $F = \prod_{i=1}^{n} F_i$ is the cartesian product of each component's final state.

Figure 4 shows an example of an automaton $A = A_1 \times A_2$ constructed according to the definition above. Alphabet elements are edge labels and final states are shown as doughnuts. In part c) of the figure, the transition from state (s_1, s_2) to (s'_1, s'_2) models the concurrent state changes in components A_1 and A_2, while other transitions show interleaved state changes. All the states in A are reachable from the initial state (s_1, s_2), however, if we change the above definition of transition relation δ to allow only synchronized transitions, the *reachable* state space of A usually has a size smaller than the cartesian product of every component's state space (e.g., see Figure 8).

2.1.3 Model Checking LTL with Automata

In the automata approach, both the system model and the requirements are given as Büchi automata. The original temporal structure $M := (\mathcal{V}, S, s_0, R, L)$ (with a unique initial state) can be viewed as a Büchi automaton $A_M := (\Sigma_M, S_M, s_{0M}, \delta_M, F_M)$ with

- $\Sigma_M = 2^{\mathcal{V}}$,

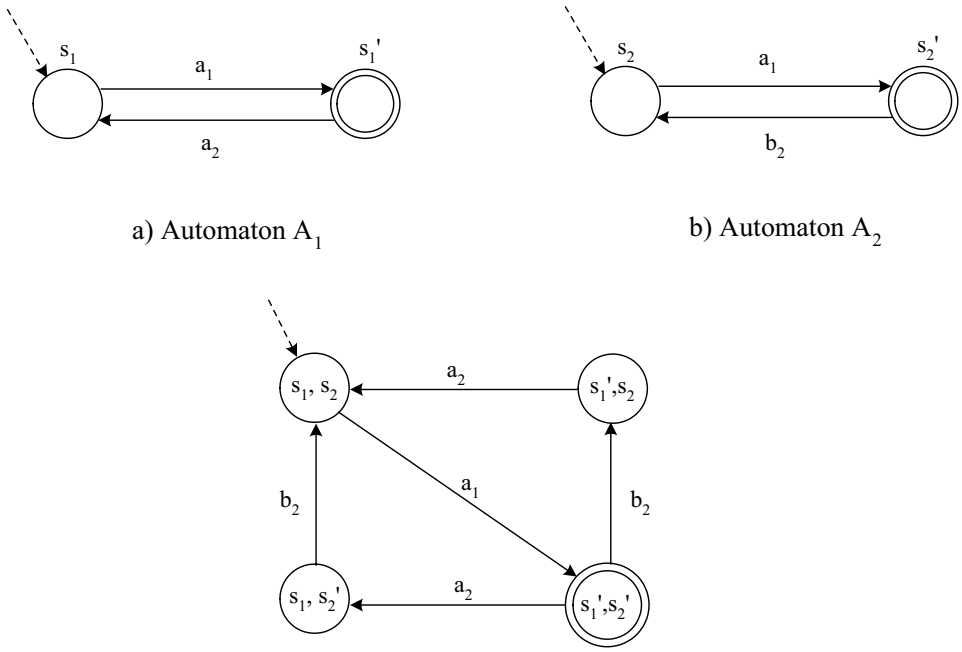


Figure 4: Example of a product automaton

- $S_M = S$,
- $s_{0_M} = s_0$,
- $\delta_M : S_M \times \Sigma_M \to 2^{S_M}$, where $s' \in \delta_M(s, a)$ iff[2] $(s, s') \in R$ and $L(s) = a \in \Sigma_M$, i.e., each transition corresponds to evaluations of system variables, and
- $F_M = S_M$, i.e., any infinite run of A_M is accepted.

So, the language $\mathcal{L}_\omega(A_M)$ is the set of *all* computations[3] of M, which gives the possible behaviours of the system.

Figure 5 shows an example of converting the temporal structure M in Figure 3 to an automaton A_M. We have $V = \{p_1, p_2\}$, $D = \{0, 1\}$, $\Sigma_M = \{a_1, a_2, a_3\}$ with $a_1 \cup a_2 \cup a_3 = 2^{\mathcal{V}}$, and $F_M = S_M$. The transition relation δ_M is defined as follows,

s	$\delta_M(s, a_1)$	$\delta_M(s, a_2)$	$\delta_M(s, a_3)$
s_1	$\{s_2, s_1\}$	$\emptyset$	$\emptyset$
s_2	$\emptyset$	$\{s_1\}$	$\emptyset$

[2]If s has no succession, we modify R so that $R(s, s)$ holds.

[3]If a computation of M is finite, we can transform it into an infinite one by letting the terminating state repeat forever so that it is accepted by A_M.

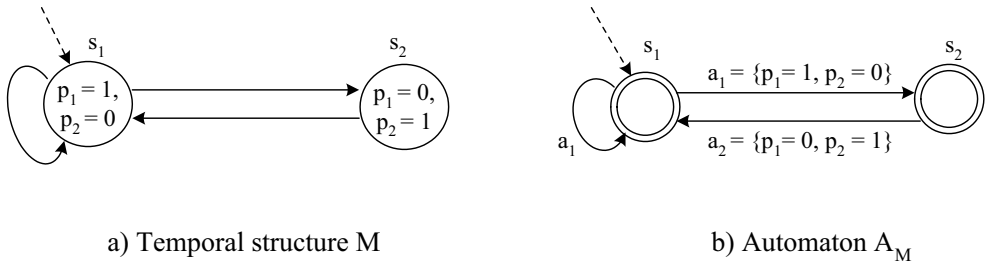


Figure 5: An example relating a temporal structure to an automaton

Note that $|\delta_M(s_1, a_1)| = 2$, meaning A_M can either stay in state s_1 or go to s_2 upon reading a_1, which models the non-determinism in M. Assuming $s_0 = s_1$, a run of the automaton can be viewed as executing the evaluations of system variables p_1, p_2, which initially are set to $p_1 = 1, p_2 = 0$. The language of A_M contains all the computations in which $p_1 = 1$ holds continuously (and of course, infinitely often).

Given an LTL formula ψ with respect to the temporal structure M, we can build a Büchi automaton $A_\psi := (\Sigma_\psi, S_\psi, s_{0\psi}, \delta_\psi, F_\psi)$, where

- $\Sigma_\psi \subseteq 2^{\mathcal{V}}$, and
- $|S_\psi| \leq 2^{O(|\psi|)}$ ($|\psi|$ is the number of sub-formulas in ψ),

such that $\mathcal{L}_\omega(A_\psi)$ is exactly the set of computations that satisfies ψ. A general method of constructing an automaton for a temporal formula is given in [55].

Figure 6 shows a Büchi automaton A_ψ constructed with respect to the LTL formula $\psi = \mathcal{GF}(p_1 = 1)$. We have $\Sigma_\psi = \{\{p_1 = 1\}, \overline{\{p_1 = 1\}}\}$ (where $\overline{\{p_1 = 1\}}$ means any symbols other than $\{p_1 = 1\}$), $S_\psi = \{t_1, t_2\}$, $s_{0\psi} = t_1$, $F_\psi = \{t_2\}$, and the transition function δ_ψ is defined as follows,

s	$\delta_\psi(s, \{p_1 = 1\})$	$\delta_\psi(s, \overline{\{p_1 = 1\}})$
t_1	$\{t_2, t_1\}$	$\{t_1\}$
t_2	$\{t_1\}$	$\{t_1\}$

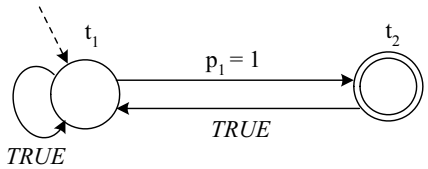


Figure 6: Büchi automaton A_ψ for the LTL formula $\psi = \mathcal{GF}(p_1 = 1)$

In Figure 6, we use $TRUE$ to denote that the automaton A_ψ can accept either $\{p_1 = 1\}$ or any symbol other than $\{p_1 = 1\}$. So A_ψ can stay in the initial state t_1 forever, or choose to move to t_2 upon reading $\{p_1 = 1\}$. But when in t_2, it has to move back to t_1 after the next transition no matter what the next symbol might be. Thus A_ψ accepts all the computations in which $p_1 = 1$ holds in the future for all the time points.

Then the model checking problem of Formula (3) reduces to the automata-theoretic problem of language containment

$$\mathcal{L}_\omega(A_M) \subseteq \mathcal{L}_\omega(A_\psi), \tag{4}$$

i.e., $\forall\omega \in \mathcal{L}_\omega(A_M)$, $\omega \in \mathcal{L}_\omega(A_\psi)$, which formalizes the notion that every possible behaviour of the system is suitable according to the requirement. Given $\overline{\mathcal{L}_\omega(A_\psi)}$, the complement of $\mathcal{L}_\omega(A_\psi)$, Formula 4 is equivalent to

$$\mathcal{L}_\omega(A_M) \cap \overline{\mathcal{L}_\omega(A_\psi)} = \mathcal{L}_\omega(A_M) \cap \mathcal{L}_\omega(A_{\neg\psi}) = \emptyset. \tag{5}$$

So, we can build a product automaton A such that $\mathcal{L}_\omega(A) = \mathcal{L}_\omega(A_M) \cap \mathcal{L}_\omega(A_{\neg\psi})$ [6, 17] and check whether $\mathcal{L}_\omega(A)$ is empty. With $A_M := (\Sigma, S_M, s_{0M}, \delta_M, F_M)$, where $F_M = S_M$ and $A_{\neg\psi} := (\Sigma, S_{\neg\psi}, s_{0\neg\psi}, \delta_{\neg\psi}, F_{\neg\psi})$, we have the *synchronous* product automaton $A := (\Sigma, S, s_0, \delta, F)$ with

- $S = S_M \times S_{\neg\psi}$,
- $s_0 = (s_{0M}, s_{0\neg\psi})$,
- $\delta : S_M \times S_{\neg\psi} \times \Sigma \to 2^{S_M \times S_{\neg\psi}}$ defined by $(s_j, t_j) \in \delta((s_i, t_i), a)$ iff $s_j \in \delta_M(s_i, a)$ and $t_j \in \delta_{\neg\psi}(t_i, a)$, and
- $F = S_M \times F_{\neg\psi} = F_M \times F_{\neg\psi}$.

Intuitively, a run of the automaton A over an input word can be viewed as two tracks r_M and $r_{\neg\psi}$, over A_M and $A_{\neg\psi}$ respectively. Whenever a track goes through an accepting state, the run shifts to the other track. The run on $r_{\neg\psi}$ shifts to r_M whenever it visits an accepting state in $F_{\neg\psi}$, however, since each state in r_M is an accepting state, the run will shift back to $r_{\neg\psi}$ immediately. Thus each transition in A_M is synchronized with a transition in $A_{\neg\psi}$ (as described in item 3 above). This guarantees, with the final state of A defined as $F_M \times F_{\neg\psi}$, that a word is accepted by A iff both tracks of the run go through their accepting states infinitely often, i.e., $\mathcal{L}_\omega(A) = \mathcal{L}_\omega(A_M) \cap \mathcal{L}_\omega(A_{\neg\psi})$. So any accepted word of A will give a computation of A_M that is accepted by $A_{\neg\psi}$, i.e., a counterexample of a path that does not satisfy the formula ψ. If A is empty, then none of the computations of the model M falsifies the specification ψ, i.e., the specification holds for the model.

Figure 7 shows a Büchi automaton $A_{\neg\psi}$ with respect to the LTL formula $\psi = \mathcal{GF}(p_1 = 1)$. The transition function δ' in $A_{\neg\psi}$ is defined as follows,

s	$\delta'(s, \{\neg(p_1 = 1)\})$	$\delta'(s, \overline{\{\neg(p_1 = 1)\}})$
t_1	$\{t_2, t_1\}$	$\{t_1\}$
t_2	$\{t_2\}$	$\{\emptyset\}$

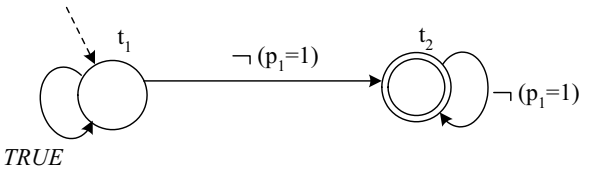


Figure 7: Büchi automaton $A_{\neg\psi}$ for the LTL formula $\psi = \mathcal{GF}(p_1 = 1)$

We can see it is exactly the automaton for the LTL formula $\phi = \mathcal{FG}(\neg(p_1 = 1))$. Actually, it is easy to show by using Formula (1) that

$$\neg(\mathcal{GF}p) = \neg(\neg\mathcal{F}\neg(\mathcal{F}p)) = \mathcal{F}(\neg(\mathcal{F}p)) = \mathcal{FG}(\neg p). \tag{6}$$

So the language of $A_{\neg\psi}$ contains only those computations in which $\neg(p_1 = 1)$ holds from some point on. Then as illustrated in Figure 8, it is easy to show that $\mathcal{L}_\omega(A) = \emptyset$, because there is no cycle in the graph that contains accepting states. So the automaton in Figure 5 is a model of the formula $\psi = \mathcal{GF}(p_1 = 1)$, i.e., $M, s_1 \models \psi$.

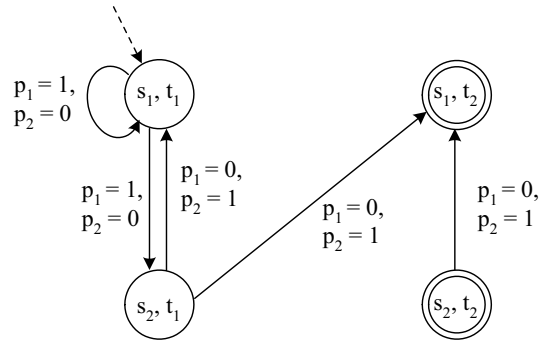


Figure 8: Büchi automaton $A = A_M \times A_{\neg\psi}$

2.1.4 Complexity Management in Model Checking

As described above, given an automaton A_M modelling the concurrent execution of finite-state transition systems A_i $(i = 1...n)$ and a temporal logic formula ψ, checking the problem of $A_M, s_0 \models \psi$ involves the following steps:

1. Compute the global behaviour of A_M by computing the interleaving product $\prod_{i=1}^{n} A_i$ of individual concurrent components.
2. Build the Büchi automaton $A_{\neg\psi}$ for the LTL formula ψ.
3. Compute the synchronous product $A = A_M \times A_{\neg\psi}$.
4. Check if A is empty or contains *accepting cycles*.

The product automaton A has $|S_M| \cdot 2^{O(|\psi|)}$ states, where S_M is the state space of the model A_M and $|\psi|$ is the size of the formula ψ. The *time upper bound* of checking whether A is empty (or equivalently $A_M \models \psi$) is given as $O(|S_M| \cdot 2^{O(|\psi|)})$ [54], which is polynomial in the size of the model and exponential in the size of the specification. The *space* upper bound is $O((\log|S_M|+|\psi|)^2)$. With the model A_M given as a product of small components $(A_1, ..., A_n)$, the model checking can be done using space of $O((\log|S_1| + \log|S_2| + ... + \log|S_n| + |\psi|)^2)$.

From Figure 8, we know that the *reachable* state space of A has a size smaller than that of the cartesian product $A_M \times A_{\neg\psi}$ or even smaller than $|S_M|$ (if no initial portion of $A_{\neg\psi}$ appears in A_M), unless many invalid behaviours appear in the system. In practice, however, the reachable portion of A can still easily become prohibitively expensive to construct exhaustively due to the constraints of computer memory. To combat this problem, a number of complexity management techniques have been developed regarding the verification procedure, state reduction and encoding methods, corresponding to which we give a brief introduction on the nested depth-first-search (NDFS) algorithm, partial order reduction and the bit-state hashing technique, respectively.

On-the-fly Model Checking Using NDFS To check if an automaton is not empty is equivalent to check if there is a cycle in the corresponding directed graph (see page 12). Courcoubetis *et al.* [9] proposed a nested depth-first-search algorithm using two interleaved DFS's: the first DFS is to determine which accepting states are reachable from the initial state s_0 and suspends upon finding one state $s \in F$; the second DFS (the nested one) then begins to find if the accepting state is also reachable from itself. If not, the first DFS resumes, otherwise the whole algorithm stops on finding one accepting cycle. Holzmann [25] implemented the algorithm with a single stack to store the path from s_0 to $s \in F$ and a single RAM (Random Access Memory) space to store the state information, e.g., descriptions of variables. If a cycle is detected, the states in the current stack correspond to a path violating the property being checked and can be exhibited immediately as a counterexample. What needs to be stored in the main memory is, therefore, only *part of* the reachable states that the depth-first search is currently exploring. It implies that the product automaton may be constructed "on-the-fly", i.e., as is needed while checking for its emptiness. In Appendix A.1, we give a more detailed description of Holzmann's NDFS algorithm.

It is also desirable to keep the property automaton A_ψ small and to avoid the exponential blowup that can occur in its construction. Gerth *et al.* [16] proposed an

algorithm to construct A_ψ in an "on-the-fly" fashion as well, in the sense that successors of a node in A_ψ are constructed only when they match the current states of the model A_M during the depth-first-search (verification) process. Thus it avoids the need to construct the entire A_ψ (or $A_{\neg\psi}$) if a violation of the checked property was found. Holzmann [26] implemented the algorithm in the model checker SPIN so that computing A_M (as the product of A_i) and the product of A_M and $A_{\neg\psi}$ can be done in the same step, and both A_M and A_ψ (or $A_{\neg\psi}$) are constructed on-the-fly during a depth-first search to check the emptiness of their language intersection.

To implement the NDFS algorithm, there are two types of memory requirement: *sequentially accessed memory* to implement a stack (possibly in secondary storage) and *randomly accessed memory* to store state space in the main memory of computer. Let N be the total number of states reached during the verification, SV (short for *state-vector*) be the number of bits required to distinguish each state for state comparison, and M be the total number of bits available in RAM. Clearly, the model checker exhausts its available memory after generating M/SV states. If $N \leq M/SV$, then the verification result is based on a full state space search, concluding $A_M \times A_{\neg\psi}$ is either empty or non-empty. Otherwise, we denote R as the real number of states required for an exhaustive search and the *state coverage* $M/(SV \times R)$ is less than 1, so only a partial search of the state space can be done in the verification.

Partial Order Reduction To perform an *exhaustive* verification within the memory constraint, two ways are possible to increase the ability to explore a large state space. One is to reduce the size of the state vector by using compression when encoding the states [27]. A second way is to generate a reduced state space which is equivalent in terms of a given correctness property [40]. The reduction is based on the observation that an LTL formula often remains valid with respect to different orderings of *concurrent and independent* events, which correspond to different interleaved state sequences in the depth-first search. So it first identifies cases where partial order reduction rules can be applied without affecting the truth or falsehood of properties, then generates only selective successors of the initial state (i.e., only one order of event executions) for actual verification. This static reduction method is not sensitive to decisions about variable orderings, while alternative reduction methods based on binary decision diagrams (BDD) have been shown to lack this feature.

Approximate Checking by Bit-State Hashing Targeting the partial search case, the bit-state hashing technique [24] can produce an *average coverage* close to 1 from a probabilistic point of view.

During a verification with 2-bit hashing, the state vector (with size SV) representing every reached state is mapped, by two independent hash-functions, to two hash values as indices in a bit-array (i.e., the hash table with a size of M). The two bits addressed in this way are set to 1 and together represent the newly reached state, so the maximum number of states that can be reached leaps from M/SV to $M/2$. If both

bits in the hash table have already been set to 1, a *hash collision* occurs and the state is matched to an already reached state. Then the DFS algorithm will not exploit any successors of this newly reached state. When $R > M/2$, this hashing automatically defines a randomized *partial* search. When $R \leq M/2$, however, there is still a possibility that parts of the reachable states are ignored, since the hash functions may map two different states to the same indices. That is why the bit-hashing technique is an approximation of a full state space search.

The probability of state losses (which is given by the average collision probability) is small if the *hash factor* H_f (defined as M/N, the ratio of the hash table size to the number of reached states[4]) is big. Based on empirical experiments of bit-state searching, it is recommended to trust only results for hash factors greater than 100.

By the nature of NDFS, the value of the parameter N cannot be predicted without performing the reachability analysis itself. As we will see in Section 4, however, both of the parameters N and SV are application dependent and can be manipulated by the model checker user to some extent. For example, SV relates to the *name space*[5] of the product automaton (i.e., the domain sizes for all system variables), and N depends on the abstraction level of system modelling. So by paying careful attention during the modelling of the system, we can either perform a full state space verification (if applicable) with fewer memory resources, or gain more confidence in the average coverage of a partial search using bit-state hashing.

2.2 Model Checker SPIN

SPIN [22, 26] is a verification tool that supports on-the-fly model checking of asynchronous process systems, including communication protocols and distributed software. A model of the system is specified in the tool's input language called PROMELA (PROcess MEta LAnguage) [50], on which SPIN (originally short for Simple Promela INterpreter) can perform simulations of the system's execution and verify the correctness requirements for the system using an automata-theoretical approach.

2.2.1 PROMELA Language

PROMELA is essentially a description language for extended finite state automata, whose syntax is loosely based on Dijkstra's guarded command language notation [11] and Hoare's CSP [21]. PROMELA supports three types of objects: *processes, variables, and channels*. A process is designed like a function in the C programming language, the body of which is a sequence of CSP-like statements that specify the behaviour of one distributed entity. So a process has an equivalent automaton model[6]. Basic data types supported for variables include booleans and integers, from which arrays and record

[4] In the case of 2-bit hashing, H_f is $M/(2N)$.

[5] Suppose, for instance, the name space has a size of $|U|$, then at least $\log|U|$ bits are needed to represent each state.

[6] In Section 3, we will give an automata definition of a distributed entity in the protocol.

structures can be built like in C. Channels, via which processes communicate with one another, are extended variables, in the sense that one channel is an array of ordered sets of variables.

```
mtype={msg};                          /* symbolic declaration of message data */
chan s2r = [1] of {mtype, bool};      /* each message with a data field and an ack bit */
chan r2s = [1] of {bool};             /* each acknowledgement with one bit */

proctype Sender() {
S0:    s2r!(msg)0 -> goto S1;
S1:    if
       :: r2s?1 -> goto S0
       :: r2s?0 -> goto S2
       fi;
S2:    s2r!(msg)1 -> goto S1
}

proctype Receiver() {
       goto R1;
R0:    r2s!0 -> goto R1;
R1:    if
       :: s2r?(msg)0 -> goto R0
       :: s2r?(msg)1 -> goto R2
       fi;
R2:    r2s!1 -> goto R1
}
```

Figure 9: PROMELA model of one version of the Alternating-Bit Protocol

Figure 9 gives a sample PROMELA model of the Alternating-Bit Protocol [2]. It has two processes, each declared in a `proctype` template, which model two terminals exchanging messages and acknowledgements on two asynchronous channels.

Line 1 in the figure shows an `mtype` declaration, which, in addition to the basic data type, allows the introduction of symbolic names for constants. Then, each channel is declared with a channel initiator `chan`, which specifies the bounded channel capacity (i.e., the array's dimensionality) as a constant, and the structure of messages as a comma-separated list of type names (i.e., an ordered set of variables).

Channel update is reflected by updating messages in it through send (!) or receive (?) operations, like the notation in CCS [38]. `s2r!(msg)0` denotes sending to the channel `s2r` a message with a header of `msg` (followed by data 0). The *send* operation is executable only when the channel is not full[7]. The *receive* operation is only executable when the channel is non-empty, and additionally, when constraints on the message fields are met. For example, `s2r?(msg)0` would be executable only if there is indeed a message having 0 in the second field at the head of the channel.

In the case of asynchronous communication between processes, messages can pile up in the channel, hence channels essentially become queues. To model a channel

[7]This is the default behaviour, however, SPIN can be instructed to lose messages which are sent to a full channel [50].

which is not necessarily a FIFO queue, PROMELA provides a *random receive* operation, `ch??msg`. It is executable as long as `msg` exists in the channel.

The sequential statements in each process are separated by semicolons (;) or arrows (`->`). Each option in an `if-fi` selection construct has a guard statement, whose executability determines if the option can proceed. For example, the `Receiver` will not move to state `R0` if there is no message sent from the `Sender`. If more than one guard statement is executable at the same time, the selection of an option is non-deterministic. In many cases, the guard statement is a boolean expression; it is executable when the expression equals true.

The appealing point of PROMELA is that it was specifically designed to have the expressiveness to specify precisely the features of a protocol. As we have seen from the above example (also described in Section 1.1.2), a communication protocol is a distributed algorithm that coordinates two or more entities to exchange messages via (a/synchronous) channels, where messages are defined with both message format and conditional sequences of messages. In PROMELA, independent *processes* are used to represent those distributed "entities"; *Boolean expressions* employed as "conditional sequences" guard the evolution of system state. *Assignments* in PROMELA update the state *variables* to "coordinate" processes. The I/O actions to "exchange" messages are succinctly abbreviated as ? (for receive) and ! (for send) through *channels*. "Messages" passing over channels are easily formatted by explicitly enumerating all fields in order (e.g., header followed by data). Moreover, the semantics of PROMELA preserve the concurrency and non-deterministic character of communication protocols.

2.2.2 The SPIN Tool

SPIN facilitates simulation and verification of a given PROMELA model. As described above, a concurrent system model in PROMELA consists of one or more user-defined process templates; each process template is then translated by SPIN into a *finite* automaton. An interleaving product of these automata then defines the global behaviour, as well as the state space, of the concurrent system. Different modes of simulation and verification can be performed on this product automaton A_M (written in a set of ANSI-C files).

SPIN supports random, interactive, or guided simulations of the automaton run. In *random* simulation, simulation runs with different random seeds will provide results that present nondeterministic choices on the automaton's transitions. On the other hand, applying the same random seed in each run will guarantee the same output. An *interactive* simulation will enable the user to make a nondeterministic choice at every execution step. If executions are deterministic, the simulation will proceed without user's intervention. In Appendix A.2, an abstract simulation algorithm used in SPIN is summarized.

Random and interactive simulation can be useful as a sanity check on the PROMELA model, to help remove apparent bugs introduced during system modelling. But by sim-

ulation alone, we can not convince ourself that the system is indeed error free. So we need to run a verification of correctness requirements against the system. If there is an error in one verification run, a *guided* simulation plays back the error trail that was produced by the NDFS algorithm as described on page 18.

When used as a verifier, SPIN can perform on-the-fly model checking, either in an exhaustive state-space-search manner or with an approximation method using bit-state hashing. In both modes, the model checking is based on the NDFS algorithm and partial order reduction to deal with the state explosion problem.

Correctness requirements can be specified in several ways for checking:

- For *temporal claims* expressed in LTL formulas[8], SPIN will mechanically translate an LTL formula ψ into PROMELA's equivalent of a Büchi automaton $A_{\neg\psi}$ and check the language intersection of A_M and $A_{\neg\psi}$ in an on-the-fly fashion. If no accepting cycle is found, the formula is satisfied.

- Some *safety* properties are usually identified by preservation of invariants and proper termination of the processes. They can be specified (as part of the model) with either PROMELA's `assert` statements or `end` labels. Then SPIN will conduct a single reachability analysis on the model A_M to trace those "bad" states where assertions are violated, or those states which are neither labelled (by the user) as end-states nor the ends of each process (e.g., a deadlock). This is more efficient than specifying the same properties separately with LTL formulas and then searching accepting cycles.

- The *liveness* property, which requires any state transitions not be postponed infinitely long, may be specified by labelling (the start of) those corresponding statements with PROMELA's `progress` labels. A *non-progress* cycle is a state sequence which only consists of states without `progress`-labels, so they may cause other statements starve. SPIN will detect, in a similar way as to find accepting cycles, any *non-progress* cycles in the model. But it is still more efficient than checking the liveness property with LTL.

In Section 3, we will discuss how to formulate the "well-formed" requirements proposed on page 6 in terms of the above properties for the WTP design in question.

2.2.3 A Brief Comparison of Model Checking Tools

In this section, we summarize the features of two model checking tools, SMV and UPPAAL, which are widely recognized in the academic world. We illustrate their limitations in the usability with respect to communication protocol modelling and providing user-friendly information during verification.

[8]For the partial order reduction rules applied to preserve liveness properties, no $\mathcal{X}$ (next) operator is allowed in formulating a requirement [23].

SMV The SMV system [7] is a *symbolic* model checking tool for verifying finite-state systems against specifications written in the temporal logic CTL. Its modelling language supports data types such as *Booleans, enumeration* types and *arrays*, compared with SPIN's additional support of *unsigned* (varied length byte), *structures* (records) and *channels*, but single *enumeration* type. SMV allows *modular* hierarchical descriptions, which may better reflect the architecture of the modelled system. CTL formulas are stated in the language construct `SPEC`, which allows one to express a rich class of temporal properties. Its model checking algorithm uses ordered binary decision diagrams (OBDDs) [4] to implicitly represent transition relations in a Kripke structure, which enable the tool to verify examples having more than 10^20 states [5].

These advantages come with some tradeoffs. SMV was designed for verification of synchronous systems (e.g., logic circuits), so there is no explicit support for asynchronous communication channels. From the experienced gained in [12], to model the same TCP/IP protocol and verify it against the livelock-free requirement, the SMV model generates a significantly larger state space (though demanding a small amount of RAM) and costs significantly more verification time, partly because extra maneuvers (consequently, more language components) are needed when using the SMV's input language for modelling.

As to its user-friendliness, the Carnegie-Mellon version[9] of SMV lacks rich support for simulating the system's execution. When a falsified CTL formula is detected by SMV, an error trace is printed out merely as the assignments of each variable/identifier in every state of the path. This primitive textual report makes it hard to read the counterexample and less helpful to find the source of a subtle error in a time-efficient manner. So we would like a tool whose interfaces are easy to use and able to provide information, in a convenient format, about the correctness of modelled system.

UPPAAL UPPAAL [35] is a verification tool targeting real-time systems. It consists of three parts: a description language, a simulator and a model checker. Its modelling language is a non-deterministic guarded command language (similar to PROMELA in some sense) with simple data types, but augmented with real-valued clock variables. So it is suitable for modelling systems with real-time constraints. A system is modelled in UPPAAL as a network of timed automata, the components of which communicate with synchronized channels (and shared variables), as opposed to asynchronous ones which are commonly used in communication protocols.

The model checker was designed to restrict to invariant and reachability analyses, the latter of which can be used to check deadlock. This restriction is crucial to the efficiency of UPPAAL's model checking performance, but it also makes it hard to verify some common liveness requirements (e.g., absence of livelock).

The temporal properties that UPPAAL can check are a subset of timed CTL. The temporal quantifies supported are, expressed in CTL syntax, $\mathcal{EF}$, $\mathcal{AF}$, $\mathcal{EG}$ and

[9]This is to differentiate it from the Cadence Berkeley Labs version, which has made progress in user interface.

$\mathcal{AG}$. But the more general quantifiers $\mathcal{AU}$ and $\mathcal{EU}$ (of which $\mathcal{EF}$ and $\mathcal{AF}$ are special cases, respectively) are not found. Nested CTL formulas are also not supported. These make it difficult for users to express some desirable requirements, e.g., the precise timing constraints in real-time systems, or the temporal claims in Section 3.5.

As to the model checking performance, UPPAAL needs about 4 times more time than SPIN to explore the same size of state space [3]. But UPPAAL enables symbolic state representation to manage a large state space. More investigation on its memory usage is needed.

2.3 Verification Procedure With SPIN

In this section, we describe the procedure involved in verifying a given protocol design with SPIN. We will use this procedure (which is inspired from Figure 3.1 in [44]) as a guideline for solving the verification problem we proposed in Section 1.

Shown as the `main()` procedure in Figure 10, the whole procedure consists of two phases: the formalization phase (procedure `Formalize()`) and the verification phase (procedure `Verify()`). It starts with the informal descriptions of both the protocol design and the properties of interest. The protocol is usually ready as documents provided after the design phase, but the requirements to be checked need to be abstracted from the design documents. The target of the verification is to model-check the formalized model M against all the properties in P, and correct the potential errors in the design, so that when finally $C = P$, we can ensure $\forall \psi \in P, M \models \psi_i$.

In the formalization phase, both the design and the requirements need to be turned into formal descriptions that the model checker SPIN can recognize. Usually, a feedback loop (line 14 to 17) based on (interactive) simulation of the PROMELA model M is helpful to improve the system modelling. Simulations, which are guided by the desired protocol's operations, can thus be used as a justification for having modelled things the right way. Equally important is to classify and specify the correctness requirements (as described in Section 2.2.2) using PROMELA syntax or linear temporal logic formulas. Actually, some basic safety requirements specified within the PROMELA model M are partly checked during the simulation procedure, since any violation of assertions or occurrence of invalid end-states will immediately interrupt the simulation at an unexpected point. When the qualities of M and P are acceptable, the verification phase begins.

The verification can be taken in two steps. First is to estimate the state space of the model, as well as the time consumed in one verification run, by using SPIN's bit-hashing verification mode. We choose *'absence of invalid end-states'* as one of the basic safety requirements for SPIN to sweep over the model M, because this property is verified by trying to reach every possible state in the corresponding automaton. It is still possible that invalid end-states may be detected after previous sanity checks done in the modelling phase, since the previous simulations can only check very limited instances of computation paths in the model. If the result comes up with a low hash factor, which

```
1   proc main()
2   {   Design: given informal (protocol) design;
3       Requirements: set of informal property descriptions;
4       M: PROMELA model of the design;
5       P = ∅: set of property specifications to be checked;
6       C = ∅: set of property specifications has been verified;
7       (M, P) = Formalize(Design, Requirements);
8       C = Verify(M, P);
9   }
10
11  proc Formalize(Design, Requirements)
12  {   ψ: basic operation procedure of the (protocol) design;
13      acceptable = FALSE;
14      while (NOT acceptable)
15      {   M = Modelling(Design);
16          acceptable = Simulating(M, ψ);
17      }
18      (safety, liveness, temporal properties) = Classify(Requirements);
19      P = safety ∪ liveness ∪ temporal properties;
20      return(M, P);
21  }
22
23  proc Verify(M, P)
24  {   ψ1: absence of invalid end-states;
25      Result = 'pseudo error';
26      while (Result ≠ 'no error')
27      {   Result = Bit-hashing(M, ψ1);
28          if (Result == 'error in model') Correct(M, ψ1);
29          else if (Result == 'low hash factor') Optimize_StateSpace(M);
30      }
31      for each ψi ∈ P
32      {   Result = Exhaustive_check(M, ψi);
33          while (Result == 'run out of memory')
34          {   Compress_StateVector(M);
35              Optimize_StateSpace(M);
36              Result = Exhaustive_check(M, ψi);
37          }
38          if (Result == 'error in model')
39          {  Correct(M, ψi); goto line 32;
40          }
41          else if (Result == 'error in design')
42          {  Halt and correct the design; goto line 7;
43          }
44          else if (Result == 'no error')    C = C ∪ ψi;
45      }
46      return(C);
47  }
```

Figure 10: Procedure for verifying a PROMELA model using SPIN

means only a small part of the whole state space can be reached, we need to optimize the model to yield a smaller state space, yet still preserve the operational features of the protocol design.

The second step involves verifying every property specified in the set P. The model checking mode taken, however, depends on the reduction of the model's state-space (as shown in line 29, 34 and 35). If the result in the 'bit-hashing' step shows no error with a high hash factor, which means most of the state space can be covered, then it is possible for us to conduct a full state space search (or exhaustive check) on other properties. If the hash factor is low, or the exhaustive check keeps running out of memory, despite the optimization efforts, then we have to sacrifice some accuracy in the verification and to use SPIN's bit-hashing mode. For most of the problems dealt with in this report, we will try to do exhaustive verification.

Analyses on the verification results may reveal modelling errors with respect to certain properties. It may also reveal that the original design violates desired requirements, e.g., as listed on page 6, in which case we have to go back to propose modifications in the design and restart the `main()` procedure. With the previous efforts, this restart will not take much time before we obtain a new version of M and begin to verify the remaining properties. The whole procedure will terminate when C, the set of verified properties, contains every element of P, the set of properties we propose to check.

2.4 Summary

In this section, we have reviewed the automata-theoretical approach of model checking and introduced a well developed model checker SPIN, which is based on this method. The verification procedure proposed in Section 2.3 brings up two main issues in model checking a protocol: (1) how to model the given protocol design with PROMELA and (2) how to verify the resulting model (preferably) with an exhaustive state-space search. Starting in the next section, we will discuss in detail these two issues with a protocol design selected from the real world.

3 Formalization of the Wireless Transaction Protocol

This section discusses the formalization of the Wireless Transaction Protocol (WTP) design [60], which provides models and specifications for verification. In Section 3.1, we summarize the TR-Service part and the TR-Protocol part of the published WTP standard, which define the protocol operation. In Section 3.2 to 3.4, we proceed to build formal models of both the TR-Service and the TR-Protocol, using finite state automata and PROMELA syntax. In Section 3.5, we abstract correctness requirements from the protocol design and formalize them as property specifications to be checked.

Table 1: WAP architecture

OSI Layers	WAP Layers
Application	Application Layer (WAE)
Presentation	
Session	Session Layer (WSP)
Transport	Transaction Layer (WTP)
	Security Layer (WTLS, optional)
	Transport Layer (WDP)
Network	**Bearers**: GSM, CDMA, PDC, IP, etc.
Data Link	
Physical	

3.1 Description of the Protocol Design

We will describe the WTP design in line with the 5 elements of a protocol definition given in Section 1.1.2. First we introduce the protocol environment, along with some conventions of protocol terminology. Then we summarize the service provided by the protocol in Section 3.1.2. The remaining 3 elements are given in Section 3.1.3, the TR-Protocol part.

3.1.1 Protocol Environment

The Wireless Transaction Protocol sits in the Wireless Application Protocol (WAP) architecture [57], which was designed by the WAP Forum [56] to provide Internet and similar data services to mobile users. The WAP architecture consists of 5 layers (from bottom to top): transport, security, transaction, session and application. Their relationship with the OSI layers is shown in Table 1 (where each pair of brackets holds the associated protocol name). Below the WAP stack are a wide range of the wireless bearer services, which this global standard aims to support.

Figure 11 (which is modified from Figures 4 and 7 in [31]) helps to understand how WTP functions in the transaction layer. In the generic OSI layered communication architecture, each layer comprises *protocol entities* that perform functions within the layer. The entities in the (N)-layer (and all layers below) provide *(N)-service* to the (N+1)-entities (referred to as *(N)-service-users*[10]), through *(N)-service-access-points* (N-SAP) at the boundary between the (N+1)-layer and (N)-layer. Peer (N)-entities use *(N)-protocol* for communication, which in turn use the (N-1)-service provided by lower layers. Therefore, the logical path for exchanging information is vertical, via SAPs. A protocol design for the N-layer defines both the (N)-service and the (N)-protocol.

Here the N-layer of interest is the Transaction layer, whose associated protocol (WTP) is defined to provide so-called "request/response duo" (referred to as a *trans-*

[10]The user initiating the service facility is called the *Initiator*; the peer user is the *Responder*.

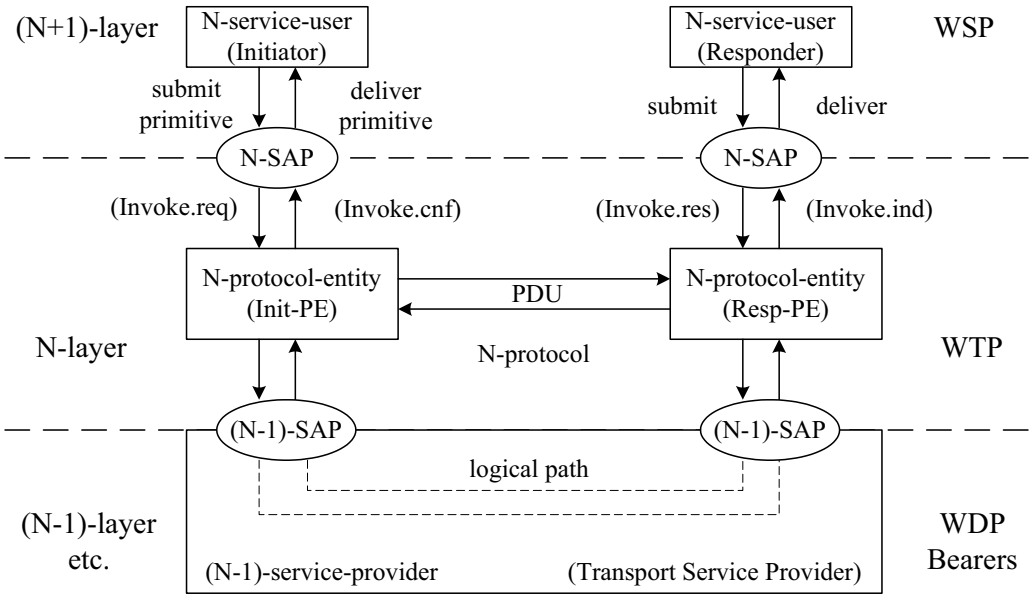


Figure 11: Abstraction of the Transaction Layer in the WAP architecture

action) services for interactive "browsing" applications, where requests are invoked by a client (the *Initiator*) for information from a server, and the server (the *Responder*) responds with the resulting information. For the upper layer users (i.e., the TR-Users), the Wireless Transaction Protocol defines 3 classes of service, among which we focus on Class 2 service. It intends to provide reliable (two-way) transaction service over an unreliable datagram service, by using a *reliable invoke message with exactly one reliable result message. The request (invoked from the Initiator) is acknowledged by the Responder and, likewise, the receipt of the result at the Initiator is acknowledged to the Responder* [60].

The protocol environment for WTP is defined by the Transport Service, which the Transaction Protocol Entities (TR-PEs) use to communicate with each other. In the Transport layer, a connection-less protocol, Wireless Datagram Protocol (WDP) [58], is used (which specifies UDP to be used over IP-supporting bearers). Therefore, the WTP design assumes messages exchanged between peer TR-PEs can be lost, re-ordered or duplicated. We will take this characteristic of the Transport-Service-Provider as an assumption on the communication channel modelled later.

3.1.2 Transaction Service

In general, the (N)-service defines the interactions between the (N)-service-users and (N)-service-provider, which are described, in an abstract way, using *service primitives* [30]. So two core elements of the TR-Service definition are: the primitives and their parameters, and the set of possible primitive sequences.

Table 2: Primitive sequence table for Transaction Service Class 2

Next Primitive (abbr.)	**Invoke**				**Result**				**Abort**	
	req	**ind**	**res**	**cnf**	**req**	**ind**	**res**	**cnf**	**req**	**ind**
Invoke.req (IREQ)										
Invoke.ind (iind)										
Invoke.res (ires)		X								
Invoke.cnf (ICNF)	X									
Result.req (rreq)		X^*	X							
Result.ind (RIND)	X^*			X						
Result.res (RRES)						X				
Result.cnf (rcnf)					X					
Abort.req (AREQ/areq)	X	X	X	X	X	X	X			
Abort.ind (aind/AIND)	X	X	X	X	X	X	X			

Note: 1. A primitive listed in the column header (printed with bold) may only be followed by a primitive in the row header that is marked with an "X" or "X^*", but "X^*" is NOT valid if Ack-Type is On.
2. Primitives abbreviated in upper case (e.g., IREQ for Invoke.req) happen at the Initiator side; those in lower case, at the Responder side. Abort primitives can happen on both sides.

Primitives Primitives are structured as "*primitive_ name .primitive_ type* [*primitive_ parameters*]" to represent data/control conveyed between service user and provider, where:

Primitive_name identifies the service provided by the primitive. We are interested in three kinds defined in WTP: *Invoke, Result* and *Abort*, which correspond to the basic messages used in the protocol.

Primitive_type is chosen from one of the four types: request (*req*), indication (*ind*), response (*res*) and confirm (*cnf*). As indicated in Figure 11, *.req* and *.ind* are a "submit/deliver duo", while *.res* and *.cnf* are another pair. For example, *Invoke.req* is a primitive submitted by the Initiator to invoke a transaction.

Primitive_parameters represent user data or control information, which may be given as a tuple following the primitive when necessary. Among those parameters given in WTP, **Ack-Type** is of special significance. If this parameter is turned On, then the TR-Users must acknowledge each *indication* primitive with a *response* primitive. If Off, then the TR-PEs may optionally provide acknowledgement without explicit acknowledgment from the TR-Users, i.e., submitting *response* instead. Therefore, the value of *Ack-Type*, which is set in the *Invoke.req* primitive and remains unchanged during one transaction, affects the possible primitive sequence (and further the protocol operation) of that transaction process.

Primitive Sequences Table 2 (based on Table 6 in [59]) defines the legal primitive sequences for the Class 2 Transaction. Though not mentioned clearly in the design document, this table only defines the *local* primitive sequences that can happen on one TR-User side (either Initiator or Responder). For the *global* sequence of primitives exchanged between peer TR-Users, those primitive types must occur in the order (*request, indication, response,* then *confirm*) and in a submit-deliver pair (*request* with *indication*, or *response* with *confirm*). This is the **end-to-end principle** used for service definitions [30]. It is not necessary, however, for all of the primitive types to occur in one primitive sequence.

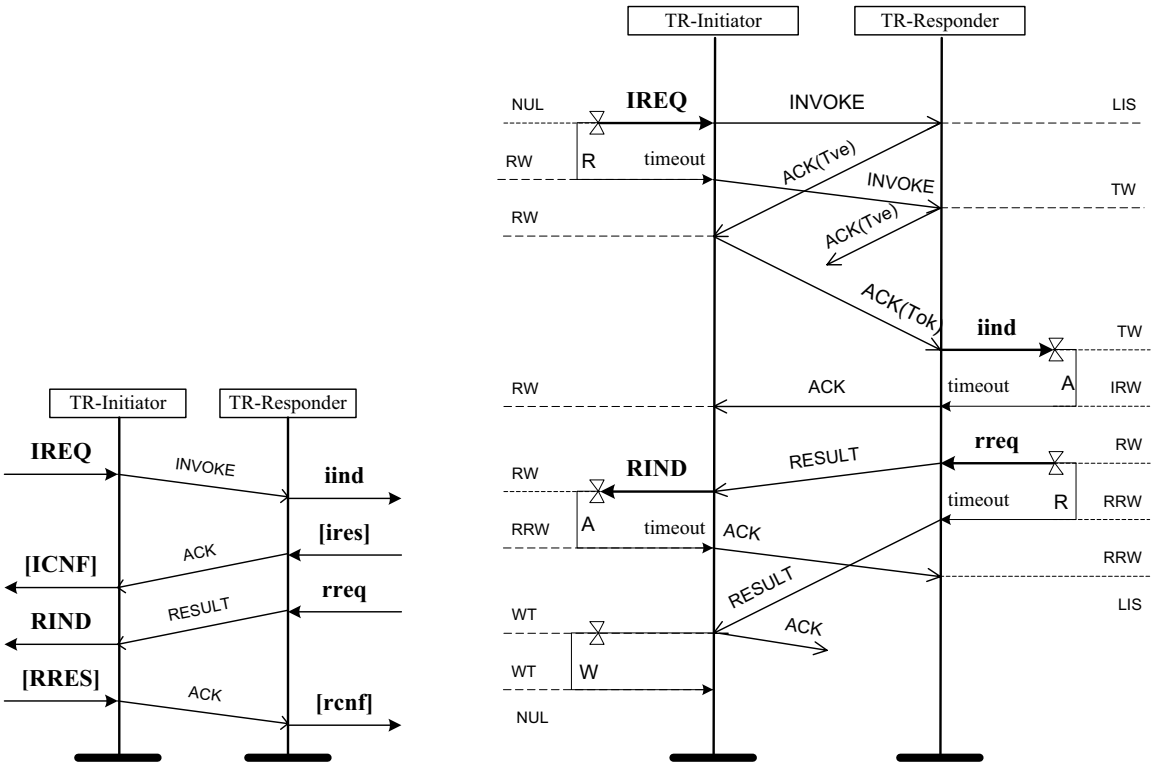
We depict, in part (a) of Figure 12 (based on Figure 2 of [59]), an MSC (Message Sequence Chart [32]) presentation of a possible primitive sequence. As abbreviated in Table 2, primitives aligned vertically (along the *instance axis* from top to bottom) are ordered in the time they occur in the local sequence. Two primitives aligned horizontally (e.g., $ICNF$ and $rreq$) can occur in any order in the global sequence, except that there is a message (described by an open-headed arrow) in between. The directed message arrow actually defines a temporal order of a submit-deliver pair (e.g., $IREQ$ and $iind$). Here, we use square brackets to hold the primitives which may optionally happen when the parameter *Ack-Type* is Off. The instance of execution of a TR-User process terminates on the solid bar at the end of the vertical axis.

Within one complete transaction (either successful or aborted), a global primitive sequence always terminates on certain ending primitives; e.g., the sequence shown in Figure 12 (a) ends with the *Result.cnf* primitive. Potential errors in the protocol design will generate illegal sequences that are neither defined in Table 2 nor accordant with the end-to-end principle, which will result in an incomplete transaction. Thus, one of the properties to be verified is whether the TR-Protocol design might have errors which enable illegal sequences. This would reveal whether the TR-Protocol is a faithful refinement of the defined TR-Service or not.

3.1.3 Transaction Protocol

The TR-Protocol part is to define the other 3 elements of the protocol: the vocabulary of messages, the format of each message and the procedures for exchanging the messages. In the generic OSI model, peer (N)-protocol-entities virtually communicate by sending and receiving Protocol Data Units (PDUs)[11], which consist of protocol control information and possibly user data. So the definition of TR-Protocol specifies the types of PDU (the vocabulary), the encoding of PDU data (the format) and the rules for exchanging the PDUs. It is these rules that define the procedures for providing the desired service of the protocol, or more explicitly, they tell a TR-User how to respond to a service primitive coming from the peer TR-User side.

[11] PDUs used by the (N)-protocol are encapsulated in Service Data Units (SDUs) that are transferred, as part of the primitive parameters, by the (N-1)-service-provider along the logical information path in lower layers.



(a) One primitive sequence defined in TR-Service with Ack-Type Off

(b) One primitive sequence defined in TR-Protocol with PDU re-transmission

Figure 12: Message Sequence Chart of sample primitive sequences

For the basic TR-Protocol behaviours (called "features" in the WTP document) modelled in this report, we are interested in four primary PDU types: *INVOKE, RESULT, ACK,* and *ABORT*. The encoding of a PDU comprises the definition of a header (for control information) and the user data, both of which are structured as an integer number of octets. For detailed encoding schemes, we refer to [60]. Table 3 lists the header fields which are modelled in this report. The significance of the fields are explained as follows:

- *Re-transmission Indicator (RID)*: "0" indicates the PDU is the first one sent; "1" identifies a re-transmitted PDU.
- *User/Protocol Entity Flag (U/P)*: "1" Indicates the primitive parameter Ack-Type is On; "0", Off.
- *Tve/Tok Flag*: Indicates if the *ACK* PDU is used for Transaction Identifier (TID) verification [60]. *Tve* indicates Resp-PE requests for a confirmation for the

Table 3: PDU header fields modelled for the TR-Protocol

Header Field (Abbrev.)	**PDU**				Number of bits used in one octet
	INVOKE	**RESULT**	**ACK**	**ABORT**	
PDU Type	Y	Y	Y	Y	4
RID	Y	Y	Y	-	1
U/P Flag	Y	-	-	-	1
Tve/Tok Flag	-	-	Y	-	1

Note: "Y" indicates the field is modelled for that PDU, "-" not applicable in the PDU.

outstanding transaction numbered with the TID (received through an *INVOKE* PDU); *Tok* indicates Init-PE confirms that Resp-PE can proceed to deliver an *Invoke.ind* primitive upon receipt of the *INVOKE* PDU.

In the WTP document, a set of state tables (listed in Appendix B) is used to specify the rules for exchanging PDUs. Peer TR-PEs change their state as the result of interacting with the operating environment (e.g., read from/update channels and global variables); the actions of interaction are guarded by some conditions (e.g., evaluations of the PDU header fields). Based on the state tables, we depict an example of one transaction iteration in part (b) of Figure 12. It can be viewed as a refinement of the service primitive sequence (with Ack-Type Off) shown in Part (a), augmented with some TR-Protocol features, e.g., PDU retransmission, hold-on acknowledgement and the three-way handshaking procedure involved in TID verification.

Taking the Initiator side for an example, the MSC describes a time order as follows:

At the beginning, the Init-PE is in state *Null* (NUL). On receiving an *IREQ* primitive submitted from the Initiator, it sends an INVOKE PDU and enters the *Result_Wait* (RW) state. When the *re-transmission interval* (R) expires before an acknowledgment comes from the peer TR-PE, the Init-PE resends an INVOKE. When receiving an ACK PDU with *Tve* flag set (requesting a TID verification), it responds with a confirmation (an ACK PDU with *Tok* flag set), which completes the three-way handshaking.

Then the Init-PE gets a hold-on acknowledgement sent from Resp-PE, as the Responder-User chooses not to send the *ires* primitive. It delivers a RIND primitive on getting a RESULT PDU and change its state to *Result_Response_Wait* (RRW). The Init-PE keeps waiting for a response (i.e., *RRES* primitive) from the TR-User until the *acknowledgment interval* (A) elapses. Then it send an ACK PDU as an implicit acknowledgement of the result. A second RESULT PDU received in the *Wait_Timeout* (WT) state is interpreted to mean the the peer PE did not receive the first ACK, so a second ACK is transmitted.

Finally, after a period of *wait time-out interval* (W), the Init-PE deletes all transaction state information and returns to its initial state, concluding one outstanding

transaction. On the peer side, this transaction is complete when the Resp-PE returns to the *Listen* (LIS) state, waiting for the next INVOKE with a new TID.

3.2 Overview of the Formalization

The aim of modelling the WTP is to provide formal descriptions of the protocol operation and the related "well-formed" requirements (see page 6). We will first build a PROMELA model of TR-Service, to generate the service language defined by Table 2 and the end-to-end principle (see page 31). Based on this abstract service model, a TR-Protocol model will be developed according to the rules defined in the state tables (see Appendix B). In both cases, we will first present a finite automata model of the protocol design, which serves as the underlying model and can be easily translated into the PROMELA language. Finally, a set of requirements will be specified with PROMELA and LTL syntax.

3.2.1 Scope

We make the following assumptions on our model. Further assumptions will be given when we model the TR-Protocol.

Assumption 3.1 Transaction Class 2. Class 2 transaction service and protocol are modelled. Its service language is much more complex than that of Class 0 and Class 1 transactions, which only provide *one-way* request service (unlike Class 2, with no result message) with different levels of reliability. So protocol operations defined for Class 0 and Class 1 are not modelled.

Assumption 3.2 Single pair of TR-Users. Transactions modelled occur between one TR-Initiator and one TR-Responder. Unlike routing protocols, for the transaction protocol, we focus more on the interaction procedure between two TR-Users involved in the "request/response" duo. So the address information of TR-Users, given in the primitive parameters, is not modelled.

Assumption 3.3 Deterministic Ack-Type. The primitive parameter *Ack-Type* is modelled as being determined *a priori* for each transaction, since it remains unchanged throughout that transaction. By determining the Ack-Type value in the first $INVOKE$ PDU, we can investigate the TR-Protocol's behaviours separately when the *User Acknowledgement* feature is turned On or Off.

Assumption 3.4 Single transaction. Different transactions between one pair of TR-Users are identified by TIDs and are independent of one another. Although TID ranges over a finite space (i.e., $0 \sim 2^{15}$), no confusion is possible in the case of TID wraparound [60]. So to generate all possible instances of one single transaction (which is facilitated in SPIN) is sufficient for verifying the functional behaviour of the TR-Protocol.

Assumption 3.5 Asynchronous channels in the Transport-Service-Provider. Messages exchanged between TR-Users (or TR-PEs) are buffered for reception. In the WTP design, there is no requirement of immediate receipt of a message sent from the peer side. So the two sides communicate with two asynchronous channels (one in each destination), rather than synchronous handshakes. Moreover, messages buffered in the channel can be re-ordered or lost due to the characteristics of the Transport-Service-Provider described on Page 29.

3.2.2 Architecture

The TR-Service and TR-Protocol models we will provide are two levels of abstraction of the WTP design. But they share a similar architecture as illustrated in Figure 13. The core of the model consists of two concurrent processes, one on the Initiator side and the other on the Responder side, which function to generate a defined service language by communicating through two asynchronous channels and shared variables (i.e., global controls).

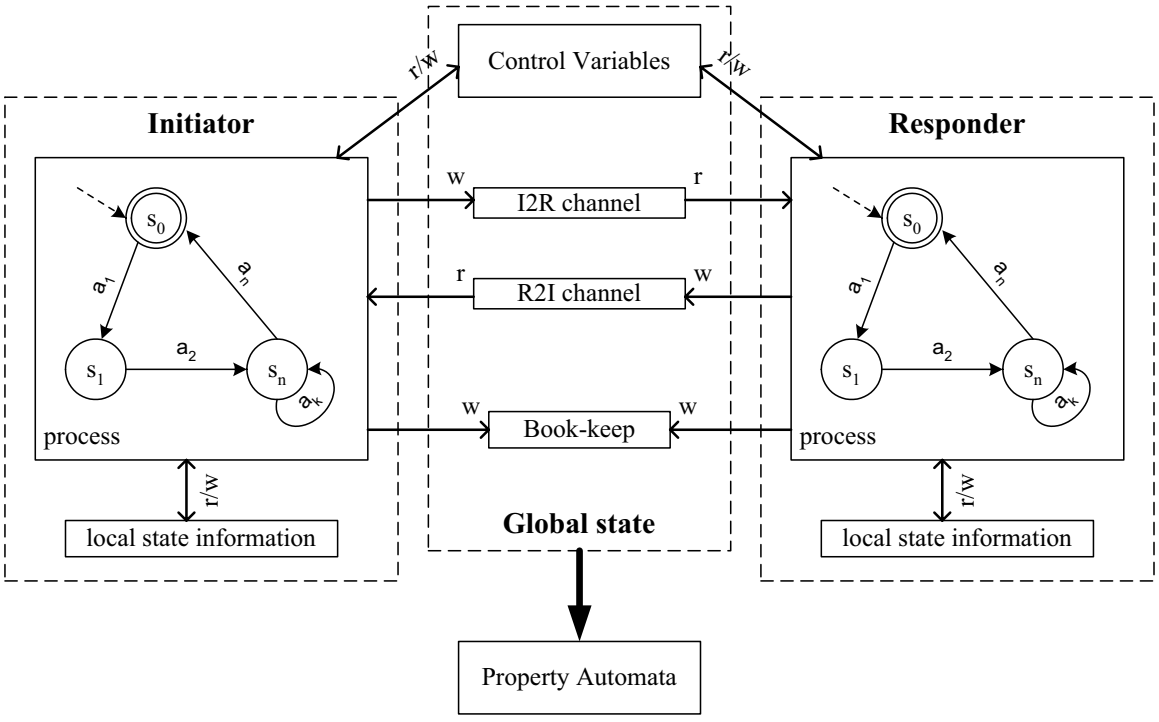


Figure 13: General model architecture

Each process can be modelled with a PROMELA `proctype` template, which has its automaton equivalent given as $A_i = (S_i, s_{0_i}, \Sigma_i, \delta_i, F_i)$. We give a generic definition of A_i, with respect to the model architecture, as follows:

- The set of states S_i defines all possible evaluations[12] of variables within the model, i.e., $S_i : V_i \rightarrow D_i$, where $V_i = (Global_i, Local_i)$ is a tuple of global/local variables used and D_i is the set of related ranges.

  Global variables include two *asynchronous* channels and other global controls (e.g., the primitive parameter *Ack-Type*). (See Section 3.3.1 and 3.4.2 for details.). Local variables comprise the state of the local SAP/TR-PE, and the local controls (e.g., counters/timers in TR-Protocol).

- The initial state s_{0_i} gives the initialization of variables.

- The alphabet $\Sigma_i = \{a \mid a = L(s),\ s \in S_i\}$ defines a set of guards for state transitions, based on the evaluations of local/global variables[13]. Given the set of atomic propositions $\mathcal{V} = \{v = d \mid v \in V_i,\ d \in D_i\}$, the label function $L : S_i \rightarrow 2^{\mathcal{V}}$ defines a symbol $a \in \Sigma_i$ is recognized in a state only when the propositions on the evaluations are true in that state.

- The transition relation $\delta_i : S_i \times \Sigma_i \rightarrow 2^{S_i}$ defines the control flow of process execution. $\delta_i(s, a)$ gives all possible successors of state s. Generally, $\mid \delta_i(s, a) \mid \geq 1$, and the evaluation of variables is selected non-deterministically.

  In the detailed model structure, the definition of δ_i is based on state tables of the local SAP or TR-PE, which can be implemented by a repetition construct (PROMELA `do-od` loop) with non-deterministic options of atomic sequences.

- F_i consists of all the possible final states of the process. In any final state $s_{F_i} \in F_i$, $s_{F_i}(Local_i) = s_{0_i}(Local_i)$ because in a complete transaction, peer SAPs/TR-PEs finally return to their initial state after clearing all state information (e.g., resetting counters). Usually $s_{F_i}(Global_i) \neq s_{0_i}(Global_i)$.

To generate the service language, we use a set of variables (each corresponding to one service primitive defined in Table 2) to record the occurrence of primitive sequences. These variables, along with the length of the channels (i.e., number of messages stored), will be used in PROMELA `assert` statements or in LTL formulas to specify correctness requirements[14]. When specified in an LTL formula, each temporal claim can be translated into a (negative) property automaton $A_{\neg\psi}$, then the automata-theoretical approach of model checking can be applied to verify if the protocol model $\prod_{i=1}^{n} A_i$ satisfies the property. On the other hand, most safety requirements specified in `assert` statements can be put in a separate process (namely, a monitor), which can also be viewed as an automaton A_m and runs with the A_i's in an interleaved way. For the (interleaving) product automaton $A_m \times \prod_{i=1}^{n} A_i$, a basic depth-first-search algorithm can

[12] In Figure 13, these evaluations are shown as open arrows inscribed with "w", i.e., write.

[13] In Figure 13, to read the evaluation results is shown as open arrows inscribed with "r".

[14] In SPIN, only global variables are accepted in LTL claims [23], even if the variables for book-keeping service primitives are actually used only by one local process.

be conducted to analyze, if a "bad" state (which violates the assertion) is reachable. Further discussion of the monitor process is given in Section 3.5.1.

3.3 PROMELA Model of TR-Service

3.3.1 Model Structure

Based on the general architecture, the underlying automata models for the *Initiator* and *Responder* processes in TR-Service are given as A_{SI} and A_{SR}, respectively.

For $A_{SI} = (S_{SI}, s_{0_{SI}}, \Sigma_{SI}, \delta_{SI}, F_{SI})$, we have the following definition:

State and Related Variables The set of states $S_{SI} : V_{SI} \rightarrow D_{SI}$ maps the set of variables V_{SI} used by the Initiator to the set of related ranges D_{SI}.

$V_{SI} = (Global_{SI}, Local_{SI})$ is a tuple consisting of global variables and local ones:

The global variable set $Global_{SI} = (channels, global\_controls, toggles)$ is a tuple, where

- *Channels* between TR-Users are two extended variables. Each channel is a tuple (*chan_name, chan_size, message_stored*), i.e., an array of *message_stored* having a maximum dimensionality of constant *chan_size* (typed as a positive integer). $Message\_stored$ is actually an ordered (finite) set of (differently typed) variables, $type_1 \times type_2 \times ... \times type_n$. The messages stored in the channels can be out of order, so they can be taken out of the array at any position.

  For the TR-Service model, *message_stored* has only one type whose range is the 4 messages conveyed between TR-Users, i.e., {INVOKE, ACK, RESULT, ABORT}. Figure 14 shows the declaration of two channels using PROMELA syntax. They are of type `chan`, and named `Init2Resp` and `Resp2Init`, with a size of 2 and 3, respectively. Clearing a message from the channel is modelled using PROMELA's *random receive* operation, e.g., `Init2Resp??INVOKE`.

```
mtype = {INVOKE, ACK, RESULT, ABORT};
chan Init2Resp = [2] of {mtype};
chan Resp2Init = [3] of {mtype};
```

Figure 14: Declaration of channels in the TR-Service model

- $Global\_controls = \{Ack-Type, NoAck, Aflag\}$ is a set of bit[15] elements. *Ack-Type* models the primitive parameter. $NoAck$ is used to control global primitive sequence when *Ack-Type* is On, in which case $Invoke.res$ primitive must precede

[15]We use *bit* type (1/0) in equivalence with *boolean* type ($true/false$).

Result.ind. $Aflag$ is to guard the atomic sequences which implement the transition relations (see page 41 for details).

- $Toggles = (PrimitiveI\_tgl, DLock\_I)$ is a tuple. $PrimitiveI\_tgl$ has 7 elements, all of PROMELA `unsigned` type[16] to toggle on the occurrence of the corresponding service primitive on the Initiator side, e.g., $IREQ\_tgl$ becomes "1" when *Invoke.req* is submitted. $DLock\_I$ has one bit-typed element to indicate deadlock in the process execution.

The local variable set $Local_{SI} = \{istate, UserAck\}$ has two elements. The value of *istate* represents the state of the local SAP as defined in Table 4. The bit $UserAck$ is a local control used to generate only one transaction iteration on the Initiator side (see the first entry of Table 6).

Table 4: SAP states on the Initiator side

State (abbr.)	**Description**
I_Null (I_NUL)	Wait for an *invoke* submitted from the Init-User.
I_Inovoke_Resp_Wait (I_IRW)	Wait for an *invoke acknowledgement* sent from the Responder side.
I_Result_Wait (I_RW)	Wait for a *result* sent from the Responder side.
I_Result_Resp_Wait (I_RRW)	Wait for a *result acknowledgement* submitted from the Init-User.

Initial State $s_{0_{SI}}$ gives the initialization of variables, shown in Table 5.

Guard Alphabet Each symbol a in the alphabet Σ_{SI} is a first-order formula, defined as

$$a(Local_{SI}, channels, global\_controls) = L(s).$$

That means, given the set of propositions $\mathcal{V} = \{v = d \mid v \in V_{SI},\ d \in D_{SI}\}$ and the label function $L : S_{SI} \to 2^{\mathcal{V}}$, $L(s)$ contains all the "true" propositions on the evaluations of variables.

Symbols are used as guards for state transitions. According to the primitive sequences allowed in Table 2, we create Table 6 to define the guards and state transitions of the Initiator process.

Each entry (row indicated with a number) corresponds to a transition. In the "Guard" column of one entry, the conjunction of atomic propositions on the local/global controls and the incoming channel, as well as the current state of the local SAP defines a symbol. For example, the symbol

$$a = (istate = I\_IRW) \wedge (ACK \in Resp2Init) \tag{7}$$

[16]Later, we will show they actually range over bit type.

Table 5: Initial state of the Initiator process

No.	V_{SI}	**Type of** D_{SI}	$s_{0_{SI}}(V_{SI})$
1	Init2Resp	chan	empty
	Resp2Init		
2	Ack-Type	bit	0/1
3	NoAck	bit	1
4	Aflag	bit	0
5	IREQ_tgl	unsigned	0
	ICNF_tgl		
	RIND_tgl		
	RRES_tgl		
	AREQ_tgl		
	AIND_tgl		
	AINDP_tgl		
6	DLock_I	bit	0
7	istate	mtype	I_NULL
8	UserAck	bit	Ack-Type

corresponds to a guard defined in the second entry of Table 6.

In the table, there is no requirement of disjointness between guards. The same guard can lead to different actions. For example, the guards defined in lines 6 to 8 can be true at the same time as any of the guards defined in lines 2 to 5. It is to model that the TR-user can abort at anytime an outstanding transaction. On the other hand, different guard conditions will not result in the same update of states.

For completeness, the last entry in the table says evaluations of variables other than those defined in the "Guard" column can make any symbol a true in any legal SAP state, but the current state remains unchanged.

Transition Relation The transition relation $\delta_{SI} : S_{SI} \times \Sigma_{SI} \rightarrow 2^{S_{SI}}$, is defined according to Table 6, whose "Action" column describes the new evaluations in state $\delta_{SI}(s, a)$.

In the table, the last column gives the update of the local *istate*, while new evaluations of other variables are defined in the next-to-last column . They include:

- submitting/delivering service primitives, modelled as incrementing $PrimitiveI\_tgls$;
- switching local/global controls;
- sending messages, modelled as adding messages into the outgoing channel, e.g., $Init2Resp \cup \{INVOKE\}$;
- receiving messages, modelled as clearing messages from the incoming channel, e.g., $Resp2Init - \{RESULT\}$.

Table 6: SAP state changes on the Initiator side

entry	**Guard**		**Action**	
	istate $\wedge$	*local_ctrl* $\wedge$ *channel* $\wedge$ *global_ctrl*	*Globals* $\wedge$ *local_ctrl*	$\wedge$ istate
1	I_NUL	UserAck = Ack-Type	IREQ_tgl + 1 Init2Resp $\cup$ {INVOKE} UserAck = $\neg$ Ack-Type	I_IRW
2	I_IRW	ACK $\in$ Resp2Init	ICNF_tgl + 1 Resp2Init $-$ {ACK} NoAck = 1	I_RW
3		RESULT $\in$ Resp2Init NoAck = 1 Ack-Type = 0	RIND_tgl + 1 Resp2Init $-$ {RESULT} NoAck = 0	I_RRW
4	I_RW	RESULT $\in$ Resp2Init NoAck = 1	RIND_tgl + 1 Resp2Init $-$ {RESULT} NoAck = 0	I_RRW
5	I_RRW		RRES_tgl + 1 Init2Resp $\cup$ {ACK}	I_NUL
6	$\neg$ I_NUL		AREQ_tgl + 1 Init2Resp $\cup$ {ABORT}	I_NUL
7		ABORT $\in$ Resp2Init	AIND_tgl + 1 Resp2Init $-$ {ABORT}	I_NUL
8			AINDP_tgl + 1	I_NUL
9		*else*	*no change*	

Note: 1. $Globals = toggles \times channels \times global\_ctrls$.
2. Variables not defined in the ‘‘Guard’’ column can take any value within their respective ranges. Variables not defined in the ‘‘Action’’ column remain unchanged.

So, the first entry in Table 6 could be read as: if $(istate = I\_IRW) \wedge (UserAck = Ack-Type) = TRUE$, then the service primitive $Invoke.req$ is submitted, a message INVOKE is sent to the $Init2Resp$ channel and the local control $UserAck$ is switched from its initial value.

If the guards of several entries are true simultaneously, one of those transitions is chosen non-deterministically to proceed. Table 7 gives an example of state transitions from some state s, where the symbol a is taken from Formula 7. Corresponding to entries 2 and 6 in Table 6, new evaluations in two possible states of $\delta_{SI}(s, a)$ are shown as $\delta_{SI}(s, a)(V_{SI})_1$ and $\delta_{SI}(s, a)(V_{SI})_2$, although, only one of the new states can be reached in one instance of process execution.

To model a single transition (i.e., an entry in the state table) in PROMELA, we can use a conjunction of boolean expressions as the guard followed by a sequence of assignment statements. It is important to note, however, that statements following the guard are not necessarily executable when running with other concurrent processes. For example, in a PROMELA sequence of `Resp2Init??[ACK] -> Resp2Init??ACK`, where

Table 7: Sample of state transitions in the Initiator process

No.	V_{SI}	$s(V_{SI})$	$\delta_{SI}(s,a)(V_{SI})_1$	$\delta_{SI}(s,a)(V_{SI})_2$
1	istate	I_IRW	I_RW	I_NUL
2	UserAck	1	1	1
3	Resp2Init	ACK included	ACK cleared	not changed
4	Init2Resp	not full	not changed	ABORT added
5	Ack-Type	0	0	0
6	NoAck	1	1	1
7	Aflag	0	0	0
8	ICNF_tgl	0	1	0
9	AREQ_tgl	0	0	1
10	IREQ_tgl	1	1	1
11	DLock_I	0	0	0

Note: `1. Other 4 PrimitiveI_tgls irrelevant to this example are not shown.`

a channel *poll* statement `Resp2Init??[ACK]` is used to model the condition $ACK \in Resp2Init$, the receiving statement that follows might be blocked even if there is an ACK message in the `Resp2Init` channel. As the result of a race condition, a second process (which shares the `Resp2Init` channel) can steal the ACK message just after the Initiator determined its presence. So each entry actually defines a critical section for updating the shared variables, including *channels* and *global_controls*.

We take advantage of the `atomic` structure in PROMELA to implement the critical section and guard it with an additional bit $Aflag$. As shown in Figure 15, all the statements within the atomic sequence are executed in sequence as one indivisible unit, non-interleaved with other processes, provided the beginning guard statement evaluates true; otherwise, the whole block (which corresponds to a transition) is disabled. If any statement within the atomic sequence is blocked (e.g., on trying to send a new message to a full channel), the atomicity is lost (with the following statements being blocked consequently) and other processes are allowed to execute.

Therefore, to ensure the exclusiveness of each critical section in the protocol model, we introduce a global control $Aflag$ in each guard. It acts like a switch, opened at the entrance to the critical section and closed immediately before leaving. In case one sequence loses its atomicity, other sequences cannot proceed. We will later put this $Aflag$ in the property automaton to monitor the safety of the whole model. To ensure the "pure atomicity" of each critical section, we must verify that $Aflag$ is always 1.

The transition relation of the whole process can be implemented using a repetition construct with non-deterministic options of atomic sequences, as shown in Figure 16. According to the mechanism of PROMELA's semantics engine (see Appendix A.2), in every iteration through the do-loop, one option (led by a double colon) is selected for execution provided that its first guard statement is executable. If none of the guards

```
1       transition Entry_i( ) {
2               atomic {
3                       if (Guard == TRUE AND Aflag == 1 ) {
4                           Aflag = 0;
5                           Actions of updating local/global variables;
6                           Display the service primitive(s);
7                           Aflag = 1;
8                       }
9               }
10      }
```

Figure 15: Pseudo-code for one transition corresponding to a state table entry

are executable, the do-construct as a whole will block until at least one of its options becomes executable again.

```
1       process TR-Init-User( ) {
2       progress_I:     do
3                       :: Entry_1 ( ) /* atomic transition defined in Figure 15 */
4                       :: ...
5                       :: Entry_n ( )
6                       :: atomic {
7                                       if (final state condition == TRUE) {
8                                           Sign transaction completed; break
9                                       }
10                        }
11                      :: atomic {
12                                      if (timeout) { Sign DLock_I; break }
13                        }
14                      od
15      }
```

Figure 16: Pseudo-code for the Initiator process

The do-loop is broken under any one of the following conditions:

- Either the whole structure is blocked in an unexpected state $s \notin F_i$ so that $(\forall a \in \Sigma_{SI})\ \delta_{SI}(s, a) = \emptyset$, which leads to a deadlock. Then we may use the condition `timeout` (pre-defined in PROMELA) as an escape from that system hang state.
- Or the process reaches a state s which satisfies the conditions defined in the "Final State" section below, i.e., $L(s) = L(s_{F_i})$ where $s_{F_i} \in F_i$. Upon jumping out of the

do-loop, one transaction between the peer TR-Users is complete, no matter if it is successful (with the RESULT message acknowledged) or aborted (with the Abort primitive occurred).

Final State The set of final states F_{SI} gives the evaluations of the variables after one transaction is complete. When they are used to terminate the execution of one TR-User process, the local variables are the only concern, since in practice a TR-User does not have to wait for the channels or the global controls to become settled before it can finish the current transaction on its side.

Table 8 shows the final evaluations of local variables for the Initiator process. $F_{SI} = \{s \in S_{SI} \mid L(s) = (UserAck = \neg\, AckType) \wedge (istate = I\_NUL)\}$, which implies that a local sequence (which contributes to a global sequence) of service primitives has been generated for one transaction.

Table 8: Final state conditions of the Initiator process

No.	V_{SI}	**Type of** D_{SI}	$s_{F_{SI}}(V_{SI})$
1	istate	mtype	I_NUL
2	UserAck	bit	¬Ack-Type
3	$Global_{SI}$	*do not care*	

As a complement to the automaton equivalent of the Initiator process shown in Figure 16, we add two guards $\{timeout, L(s_{F_{SI}})\}$ to the alphabet Σ_{SI}. So the transition relation $\delta_{SI}(s, timout)$ or $\delta_{SI}(s, L(s_{F_{SI}}))$ maps to the *end* state, which corresponds to the closing curly bracket of the PROMELA process.

The *Responder* process has a similarly structured automaton model, $A_{SR} = (S_{SR}, s_{0_{SR}}, \Sigma_{SR}, \delta_{SR}, F_{SR})$, except for the following differences:

- In the variable set $V_{SR} = (Global_{SR}, Local_{SR})$, we define the local variable set $Local_{SR} = \{rstate, FirstInvoke\}$. The value of $rstate$, representing the local SAP state, is defined in Table 9. The bit $FirstInvoke$ is a local control on the Responder side, which functions similarly as $UserAck$ does for the Initiator process (see the first entry of Table 11).

  In the global variable set $Global_{SR} = (channels, global\_controls, toggles)$, $channels$ and $global\_controls$ are the same as defined in the Initiator process, but the duple $toggles = (PrimitiveR\_tgl, DLock\_R)$ is different. $PrimitiveR\_tgl$ corresponds to the other 7 service primitives on the Responder side, e.g., $iind\_tgl$ for $Invoke.ind$. $DLock\_R$ is to signal deadlock in the Responder process.

- $s_{0_{SR}}$ gives the initialization of variables, shown in Table 10.

Table 9: SAP states on the Responder side

State (abbr.)	**Description**
R_Listen (R_LIS)	Wait for an *invoke* sent from the Initiator side.
R_Invoke_Resp_Wait (R_IRW)	Wait for an *invoke acknowledgement* submitted from the Resp-User.
R_Result_Wait (R_RW)	Wait for a *result* submitted from the Resp-User.
R_Result_Resp_Wait (R_RRW)	Wait for a *result acknowledgement* sent from the Initiator side.

Table 10: Initial state of the Responder process

No.	V_{SR}	**Type of** D_{SR}	$s_{0_{SR}}(V_{SR})$
1	iind_tgl	unsigned	0
	ires_tgl		
	rreq_tgl		
	rcnf_tgl		
	areq_tgl		
	aind_tgl		
	aindp_tgl		
2	DLock_R	bit	0
3	rstate	mtype	R_LIS
4	FirstInvoke	bit	1

Note: `1. Other global variables used are initialized in Table 5.`

- The guard symbol $b \in \Sigma_{SR}$ and transition relation δ_{SR} are defined in accordance with Table 11. The PROMELA implementations of both a single transition and the whole process are similar as shown in Figure 15 and Figure 16.

  It is possible in many cases that more than one guard statement in both TR-User processes are executable at the same time, which models the concurrency property of the protocol operation. Consider, for example, the guard $a \in \Sigma_{SI}$ in Formula 7 and the guard in entry 4 of Table 11 (given as $b = (rstate = R\_RESULT\_WAIT) \in \Sigma_{SR}$), which can be true at the same time. Consequently, the service primitives, $Invoke.cnf$ and $Result.req$, can be delivered and submitted, respectively, in any order or even simultaneously. This is the situation we have seen in part (a) of Figure 12, where non-correlated service primitives can happen concurrently.

- In the final state set F_{SR}, evaluations of local variables are given in Table 12. $F_{SR} = \{s \in S_{SR} \mid L(s) = (FirstInvoke = 0) \wedge (rstate = R\_LIS)\}$

Table 11: SAP state changes on the Responder side

entry	**Guard**		**Action**	
	rstate $\wedge$	*local_ctrl* $\wedge$ *channel* $\wedge$ *global_ctrl*	*Globals* $\wedge$ *local_ctrl*	$\wedge$ rstate
1	R_LIS	INVOKE $\in$ Init2Resp	iind_tgl + 1 Init2Resp $-$ {INVOKE} FirstInvoke = 0	R_IRW
2	R_IRW	NoAck = 1	ires_tgl + 1 Resp2Init $\cup$ {ACK} NoAck = 0	R_RW
3		Ack-Type = 0	rreq_tgl + 1 Resp2Init $\cup$ {RESULT}	R_RRW
4	R_RW		rreq_tgl + 1 Resp2Init $\cup$ {RESULT}	R_RRW
5	R_RRW	ACK $\in$ Init2Resp	rcnf_tgl + 1 Init2Resp $-$ {ACK}	R_LIS
6	$\neg$ R_LIS		areq_tgl + 1 Resp2Init $\cup$ {ABORT}	R_LIS
7		ABORT $\in$ Init2Resp	aind_tgl + 1 Init2Resp $-$ {ABORT}	R_LIS
8			aindp_tgl + 1	I_NUL
9		*else*	*no change*	

Table 12: Final state conditions of the Responder process

No.	$Local_{SR}$	**Type of** D_{SR}	$s_{F_{SR}}(Local_{SR})$
1	rstate	mtype	R_LIS
2	FirstInvoke	bit	0

Product Automaton of the Protocol Model The basic operation of the protocol can be modelled as an interleaving product automaton $A_{MS} = A_{SI} \times A_{SR}$. We define $A_{MS} = (S_{MS}, s_{0_{MS}}, \Sigma_{MS}, \delta_{MS}, F_{MI})$ as follows:

- $S_{MS} : V_{SI} \times V_{SR} \rightarrow D_{SI} \times D_{SR}$.
- $s_{0_{MS}} = (s_{0_{SI}}, s_{0_{SR}})$.
- $\Sigma_{MS} = \Sigma_{SI} \cup \Sigma_{SR}$ is the union of each component's guard alphabet.
- δ_{MS} is defined by $(s'_{SI}, s'_{SR}) \in \delta_M((s_{SI}, s_{SR}), c)$ *iff*
  - $s'_i \in \delta_i(s_i, c)$ for each i $(i = SI, SR)$ such that $c \in \Sigma_i$ and
  - $s'_i = s_i$ for each i such that $c \notin \Sigma_i$.

Since by nature $\Sigma_{SI} \cap \Sigma_{SR} = \emptyset$, each symbol $c \in \Sigma_{MS}$ is exclusive to one automaton (either A_{SI} or A_{SR}), so state changes are interleaved.

- $F_{MS} = F_{SI} \times F_{SR}$ is the cartesian product of each component's final state.

3.3.2 Sample Simulation Run

As proposed in the formalizing procedure (Figure 10), (random/ interactive) simulation can facilitate a sanity check on the PROMELA model to remove apparent bugs which fail to realize the basic operations of the protocol design. As the result of one simulation run, SPIN can generate a graphical form of a *Message Sequence Chart* (MSC [32]) that presents the execution of the PROMELA model, in addition to a textual report on the updated evaluations of variables and channels.

Given the model in Appendix C.1, Figure 17 shows an MSC presentation of an interactive simulation run on the TR-Service model A_{MS}. Compared with Figure 12 part (a), each vertical line beginning with the id number and the name of a PROMELA process is an instance axis, which describes a partial order on the execution of the process.

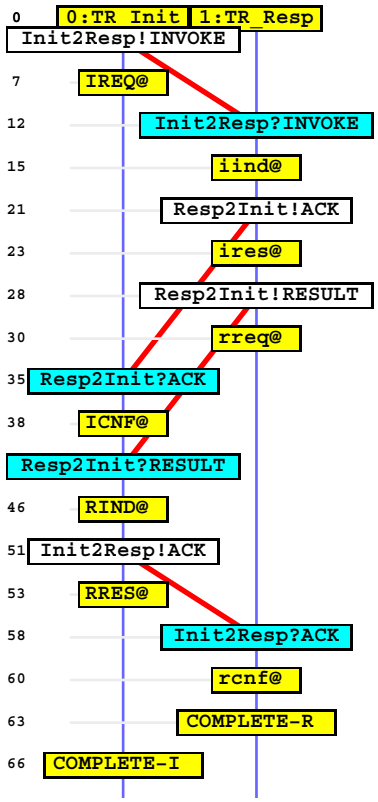


Figure 17: A sample simulation run on the TR-Service PROMELA model

Actions of updating variables are graphically specified by means of rectangles along the instance axis. Each line between two instance axes correlates two actions of sending and receiving the *same* message For example, the INVOKE message sent from the Initiator (`Resp2Init!INVOKE`) is received by the Responder (`Init2Resp?INVOKE`).

According to SPIN's simulation algorithm (see Appendix A.2), the interleaving product automaton (representing the whole model) runs in a stepwise manner; at one step only one statement in the model can be selected to execute. So the serial number left to an action box indicates at which step the statement is executed[17].

These serial numbers just present one possible temporal order of service primitives in a global sequence. In this example, the global sequence is $IREQ$ (step 7) $\rightarrow$ *iind* (step 15) $\rightarrow$ *ires* (step 23) $\rightarrow$ *rreq* (step 30) $\rightarrow$ $ICNF$ (step 38) $\rightarrow$ $RIND$ (step 46) $\rightarrow$ $RRES$ (step 53) $\rightarrow$ *rcnf* (step 60).

The two instance axes terminate on signalling the transaction complete (steps 63 and 66). This is the desired behaviour we defined in Figure 16, line 8. So the MSC provided by SPIN is a convenient means of visualizing traces of model execution, which makes it easier to locate errors in the protocol design during verification.

3.4 PROMELA Model of TR-Protocol

3.4.1 Further Assumptions

Assumption 3.6 Protocol features modelled. The TR-Protocol refines the TR-Service by supporting different protocol features. Out of the 15 features defined in WTP Version 2.0 [60], 6 are modelled and analyzed, for we consider them as representative of the core behaviour of the TR-Protocol. They are (1) message transfer, (2) re-transmission until acknowledgment, (3) user acknowledgment, (4) transaction abort, (5) TID (transaction identifier) verification and (6) error handling. Other features, such as segmentation and re-assembly, etc., are left for future work.

Assumption 3.7 Lossy channels. As assumed in Section 3.2.1, messages (PDUs) can be lost when delivered via the Transport-Service-Provider and TR-PE input/output buffers (collectively modelled as *channels*). We use a separate PROMELA process `proctype Stealing()` to model this characteristic of the protocol environment (see page 54). It is to determine if the protocol design can recover from the environmental error, through the mechanism of the "re-transmission until acknowledgment" feature.

Assumption 3.8 Non-deterministic TID Verification. TID Verification is another protocol feature to deal with the receipt of out-of-order or duplicated *Invoke* messages. On receiving a new Invoke PDU, the TR-Resp-PE is modelled to make an arbitrary choice between initiating a TID verification or not. This allows not to model

[17]For clarity, only actions of sending/receiving messages and submitting/delivering primitives are shown, so there are gaps between the serial numbers.

a specific TID caching mechanism[18], since only one transaction (hence, a fixed TID) is considered.

Assumption 3.9 Conditions for process states remaining unchanged. The protocol operation is mainly defined in the 9 state tables listed in Appendix B. For completeness, we assume that the two processes (TR-Init-PE and TR-Resp-PE) involved in the operation will maintain evaluations of local/global variables (i.e., their current states), when the variables take other values than those defined in the "Event" and "Condition" columns.

Process states are also assumed to remain unchanged when the arriving PDU is *ignored*, which includes two cases: (1) the following "Action" column is defined as "*Ignore*"; (2) the PDU reaches in a TR-PE state (see Table 13) where it is not required, due to re-transmission or overtaking. For example, there is no *RcvAck* event defined in Table 38, since an ACK PDU is not expected when TR-Init-PE is in the *Result_Response_Wait* state. So ignored PDUs will be left in the communication channels. But they will not block the following PDUs from being received, as the channels allow for re-ordering of PDUs and the channel sizes are defined not smaller than the maximum number of PDUs buffered in them (see Section 4.2.3).

Assumption 3.10 Selective PDU header fields. The PDU header fields modelled are listed in Table 3. Other fields, e.g., *TID, AbortType*, *AbortReason*, etc., are not modelled, because they have no impact on the behaviour of a single transaction.

Assumption 3.11 Timer with no actual timing. In the original protocol design, a local timer was used to count the *re-transmission interval* (R), the *acknowledgement interval* (A) and the *wait timeout interval* (W). We consider, however, that the changing of TR-PE states is not based on the actual timing (number of seconds), but on the possible event of time-out (which can be seen as a binary event). Therefore the timer is modelled as a local `bit`, i.e., set to 1 when the actual timer starts and 0 on stopping the timer. This maintains the basic functionality of the TR-Protocol timers, while simplifying the state space analysis. We will discuss in Section 4.2.2 whether this assumption may lead to find an error that might be physically impossible, or say an error might only occur under extreme conditions in the real networks.

Tables 38 and 42 indicate that PDUs are queued for a certain time interval before being sent. This mechanism is simplified by directly sending PDUs to the outgoing channel, which is independent of their queuing discipline, because the TR-PEs simply view the communication channel as allowing re-ordering.

Assumption 3.12 Selective counters. Two counters were defined to count the number of times timer intervals R and A expire, both of which have a maximum value.

[18]By using TID caching, the last TID is stored for comparison with the current one, to help judge if TID verification is necessary.

We model the Re-transmission Counter (RCR) explicitly, because a time-out on interval R triggers transmission of PDUs, impacting the subsequent operation of the TR-Protocol. The other one, the Acknowledgment Expiration Counter (AEC), is modelled non-deterministically, for time-out on interval A only causes the current transaction to abort. Based on this assumption, our verification results will be independent of the value of AEC_MAX, but dependent on RCR_MAX.

Assumption 3.13 Abort initiated by Transport-Service-Provider. In addition to the aborts given in the state tables, which are initiated by TR-Users and TR-PEs, we model the abstract handling of errors in the lower layers of the protocol stack, i.e., abort initiated by the Transport-Service-Provider. Abort can occur non-deterministically in any step of one transaction.

3.4.2 Model Structure

The model structure for the TR-Protocol is similar to that of TR-Service, except that there are more variables and guard conditions needed to implement those protocol features. The automata models for the *Initiator* and *Responder* processes in TR-Protocol are given as: $A_i = (S_i, s_{0_i}, \Sigma_i, \delta_i, F_i)$, where $i = PI$ or PR. Each component is defined in the following subsections.

State and Related Variables The set of states $S_i : V_i \rightarrow D_i$ maps the set of variables $V_i = (Global_i, Local_i)$ to the set of corresponding ranges D_i.

The set of global variables $Global_i = (channels, global\_control, toggles)$ is a tuple, where

- Both *channels* are a tuple (*chan_name, chan_size, message_stored*). For the TR-Protocol model, *message_stored* is an ordered set of variables, $mtype \times bit \times bit \times bit$, which correspond to the header fields in Table 3. The first field of PDU type is modelled symbolically using the PROMELA `mtype` declaration, which ranges over {INVOKE, ACK, RESULT, ABORT}. The next two fields (if applicable) are modelled as two bits, and the third bit is reserved for modification. Figure 18 shows the declaration of two channels [19] using PROMELA syntax. The messages can be randomly taken out of the (channel) array without respect to their queuing discipline, which models the reordering occurred in the transmission medium.

- *Global_control* $= \{Aflag\}$. The bit $Aflag$ is still needed to guard the atomic action sequence as described in Figure 15. The other two controls, *Ack-Type* and $NoAck$, previously defined in TR-Service are not used, as their functions are realized by the U/P flag in the INVOKE PDU header field.

[19] The two channel sizes defined here are under the configuration where RCR_MAX is 2 on the Initiator side and 1 on the Responder side.

```
mtype = {INVOKE, ACK, RESULT, ABORT};
chan Init2Resp = [4] of {mtype, bit, bit, bit};
chan Resp2Init = [6] of {mtype, bit, bit, bit};
```

Figure 18: Declaration of channels in the TR-Protocol model

- $Toggles = (Primitive\_tgls, DLock)$ is a tuple. $Primitive\_tgls$ has 14 elements, recording the occurrence of those service primitives. $DLock = \{DLock\_I, DLock\_R\}$ has two bit-typed elements to signal deadlock in either process execution.

The set of local variables $Local_{PI} = (istate, local\_control)$ is a tuple, where

1. The value of $istate$ represents the states of TR-Init-PE as defined in Table 13; they follow the names as given in the state tables (in Appendix B).

Table 13: TR-PE states on the Initiator side

State (abbr.)	**Description**
I_Null (I_NUL)	Same as that defined in Table 4.
I_Result_Wait (I_RW)	A combination of I_IRW and I_RW defined in Table 4.
I_Result_Resp_Wait (I_RRW)	Same as that defined in Table 4
I_Wait_Timeout (I_WT)	Wait for the time interval W to expire before returning to the initial state I_Null.

2. $Local\_controls = (UserAck, idata)$ is a tuple, where

   - The bit $UserAck$ has the same functionality as defined for the TR-Service model (see page 38).
   - The set $idata = \{RCR, Timer, Uack, AckSent, HoldOn\}$ consists of the retransmission counter RCR, the (bit) Timer and other bit variables described in the WTP design, which are used only on the Initiator side.

Similarly on the Responder side, $Local_{PR} = (rstate, local\_control)$ is a tuple, where

1. The value of $rstate$ represents the states of TR-Resp-PE. In addition to the 4 states which have the same meanings as defined in Table 9, there is another state R_TIDOK_WAIT required (see Table 14).

Table 14: TR-PE states on the Responder side

State (abbr.)	**Description**
R_Listen (R_LIS)	Same as that defined in Table 9.
R_TIDOK_WAIT (R_TW)	Wait for a positive acknowledgement of TID verification sent from the Initiator (see Figure 41).
R_Invoke_Resp_Wait (R_IRW)	Same as that defined in Table 9.
R_Result_Wait (R_RW)	Same as that defined in Table 9.
R_Result_Resp_Wait (R_RRW)	Same as that defined in Table 9.

2. $Local\_controls = (FirstInvoke, rdata)$ is a tuple, where

   - The bit $FirstInvoke$ has the same functionality as defined for the TR-Service model (see page 43).
   - The set $rdata = \{RCR, Timer, Uack, AckSent\}$ consists of the retransmission counter RCR, the (bit) Timer and other bit variables described in the WTP design, which are used on the Responder side.

Initial State s_{0_i} gives the initialization of variables, shown as Table 15.

Table 15: Initial state of the TR-Protocol model

i	V_i	**Type of** D_i	$s_{0_i}(V_i)$
	2 channels	chan	empty
	Aflag	bit	0
	14 Primitive_tgls	unsigned	0
	DLock_I	bit	0
	DLock_R	bit	0
PI	istate	mtype	I_NULL
	UserAck	bit	ACK_TYPE (1 or 0)
	idata.RCR	unsigned	0
	idata.Timer	bit	0
	idata.Uack	bit	0
	idata.AckSent	bit	0
	idata.HoldOn	bit	0
PR	rstate	mtype	R_LISTEN
	FirstInvoke	bit	1
	rdata.RCR	unsigned	0
	rdata.Timer	bit	0
	rdata.Uack	bit	0
	rdata.AckSent	bit	0

Guard Alphabet and Transition Relation Each symbol a in the alphabet Σ_i is a first-order formula, defined as

$$a(channels, global\_control, Local_i) = L(s).$$

Label function $L : S_i \rightarrow 2^{\mathcal{V}}$ maps each current state to a set of "true" propositions, where $\mathcal{V} = \{v = d \mid v \in V_i,\, d \in D_i\}$.

The transition relation, $\delta_i : S_i \times \Sigma_i \rightarrow 2^{S_i}$, gives the new evaluations of variables in state $\delta_i(s, a)$ when the symbol a is true in the current state s. When $\mid \delta_i(s, a) \mid \geq 1$, or for all symbols which are true at the same time, the choice of a new evaluation of variables is non-deterministic.

Table 16: TR-Init-PE state change (current state = *I_Result_Wait*)

Guard	**Action**		original	transition
channel $\wedge$ *local_ctrl*	*channels* $\wedge$ *local_ctrl*	$\wedge$ istate	entry #	name
{ACK,RID,0,tmp} $\in$ Resp2Init idata.HoldOn = 0	Resp2Init $-$ {ACK,RID,0,tmp} SetHoldOn(idata) StopTimer(idata)	I_RW	2	ICNF
{ACK,RID,1,tmp} $\in$ Resp2Init idata.RCR $<$ RCR_MAX_I	Resp2Init $-$ {ACK,RID,1,tmp} Init2Resp $\cup$ {ACK,0,1,tmp} SetAckSent(idata) IncRCR(idata) StartTimer(idata)	I_RW	5	RcvAck_TIDok
idata.Timer = 1 idata.RCR $<$ RCR_MAX_I idata.AckSent = 0	Init2Resp $\cup$ {INVOKE,1,idata.Uack,tmp} IncRCR(idata) StartTimer(idata)	I_RW	9	T2R_2ndINVOKE
idata.Timer = 1 idata.RCR $<$ RCR_MAX_I idata.AckSent = 1	Init2Resp $\cup$ {ACK,1,1,0} IncRCR(idata) StartTimer(idata)	I_RW	10	T2R_2ndTIDok
idata.Timer = 1 idata.RCR = RCR_MAX_I	ClearI(idata) AINDP_tgl + 1	I_NUL	11	AINDP_T2R
{RESULT,RID,tmp,tmp} $\in$ Resp2Init idata.HoldOn = 1	Resp2Init $-$ {RESULT,RID,tmp,tmp} RIND_tgl + 1 StartTimer(idata)	I_RRW	12	RIND
{RESULT,RID,tmp,tmp} $\in$ Resp2Init idata.HoldOn = 0	Resp2Init $-$ {RESULT,RID,tmp,tmp} ICNF_tgl + 1 RIND_tgl + 1 StartTimer(idata)	I_RRW	13	ICNF_RIND
else	*no change*		-	-

We take Table 16 (which is based on Figure 37) as an example to explain how we formalize the guard alphabet and transition relations from the original TR-Protocol design. The "Guard" column combines the "Event" and "Condition" columns in the original state tables. Hence, the conjunction of atomic propositions on the incoming channel and the global/local controls, as well as *current* state of the local TR-PE, defines a symbol which serves as a guard for state transitions. The "Action" column defines the new evaluations of variables, including the update of the TR-PE state.

Treatment of Events The event of receiving a PDU from the peer TR-PE (e.g., *RcvAck* in entry 5), together with the conditions on its header field (e.g., $TIDve =$

1 and $RCR < MAX\_RCR$), is modelled as a symbol which says a corresponding message is buffered in the incoming channel, i.e., $a = (\{ACK, RID, 1, tmp\} \in Resp2Init) \wedge (idata.RCR < RCR\_MAX\_I)$. When a field of the message stored are not specified with bit value 0 or 1 (e.g., the second field RID and the forth one, tmp), it can take any values to make this symbol true.

The second kind of event, time-out (e.g., TimerTO_R in entry 9), is modelled as a proposition $idata.Timer = 1$, which assumes that time-out can only occur when the timer is currently turned on.

The third kind of events, receipt of *request* or *response* primitives from the TR-User (e.g., indicated as *TR-Abort.req* in entry 1), has no actual conditions on the local/global variables. So they are modelled as switching corresponding primitive toggles in the "Action" column, to book-keep their occurrence.

It is possible that several events can happen simultaneously. For instance, the guards of entries 5, 9 and 12 can be true at the same time. But only one transition can be taken at a time, and the choice is made non-deterministically.

Clarification of Actions The update of the RID field in outgoing PDUs (e.g., resend ACK(TIDok) in entry 10), which is important to the global variable *channels*, was not explicitly described in the WTP design document. Based on our understanding of the protocol operation, we define the update as follows :

1. If an ACK PDU is sent upon receipt of a re-transmitted INVOKE or RESULT PDU, the RID field of this ACK is set to 1. This means the local TR-PE has to resend an acknowledgement, because the peer TR-PE has resent an INVOKE or a RESULT as the result of missing the first acknowledgement.

2. If any PDU is sent with the local RCR being incremented, its RID is set to 1, which means this PDU is a re-transmitted one.

3. In any cases other than the ones stated above, the RID field remains zero.

As to the update of variables, some common actions, such as sending PDUs or generating service primitives, can be modelled in a similar way as described in the TR-Service model (see page 39). For modelling of local $(r/i)data$ updating actions (e.g., StopTimer in entry 2) which are explicitly described in the original state tables, we refer to Appendix C.2.2.

Reorganization of state table entries We add into Table 16 two columns of *entry numbers* and *transition names* for easy reference to the original TR-PE state table (Figure 37), but it can be seen that not every entry in Figure 37 is modelled. The original entries 3 and 6 are not modelled, since their corresponding actions are defined as "ignore", which means there will be no changes in the process. Entry 4 is not modelled, for it only relates to Class 1 service. *Transaction aborts* described in entries 1

and 7 (and similar entries in other state tables) are modelled collectively in Table 17 as error handling in the *Transaction*-Service-Provider (i.e., TR-PE). In the same table, the two entries next-to-the-last model an error initiated by the *Transport*-Service-Provider (TSP) [18], which occurs in the lower layer of the protocol stack.

Table 17: Modelling of TR-Protocol Error Handling

Guard		**Action**		original	transition
$(i/r)state$	$\wedge local\_ctrl \wedge channel$	$channels \wedge local\_ctrl \wedge$	$(i/r)state$	entry #	name
¬ I_NUL		AREQ_tgl + 1 Init2Resp ∪ {ABORT,0,0,0} ClearI(idata)	I_NUL	Fig. 37-1 Fig. 38-4 Fig. 39-7	AREQ
¬ I_NUL	{ABORT,tmp, tmp, tmp} ∈ Resp2Init	AIND_tgl + 1 Resp2Init − {ABORT,tmp,tmp,tmp} ClearI(idata)	I_NUL	Fig. 37-7 Fig. 38-3 Fig. 39-4	AIND
¬ R_LIS ¬ R_TW		areq_tgl + 1 Resp2Init ∪ {ABORT,0,0,0} ClearR(rdata)	R_LIS	Fig. 42-5 Fig. 43-6 Fig. 44-1	areq
¬ R_LIS	{ABORT,tmp, tmp, tmp} ∈ Init2Resp	aind_tgl + 1 Init2Resp − {ABORT,tmp,tmp,tmp} ClearR(rdata)	R_LIS	Fig. 41-3 Fig. 42-6 Fig. 43-7 Fig. 44-2	aind
¬ I_NUL		AIND_TSP_tgl + 1 ClearI(idata)	I_NUL	-	AINDP_TSP
¬ R_LIS		aind_TSP_tgl + 1 ClearR(rdata)	R_LIS	-	aindp_TSP
	else	*no change*		-	-

The PROMELA implementations of the transitions, as well as the Initiator and Responder processes, are similar to the structures shown in Figures 15 and 16. We can see both structures of a single transition (i.e., a PROMELA `inline` sequence) and the whole process (a `proctype` template), have rather fixed formats, so that we can use GNU m4 [47], a macro processor, to generate the PROMELA code. As can be seen from Appendix C.2.3, the implementation of Table 16 can follow a straightforward pattern. Those tens of inline transitions used for one process can be generated in a batch through a common structure which is based on Figure 15.

Final State Table 18 shows the final evaluations of local variables for both Initiator and Responder processes. For example, $F_{PI} = \{s \in S_{PI} \mid L(s) = (UserAck = \neg\, ACK\_TYPE) \wedge (istate = I\_NUL) \wedge (idata = 0)\}$ implies that a local sequence of service primitives has been generated on the Initiator side for one transaction. Again, the alphabet Σ_i in each process is complemented with two guards $\{timeout, L(s_{F_i})\}$ which enable the do-loop to terminate properly.

Modelling of Lossy Channels We have modelled the retransmission feature of the TR-Protocol (e.g., entry 9 in Table 16), which was designed to cope with the errors (specifically, loss of messages) caused by the lower layer services. So it is natural to model

Table 18: Final state conditions of the TR-Protocol model

i	V_i	**Type of** D_i	$s_{F_i}(V_i)$
PI	istate	mtype	I_NUL
	UserAck	bit	¬ACK-TYPE
	idata.RCR	unsigned	0
	idata.Timer	bit	0
	idata.Uack	bit	0
	idata.AckSent	bit	0
	idata.HoldOn	bit	0
PR	rstate	mtype	R_LISTEN
	FirstInvoke	bit	0
	rdata.RCR	unsigned	0
	rdata.Timer	bit	0
	rdata.Uack	bit	0
	rdata.AckSent	bit	0
	$Global_i$	*do not care*	

lossy channels and verify the model of the TR-protocol in such a faulty environment. If all the properties we desire are proved with the existence of lossy channels, we gain confidence that the protocol is indeed designed to be able to recover from the error and preserve the correctness requirements.

There are several ways to model lossy channels in PROMELA, as described in [44]. We choose to use a separate `Stealing` process, because it does not affect the present transition structure, saving the trouble of severe modification. The cost is bearable: although an additional 4 bytes in the state vector are required for an extra process, the RAM space requirement increases not more than 4%.

```
1       process Stealing( ) {
2       end:    do
3               :: Init2Resp??_, _, _, _
4               :: Resp2Init??_, _, _, _
5               do
6       }
```

Figure 19: Pseudo-code for the *Stealing* process

Figure 19 shows the structure of the `Stealing` process. It runs in parallel with the product automaton $A_{MP} = A_{PI} \times A_{PR}$ in an interleaved way. In any state $s \in S_{MP}$, as long as either channel (*Init2Resp* or *Resp2Init*) is not empty, a PDU may be stolen by executing the PROMELA *random receive* operation (e.g., `Init2Resp??_, _, _, _`). PROMELA's write-only variable '`_`' is used to store the lost PDU, since we do not care

about its contents.

3.4.3 Sample Simulation Run

Given a PROMELA model which can be generated from the GNU m4 code in Appendix C.2, SPIN can produce a Message Sequence Chart of a simulation run on the TR-Protocol model $A_{PI} \times A_{PR}$ with the `Stealing` process. In Figure 20, those rectangles along the three instance axes show the actions of sending/receiving PDUs, as well as the corresponding transition names (prefixed by TR-PE state table names and entry numbers).

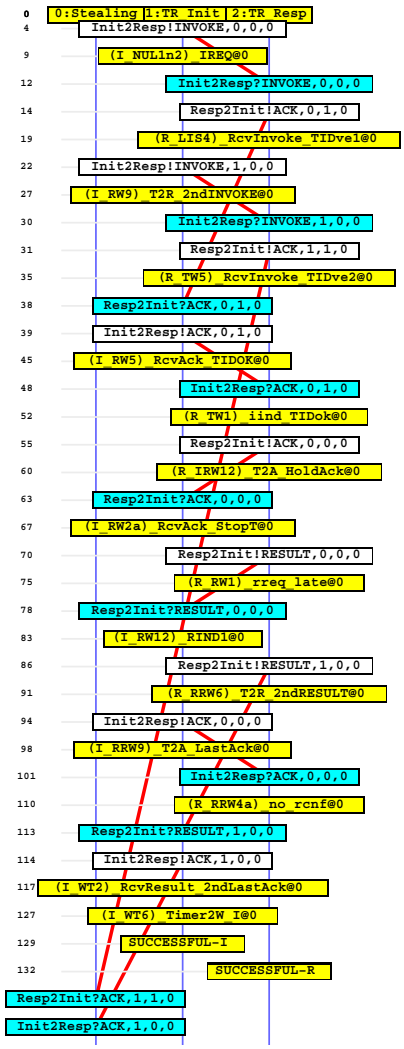


Figure 20: A sample simulation run on the TR-Protocol PROMELA model

Referring to page 32, we can see this MSC illustrates the very transition iteration

which has been depicted in part (b) of Figure 12. Steps 22 to 27, 86 to 91 and 114 to 117 model the retransmission of INVOKE, RESULT and ACK PDUs, respectively, upon time-out events. Steps 14 to 52 model the three-way handshaking during a TID verification (note that the second bit of ACK PDUs, the Tve/Tok flag, is set to one). Steps 55 to 60 show a hold-on acknowledgement sent from TR-Resp-PE. The two TR-PE processes terminate on signalling the transaction "successful" (steps 129 and 132), generating a global sequence of service primitive $IREQ$ (step 9) $\rightarrow$ $iind$ (step 52) $\rightarrow$ $rreq$ (step 75) $\rightarrow$ $RIND$ (step 83). The two ACK PDUs which are not accepted by either TR-PE are finally "stolen" by the `Stealing` process.

We use Figure 20 to show that our TR-protocol model, when driven by the PROMELA semantic engine in SPIN (see Appendix A.2), can capture the main features of the protocol operation. Based on this model, SPIN can produce a similar MSC to show the error trail of a property which is verified to be unsatisfiable. This will facilitate the detection of the cause of the error, either in the modelling phase or in the original protocol design.

3.5 Specification of Correctness Requirements

In this section, we will classify the correctness requirements for a well-formed protocol design, and express the desired properties of WTP in LTL syntax and PROMELA statements. Consequently, we will obtain a set of property specifications to be checked, denoted as P (consistent with Section 10). It consists of three subsets, namely P_S, P_L and P_T, which correspond to safety properties, liveness properties and temporal behaviours, respectively.

3.5.1 Safety Properties

A safety property p_S generally says "nothing bad happens". We consider it to specify conditions that must not be violated at any time instance. Formally, given the protocol automaton A_M, $(\forall p_S \in P_S)$ $(\forall s \in S_M)$ $p_S \in L(s)$, i.e., any safety property p_S is true in any state s.

For consistency of variables, safety requires:

- The protocol is bounded, i.e., each variable used cannot overflow its range. Given $S_M : V_M \rightarrow D_M$, $(\forall v \in V_M)$ $(\forall s \in S_M)$ $s(v) \in D_M$. We are especially interested in the channels and local counters used in the TR-Protocol.

  Given two channels: $chan\_name_1 = Resp2Init$ with $chan\_size_1 = R2ISIZE$ and $chan\_name_2 = Init2Resp$ with $chan\_size_2 = I2RSIZE$, the function $len(chan\_name) \in s$ returns the current number of messages stored and we give safety property 1 as

$$p_{S_1} = (len(Resp2Init) \leq R2ISIZE) \wedge (len(Init2Resp) \leq I2RSIZE). \quad (8)$$

Given two local retransmission counters: $counter_1 = idata.RCR$ with $maximum_1 = RCR\_MAX\_I$ and $counter_2 = rdata.RCR$ with $maximum_2 = RCR\_MAX\_R$, we have safety property 2 as

$$p_{S_2} = (idata.RCR \leq RCR\_MAX\_I) \wedge (rdata.RCR \leq RCR\_MAX\_R). \quad (9)$$

- Pure atomicity, i.e., updating of shared channels/variables must be mutually exclusive.

  Referring to Figure 15, given the global control $Aflag$, safety property 3 is

$$p_{S_3} = (Aflag = 1). \quad (10)$$

For progress of execution, safety requires:

- No deadlock, i.e., $(\forall s \notin F_M)\ (\exists\, a \in \Sigma_M)\ \delta_M(s, a) \neq \emptyset$; $(\forall s \in F_M)\ \delta_M(s, L(s)) = s$, where $L(s)$ is in accordance with those final state conditions defined in Tables 8, 12 and 18.

  Based on the process structure shown in Figure 16, it is desired that the `timeout` condition is never enabled, hence the toggle bit $DLock\_I$ (or $DLock\_R$) remains unchanged from its initial value. So, safety property 4 is

$$p_{S_4} = (DLock\_I = 0) \wedge (DLock\_R = 0). \quad (11)$$

- The protocol is not under-specified, i.e., there is no such case that a message reaches a receiver that cannot respond to it.

  So we define that if a message or a PDU is buffered in the incoming channel, but not used in any guard conditions in the SAP/TR-PE state tables, then it is ignored and the current state of the process automaton remains unchanged (see Assumption 3.9). This is a complement of those receiving-message actions, to ensure the protocol is specified with completeness.

Actually, each p_{S_i} $(i = 1...4)$ defined above is an invariance property of the model A_M, expressed in LTL formula as $A_M \models \mathcal{G}p_{S_i}$. It is equivalent (and in some cases, more efficient) to verify $A_M \not\models \mathcal{F}(\neg\, p_{S_i})$. So instead of checking each LTL formula $\mathcal{G}p_{S_i}$ separately, we can use a monitor automaton A_m to run in parallel with the model A_M, reporting any violation of these invariants in all computations of A_M.

Assume $V_m = \{Resp2Init,\ Init2Resp,\ idata.RCR,\ rdata.RCR,\ Aflag,$ $DLock\_I,\ DLock\_R\}$, $D_m = \{d_m \mid d_m = s_M(v_m)\}$ (i.e., each d_m is the current evaluation of $v_m \in V_m$ in the protocol model A_M), we define $A_m = (S_m, s_{0_m}, \Sigma_m, \delta_m, F_m)$ as follows:

- $S_m = S'_m \times \{good, bad\}$, where $S'_m : V_m \rightarrow D_m$.
- $s_{0_m} = s_{0_M} \times \{good\}$.

- $\Sigma_m = \{\neg\, p_{S_i} \mid i = 1...4\}$.
- $\delta_m(s_m, a) = s_{F_m}$, where $a \in \Sigma_m$ and $s_{F_m} \in F_m$.
- $F_m = S'_m \times \{bad\}$, i.e., each final state is attached with a sign *bad*, but the current evaluations of variables are not changed.

Then we can define a new automaton $A_m \times A_M$ in a similar way as we did for A_M (see page 45), where the transition relation of $A_m \times A_M$ defines that A_m runs with A_M in an interleaved way. In the verification phase, SPIN will perform a depth first search on $A_m \times A_M$ to locate any bad states. It is also very helpful in the modelling phase, since a simulation run on the model for a sanity check will stop on the computation path where an invariant is violated.

The above property automaton A_m can be implemented as a PROMELA process, shown in Figure 21. A PROMELA statement `assert(`p_{S_i}`)` can be used to report whether assertion p_{S_i} is violated and then locate a bad state. In the guarded sequence, the `assert(`p_{S_i}`)` statement is only executable when p_{S_i} is not true; this is more efficient than using `assert(`p_{S_i}`)` without a guard (i.e., to verify $\mathcal{G}p_{S_i}$ directly) as proposed in [50].

```
1       process Monitor( ) {
2       end:    atomic {
3                       if
4                       :: if (p_S1 = FALSE)    { Report p_S1 violated }
5                       :: ...
6                       :: if (p_S4 = FALSE)    { Report p_S4 violated }
7                       fi
8               }
9       }
```

Figure 21: Pseudo-code for the *Monitor* process

It is noted that SPIN requires each PROMELA process either terminate on its closing curly bracket or stay in a state labelled with `end`. In the previous implementations of A_i ($i = SI, SR, PI, PR$), a final state is always reachable by the transition relation $\delta_i(s, L(s_{F_i}))$ or $\delta_i(s, timeout)$. However, in the monitor process, when none of the p_{S_i}'s are ever false (which is desirable), the atomic sequence never becomes executable to reach the end of the process. So an `end` label is needed at the beginning of the process, to avoid an unexpected "invalid-end-state" report from SPIN.

3.5.2 Liveness Properties

The liveness property says "something good happens". We consider it to specify conditions that will eventually be reached. For progress of execution, liveness requires:

- The protocol does not contain unreachable states, due to either over-specification of the design or occurrence of unexpected behaviours. Formally, let Σ_M^* be the set of all possible words in the model A_M, then $(\forall s_i \in S_M)\ (\exists\, \omega_{0(i-1)} \in \Sigma_M^*)$ $s_i \in \delta_M^*(s_0, \omega_{0(i-1)})$, where $\omega_{0(i-1)} = a_0 a_1 ... a_{i-1}$ and δ_M^* is the extended transition relation as defined in Section 2.1.2.

  When verifying the PROMELA model, SPIN will report the "unreached statements". Later we will see in the TR-Protocol model, there are some *expected* unreachable statements due to limitations imposed by the protocol's configuration. Apart from the expected, other unreached statements are undesirable.

- No starvation: both the Initiator and Responder processes can eventually access shared variables.

  In the corresponding PROMELA processes, we can label the start of each do-loop as a `progress` state (see Figure 16). The label specifies that it cannot be postponed infinitely long to select those non-deterministic options within either do-loops, and each atomic entry must be executed to make progress. In the computations of the PROMELA model, any cyclic sequence that does not contain a `progress`-labelled state will cause either the Initiator or the Responder to starve. SPIN will report any "non-progress cycles".

3.5.3 Temporal Behaviours

Temporal behaviours mainly deal with the ordering of the service primitives in the local/global primitive sequences. Using those toggle variables *Primitive_tgl*s as defined previously, these temporal requirements can be specified in a set of LTL formulas p_{T_i}, which make up the property set P_T.

End-to-end behaviour in the global sequence The *end-to-end principle* (introduced on page 31) says for the *global* sequence of primitives exchanged between peer TR-Users, we require those primitive types occur in the order (*request, indication, response,* then *confirm*). It is easy to guarantee that *indication* primitives always precede the *responses*, because we have previously defined that within one local process, the guard for submitting *response* becomes true only after *indication* has been delivered (see Table 6 entries 4, 5 and Table 11 entries 1, 2). It is, however, more difficult to guarantee that primitives in different TR-User/TR-PE processes occur in the required order, i.e., *request* precedes *indication*, and *response* comes before *confirm*. If it is verified, we can assure that in the global sequence, a primitive type cannot occur before any of its predecessors have occurred.

Using the concepts in Dwyer's Specification Pattern System [13], the requirement on the above orderings can be expressed as a *precedence* pattern in the *global* scope: in the entire execution of the model, the occurrence of the *cause* is a necessary pre-condition for an occurrence of the *effect*. The occurrence of the *effect* is not allowed without

the occurrence of the *cause*, however, *causes* are allowed to occur without subsequent *effects*. For example, if the transaction is aborted after a *request* (or *response*) has been submitted, there is no *indication* (or *confirm*) to be delivered.

With LTL syntax, the above precedence pattern can be expressed as $\mathcal{G}(\neg\, effect\, \mathcal{W}\, cause)$, where $\mathcal{W}$ is the *weak until* operator. Since SPIN only supports the *strong until* operator $\mathcal{U}$, we can specify it equivalently as

$$p_{T_i} = \mathcal{G}((\mathcal{F}\, effect) \Rightarrow (\neg\, effect\, \mathcal{U}\, cause)). \tag{12}$$

To be recognized by SPIN, Formula (12) needs to be re-written with PROMELA syntax as `[]((<>effect) -> (!effect U cause))`, where `effect` and `cause` are of boolean type.

Table 19: Cause-effect primitive pairs with end-to-end behaviours

p_{T_i}	***effect***	***cause***
1	iind	IREQ
2	ICNF	ires
3	RIND	rreq
4	rcnf	RRES
5	aind	AREQ $\vee$ AINDP
6	AIND	areq $\vee$ aindp

We define, in Table 19, 6 pairs of cause-effect primitives. Each pair is a submit-deliver duo consisting of primitives on different TR-User/TR-PE sides, which need to be checked for the end-to-end property. Toggle variables *Primitive_tgl* defined in the previous protocol models are actually used to instantiate the boolean expressions *effect* and *cause* in each LTL formula.

Special ordering with User Acknowledgement According to the WTP protocol design, when the feature of User Acknowledgement is turned on (correspondingly, the primitive parameter *Ack-Type* set to 1), *Invoke.res* must be submitted before *Result.req* on the Responder side, and consequently *Invoke.conf* must be delivered before *Result.ind* on the Initiator side. This requirement can also be specified as the previous *precedence* pattern, using Formula (12), though the primitives involved do not necessarily have a cause-effect relation.

$$p_{T_7} = \mathcal{G}((\mathcal{F}\, rreq\_tgl) \Rightarrow (\neg\, rreq\_tgl\, \mathcal{U}\ ires\_tgl)). \tag{13}$$

$$p_{T_8} = \mathcal{G}((\mathcal{F}\ RIND\_tgl) \Rightarrow (\neg\, RIND\_tgl\ \mathcal{U}\ \ ICNF\_tgl)). \tag{14}$$

Local primitives ordering Based on the primitive ordering defined in Table 2, we are to verify the following properties, violations of which are considered to be unreasonable:

- No Abort primitives occur before the first invoke primitive has occurred on either side. It is natural that the Initiator could not abort a transaction before it has submitted the *Invoke.req* primitive, when the actual transaction has not begun yet. Similarly on the Responder side, no abort occurs before the *Invoke.ind* primitive has been delivered.

  This requirement can be expressed as an *absence* pattern in the *before* scope [13], which says proposition ϕ is false up to proposition ψ becomes true. In our case, we denoting ϕ as the occurrence of any Abort primitive, while ψ as the occurrence of *Invoke.req* in the Initiator process (or *Invoke.ind* for the Responder).

  Generally, the pattern can be specified in LTL formula as $\mathcal{G}((\mathcal{F}\,\psi) \Rightarrow (\neg\,\phi\,\mathcal{U}\,\psi))$. Specifically, for the Initiator process, we have

  $$p_{T_9} = \mathcal{G}((\mathcal{F}\ \ IREQ\_tgl) \Rightarrow (\neg\, ABORT\_tgl\ \mathcal{U}\ \ IREQ\_tgl)). \tag{15}$$

  where $ABORT\_tgl = AREQ\_tgl \vee AIND\_tgl \vee AINDP\_tgl$.

  For the Responder process, we have

  $$p_{T_{10}} = \mathcal{G}((\mathcal{F}\ \ iind\_tgl) \Rightarrow (\neg\, abort\_tgl\ \mathcal{U}\ \ iind\_tgl)). \tag{16}$$

  where $abort\_tgl = areq\_tgl \vee aind\_tgl \vee aindp\_tgl$.

- No Abort primitives occur after *Result.cnf* has been delivered on the Responder side[20].

  This requirement can be expressed as an *absence* pattern in the *after* scope [13], which says, in our case, proposition ϕ (denoting the occurrence of any Abort primitive) is false after proposition ψ becomes true, where ψ denotes the occurrence of *Respond.cnf* in the Responder process.

  Generally, the pattern can be specified in LTL formula as $\mathcal{G}(\psi \Rightarrow \mathcal{G}(\neg\,\phi))$. So we have the following specifications:

  $$p_{T_{12}} = \mathcal{G}(rcnf\_tgl \Rightarrow \mathcal{G}(\neg\, abort\_tgl)). \tag{17}$$

  where $abort\_tgl$ are defined in Formula (16).

[20]It is noted that, according to Table 2, the Initiator could still abort the current transaction after it has submitted *Respond.res*. This provides the Initiator an additional "safety exit" from the *Wait_Timeout* state to its initial *Null* state when it is waiting to finish the current transaction.

3.6 Summary

Following the `Formalize` procedure in Figure 10, we presented in this section two PROMELA models of different abstraction levels, namely TR-Service and TR-Protocol, for the WTP protocol design version 2.0. They were both built on a formal basis of finite state automata (complemented with *Horizontal Condition Table*s), as well as under reasonable assumptions to obtain a manageable state space.

As to the correctness requirements, safety properties and desired temporal behaviours were given in 15 formulas, i.e., $P_S = \{p_{S_i} \mid i = 1 \sim 4\}$ and $P_T = \{p_{T_i} \mid i = 1 \sim 11\}$, as shown in Formulas (8) to (17). The temporal behaviours of TR-Protocol are of special interest, because for it to be a faithful refinement of TR-Service, TR-Protocol must be designed to generate a service language (i.e., set of possible primitive sequences) which is a subset of that of TR-Service. We will discuss in the next section how to verify these requirements.

4 Verification of WTP Models

In this section, we present the verification experiments conducted on our WTP PROMELA models. Section 4.1 describes the experimental environment. In Section 4.2, we discuss the design errors revealed from the verification of the TR-Protocol model, and the properties satisfied after modifying the TR-Protocol design. Section 4.3 shows the state space analysis of the PROMELA models and finally, a comparison with results obtained from the Coloured Petri Nets method [18] is given.

4.1 Verification Setup

This section deals with a general experimental setup for verifying PROMELA models (Section 4.1.1), and the specific parameters selected for the analysis of the TR-Protocol model (Section 4.1.2).

4.1.1 Hardware and Software setup

All analyses (modelling and verification) were performed on a computer with specifications as given in Table 20.

The whole process of verifying a PROMELA model using the model checker SPIN is graphically depicted in Figure 22 (extended from page 26 in [45]). It begins with a PROMELA model (`*.pml`) and (when necessary) a set of LTL formulas written with PROMELA syntax, and ends with a set of textual reports to notify whether the PROMELA model satisfies the desired safety, liveness and temporal requirements.

In our verification of the TR-Protocol model, we first parameterize an *m4* source file to generate a PROMELA file which is specifically tailored for verifying one configuration of the TR-Protocol (see Section 4.1.2). Then we can use SPIN to create a

Table 20: Specifications of hardware and software used for analyses

Items	Specification
CPU model	Intel Pentium III (Coppermine)
CPU clock	863.877MHz
RAM	256MB
Operating system	Red Hat Linux 7.3 (Valhalla)
OS kernel release	2.4.18
C compiler	gcc version 2.96 (20000731)
SPIN	version 3.5.3 (20021208)
GNU m4	version 1.4

verifier written in C. This verifier is equivalent to the product automaton, which results from the protocol model (either A_{MS} or A_{MP}) and the property automaton (e.g., A_m for safety or $A_{\neg\psi}$ for temporal requirements). When an LTL formula ψ is presented, it can be mechanically translated into PROMELA's `never claim` structure [26], which is an equivalent of the Büchi automaton $A_{\neg\psi}$.

The verifier source `pan.c` needs to be compiled with a set of directives to produce an *efficient* executable verifier. The purpose of these directives includes choice of correctness requirements (e.g., searching for non-progress cycles or acceptance cycles), choice of search mode (e.g., exhaustive search or bit-hashing) and other optimization options for complexity management. A typical compilation command for an exhaustive search for an unsafe state (e.g., which is an invalid end-state or where an assertion is violated) may look like:

```
gcc -w -o pan -DSAFETY -DCOLLAPSE -DNOCLAIM pan.c
```

where the directive `-DNOCLAIM` modifies the verifier to exclude the `never claim` (if present) from the verification, and `-DCOLLAPSE` can invoke the state vector compression which collapses state vector sizes by up to 80%. For details of other compile-time and run-time options, we refer to [51].

After "pressing the *run* button" (as is done in most model checkers), we expect a waiting period as short as possible before we are prompted with the verification result. If a property is not satisfied, the error trail leading to the violated state has been recorded by SPIN's NDFS algorithm. We only need to perform a *guided* simulation to get an MSC figure of that error trace. This visualized erroneous operation of the model, along with the textual report on the final evaluations of variables, will help us in locating the source of error.

4.1.2 Configuration of TR-Protocol

The configuration of the TR-Protocol is defined by four parameters, as listed in Table 21. Later, we use an ordered list to denote different configurations, e.g., *F-2-1-0* for the default one.

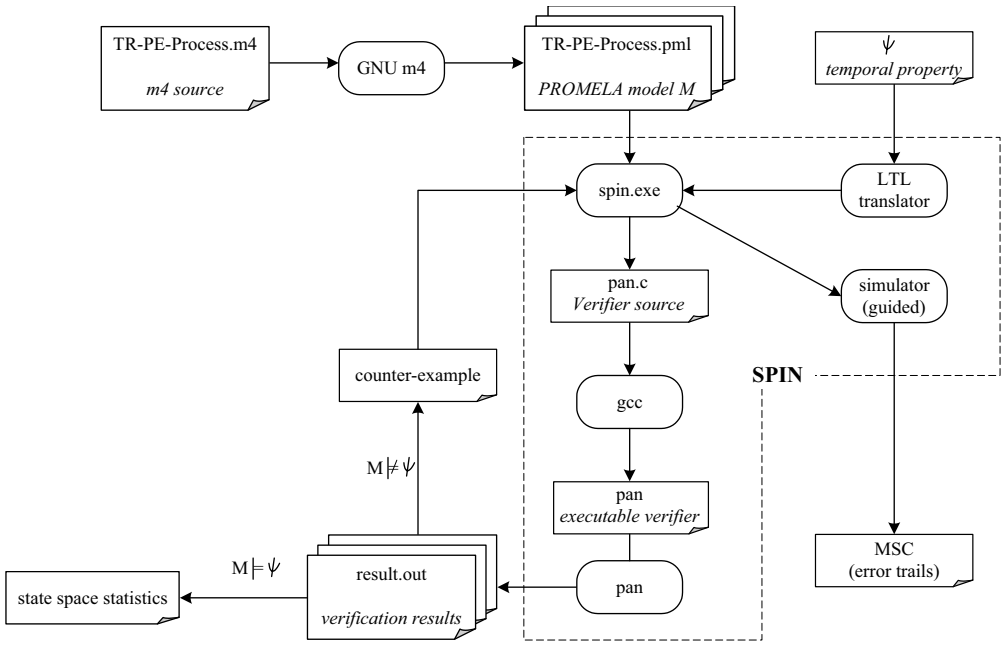


Figure 22: Laboratory setup to verify PROMELA models

Table 21: Default configuration vector of TR-Protocol

#	Parameters	Type	Defaults	Significance
1	ACK_TYPE	bit	0	Indicate whether the UserAck feature is turned on or off.
2	RCR_MAX_I	natural	2	The maximum value of RCR at the TR-Init-PE.
3	RCR_MAX_R	natural	1	The maximum value of RCR at the TR-Resp-PE.
4	LOSSY	bit	0	Indicate whether the two channels are lossy or not.

The parameter ACK_TYPE is used to test the differences in behaviour when the *User Acknowledgement* (UserAck) feature is On or Off. We can see from Table 2, the service language in the situation of UserAck On is a subset of that when UserAck is Off, so our analysis will mainly focus on the latter case. When to verify the special primitive orderings with UserAck On, we will set ACK_TYPE to 1.

As pointed out in Section 3.4.1, our analysis of the TR-Protocol is limited by the maximum value of the counter RCR, RCR_MAX. It could be ideal to prove the desired properties are satisfied under any evaluation of RCR_MAX. But in practice, the WTP protocol is to be implemented with RCR_MAX being only 4 in GSM Short Message Service (SMS) networks (or being 8 in IP bearers, see Appendix A in [60]); on the other hand, to allow a large number of PDU retransmissions will result in poor performance of the protocol. So it is reasonable for us to gain confidence in the correctness of the

TR-Protocol within this parameter limit.

We have gained the experience that different configurations of RCR_MAX will result in big differences in the state space size, which in turn impacts the efficiency of verification (see Section 4.3.1). So we prefer to start with a small value of RCR_MAX, which yields a small state space. On the other hand, the value cannot be too small, otherwise some transitions will be disabled, e.g., TR-Init-PE will not respond to TID verification if $RCR\_MAX\_I < 2$, so that the resulting state space will not be satisfactory for a convincing verification. It can also be estimated that using an additional process to model the lossy channels (as described in Figure 19) will lead to a considerable increase of the state space size. Therefore, we choose $RCR\_MAX\_I = 2$, $RCR\_MAX\_R = 1$ and $LOSSY = 0$ as the starting point, to analyze the behaviours of the TR-Protocol.

To specify different configurations of the TR-Protocol, we can use an *m4* command as follows:

```
m4 -DACK_TYPE=1 -DRCR_MAX_I=4 -DRCR_MAX_I=4 -DLOSSY=1 TR-PE-
Process.m4 > TR-PE-T-4-4-1.pml
```

This will produce a PROMELA model with a configuration of UserAck turned on, $RCR\_MAX\_I = RCR\_MAX\_R = 4$ (for use in GSM SMS networks), and $LOSSY = 1$ (lossy channels).

4.2 Verification Results

In this section, we present verification results on both the TR-Service model and the TR-Protocol model. More emphasis will be given to the description of errors uncovered in the original TR-Protocol design, and the possible modifications.

4.2.1 Verified Properties for the TR-Service Model

The purpose of verifying the TR-Service model is to prove that the PROMELA model we have proposed can indeed generate the service language defined in the WTP design. The control flow defined within the Initiator/Responder processes, as well as other model structures, then can be safely extended for use in the TR-Protocol model.

All the properties specified in Section 3.5 have been proved to be satisfiable in the TR-Service PROMELA model. Figure 23 shows a sample verification result provided by SPIN. The model checking is performed in a full state space search, for the RAM space requirement is very small. Actually, the resulting state vector $40\,byte/state$ multiplied by the total 155 states gives a memory usage less than $2.542\,Mbyte$. This small transition system results from a high level specification and some complexity management techniques, such as partial order reduction and compression in state encoding.

There are no errors reported regarding the assertion violations and invalid end-states. The unreached states are just what we desired, which means their guards are never enabled to falsify the safety properties we defined in Section 3.5.1. Verification

```
(Spin Version 3.5.3 -- 8 December 2002)
    + Partial Order Reduction
    + Compression

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

State-vector 40 byte, depth reached 49, errors: 0
     155 states, stored
     335 transitions

2.542   memory usage (Mbyte)

unreached in proctype TR_Init
    line 242, state 2, "DLock_I = 1"
    line 242, state 3, "printf('MSC: DEADLOCK-I\n')"
unreached in proctype TR_Resp
    line 265, state 2, "DLock_R = 1"
    line 265, state 3, "printf('MSC: DEADLOCK-R\n')"
unreached in proctype monitor
    line 288, state 2, "assert(((len(Resp2Init)<=3) && (len(Init2Resp)<=2)))"
    line 289, state 4, "assert(Aflag)"
    line 290, state 6, "assert((!(DLock_R) && !(DLock_I)))"
    line 293, state 10, "-end-"
```

Figure 23: Verification of safety properties for the TR-Service model

results of the liveness and temporal behaviours are reported in a similar way. After we have obtained these positive results, it is reasonable for us to attribute errors found in the TR-Protocol model to defects in the original WTP design.

4.2.2 Errors in the TR-Protocol Design

Errors revealed from the TR-Protocol model indicate that the TR-Protocol design is not a faithful refinement of the TR-Service definition. We use the default configuration (as described in Table 21) to illustrate 4 main errors which generate some unexpected service languages. But it should be clear that these errors are independent of the values of those protocol parameters; neither are they dependent of one another, in that one error can be reproduced when all the others are fixed.

Error 4.1 Deadlock during TID verification

Description: This error occurs as a violation of the safety property p_{S_4}, no deadlock. Two error trails are shown in Figure 24, from separate guided simulation runs.

In the chart on the left-hand side, the TR-Resp-PE process comes to a deadlock (step 97) after its last transition, `(R_LIS4)_RcvInvoke_TIDve1` at step 66. This last transition corresponds to entry 4 in the *R_LISTEN* state table (Figure 41), whose action is to request a TID verification upon receipt of a delayed INVOKE PDU with invalid

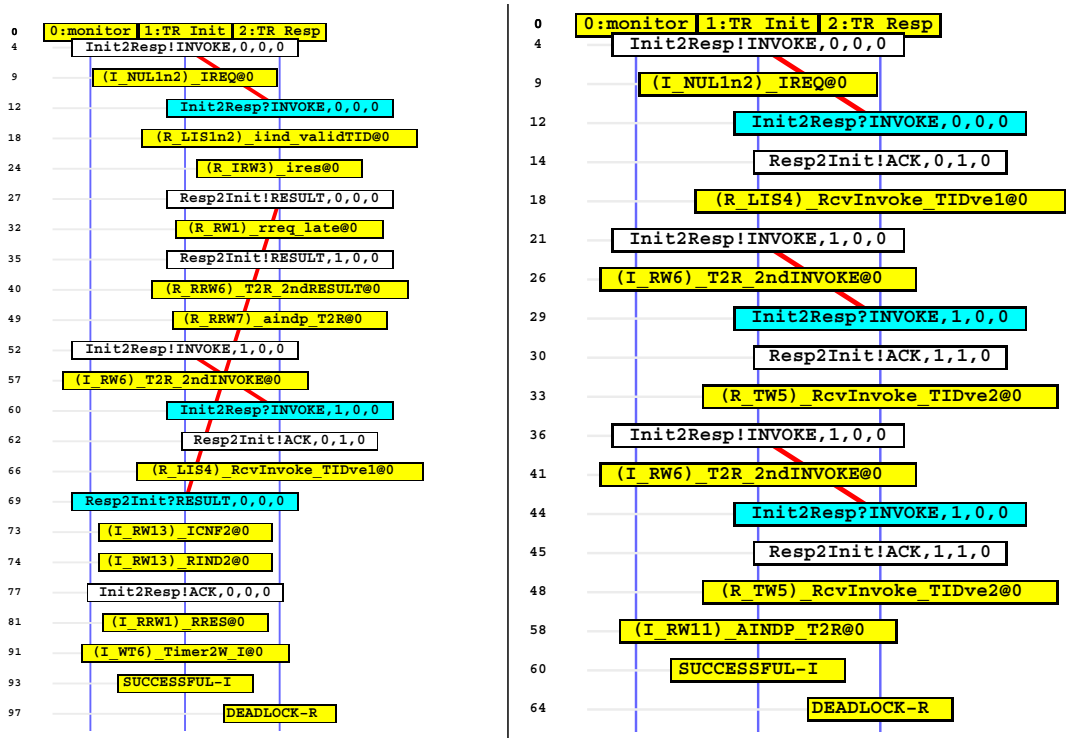


Figure 24: Deadlock trails of TR-Protocol

TID. But on the peer side, the TR-Init-PE (non-deterministically, or simply because the TID verification request arrives late at the Initiator side) chooses to receive a RESULT PDU (step 69), rather than responding to the TID verification, then it completes the current transaction. This leaves TR-Resp-PE blocked in the *TIDOK_WAIT* state after step 66, since there is no new PDU sent to it to enable new updates.

Analysis: At first glance, the last transition happens *after* an abort primitive has been delivered to the Responder (step 49). It was unexpected within one single transition (Assumption 3.4), in that the TR-Resp-PE does not remain in its initial state (*R_LISTEN*) after it has returned to it. Is this the reason to be blamed for the deadlock?

We found another trace, as presented on the right-hand side, which can also lead the TR-Resp-PE process to deadlock. In this chart, TR-Resp-PE has not returned to its initial state, but is again blocked in the *TIDOK_WAIT* state after requesting a TID verification (i.e., `(R_TW5)_RcvInvoke_TIDve2` at step 48). We can see this transition is again triggered by a re-transmitted INVOKE PDU from the TR-Init-PE (steps 36 to 44), which is similar to the situation in steps 52 to 60 in the left-hand chart. Therefore, this suggests that a re-transmitted INVOKE PDU may cause TR-Resp-PE to be trapped in the *TIDOK_WAIT* state.

Suggested solution: Table 22 illustrates what changes (indicated in italics) are suggested to modify the TR-Protocol design (and consequently the PROMELA model). Whenever TR-Resp-PE requests a TID verification, it starts the timer for an interval W. When the interval expires, TR-Resp-PE can safely return to its initial state, as can TR-Init-PE (see Figure 39 entry 6). The time interval W is usually long enough so that any ACK PDU sent from TR-Init-PE can reach the Responder side within the average response time.

Table 22: Modification of TR-Protocol to correct Error 4.1

Guard		**Action**		original	transition
rstate	$\wedge$*local_ctrl* $\wedge$ *channel*	*channels* $\wedge$ *local_ctrl*$\wedge$	*rstate*	entry #	name
R_LIS	{INVOKE,RID,UPack, tmp} $\in$ Init2Resp	Init2Resp $-$ {INVOKE,RID,UPack,tmp} Resp2Init $\cup$ {ACK,0,1,0} ... *StartTimer(rdata)*	R_TW	Fig. 40-4	Invk_TIDve1
R_TW	{INVOKE,1,UPack, tmp} $\in$ Init2Resp	Init2Resp $-$ {INVOKE,1,UPack,tmp} Resp2Init $\cup$ {ACK,1,1,0} *StartTimer(rdata)*	R_TW	Fig. 41-5	Invk_TIDve2
	rdata.Timer $= 1$	*ClearR(rdata)*	*R_LIS*	-	*Timer2W_R*

Aside: From the trail on-the-left in Figure 24, we see that the original TR-protocol design does not require a TR-PE terminate its operation immediately on returning to its initial state, even within the context of a single transition. As long as they do not generate new primitives, we can treat some "extra" transitions as legal ones, e.g., those to clear up delayed PDUs in the channels.

It is also noted that this error cannot be uncovered if the transition `aindp_TSP` (see Table 17) is enabled unconditionally in every TR-Resp-PE state. This transition was to model a possible abort initiated by the Transport-Service-Provider [18] (as part of the "Error Handling" feature), but it could also become an arbitrary exit from a hang state even when there is actually no errors in the Transport-Service-Provider. To eliminate this side effect, we disable these two transitions (`aindp_TSP` and `AINDP_TSP`) before all the errors have been corrected in the WTP design.

Assumption 3.11 enables this deadlock scenario to happen no matter how good (or poor) the real network condition could be, as long as the bit event of re-transmission time-out occurs on the Initiator side before the *Result.req* primitive reaches. This is a little more conservative (in another word, more strict) than the normal situation, as the re-transmission interval R is usually set longer than the PDU round-trip-time (RTT). But the RTT can only be obtained through statistics, thus varies a lot especially in wireless networks, so a fixed interval R cannot eliminate the deadlock possibility.

Error 4.2 Erroneous restart of primitive sequence

Description: This error appears as a violation of the safety property $p_{S_3} = \mathcal{G}(Aflag = 1)$, pure atomicity. The error trail is produced as Figure 25, along with a textual result of the guided simulation shown in Figure 26.

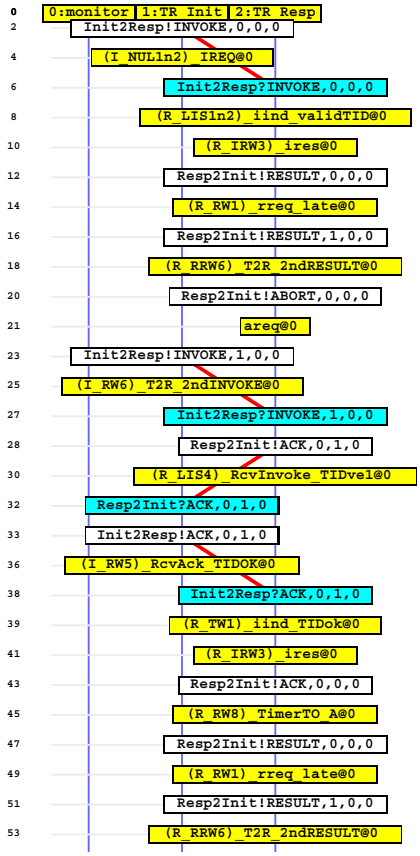


Figure 25: An erroneous restart trail of TR-Protocol

Figure 26 shows the atomicity is broken after step 54, where the TR-Resp-PE is trying to send an ABORT PDU (upon the *Abort.req* primitive) to a full channel (*Resp2Init*)[21].

Analysis: When we re-examine the error trail in Figure 25, we can see it is the second round of the local primitive sequence (starting from step 38) that leads to extra PDUs transmitted to the *Resp2Init* channel. This restart of the primitive sequence is not allowed within the context of a single transition, after TR-Resp-PE has returned to its initial R_LISTEN state (at step 21).

[21]By counting the PDUs from Figure 25, we can see there have already been 6 PDUs buffered, reaching the limit of channel capacity for our default protocol configuration.

```
(step 0 to 52 skipped)
53:    proc  2 (TR_Resp) line 1154 "pan_in" (state 182)
               [printf('MSC: (R_RRW6)_T2R_2ndRESULT)]
54:    proc  2 (TR_Resp) line 729 "pan_in" (state 198)
               [(areq_a && Aflag && !(rstate==R_LISTEN) && !(rstate==R_TIDOK_WAIT))]
54:    proc  2 (TR_Resp) line 730 "pan_in" (state 199) [Aflag = 0]
55:    proc  0 (monitor) line 1287 "pan_in" (state 3)  [!(Aflag)]
spin: line 1287 "pan_in", Error: assertion violated
spin: text of failed assertion: assert((A_Flag))
```

Figure 26: Guided simulation result of the erroneous restart sequence

The cause of this erroneous restart is a TID verification procedure (from step 23 to 38) initiated by a re-transmitted (but delayed) INVOKE PDU. It can be predicted that if TID verification is allowed with no further condition in the R_LISTEN state, the local primitive sequence will repeat in the same pattern as shown in the error trail, which finally makes any arbitrarily large channel overflow.

Table 23: Modification of TR-Protocol to correct Error 4.2

Guard		**Action**		original	transition
rstate	$\wedge$*local_ctrl* $\wedge$ *channel*	*channels* $\wedge$ *local_ctrl*$\wedge$	*rstate*	entry #	name
R_LIS	{INVOKE,RID,UPack} $\in$ Init2Resp First_Invoke = 1 $rdata.Timer = 0$	Init2Resp $-$ {INVOKE,RID,UPack} iind_tgl + 1 SetUack(rdata,invoke.UPack) ...	R_IRW	Fig. 40-1&2	iind_validTID
	{INVOKE,RID,UPack} $\in$ Init2Resp $rdata.Timer = 0$	Init2Resp $-$ {INVOKE,RID,UPack} Resp2Init $\cup$ {ACK,0,1,0} ...	R_TW	Fig. 40-4	Invk_TIDve1
	else	no change		-	-
R_TW	...	...	R_LIS	-	Timer2W_R
R_IRW		*StartTimer(rdata)*		Fig. 42-9&10	aindp_T2A
R_RRW				Fig. 44-4 Fig. 44-7	rcnf aindp_T2R
				- - -	areq aind aindp_TSP

Suggested solution: To avoid this error, TID verification (on the same TID) needs to be disabled for some period of time T after TR-Resp-PE returns to the R_LISTEN state. This time interval should be no shorter than two maximum packet lifetimes [18], so that all the PDUs (including the delayed ones) are cleared from the network when T expires. This requires a new action, starting the local timer for a time interval T, be added immediately before the TR-Resp-PE updates its local state to R_LISTEN in every related transition. In addition, a new condition "TimerTo_T" (which says the time interval T expires on the timer and is modelled as $rdata.Timer = 0$)

needs to guard the transitions of TID verification and delivering the *iind* primitive when TR-Resp-PE is in the R_LISTEN state (see Table 23).

Error 4.3 Erroneous end-to-end behaviours

Description: There are violations of temporal claims p_{T_2}, p_{T_4} and p_{T_5} defined in Table 19.

Figure 27 shows two error trails where the *Invoke.cnf* primitive is delivered to the Initiator (at steps 78 to 83 in the left chart and steps 69 to 72 on the right) *without* *Invoke.res* having been submitted on the Responder side. This is a violation of $p_{T_2} = \mathcal{G}(\neg\, ICNF\ \mathcal{W}\ ires)$.

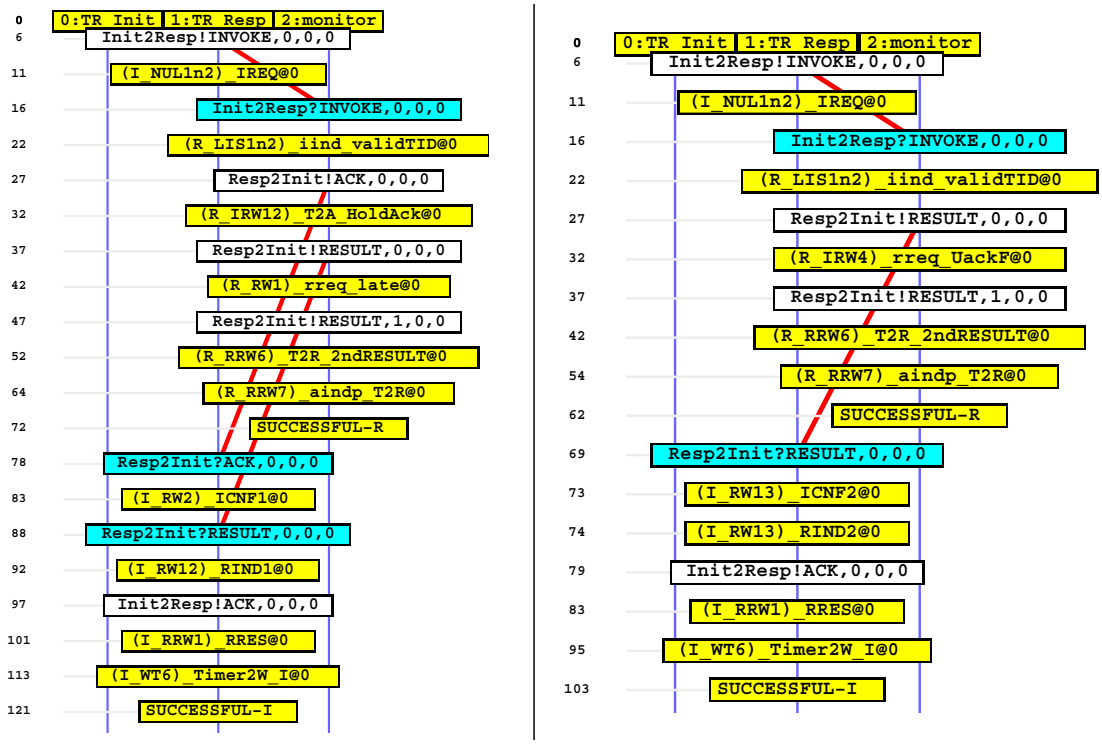


Figure 27: Two erroneous end-to-end trails of TR-Protocol ($ICNF$ without $ires$)

Figure 28 shows the error trail where the *Result.cnf* primitive is delivered to the Responder (at steps 77 to 86) *without* *Result.res* having been submitted on the Initiator side. This is a violation of $p_{T_4} = \mathcal{G}(\neg\, rcnf\ \mathcal{W}\ RRES)$.

Figure 29 shows the error trail where the *Abort.ind* primitive is delivered to the Responder (at step 82 to 91) *without* *Aind.ind* having been generated by TR-Init-PE, which means the Initiator is not notified when the transaction has been aborted. This is a violation of $p_{T_5} = \mathcal{G}(\neg\, aind\ \mathcal{W}\ AINDP)$.

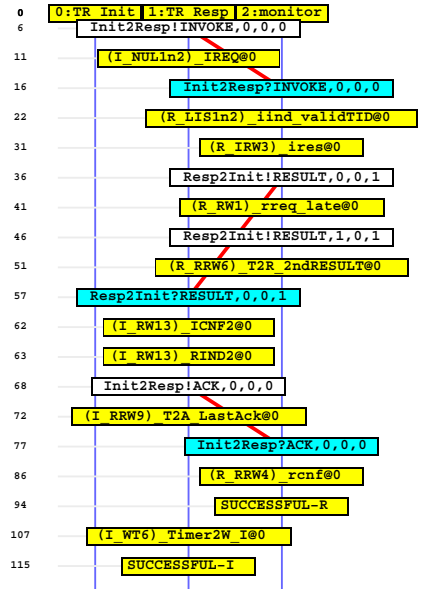


Figure 28: A 3rd erroneous end-to-end trail of TR-Protocol (*rcnf* without *RRES*)

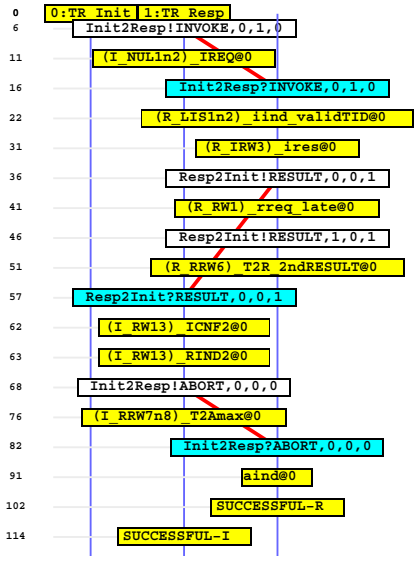


Figure 29: A 4th erroneous end-to-end trail of TR-Protocol (*aind* without *AINDP*)

Analysis: In the left chart of Figure 27, the delivery of *Invoke.cnf* is triggered by an ACK PDU sent from TR-Resp-PE in the `(R_IRW12)_T2A_HoldAck` transition (steps 27 to 32). This ACK is an acknowledgement from TR-PE, rather than one submitted by the TR-*User* within the acknowledgement interval A. But the Initiator cannot tell the difference on receipt of the ACK. Similarly, in the right chart, *Invoke.cnf* is generated when receiving a RESULT PDU sent in the `(R_IRW4)_rreq_UackF` transition (steps 27 to 32). This RESULT is an implicit acknowledgement to the invoke, instead of an explicit one of the *Invoke.res* primitive, when the UserAck feature is turned off. Therefore, we learn that the TR-Protocol design lacks the mechanism of enabling TR-PE to discriminate between incoming PDUs with different semantics (i.e., an explicit or implicit acknowledgement), thus leading TR-PE to generate an improper response.

Figure 28 is just another example that shows the Responder cannot tell whether an ACK PDU is the acknowledgement from TR-PE (after time-out), or the one from TR-User (when the *Result.res* primitive has been submitted).

Figure 29 guided us to find a minor error in entry 8 of the state table *I_RESULT_RESP_WAIT* (Figure 38), where an action of "generate Abort.ind" is missing when the event "TimerTo_A" occurs.

Suggested solution: For the detailed modification made on the TR-Protocol state tables, which aims to correct the above errors, we refer to [18]. The main idea is to activate a reserved bit (named CNF) in the header field of the RESULT and ACK PDUs, by which TR-PE can tell whether to generate a *confirm* primitive or not, as the CNF bit indicates if a *response* primitive has been submitted as an explicit acknowledgement. Associated changes are incorporated in our PROMELA model (see Appendix C.2).

Error 4.4 Erroneous primitive ordering with UserAck turned on

This error is a violation of $p_{T_7} = \mathcal{G}(\neg\, rreq\ \mathcal{W}\ ires)$ when the TR-Protocol configuration is changed to $ACK\_TYPE = 1$. Figure 30 shows the error trail. At step 32, the *Result.req* primitive is generated, but no *Invoke.res* primitive has been submitted to acknowledge the invoke. This is not allowed when the "User Acknowledgement" feature is turned on.

With the transition name `(R_IRW4)_rreq_UackF` in the action box of step 32, we can easily locate the cause of the error: a lack of condition $Uack = False$ in entry 4 of the *R_INVOKE_RESP_WAIT* state table (Figure 42). Table 24 shows the modification of the problematic transition.

4.2.3 Verified Properties for the Modified TR-Protocol Model

After we have modified our TR-Protocol model according to the suggested corrections, we verify, in a *full* state space search mode, that all the properties specified in Section 3.5 are satisfied in a model with the configuration given in Table 25, i.e., *Config F-4-4-1* and *T-4-4-1*.

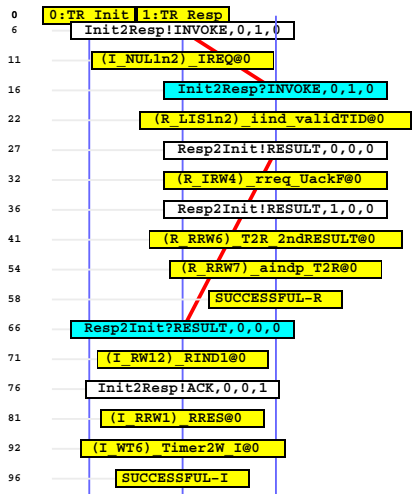


Figure 30: An erroneous primitive ordering with *UserAck* turned on

Table 24: Modification of TR-Protocol to correct Error 4.4

Guard		**Action**		original	transition
rstate	$\wedge local\_ctrl \wedge channel$	$channels \wedge local\_ctrl \wedge$	*rstate*	entry #	name
R_IRW	$rdata.Uack = 0$	Resp2Init $\cup$ {RESULT,0,0,0} ResetRCR(rdata) StartTimer(rdata)	R_RRW	Fig. 42-4	rreq_UackF

This means that our PROMELA model of TR-Protocol can preserve the safety and liveness properties, no matter whether the UserAck feature is available or not. Even though messages may get lost or reordered in the lower layer service (which includes a GSM SMS bearer), the TR-Protocol model will not generate unexpected service languages. These properties give high confidence that the design of the *revised* TR-Protocol is well-formed.

Relationship between channel size and RCR maximum From the safety property $p_{S_1} = (len(Resp2Init) \leq R2ISIZE) \wedge (len(Init2Resp) \leq I2RSIZE)$, we learn that the minimum sizes of the two channels satisfy the following relationship with the maximum values of the RCR counters:

$$I2RSIZE = \begin{cases} RCR\_MAX\_I + RCR\_MAX\_R + 1, & \text{for } RCR\_MAX\_R > 0 \\ RCR\_MAX\_I + 2, & \text{for } RCR\_MAX\_R = 0 \end{cases}$$

$$R2ISIZE = RCR\_MAX\_I + RCR\_MAX\_R + 3 \tag{18}$$

This empirical relationship has been reported in [18]. Using our PROMELA model, the underlying mechanism of the relationship can be easily revealed by modi-

Table 25: Configuration vector of the TR-Protocol for GSM SMS network

#	Parameters	Type	Values
1	ACK_TYPE	bit	0/ 1
2	RCR_MAX_I	natural	4
3	RCR_MAX_R	natural	4
4	LOSSY	bit	1

fying the invariant p_{S_1} within the monitor process. For example, if p_{S_1} is changed to $p_{I2R} = (len(Resp2Init) \leq R2ISIZE) \wedge (len(Init2Resp) < I2RSIZE)$, a counterexample showing the violation of p_{I2R} will tell which PDUs fill the maximum size of the channel buffer $Init2Resp$.

As illustrated in Figure 31, the maximum number of PDUs consists of at most RCR_MAX_I many INVOKE PDUs resulting from retransmission (e.g., at steps 57 and 65 in the default configuration), $(RCR\_MAX\_R + 1)$ many ACK PDUs (steps 81 and 89) in response to the RESULT (which is resent RCR_MAX_R times).

Similarly, in Figure 32, the maximum number of PDUs in the $Resp2Init$ channel consists of at most $(RCR\_MAX\_I + 1)$ many ACK PDUs (at step 51, 60 and 67) in response to the INVOKE, $(RCR\_MAX\_R{+}1)$ many RESULT PDUs and a final ABORT PDU.

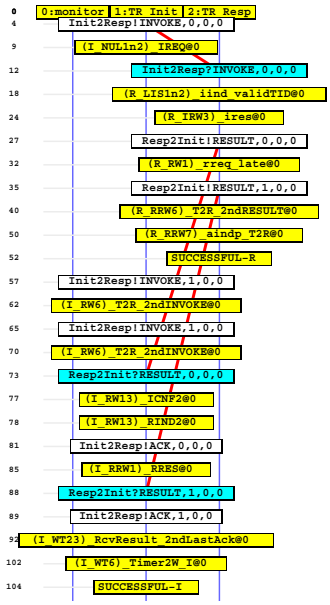


Figure 31: A primitive sequence with full $Init2Resp$ channel ($RCR\_MAX\_I = 2$, $RCR\_MAX\_R = 1$)

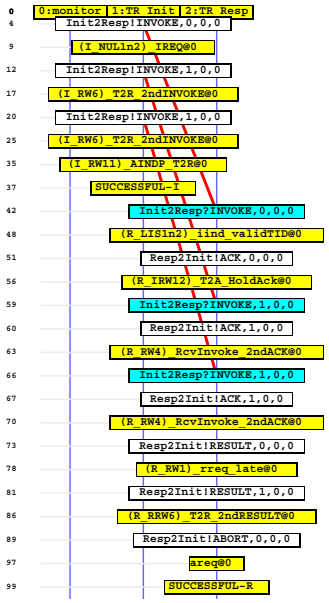


Figure 32: A primitive sequence with full *Resp2Init* channel ($RCR\_MAX\_I = 2$, $RCR\_MAX\_R = 1$)

Unreachable transitions As to the liveness property, we list all the *expected* unreachable transitions in Table 26. They are due to the restriction on protocol parameters. Taking row 1 as an example, when the UserAck feature is Off, TR-Init-PE does not need to wait another interval A for an acknowledgement sent from TR-User, so entries 7 and 8 Figure 38 are not enabled and consequently not reachable during the verification. There are no other *unexpected* unreachable transitions reported.

4.3 Analysis of Verification Results

4.3.1 State Space Statistics

The purpose of this section is to show: (1) the state space statistics collected for the verification of our revised TR-Protocol model with *Config F-4-4-1* (see Table 25); (2) verification results beyond the configuration *F-4-4-1*, towards *Config F-8-8-1*.

Table 27 shows the statistics for verification of the safety properties, absence of a non-progress cycle and the 11 temporal claims, respectively. Take the first row as an example, SPIN explores all the reachable states of the protocol model for those safety properties (which are included in one monitor process). By using compression encoding, each state is stored in fewer bits than the state-vector size. Here, only 38.54 Megabytes [22] of RAM are needed to store 634,556 distinguished states, which is about 40% of the

[22]We denote 10^6 bytes as 1 Megabytes, while 2^{20} bytes as 1 MB (the common unit of RAM).

Table 26: Expected unreachable transitions of TR-Protocol

#	Condition	Entry Number in State Tables	Transition Name
1	$ACK\_TYPE = 0$	Fig. 38-7&8	`AINDP_T2A`
2		Fig. 42-9&10	`aindp_T2A`
3	$ACK\_TYPE = 1$	Fig. 38-9	`TimerTO_A_LastAck`
4		Fig. 42-4	`rreq_UackF`
5		Fig. 42-12	`T2A_HoldAck`
6	$RCR\_MAX\_R = 0$	Fig. 44-6	`T2R_2ndRESULT`
7		Fig. 39-2	`RcvResult_2ndLastAck`
8	$RCR\_MAX\_I = 0$	Fig. 37-9	`T2R_2ndINVOKE`
9		Fig. 37-5	`RcvAck_TIDOK`
10		Fig. 41-1	`iind_TIDok`
11		Fig. 41-5	`RcvInvoke_TIDve2`
12		Fig. 44-4	`RcvInvoke_2ndACK`
13	$RCR\_MAX\_R \leq 1$	Fig. 37-10	`T2R_2ndTIDok`

memory size required without compression.

The verifications of the liveness property and LTL formulas require additional bytes in each state-vector, for they both need an extra process for searching for a non-progress cycle which violates the liveness property and for an acceptance cycle which falsifies the LTL formula. Different numbers of states stored for different LTL formulas (some of which may share the same pattern) reflect the fact that using on-the-fly model checking, the effort of constructing (while exploring) the state space of the product automaton $A_M \times A_{\neg\psi}$ has to be done anew for every verification run. Because the construction depends on whether the *current* state of A_M satisfies the specific LTL formula ψ, a different state space is computed.

Our parameterized TR-Protocol model has served not only as a platform for proving properties of specific configurations, but also for experimenting with the impact of the parameters (specifically, the maximum values of counters) in the TR-Protocol. It has been predicted [18] that the relationship between RCR_MAX_I and the state space size of the TR-Protocol (CPN) model can be expressed as a 7th order polynomial [23], and for RCR_MAX_R is quadratic. Since the memory requirement and the time consumed for *Config F-4-4-1* is quite acceptable with regard to the safety properties, we are optimistic that we can explore more complex configurations, e.g., *Config F-8-8* as suggested for the WTP used in an IP network [60].

Table 28 presents our trials on some configurations beyond *Config F-4-4-1*. For a protocol running on lossless channels, we are able to verify the TR-Protocol model (against the safety properties) with configurations up to *F-6-6-0*, using full state space search mode. When the channels are lossy, an exhaustive check for non-safe states is

[23] We notice that in our PROMELA model (with lossless channels), the number of states grows at a slower speed of $O(n^5)$, where $n = RCR\_MAX\_I$.

Table 27: Full state-space search of the revised TR-Protocol model (*Config F-4-4-1*)

Property	**State-vector** (bytes)	**No. of States stored**	**No. of transitions**	**Memory usage for states** (Megabytes)	**Compression ratio**	**User time** (seconds)
$safety$	140	634,556	3,488,120	38.54	39.96%	35.17
$progress$	144	672,982	3,788,110	42.24	40.23%	42.47
p_{T_1}	144	635,045	3,489,600	41.10	41.49%	41.56
p_{T_2}	144	925,319	6,459,000	50.11	34.72%	76.80
p_{T_3}	144	666,580	3,674,980	42.43	40.81%	43.82
p_{T_4}	144	1,097,150	7,979,090	54.93	32.09%	94.85
p_{T_5}	148	1,338,800	10,168,900	64.15	29.95%	123.54
p_{T_6}	148	1,298,900	7,998,480	64.55	31.06%	97.54
p_{T_7}	144	256,143	1,368,630	16.21	40.57%	16.10
p_{T_8}	144	318,156	1,785,410	18.36	36.99%	20.71
p_{T_9}	148	774,685	3,936,490	49.40	39.85%	47.21
$p_{T_{10}}$	148	799,687	4,045,740	50.73	39.65%	48.56
$p_{T_{11}}$	148	860,501	4,499,410	52.58	38.19%	54.14

`Note:` p_{T_7} `and` p_{T_8} are verified with *Config T-4-4-1* (see page 61).

applicable only for *Config F-5-5-1* with 256MB RAM available.

Table 28: Verification of safety for the revised TR-Protocol model (beyond *F-4-4-1*)

Config	**State-vector** (bytes)	**No. of States stored**	**No. of transitions**	**Memory usage for states** (Megabytes)	**Compression ratio/ Hash factor**	**User time** (seconds)
F-5-5-1	156	1,937,380	10,693,600	120.90	37.15%/-	114.44
F-5-5-1	156	1,937,350	10,693,400	134.22	-/277.12	114.38
F-6-6-1	172	5,280,140	29,187,000	134.22	-/101.68	329.99
F-8-8-1	204	30,755,600	170,360,000	134.22	-/17.46	2208.37
F-5-5-0	152	1,058,540	2,893,880	59.65	34.36%/-	28.46
F-6-6-0	168	2,997,940	8,116,770	173.35	32.12%/-	90.16
F-8-8-0	200	18,458,600	49,569,200	134.22	-/29.08	627.84

To search the state space of a more complex configuration, we resort to bit-state hashing, the approximation of full state space search. We limit the size of the hash table to 134.22 Megabytes, (i.e., 128MB = 2^{30} bits), which can store 2^{29} (approx. 5.4×10^8) states using 2-bit hashing (for reachability analysis). The hash table size is 277 (i.e., the value of the hash factor) times the size of the known state space of *Config F-5-5-1*, so this approximation method performs quite well, only missing 0.0015% of states.

For *Config F-6-6-1*, we manage to obtain a hash factor greater than 100. The verification has been repeated 16 times using 16 different hash functions, which aims to

increase the coverage of a (super-trace) search using sequential bit-state hashing [24]. All runs return the same number of states stored (explored), which means the 99.9% coverage (of the actual state space) reported by SPIN is not merely a guess [24]. Unfortunately, for *Config F-8-8* whose state space is estimated at the order of 10^7, we have only partial confidence [25] to say the safety properties *are* satisfiable in the TR-Protocol model, for the hash factor is not big enough.

Figure 33 shows the impact of protocol parameters on the state space size of the TR-Protocol model. The size of a model with lossy channels (*Config F-I-R-1*) is about twice that of the model with lossless channels (*Config F-I-R-0*). With the reference of two linear lines (based on the data of *Config 1-1, 2-2* and *3-3*), we can see the state space of either model grows, as both RCR_MAX increase, at a less-than exponential rate. Actually, both trend lines can be expressed by a 5th order polynomial, i.e., the state space growth is mainly affected by the parameter RCR_MAX_I.

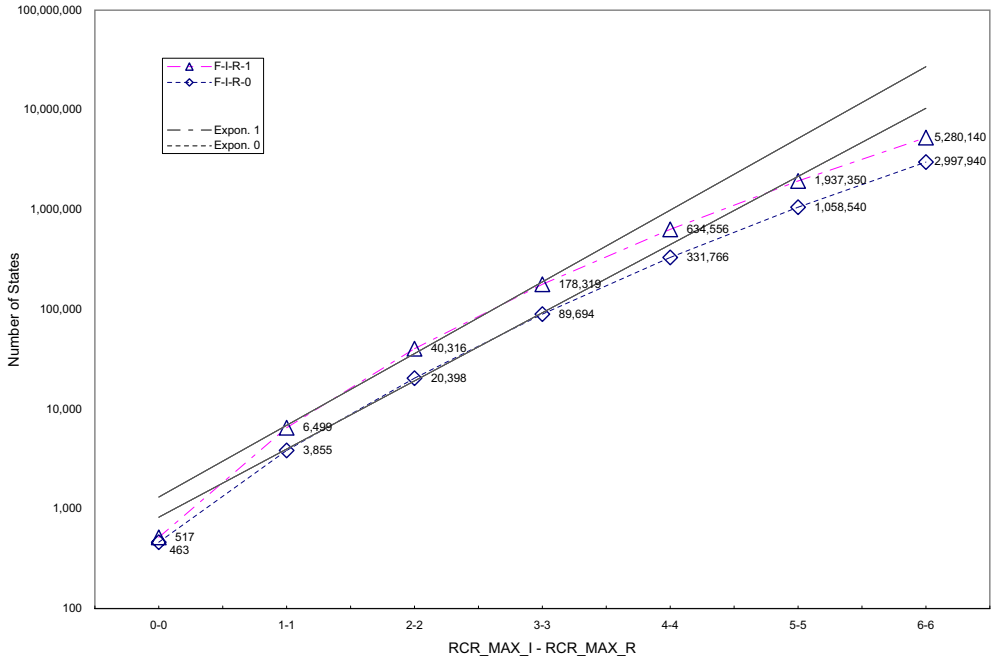


Figure 33: State space growth in the TR-Protocol PROMELA model (logarithmic scale)

4.3.2 Comparison with the Coloured Petri Nets method

Gordon [18] has applied Coloured Petri Nets (CPN) in modelling and analyzing an old version of the WTP design [59]. In this section, we make a comparison between his

[24] Actually, we have tried to double the hash table size (which increases the hash factor to 203), and found the states covered only increase by 0.05%.

[25] The coverage reported by SPIN is 98%.

method and ours.

Efficiency in memory and time consumption Gordon conducted the analysis in two steps: (1) model the TR-Service and TR-Protocol design in two separate CPNs and verify the safety and liveness requirements (mainly, absence of deadlocks, absence of livelocks and dead transitions) through state space analysis, using Meta Software's Design/CPN tool; (2) convert the state spaces of these two CPNs into two finite state automata (FSAs) (through an intermediate representation in Standard ML code) and compare their service languages (i.e., sets of primitive sequences) through FSA language analysis, using AT&T's FSM tool [1].

Both steps require a full state space of CPN generated before any correctness requirement can be checked against the model. This limits the applicability of the CPN method in verifying TR-Protocol models with complex configurations. Although CPN verification has been run on a computer equipped with 512MB RAM (two times ours) and a 366MHz CPU, the maximum configuration handled is *F-4-3-0* using *ordinary* state space analysis. To deal with the same configuration, our PROMELA model needs less than 16MB memory to complete the verification of safety and liveness properties. The state space of the PROMELA model has a similar size with that of the CPN model, so we conclude that the state encoding of SPIN is more efficient than that of Design/CPN.

To combat the state explosion problem, the sweep-line method (implemented as a library in Design/CPN) has been exploited in the CPN analysis. Storage space is saved by deleting some states that have been visited but are guaranteed not to be reached again [18]. An alteration of on-the-fly verification as it is, the method differs from SPIN's NDFS algorithm in that (1) it traverses the full state space in a *breadth* first manner; (2) a path leading to an erroneous state may not have been stored, which helps little for debugging purposes; (3) models that can be checked are of a limited kind and must exhibit a *progress* property, in other words, a user must create a *progress measure* function for each state that gives it a value no less than its predecessor state; (4) absence of livelock cannot be verified.

Even using the sweep-line method, CPN verification can only reach *Config F-4-4-0* (i.e., for WTP used in a GSM SMS network, but on *lossless* channels), whose state space is much smaller than our *lossy*-channel model (see Table 28).

Since no memory usage is reported in [18], we list in Table 29 the time consumed in the verification of the safety property for the CPN (mainly no deadlock) and PROMELA models respectively, for some sample configurations. We can see SPIN again significantly outperforms Design/CPN. As the state space grows, the time consumed in verifying the CPN model increases from 100 times to 880 times that spent on our PROMELA model. In the CPN model, most of the time is spent in generating the whole state space while checking for deadlock; comparatively, the time spent on FSA language comparison is negligible if the FSA models are already available. To estimate SPIN's checking time of all properties (including the comparison of service languages), however, one has to take into account the time spent on every single property, since SPIN builds the state space

anew for every verification run. Even that, the total run time[26] of a PROMELA model is still much less than that of a CPN model for the same configuration.

Table 29: Comparison of time consumption between CPN and PROMELA verification (*Config F-I-R-0*, time in seconds.)

Config	CPN Time	Adjusted CPN Time	PROMELA Time
4-0	317	134.30	1.26
4-1	1799	762.18	2.83
4-2	6004	2543.72	5.03
4-3	16242	6881.27	7.79
4-4	12289	5206.50	11.35

Note: 1. The *adjusted CPN Time* is scaled down by a factor of 2.36, which equals the CPU clock ratio (864/366).
2. The *CPN Time* of *Config F-4-4-0* is obtained by using sweep-line method, others by ordinary state space analysis.

Moreover, SPIN's bit-hashing enables us to gain confidence, to a certain extent, in some basic safety requirements (e.g. proper termination and no assertion violations), when they are verified against a bigger (1 to 2 orders of magnitude) state space. But we cannot see support in Design/CPN to facilitate such approximated verification.

Usability in modelling, verification and implementation As modelling methods, both CPN and PROMELA are suitable for protocol formalization. They support an incremental approach applied in the protocol analysis, which means (1) different levels of abstraction (e.g., service and its refinement protocol) can be built from a similar model structure; (2) to cope with the inherent complexity of a protocol design, different features or modelling assumptions can be added into the model bit by bit. For example, we can first disable the "Transaction Abort" feature, then include it after we have investigated the basic "Message Transfer" function. Or we can disable the lossy property of channels and begin with a model of smaller state space.

However, a different amount of human effort is required in using these two kinds of models for verification. In CPN verification, nearly 60,000 different primitive sequences are reported as the difference in service language between the TR-Service model and the TR-Protocol model, by a *separate* analysis tool FSM. These "error" sequences are classified (with additional effort) to show that they are caused only by 4 errors in the design, since a single error may lead to a large amount of non-deterministic execution paths. But to locate the cause of error, one still has to return to Design/CPN and inspect parts of the state space of the CPN to find the error trail. This task is laborious, because

[26]For example, the total added from Table 27 is 741 seconds for *Config F-4-4-1*, which is estimated to be 3 times that of *Config F-4-4-0*, for comparing with the same configuration in Table 29.

the path shown in the CPN state space is quite far away from a convenient graphical representation (e.g., the standard Message Sequence Chart) of protocol operation.

On the contrary, within a *unified* analysis environment, SPIN's LTL verification enables us to specify various temporal requirements (e.g., on the primitive orderings) of users' interests, although it requires some skills to propose an appropriate amount of LTL formulas (in the appropriate syntax) to be checked. But in that case that a LTL formula is violated, we are only faced with one error trail and will not be bothered by many other trails taking different appearances but caused by the same defect. We can take advantage of the MSCs (automatically) generated by SPIN to visualize an execution run and locate the cause of errors.

There are also differences in the usability of CPN and PROMELA models for protocol implementation. The syntax of PROMELA is close to ANSI-C and the verifier of a PROMELA model (which has already been used to simulate the protocol execution as a "running" prototype) is directly translated into a set of C files. So from a PROMELA model which has been verified to be error free, a protocol implementation written in C (which is widely accepted in both the academic and industrial worlds) can be generated [37] and used with assurance.

It is said that the *graphical* CPN model in Design/CPN can be converted into Standard ML code, which can be executed for simulation purposes. But from that to a protocol implementation with good portability, more steps have to be taken in transforming the language. This process is vulnerable to errors due to differences in syntax and semantics, which (unfortunately) is inevitable in different languages.

Syntax and semantics CPN and FSA (the foundation of PROMELA) both have formal syntax and semantics. In a certain sense, one formalism can be translated to the other due to their similar semantics. For example, binding elements in CPN function like an alphabet in FSA; transitions in CPN are used in a similar way as transition relations in FSA. There are also methods to directly model (Coloured) Petri Nets using PROMELA syntax [15, 19], e.g., replacing places in Petri Nets with channels in PROMELA. But it turns out that for a real problem like WTP, direct translation from a CPN model (which already has a large number of nodes and transitions) could result in a PROMELA model having a huge state space even for a small protocol configuration, so that the verification is infeasible. So we still need to use a proper abstraction (e.g., with the FSA formalism) of the protocol design, with appropriate applications of PROMELA components (variables, channels and processes), to obtain a PROMELA model.

Still some differences exist in syntax which affect the system modelling. For instance, the concept of *colour set* in CPN enables users to define various user types. In Gordon's CPN model, 4 kinds of PDUs (INVOKE, ACK, RESULT, ABORT) are grouped in the type `PDU`, while 4 kinds of TR-Init-PE states (I_NUL, I_RW, I_RRW, I_WT) are classified as another type `IStateName` [18]. But, to enhance the model checking efficiency, PROMELA is restrictive on some language features, among which user type definition is supported through only one kind of `mtype` declaration. All the

above PDU types and TR-PE states can be declared symbolically but have to be declared as the same "`mtype`". This makes it difficult to type a variable which should have a specific range; e.g., the first field of a message should only take the symbolic names of PDUs, rather than the names of TR-PE states. This requires users to be careful in assigning values to corresponding variables.

It seems that each CPN place can be defined in Design/CPN without a bound on size, as if a place had infinite capacity (this may cause inefficiency in memory management, since in a real implementation it is not practical to allocate infinite space). A channel in PROMELA, however, must be declared with bounded capacity, because each component in PROMELA must take a finite range to result a finite state model. This requires a good estimation of the channel size, as a large channel size can significantly increase the state space of the model, while a too small one will always falsify the safety property on boundedness.

4.4 Summary

This section gives the procedures and results of verifying WTP 2.0 with the model checker SPIN. Five main defects in TR-protocol design, which can lead to deadlock, buffer overflow and unfaithful refinement of TR-Service, are illustrated, along with modification solutions. The modified TR-Protocol model meets all the correctness requirements we have proposed, which implies that its revised design is well-formed. Additionally, our optimized PROMELA model obtains a reasonable state space size, which enables us to investigate the WTP protocol with more complex configurations beyond the suggested configuration for a GSM SMS network.

The comparison of Coloured Petri Net analysis and model checking (on PROMELA models) is useful because the representations yield models with slightly different semantics. Not only have we seen the superior performance of SPIN in model checking, we are also more confident that our PROMELA models are correct, rather than lucky by-products of modelling decisions in a particular notation. A verified PROMELA model can then be used to directly generate a protocol implementation written in ANSI-C. Fig. 20

5 Conclusions

5.1 Results

The contributions of this report are:

- Building formalized models, with finite state automata (FSA), of Class 2 Service and Protocol design of the Wireless Transaction Protocol Version 2.0 [60]. The modelling procedure helps to uncover unstated assumptions in the TR-Protocol design and we correct its under-specification as follows: (1) in each *Condition Table*

(see e.g., Table 16, which formulates the informal state tables of the TR-Protocol), an "*else*" entry is added for the completeness of guard conditions. It defines that the current state of the FSA remain unchanged if variables take values other than those explicitly defined in the table; (2) Update rules of the RID field in PDUs are clarified when defining the transition relations of FSA models (see Section 3.4.2).

- Establishing a relationship between the FSA formalism and the modelling language PROMELA. Instead of building PROMELA models directly from a protocol design, we derive the TR-Service PROMELA model and the TR-Protocol PROMELA model from their FSA representations. This is to illustrate how PROMELA can be applied to formalize an artefact with methodology, as well as in regular formats, rather than be used like a programming language (though its syntax is close to ANSI-C) in an ad-hoc way.

- Uncovering errors in the TR-Protocol design when checking the model against correctness requirements on safety, liveness and temporal behaviours. These defects, which are difficult to reveal by human inspection, include: (1) deadlock during TID verification; (2) erroneous re-start of the primitive sequence, which may lead to overflow of channel buffers; (3) erroneous end-to-end behaviours caused by ambiguity in INVOKE and RESULT acknowledgements; (4) erroneous primitive ordering with UserAck turned on, which is caused by under-specification in guard conditions. Solutions are proposed to correct these errors.

- Verifying correctness of a revised TR-Protocol model with the configuration used in GSM SMS networks, providing a result of practical significance. The resulting model checking performance is encouraging, so that we extend Gordon's verification result [18] to a more general protocol environment which includes lossy channels. We also gain confidence in the model with more complex configurations to be used in IP networks.

5.2 Future Work

The work presented in this report can be extended in several ways.

- Some assumptions made in Section 3.4.1 can be re-considered.

    1. With an incremental approach, more protocol features can be modelled, e.g., segmentation and re-assembly (SRA) of messages. When SRA is modelled, we may also investigate the sliding window procedure incorporated in the protocol model. Verification of new properties will increase our confidence level in the more complex behaviour of WTP;
    2. Verification results are desired to be independent of protocol parameters. To obtain a manageable state space, new abstraction methods are needed in

system modelling (this may yield a model which goes further from implementation). Or we may apply theorem proving to provide an inductive proof;

3. In some cases, correct timing is important to protocol performance. To investigate timing requirements, we may need new extensions of SPIN, as its main version does not support real-time verification. Or we may have to use other suitable tools. But the state space size is still a concern, as we can imagine that even a WTP model with a small configuration can have a considerably large state space, if it is modelled with timing behaviours.

- Proper formalization methodology can facilitate automatic generation of a model to be verified with the model checker SPIN. So it is desirable that tools are to be developed to mechanically translate a tabular representation of a protocol design (or even C implementation of the protocol) into SPIN's input language PROMELA.

- Given a verified PROMELA model, a protocol implementation in ANSI-C is expected to be automatically produced from its verifier. Thus a complete process throughout protocol development (from formalization to verification to implementation) can be established.

Acknowledgments

I am grateful to Drs. Ryszard Janicki, Michael Soltys, Doug Down and Mark Lawford for their careful review of this report and for sharing with me their uncommon wealth of insights. I shall thank Drs. Steven Gordon and Gerard Holzmann for their helping me better understand the WTP operation and the SPIN tool, respectively.

References

[1] AT&T. *FSM Library*. Available via: http://www.research.att.com/sw/tools/fsm

[2] K. A. Bartlett, R. A. Scantlebury and P. T. Wilkinson. *A note on reliable full-duplex transmission over half-duplex lines*. Communications of the ACM, Volume 12, No. 5, pp. 260–265. ACM Press, 1969.

[3] G. Behrmann, A. Fehnker, et al. *Efficient Guiding Towards Cost-Optimality in UPPAAL*. LNCS 2031: Proceeding of TACAS'01. Springer-Verlag, 2001.

[4] R. E. Bryant. *Graph-based algorithms for boolean function manipulation.* IEEE Transactions on Computers, C-35(8), 1986.

[5] J. R. Burch, E. M. Clarke, et al.. *Symbolic model checking:* 10^{20} *states and beyond.* Information and Computation, 98(2):142–170, June 1992.

[6] Y. Choueka. *Theories of automata on ω-tapes: A simplified approach.* In Journal of Computer and System Sciences, 8:117–141, 1974.

[7] E. M. Clarke, K. McMillan, et al.. *Symbolic model checking.* In LNCS 1102: Proceedings of CAV'96. Springer-Verlag, 1996.

[8] E. M. Clarke and J. M. Wing. *Formal methods: State of the art and future directions.* ACM Computing Surveys, 28(4):626–643, December 1996.

[9] C. Courcoubetis, M. Vardi, P. Wolper and M. Yannakakis. *Memory-efficient algorithms for the verification of temporal properties.* In Formal methods in system design, Volume 1, pp. 275–288, 1992.

[10] P. R. D'Argenio, J-P. Katoen, T. C. Ruys and J.Tretmans. *The Bounded Retransmission Protocol must be on time!*, In LNCS 1217: Proceedings of TACAS'97, pp. 416–431. Springer-Verlag, 1997.

[11] E. W. Dijkstra. *Guarded commands, non-determinacy and formal derivation of programs.* Communication of the ACM Volume 18, No. 8, pp. 453–457. ACM Press, 1975.

[12] Y. Dong, X. Du, et al.. *Fighting livelock in the i-protocol: a comparative study of verification tools.* In Proceedings of TACAS'99, Amsterdam, The Netherlands, March 1999.

[13] M. B. Dwyer, G. S. Avrunin and J. C. Corbett. *Patterns in Property Specifications for Finite-State Verification.* In Proceedings of the 1999 International Conference on Software Engineering (ICSE99), pp. 411–420. ACM Press, May 1999.

[14] E. A. Emerson. *Temporal and modal logic.* In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, volume B, pp. 996–1072. Elsevier Science Publishers, 1990.

[15] G. Gannod and S. Gupta. *An Automated Tool for Analyzing Petri Nets using Spin.* In Proceedings of the 16th Automated Software Engineering Conference. IEEE, 2001.

[16] R. Gerth, D. Peled, M. Vardi and P. Wolper. *Simple on-the-fly automatic verification of linear temporal logic.* In Proceedings of PSTV'95. Chapman & Hall, 1995.

[17] P. Godefroid and G. J. Holzmann. *On the verification of temporal properties.* In Proceedings of PSTV'93. North-Holland, 1993.

[18] S. Gordon. *Verification of the WAP Transaction Layer using Coloured Petri nets.* Ph.D. thesis, University of South Australia, 2001.

[19] B. Grahlmann and C. Pohl. *Profiting from Spin in PEP*. In Proceedings of the SPIN'98 Workshop, 1998.

[20] J. V. Guttag, J. J. Horning and J. M. Wing. *Some Remarks on Putting Formal Specifications to Productive Use*. Science of Computer Programming, Vol. 2, No.1, pp. 53–68, North-Holland, 1982.

[21] C. A. R. Hoare. *Communicating Sequential Processes*. Communications of the ACM, Vol. 21, No. 8, pp. 666–677, August 1978.

[22] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[23] G. J. Holzmann and D. Peled. *An improvement in formal verification*. In Proceedings of the 7th International Conference on Formal Description Techniques, pp. 177–194, 1994.

[24] G. J. Holzmann. *An Analysis of Bit-State Hashing.* In Proceedings of PSTV'95, pp. 301–314. Chapman & Hall, 1995.

[25] G. J. Holzmann, D. Peled and M. Yannakakis. *On nested depth-first search*. In Proceedings of the 2nd SPIN Workshop. American Mathematical Society, 1996.

[26] G. J. Holzmann. *The model checker SPIN*. IEEE Transactions on Software Engineering, Vol 23, No. 5, pp. 279–295, May 1997.

[27] G. J. Holzmann. *State Compression in SPIN: Recursive Indexing and Compression Training Runs.* In R. Langerak, editor, Proceedings of the 3rd SPIN Workshop. Twente University, The Netherlands, 1997.

[28] G. J. Holzmann. *The Engineering of a Model Checker: the Gnu i-Protocol Case Study Revisited.* LNCS 1680: In Proceedings of SPIN'99 Workshop, pp. 232–244. Springer-Verlag, 1999.

[29] J. E. Hopcroft and J. D. Ullman. *Formal languages and their relation to automata.* Addison-Wesley, 1969.

[30] ITU-T. *Information Technology–Open Systems Interconnection–Basic reference model: Conventions for the definition of OSI services.* ITU-T Recommendation X.210, Nov. 1993.

[31] ITU-T. *Information Technology–Open Systems Interconnection–Basic reference model: The basic model.* ITU-T Recommendation X.200, July 1994.

[32] ITU-T. *Message Sequence Chart (MSC)*. ITU-T Recommendation Z.120, 1997.

[33] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997.

[34] A. Josang. *Security Protocol Verification using SPIN.* In Proceedings of SPIN'95 Workshop. Available via: http://netlib.bell-labs.com/ netlib/spin/ws95/papers.html.

[35] K. Larsen, P. Pettersson and W. Yi. *UPPAAL in a Nutshell.* In Journal for Software Tools for Technology Transfer, vol 1, 1997.

[36] M. T. Liu. *Protocol engineering.* Advances in Computers, Vol. 27, pp. 79–195. Academic, 1989.

[37] S. Löffler and A. Serhrouchni. *Creating Implementations from Promela Models.* In Proceedings of the 2nd SPIN workshop, 1996.

[38] R. Milner. *A Calculus of Communication Systems.* Lecture Notes in Computer Science, Vol. 92. Springer-Verlag, 1977.

[39] D. Obradovic. *Formal Analysis of Routing Protocols.* Ph.D. thesis, University of Pennsylvania, 2002.

[40] D. Peled. *Combining Partial Order Reductions with On-The-Fly Model-Checking.* LNCS 818: Proceedings of CAV'94, pp. 377–390. Springer-Verlag, 1994.

[41] A. Pnueli. *The Temporal Logic of Programs.* In Proceedings of the 18th IEEE Symposium on Foundations of Computer Science, pp. 46–57, 1977.

[42] J. M. T. Romijn. *Analyzing Industrial Protocols with Formal Methods.* PhD thesis, University of Twente, September 1999.

[43] W. W. Royce. *Managing the development of Large Software Systems: Concepts and Techniques.* In Proceedings of WESCON, August 1970.

[44] T. C. Ruys. *Towards Effective Model Checking.* PhD thesis, University of Twente, 2001.

[45] T. C. Ruys. *SPIN Beginners' Tutorial.* Slides presented at the SPIN 2002 Workshop. Available via: http://wwwhome.cs.utwente.nl/ ruys/.

[46] K. Saleh. *Synthesis of communications protocols: An annotated bibliography.* Computer Communication Review, 26(5):40–59, October 1996.

[47] R. Seinal. *GNU m4, version 1.4.* Free Software Foundation, Inc., November 1994. Available via: http://www.gnu.org.

[48] M. Sipser. *Introduction to the Theory of Computation.* PWS publishing company, 1997.

[49] Spin Online References. *SPINE homepage.* Available via: http://spinroot.com/spin/whatispin.html.

[50] Spin Online References. *SPIN Version 3.3: Language Reference*. Available via: http://spinroot.com/spin/Man/index.html.

[51] Spin Online References. *SPIN and PAN verification options*. Available via: http://spinroot.com/spin/Man/index.html.

[52] K. Stahl, K. Baukus, Y. Lakhnech and M. Steffen. *Divide, abstract, and model-check.* In LNCS 1680: Proceedings of the 5th International SPIN Workshop on Theoretical Aspects of Model Checking. Springer-Verlag, 1999.

[53] W. Thomas. *Automata on infinite objects.* In J. can Leeuwn, editor, Handbook of Theoretical Computer Science, Volume B, pp. 133–191. Elsevier Science Publishers, 1990.

[54] M. Y. Vardi and P. Wolper. *An Automata-Theoretic Approach to Automatic Program Verification.* In Proceedings of the First IEEE Symposium on Logic in Computer Science, pp. 322–331, 1986.

[55] M. Y. Vardi and P. Wolper. *Reasoning About Infinite Computations*. Information and Computation, vol. 115, pp. 1–37, 1994.

[56] WAP Forum. *Wireless Application Protocol*. Web site: http://www.wapforum.org/

[57] WAP Forum. *WAP Architecture Specification. Version 2.0.* Available via: http://www.wapforum.org/, 12 July 2001.

[58] WAP Forum. *WAP Wireless Datagram Protocol Specification, Version 2.0.* Available via: http://www.wapforum.org/, 14 Jun 2001.

[59] WAP Forum. *WAP Wireless Transaction Protocol Specification, June 2000 Conformance Release*. Available via: http://www.wapforum.org/, 19 Feb. 2000.

[60] WAP Forum. *WAP Wireless Transaction Protocol Specification, Version 2.0.* Available via: http://www.wapforum.org/, 10 July 2001.

Notes:

CAV:	International Conference on Computer Aided Verification
LNCS:	Lecture Notes in Computer Sciences
PSTV:	IFIP/WG6.1 Symposium on Protocols Specification, Testing, and Verification
TACAS:	International Workshop on Tools and Algorithms for the Construction and Analysis of Systems

A SPIN's Verification and Simulation Algorithms

A.1 Nested Depth-First-Search Algorithm

Adopted from [25], Figure 34 shows one version of the nested depth-first-search algorithm, which is compatible with the partial order reduction method given in [40]. It begins (at line 3) with a basic depth-first-search from the initial state s_0 of the synchronous product automaton $A_M \times A_{\neg\psi}$. *Stack* is used to store a path from s_0 to an accepting state s. *StateSpace* is implemented in random access memory to store states for comparison; it could be a hashtable.

```
1       Initialize: Stack is empty; StateSpace is empty;
2       proc Search_for_Accepting_Cycle()
3       {     dfs(s0);
4       }
5
6       proc dfs(s)       /* reachability search*/
7       {     add {s, 0} to StateSpace;
8             push s onto Stack;
9             for each (selected) successor s' of s
10            {   if ({s', 0} NOT in StateSpace)
11                { dfs(s');
12                }
13            }
14            if (accepting(s)== TRUE)
15            {   ndfs(s);
16            }
17            pop s out of Stack;
18      }
19
20      proc ndfs(s)      /* the nested search */
21      {     add {s, 1} to StateSpace;
22            for each (selected) successor t of s
23            {   if ({t, 1} NOT in StateSpace)
24                { ndfs(t);
25                }
26                else if (t in Stack)
27                { report cycle; return;
28                }
29            }
30      }
```

Figure 34: Nested Depth-First-Search algorithm in SPIN

As can be seen from line 9, only the successors (of the current state) selected by the partial reduction method are stored in *StateSpace* to extend the search. Upon

reaching an accepting state (in line 14), a nested search is launched to find an accepting cycle. Each state is a pair, $\{s, 1\}$ or $\{s, 0\}$, suffixed with a tag to indicate whether the state is in the set of final states or not. If an accepting state $\{s, 1\}$ is reachable from both itself and the initial state (as stated in line 26), then an accepting cycle is found. Currently *Stack* stores a trace which leads to a state where the model M falsifies the property ψ.

If an accepting state $\{s, 1\}$ is reachable from s_0 but not reachable from itself, the nested search terminates and the accepting state is abandoned (as in 17). In the "worst" case, all the accepting states reachable from s_0 are analyzed and *StateSpace* holds all the states reachable along every computation path, as long as the memory constraint permits. On the other hand, if the whole search procedure terminates before exhausting the memory, it is guaranteed that the entire state space has been checked to have no instance of violating the property.

A.2 Random Simulation Algorithm

As discussed in Section 2.2.2, a PROMELA model can be viewed as an *interleaving product* automaton A_M of several process templates (each declared as `proctype`). Figure 35 shows an abstract simulation algorithm which runs the automaton in a stepwise manner:

```
1       while ((NO error) ∧ (executable(s) ≠ ∅))
2       {       E = executable(s);
3               a = random_choose(E);
4               (s', error) = execute(s, a);
5               s = s'
6       }
```

Figure 35: Random simulation algorithm in SPIN

In each current global state s, the function `executable()` returns a set E, which contains all the executable statements of every process in the model. Then the function `random_choose()` selects from E an executable option; this selection may either be made by the user or be automated by SPIN. The function `execute()` returns two results of executing the selected statement a: one is the updating of the current state; the other is a report of errors, if any, which may include invalid final states or violation of assertions.

In SPIN, this algorithm for random simulation actually defines the behaviour of PROMELA's *semantics engine* [50], which executes the model by selecting and executing one statement at a time. The semantic engine drives the model to execute until either an error occurs or no more executable statements exist in the current state (i.e., the model comes to its final state). If a non-terminating repetition structure is required in a

model, then the user is advised to label a proper end-state to avoid an invalid-end-state report during the verification.

B Transaction Protocol State Tables

In this appendix, we use 9 state tables defined (in a new version of the WTP design [60]) for Class 2 transaction service to show the operation of the Transaction Protocol. There are four state tables for describing the state changes of the Init-PE and five for the Resp-PE, one table for each possible state of the TR-PE. Each state table entry can be referred to as a tuple of (*event, condition, action, next state*), where

- The *event* may be one of three types: the receipt of *request* or *response* primitives from the TR-User, the receipt of PDUs from the peer TR-PE, or internal events (e.g., time-outs) of the TR-PE.
- The *conditions*, if any, of the incoming event are predicates on: the parameters of service primitives (e.g., Ack-Type), the header fields of PDUs (e.g., RID), and the counters or variables (e.g., Re-transmission Counter, RCR) defined for WTP.
- The *actions* include any combination of the following: sending or queuing PDUs, delivering primitives to TR-Service-User, modifying variables, resetting or incrementing counters, and starting or stopping timers. A certain set of actions is taken when an event occurs and its conditions are met.
- The *next state* specifies the next state for the TR-PE to enter after the actions have been taken. It must be one of the states defined by the state tables.

For further analysis of the state tables, please refer to Section 3.4.2 in the report.

WTP Initiator NULL			
Event	**Condition**	**Action**	**Next State**
TR-Invoke.req	Class == 2 | 1	SendTID = GenTID Send Invoke PDU Reset RCR Start timer, R [RCR] Uack = False	RESULT WAIT
	Class == 2 | 1 UserAck	SendTID = GenTID Send Invoke PDU Reset RCR Start timer, R [RCR] Uack = True	
	Class == 0	SendTID = GenTID Send Invoke PDU	NULL

Figure 36: State table for Init-PE in *Null* state

WTP Inititator RESULT WAIT			
Event	**Condition**	**Action**	**Next State**
TR-Abort.req		Abort transaction Send Abort PDU (USER)	NULL
RcvAck	Class == 2 HoldOn == False	Stop timer Generate T-TRInvoke.cnf HoldOn = True	RESULT WAIT
	Class == 2 HoldOn == True	Ignore	RESULT WAIT
	Class == 1	Stop timer Generate T-TRInvoke.cnf	NULL
	TIDve Class == 2 | 1 RCR < MAX_RCR	Send Ack(TIDok) Increment RCR Start timer, R [RCR]	RESULT WAIT
	TIDve Class == 2 | 1	Ignore	RESULT WAIT
RcvAbort		Abort transaction Generate TR-Abort.ind	NULL
RcvErrorPDU		Abort Transaction Send Abort PDU (PROTOERR) Generate TR-Abort.ind	NULL
TimerTO_R	RCR < MAX_RCR	Increment RCR Start timer R [RCR] Send Invoke PDU	RESULT WAIT
	RCR < MAX_RCR, Ack(TIDok) already sent	Increment RCR Start timer R [RCR] Send Ack(TIDok)	RESULT WAIT
	RCR == MAX_RCR	Abort transaction Generate TR-Abort.ind	NULL
RcvResult	Class == 2 HoldOn == True	Stop timer Generate TR-Result.ind Start timer, A	RESULT RESP WAIT
	Class == 2 HoldOn == False	Stop timer Generate TR-Invoke.cnf Generate TR-Result.ind Start timer, A	

Figure 37: State table for Init-PE in *Result_Wait* state

WTP Inititator RESULT RESP WAIT (class 2 only)			
Event	**Condition**	**Action**	**Next State**
TR-Result.res		Queue(A) Ack PDU Start timer, W	WAIT TIMEOUT
	ExitInfo	Queue(A) Ack PDU with Info TPI Start timer, W	
RcvAbort		Abort transaction Generate T-TRAbort.ind	NULL
TR-Abort.req		Abort transaction Send Abort PDU (USER)	
RcvErrorPDU		Abort Transaction Send Abort PDU (PROTOERR) Generate TR-Abort.ind	NULL
RcvResult		Ignore	RESULT RESP WAIT
TimerTO_A	AEC < AEC_MAX	Increment AEC Start timer, A	RESULT RESP WAIT
	AEC == AEC_MAX	Abort transaction Send Abort PDU (NORESPONSE)	NULL
	Uack == False	Queue(A) Ack PDU Start timer, W	WAIT TIMEOUT

Figure 38: State table for Init-PE in *Result_Response_Wait* state

WTP Inititator WAIT TIMEOUT (class 2 only)			
Event	**Condition**	**Action**	**Next State**
RcvResult	RID=0	Ignore	WAIT TIMEOUT
RcvResult	RID=1	Send Ack PDU	WAIT TIMEOUT
RcvResult	RID=1, ExitInfo	Send Ack PDU with info TPI	WAI T TIMEOUT
RcvAbort		Abort transaction Generate T-TRAbort.ind	NULL
RcvErrorPDU		Abort Transaction Send Abort PDU (PROTOERR) Generate TR-Abort.ind	NULL
TimerTO_W		Clear Transaction	NULL
TR-Abort.req		Abort transaction Send Abort PDU (USER)	NULL

Figure 39: State table for Init-PE in *Wait_Timeout* state

WTP Responder LISTEN			
Event	**Condition**	**Action**	**Next State**
RcvInvoke	Class == 2 | 1 Valid TID U/P flag	Generate TR-Invoke.ind Start timer, A Uack = True	INVOKE RESP WAIT
	Class == 2 | 1 Valid TID	Generate TR-Invoke.ind Start timer, A Uack = False	
	Class == 0	Generate TR-Invoke.ind	LISTEN
	Class == 2 | 1 Invalid TID	Send Ack(TIDve)	TIDOK WAIT
RcvErrorPDU		Send Abort PDU (PROTOERR)	LISTEN

Figure 40: State table for Resp-PE in *Listen* state

WTP Responder TIDOK WAIT			
Event	**Condition**	**Action**	**Next State**
RcvAck	Class == 2 | 1 TIDok	Generate TR-Invoke.ind Start timer, A	INVOKE RESP WAIT
RcvErrorPDU		Send Abort PDU (PROTOERR) Abort Transaction	LISTEN
RcvAbort		Abort transaction	LISTEN
RcvInvoke	RID=0	Ignore	TIDOK WAIT
	RID=1	Send Ack(TIDve)	TIDOK WAIT

Figure 41: State table for Resp-PE in *TIDok_Wait* state

WTP Responder INVOKE RESP WAIT			
Event	**Condition**	**Action**	**Next State**
TR-Invoke.res	Class == 1 ExitInfo	Queue(A) Ack PDU with InfoTPI Start timer,W	WAIT TIMEOUT
	Class == 1	Queue(A) Ack PDU Start timer, W	
	Class == 2	Start timer, A	RESULT WAIT
TR-Result.req		Reset RCR Start timer R[RCR] Send Result PDU	RESULT RESP WAIT
TR-Abort.req		Abort transaction Send Abort PDU (USER)	LISTEN
RcvAbort		Generate TR-Abort.ind Abort transaction	LISTEN
RcvInvoke		Ignore	INVOKE RESP WAIT
RcvErrorPDU		Abort Transaction Send Abort PDU (PROTOERR) Generate TR-Abort.ind	LISTEN
TimerTO_A	AEC < AEC_MAX	Increment AEC Start t imer, A	INVOKE RESP WAIT
	AEC == AEC_MAX	Abort transaction Send Abort PDU (NORESPONSE) Generate TR-Abort.ind	LISTEN
	Class == 1 Uack == False	Queue(A) Ack PDU Start timer, W	WAIT TIMEOUT
	Class == 2 Uack == False	Send Ack PDU	RESULT WAIT

Figure 42: State table for Resp-PE in *Invoke_Response_Wait* state

WTP Responder RESULT WAIT (class 2 only)			
Event	**Condition**	**Action**	**Next State**
TR-Result.req		Reset RCR Start timer, R[RCR] Send Result PDU	RESULT RESP WAIT
RcvInvoke	RID=0	Ignore	RESULT WAIT
	RID=1	Ignore	RESULT WAIT
	RID=1, Ack PDU already sent	Resend Ack PDU	RESULT WAIT
RcvErrorPDU		Abort Transaction Send Abort PDU (PROTOERR) Generate TR-Abort.ind	LISTEN
TR-Abort.req		Abort transaction Send Abort PDU (USER)	LISTEN
RcvAbort		Generate T-TRAbort.ind Abort transaction	LISTEN
TimerTO_A		Send Ack PDU	RESULT WAIT

Figure 43: State table for Resp-PE in *Result_Wait* state

WTP Responder RESULT RESP WAIT (class 2 only)			
Event	**Condition**	**Action**	**Next State**
TR-Abort.req		Abort transaction Send Abort PDU (USER)	LISTEN
RcvAbort		Generate T-TRAbort.ind Abort transaction	LISTEN
RcvAck	TIDok	Ignore	RESULT RESP WAIT
RcvAck		GenerateTR-Result.cnf	LISTEN
RcvErrorPDU		Abort Transaction Send Abort PDU (PROTOERR) Generate TR-Abort.ind	LISTEN
TimerTO_R	RCR < MAX_RCR	Increment RCR Send Result PDU Start timer, R [RCR]	RESULT RESP WAIT
	RCR == MAX_RCR	Generate T-TRAbort.ind Abort transaction	LISTEN

Figure 44: State table for Resp-PE in *Result_Response_Wait* state

C PROMELA models

C.1 PROMELA code for the TR-Service model

C.1.1 DeclareVar.pml

```
/*****************************************************\
* Configuration of model parameters                   *
\*****************************************************/

#define ACK_TYPE 0      /*set 1 to turn on UserAck*/
 #define ABORTallowed 1  /*set 0 to disable Abort.req/ind */
 #define MONITOR 0       /*set 1 to turn on the monitor for verification, 0 to turn off in simulation.*/

/*****************************************************\
* Declaration of global variables (channels and controls)*
\*****************************************************/

#define I2RSIZE 2  /*channel capacity*/
 #define R2ISIZE 3

mtype = { INVOKE, ABORT, ACK, RESULT }; /*Messages*/

chan Init2Resp = [I2RSIZE] of {mtype};  /*channels*/
chan Resp2Init = [R2ISIZE] of {mtype};

bit NoAck=1;    /*global primitive sequence control*/
bit Aflag=1;    /*Check atomicity*/
#define SET_ATOM    Aflag = 0
#define RESET_ATOM  Aflag = 1

local bit DLock_I,DLock_R; /*book-keeps*/

#define W 1     /*1 or 2 */
 local unsigned  /*14 primitive toggles*/
    IREQ_tgl:W, ICNF_tgl:W, RIND_tgl:W, RRES_tgl:W, AREQ_tgl:W, AINDP_tgl:W, AIND_tgl:W,
    ires_tgl:W, rcnf_tgl:W, rreq_tgl:W, iind_tgl:W, areq_tgl:W, aind_tgl:W, aindp_tgl:W;

/*****************************************************\
* Declaration of local variables                      *
\*****************************************************/
```

```
mtype = {   /*local SAP states*/
    I_NULL, I_INVOKE_RESP_WAIT, I_RESULT_WAIT, I_RESULT_RESP_WAIT,
    R_LISTEN, R_INVOKE_RESP_WAIT, R_RESULT_WAIT, R_RESULT_RESP_WAIT
};
local mtype istate = I_NULL, rstate = R_LISTEN;

local bit User_Ack = ACK_TYPE; /*local control on Initiator side*/
local bit First_Invoke = 1;   /*local control on Responder side*/

/*****************************************************\
* Final state  conditions                             *
\*****************************************************/

#define COMPLETE_I (istate == I_NULL) && (User_Ack == !ACK_TYPE)
#define COMPLETE_R (rstate == R_LISTEN) && (First_Invoke == 0)
```

C.1.2 Transitions.pml

```
/*****************************************************\
* Guard Alphabet and Atomic Actions on Initiator side   *
\*****************************************************/

inline IREQ() {
    atomic {
        (Aflag && (istate==I_NULL) && User_Ack==ACK_TYPE) ->
        SET_ATOM;
            Init2Resp!INVOKE;
            User_Ack=!ACK_TYPE;
            istate=I_INVOKE_RESP_WAIT;
            IREQ_tgl++; printf("MSC: IREQ@\n");
        RESET_ATOM;
    }
}

inline ICNF() {
    atomic {
        (Aflag && (istate==I_INVOKE_RESP_WAIT) && Resp2Init??[ACK] ) ->
        SET_ATOM;
            Resp2Init??ACK;
            NoAck=1;
            istate=I_RESULT_WAIT;
            ICNF_tgl++; printf("MSC: ICNF@\n");
        RESET_ATOM;
    }
}

inline RIND() {
    atomic {
        (Aflag
        && ((istate==I_RESULT_WAIT)
            || ((istate==I_INVOKE_RESP_WAIT) && !ACK_TYPE))
        && (NoAck && Resp2Init??[RESULT])) ->
        SET_ATOM;
            Resp2Init??RESULT;
            NoAck=0;
            istate=I_RESULT_RESP_WAIT;
            RIND_tgl++; printf("MSC: RIND@\n");
        RESET_ATOM;
    }
}

inline RRES() {
    atomic {
```

```
        (Aflag && (istate==I_RESULT_RESP_WAIT)) ->
        SET_ATOM;
            Init2Resp!ACK;
            istate=I_NULL;
            RRES_tgl++; printf("MSC: RRES@\n");
        RESET_ATOM;
    }
}

inline AREQ() {
    atomic {
        (Aflag && !(istate==I_NULL)) ->
        SET_ATOM;
            Init2Resp!ABORT;
            istate=I_NULL;
            AREQ_tgl++; printf("MSC: AREQ@\n");
        RESET_ATOM;
    }
}

inline AIND() {
    atomic {
        (Aflag && !(istate==I_NULL) && Resp2Init??[ABORT]) ->
        SET_ATOM;
            Resp2Init??ABORT;
            istate=I_NULL;
            AIND_tgl++; printf("MSC: AIND@\n");
        RESET_ATOM;
    }
}

inline AINDP(){
    atomic {
        (Aflag && !(istate==I_NULL)) ->
        SET_ATOM;
            istate=I_NULL;
            AINDP_tgl++; printf("MSC: AINDP@\n");
        RESET_ATOM;
    }
}

/*****************************************************\
* Guard Alphabet and Atomic Actions on Responder side   *
\*****************************************************/

inline iind() {
    atomic {
        (Aflag && (rstate==R_LISTEN) && Init2Resp??[INVOKE]) ->
        SET_ATOM;
            Init2Resp??INVOKE;
            First_Invoke=0;
            rstate=R_INVOKE_RESP_WAIT;
            iind_tgl++; printf("MSC: iind@\n");
        RESET_ATOM;
    }
}

inline ires() {
    atomic {
        (Aflag && (rstate==R_INVOKE_RESP_WAIT) && NoAck) ->
        SET_ATOM;
            Resp2Init!ACK;
            NoAck=0;
            rstate=R_RESULT_WAIT;
```

```
            ires_tgl++; printf("MSC: ires@\n");
        RESET_ATOM;
    }
}

inline rreq() {
    atomic {
        (Aflag
        && ((rstate==R_RESULT_WAIT)
            || ((rstate==R_INVOKE_RESP_WAIT) && !ACK_TYPE))) ->
        SET_ATOM;
            Resp2Init!RESULT;
            rstate=R_RESULT_RESP_WAIT;
            rreq_tgl++; printf("MSC: rreq@\n");
        RESET_ATOM;
    }
}

inline rcnf() {
    atomic {
        (Aflag && (rstate==R_RESULT_RESP_WAIT) && Init2Resp??[ACK]) ->
        SET_ATOM;
            Init2Resp??ACK;
            rstate=R_LISTEN;
            rcnf_tgl++; printf("MSC: rcnf@\n");
        RESET_ATOM;
    }
}

inline areq() {
    atomic {
        (Aflag && !(rstate==R_LISTEN)) ->
        SET_ATOM;
            Resp2Init!ABORT;
            rstate=R_LISTEN;
            areq_tgl++; printf("MSC: areq@\n");
        RESET_ATOM;
    }
}

inline aind() {
    atomic {
        (Aflag && !(rstate==R_LISTEN) && Init2Resp??[ABORT]) ->
        SET_ATOM;
            Init2Resp??ABORT;
            rstate=R_LISTEN;
            aind_tgl++; printf("MSC: aind@\n");
        RESET_ATOM;
    }
}

inline aindp() {
    atomic {
        (Aflag && !(rstate==R_LISTEN)) ->
        SET_ATOM;
            rstate=R_LISTEN;
            aindp_tgl++; printf("MSC: aindp@\n");
        RESET_ATOM;
    }
}
```

C.1.3 Monitor.pml

```
#define BOUNDNESS (len(Resp2Init)<=R2ISIZE && len(Init2Resp)<=I2RSIZE)
```

```
active proctype monitor() {
end: atomic {
        if
        :: !BOUNDNESS -> assert(BOUNDNESS);
        :: !Aflag -> assert(Aflag);
        :: (DLock_R || DLock_I) -> assert(!DLock_R && !DLock_I);
        fi
    }
}
```

C.1.4 TR-User-Process.pml

```
 /**************Beginning of Code**********************/

#include "DeclareVar.pml"
#include "Transitions.pml"

/*****************************************************\
* PROCESS TYPE                                        *
\*****************************************************/

active proctype TR_Init_User() {

progressI:  do
            :: atomic {
                timeout ->
                    DLock_I=1 -> printf("MSC: DEADLOCK-I\n")-> break
                }
            :: atomic {
                COMPLETE_I ->
                    printf("MSC: COMPLETE-I\n") -> break
                }
            :: IREQ()
            :: ICNF()
            :: RIND()
            :: RRES()
    #if ABORTallowed
            :: AREQ()
            :: AIND()
            :: AINDP()
    #endif
            od
}

active proctype TR_Resp_User() {

progressR:  do
            :: atomic {
                timeout ->
                    DLock_R=1 -> printf("MSC: DEADLOCK-R\n")-> break
                }
            :: atomic {
                COMPLETE_R ->
                    printf("MSC: COMPLETE-R\n") -> break
                }
            :: iind()
            :: rreq()
            :: ires()
            :: rcnf()
    #if ABORTallowed
            :: aind()
            :: aindp()
            :: areq()
```

```
    #endif
            od
}

 #if MONITOR
 #include "Monitor.pml"
 #endif

 /*********************End of Code**********************/
```

C.2 GNU m4 input for generating the TR-Protocol PROMELA model

C.2.1 SetConfig.m4

```
divert(-1) changecom(‘/*’,‘*/’)

define(‘m4_forloop’,‘ ‘’dnl
    pushdef(‘$1’,‘$2’)_forloop(‘$1’,‘$2’,‘$3’,‘$4’)popdef(‘$1’)
’)
define(‘_forloop’,‘$4‘’ifelse($1,‘$3’, ,‘ ‘’dnl
   define(‘$1’,incr($1))_forloop(‘$1’,‘$2’,‘$3’,‘$4’)’) ‘’dnl
’)

 /*****m4 command options*****************************/

ifdef(‘usr_ack’,‘define(‘ACK_TYPE’, usr_ack)’,
                ‘define(‘ACK_TYPE’, 0)’
)

ifdef(‘RCRmaxI’,‘define(‘RCR_MAX_I’, RCRmaxI)’,
                ‘define(‘RCR_MAX_I’, 2)’
)

ifdef(‘RCRmaxR’,‘define(‘RCR_MAX_R’,RCRmaxR)’,
                ‘define(‘RCR_MAX_R’,1)’
)

ifdef(‘lossy’,‘define(‘LOSSY’, lossy)’,
                ‘define(‘LOSSY’, 0)’
)

ifdef(‘tsp_abort’,‘define(‘TSP_ABORT’, tsp_abort)’,
                ‘define(‘TSP_ABORT’, 0)’
)

ifdef(‘w’,‘define(‘W’, w)’,
            ‘define(‘W’,0)’
) /*bit numbers of unsigned variables*/

ifdef(‘monitor’,‘define(‘MONITOR’, monitor)’,
                ‘define(‘MONITOR’, 1)’
)

ifdef(‘abort’,‘define(‘ABORTallowed’, abort)’,
                ‘define(‘ABORTallowed’, 1)’
)

 /*set modifications*/
define(‘modificationNo’, 7)
 define(‘modify’, ‘modified$1’)
 define(‘MODIFY’, ‘MODIFIED$1’)
```

```
m4_forloop(‘i’, 1, modificationNo, ‘
    ifelse(modify(i),0,‘define(MODIFY(i), 0)’,
                        ‘define(MODIFY(i), 1)’
    )
’)

 ifelse(modify(3a),0,‘define(MODIFY(3a), (modify(3a) && modify(3)))’,
                    ‘define(MODIFY(3a), (modify(3a) && modify(3)))’
 )       /*bit MODIFIED3a set 1 to correct error of rcnf without RRES*/

divert‘’dnl
```

C.2.2 DeclareVar.m4

```
/*****************************************************\
* Type declarations                                   *
\*****************************************************/

mtype = {
 /*PDUs*/                INVOKE, ABORT, ACK, RESULT,
 /*TR-PE-Init states*/   I_NULL, I_RESULT_WAIT, I_RESULT_RESP_WAIT, I_WAIT_TIMEOUT
 /*TR-PE-Resp states*/   R_LISTEN, R_TIDOK_WAIT, R_INVOKE_RESP_WAIT, R_RESULT_WAIT, R_RESULT_RESP_WAIT
};

/*PDU header fields*/

typedef InovkePDU {
    bit RID;   /*Re-transmission Indicator*/
    bit UPack  /*Acknowledgement Type, 1=by User, 0=by Protocol Entity */
};

typedef ResultPDU {
    bit RID    /*Re-transmission Indicator*/
ifelse(MODIFIED3,1,‘‘’dnl
    ;bit CNF   /*Confirm Indicator*/’, ‘‘’dnl’ dnl
) };

typedef AckPDU {
    bit RID;   /*Re-transmission Indicator*/
    bit TveTok /*TID Verification or TID Ok*/
ifelse(MODIFIED3,1, ‘‘’dnl
    ;bit CNF   /*Confirm Indicator*/’, ‘‘’dnl’ dnl
) };

/*local control data: Timers, Counters and Variables*/

typedef InitData {
    bit Uack;      /*1 when UserAck=On*/
ifelse(RCR_MAX_I,8, ‘‘’dnl
    unsigned RCR: 4;’, ‘‘’dnl
    unsigned RCR: 3; /*Retransmission Counter, !RCR==0 if RCR>0*/’ dnl
)
    bit AckSent;   /*1 when True if Ack(TIDok) PDU sent*/
    bit Timer;     /*1 when Timer on*/
ifelse(MODIFIED3,1, ‘‘’dnl
    bit Ucnf       /*1 when ires/RRES primitive sent*/’, ‘‘’dnl
    bit HoldOn     /*only used on Init side*/’ dnl
) };

typedef RespData {
    bit Uack;
ifelse(RCR_MAX_R,8,   ‘‘’dnl
    unsigned RCR:4;’, ‘‘’dnl
```

```
    unsigned RCR:3;     'dnl
)
    bit AckSent;
    bit Timer;
ifelse(MODIFIED3,1,` ''dnl
    bit Ucnf', ``'dnl'dnl
) };

/*****************************************************\
* Declaration of global variables                     *
\*****************************************************/
divert(-1)
 /*channel capacity*/
ifelse(RCR_MAX_R, 0, `define(`I2RSIZE', eval(RCR_MAX_I+2))',
                     `define(`I2RSIZE', eval(RCR_MAX_I+RCR_MAX_R+1))'
)
 define(`R2ISIZE',eval(RCR_MAX_I+RCR_MAX_R+3))
divert``'dnl

/*channels*/
 chan Init2Resp = [I2RSIZE] of {mtype, bit, bit, bit};
 chan Resp2Init = [R2ISIZE]of {mtype, bit, bit, bit};

/*PDUs*/
 InovkePDU invoke;
 ResultPDU result;
 AckPDU    ack;

/*Check atomicity*/
 bit Aflag=1;
 #define SET_ATOM    Aflag = 0
 #define RESET_ATOM  Aflag = 1

/*book-keeps*/
 local bit DLock_I,DLock_R;

local unsigned  /*14 primitive toggles*/
    IREQ_tgl:W, ICNF_tgl:W, RIND_tgl:W, RRES_tgl:W, AREQ_tgl:W, AINDP_tgl:W, AIND_tgl:W,
    ires_tgl:W, rcnf_tgl:W, rreq_tgl:W, iind_tgl:W, areq_tgl:W, aind_tgl:W, aindp_tgl:W,
    AREQ1_tgl:W, AINDP1_tgl:W;

define(`CNT_TGL1', 1)

/*****************************************************\
* Declaration of local variables                      *
\*****************************************************/

local mtype istate=I_NULL, rstate=R_LISTEN; /*TR-PE state*/

local InitData idata;   /*local controls on the Initiator side*/
 local bit UserAck = ACK_TYPE;
local RespData rdata;   /*Responder side*/
 local bit First_Invoke = 1;

/*****************************************************\
* Local (r/i)data updating actions                    *
\*****************************************************/

inline SetUack(t,u) {
    t.Uack= u
}

inline SetAckSent(t) {
    t.AckSent=1
```

```
}

inline IncRCR(t) {
    t.RCR++
}

inline ResetRCR(t) {
    t.RCR=0
}

inline StartTimer(t) {
    t.Timer=1
}

inline StopTimer(t) {
    t.Timer=0
}

ifelse(MODIFIED3,1, `inline SetUcnf(t) {
    t.Ucnf=1
}

inline ResetUcnf(t) {
    t.Ucnf=0
}', `inline SetHoldOn(t) {
    t.HoldOn=1
}' dnl )

inline ClearI(t) {
    t.Uack= 0;
    t.RCR= 0;
    t.AckSent= 0;
    t.Timer=0;
ifelse(MODIFIED3,1, ``'dnl
    t.Ucnf=0;', ``'dnl
    t.HoldOn=0;' dnl
) }

inline ClearR(t) {
    t.Uack= 0;
    t.RCR= 0;
    t.AckSent= 0;
    t.Timer=0;
ifelse(MODIFIED3,1, ``'dnl
    t.Ucnf=0;', ``'dnl' dnl
) }

/*****************************************************\
* Final state  conditions                             *
\*****************************************************/

ifelse(MODIFIED3, 1,
 `#define RETURN_I (``'dnl
   (idata.Ucnf==0)&&(idata.Uack==0)&&(idata.RCR==0)&&(idata.AckSent==0)&&(idata.Timer==0)
   &&(istate==I_NULL)``'dnl
 )',
 `#define RETURN_I (``'dnl
   (idata.HoldOn==0)&&(idata.Uack==0)&&(idata.RCR==0)&&(idata.AckSent==0)&&(idata.Timer==0)
   &&(istate==I_NULL)``'dnl
 )'
)dnl

#define SUCCESS_I (RETURN_I && (UserAck==!ACK_TYPE))
```

```
ifelse(MODIFIED1, 1,
 `#define RETURN_R (``'dnl
    (rdata.Timer==1) && (rdata.Uack==0) && (rdata.RCR==0) && (rdata.AckSent==0) && (rstate==R_LISTEN)``'dnl
 )',
 `#define RETURN_R (``'dnl
    (rdata.Timer==0) && (rdata.Uack==0) && (rdata.RCR==0) && (rdata.AckSent==0) && (rstate==R_LISTEN)``'dnl
 )'
)dnl

#define SUCCESS_R       (RETURN_R && (First_Invoke == 0))
 #define ABORTED_I      (AREQ1_tgl || AINDP1_tgl)
 #define NO_INVOKE_R    ABORTED_I
```

C.2.3 ITransitions.m4

```
/*****************************************************\
* TR-Init-PE State Table Structure                    *
\*****************************************************/

divert(-1)

define(`NumberOfStatesI', 5) /*extra one for abort primitives*/
 define(`StateI', `StateI$1')
 define(`TableEntryI', `TableEntryI$1')
 define(`ActionNameI', `ActionNameI$1$2')
 define(`GuardStateI', `GuardStateI$1')
 define(`GuardDataI', `GuardDataI$1$2')
 define(`TglI', `TglI$1$2')
 define(`ActionsI', `ActionsI$1$2')
 define(`NextStateI', `NextStateI$1$2')
 define(`StateMsgI', `StateMsgI$1')

 /*initialization*/
define(`StateI1', `I_NULL')
 define(`StateI2', `I_RESULT_WAIT')
 define(`StateI3', `I_RESULT_RESP_WAIT')
 define(`StateI4', `I_WAIT_TIMEOUT')

define(`TableEntryI1', 2)
 define(`TableEntryI2', 15)
 define(`TableEntryI3', 9)
 define(`TableEntryI4', 6)
 define(`TableEntryI5', 3)

m4_forloop(`i', 1, NumberOfStatesI, ` ``'dnl
    ifelse(i, NumberOfStatesI,`define(GuardStateI(i), `Aflag && !(istate == ``'StateI(1))')',  /*abort*/
                               `define(GuardStateI(i), `Aflag && (istate == ``'StateI(i))')'  /*others*/
    )
')

 /*messages to print out*/
define(`StateMsgI1', `I_NUL') define(`StateMsgI2', `I_RW')
define(`StateMsgI3', `I_RRW') define(`StateMsgI4', `I_WT')

/*****************************************************\
* Definition of each table entry                      *
\*****************************************************/

/*****I_NULL*******************************************/

define(`State', 1)
define(`Entry', 1)
define(ActionNameI(State,Entry), `IREQ')
define(GuardDataI(State,Entry), `UserAck==ACK_TYPE')
```

```
define(TglI(State,Entry), ‘IREQ_tgl’)
define(ActionsI(State,Entry),
        ‘UserAck=!ACK_TYPE;
        Init2Resp!INVOKE(0, ACK_TYPE, pidtmp);
        SetUack(idata,ACK_TYPE) -> StartTimer(idata) ->  ‘’dnl
’)
 define(NextStateI(State,Entry), ‘I_RESULT_WAIT’)

define(‘State’, 1) define(‘Entry’, 2)
define(ActionNameI(State,Entry), ‘RcvAck_Tve_T2’)
define(GuardDataI(State,Entry), ‘Resp2Init??[ACK‘,ack.RID,’1]’)
define(ActionsI(State,Entry),
        ‘Resp2Init??ACK(ack.RID, 1, pidtmp);
        Init2Resp!ABORT(0,0,pidtmp); ‘’dnl
’)
 define(NextStateI(State,Entry), ‘I_NULL’)

/*****I_RESULT_WAIT************************************/

ifelse(ABORTallowed,0, ,‘
 define(‘State’, 2)
 define(‘Entry’, 1)
 define(ActionNameI(State,Entry), ‘AREQ1’)
 define(GuardDataI(State,Entry), ‘Init2Resp??[INVOKE] && !idata.RCR’)
 define(TglI(State,Entry), ‘AREQ_tgl’)
 define(ActionsI(State,Entry),
        ‘AREQ1_tgl++;
        Init2Resp??INVOKE(invoke.RID, invoke.UPack, pidtmp);
        ClearI(idata) ->   ‘’dnl
 ’)
 define(NextStateI(State,Entry), ‘I_NULL’)

 define(‘State’, 2)
 define(‘Entry’, 14)
 define(ActionNameI(State,Entry), ‘AINDP1’)
 define(GuardDataI(State,Entry), ‘Init2Resp??[INVOKE] && !idata.RCR’)
 define(TglI(State,Entry), ‘AINDP_tgl’)
 define(ActionsI(State,Entry),
        ‘AINDP1_tgl++;
        Init2Resp??INVOKE(invoke.RID, invoke.UPack, pidtmp);
        ClearI(idata) ->   ‘’dnl
 ’)
 define(NextStateI(State,Entry), ‘I_NULL’)

’)  /*end of ABORTallowed*/

define(‘State’, 2)
define(‘Entry’, 2)
define(ActionNameI(State,Entry), ‘ICNF1’)
 ifelse(MODIFY(3),1,‘define(GuardDataI(State,Entry), ‘!idata.Ucnf && Resp2Init??[ACK‘,ack.RID,0,’1]’)’,
                    ‘define(GuardDataI(State,Entry), ‘!idata.HoldOn && Resp2Init??[ACK‘,ack.RID,’0]’)’
)
 ifelse(CNT_TGL1,1,‘define(TglI(State,Entry), ‘ICNF_tgl’)’,
                    ‘define(TglI(State,Entry), ‘ICNF1_tgl’)’
)
 define(ActionsI(State,Entry),
    ‘ifelse(MODIFY(3),1,
        ‘Resp2Init??ACK(ack.RID, 0, 1);
        SetUcnf(idata) -> ’,
        ‘Resp2Init??ACK(ack.RID, 0, pidtmp);
        SetHoldOn(idata) -> ’ ‘’dnl
    )
        StopTimer(idata) -> ‘’dnl
’)
```

```
 define(NextStateI(State,Entry), ‘I_RESULT_WAIT’)

define(‘State’, 2)
define(‘Entry’, 5)
define(ActionNameI(State,Entry), ‘RcvAck_TIDOK’)
 define(GuardDataI(State,Entry), ‘(idata.RCR < RCR_MAX_I) && Resp2Init??[ACK‘,ack.RID,’1]’)
define(ActionsI(State,Entry),
    ‘ifelse(MODIFY(3),1,
        ‘Init2Resp!ACK(0,1,0);’,
        ‘Init2Resp!ACK(0,1,pidtmp);’ ‘’dnl
    )
        Resp2Init??ACK(ack.RID, 1, pidtmp);
        SetAckSent(idata) -> IncRCR(idata) -> StartTimer(idata) -> ‘’dnl
’)
 define(NextStateI(State,Entry), ‘I_RESULT_WAIT’)

define(‘State’, 2)
define(‘Entry’, 9)
define(ActionNameI(State,Entry), ‘T2R_2ndINVOKE’)
 define(GuardDataI(State,Entry), ‘(idata.RCR < RCR_MAX_I) && !idata.AckSent && idata.Timer’)
 define(ActionsI(State,Entry),
        ‘Init2Resp!INVOKE(1, idata.Uack, pidtmp);
        IncRCR(idata) -> StartTimer(idata) ->‘’dnl
’)
 define(NextStateI(State,Entry), ‘I_RESULT_WAIT’)

define(‘State’, 2)
define(‘Entry’, 10)
define(ActionNameI(State,Entry), ‘T2R_2ndTIDok’)
 define(GuardDataI(State,Entry), ‘(idata.RCR < RCR_MAX_I) && idata.AckSent && idata.Timer’)
 define(ActionsI(State,Entry),
        ‘Init2Resp!ACK(1,1,0);
        IncRCR(idata) -> StartTimer(idata) ->‘’dnl
’)
 define(NextStateI(State,Entry), ‘I_RESULT_WAIT’)

define(‘State’, 2)
define(‘Entry’, 11)
define(ActionNameI(State,Entry), ‘AINDP_T2R’)
define(GuardDataI(State,Entry), ‘(idata.RCR == RCR_MAX_I) && idata.Timer’)
 ifelse(CNT_TGL1,1,‘define(TglI(State,Entry),‘AINDP_tgl’)’,
                    ‘define(TglI(State,Entry), ‘AINDP_T2R_tgl’)’
 )
define(ActionsI(State,Entry),
        ‘ClearI(idata) ->‘’dnl
 ’)
define(NextStateI(State,Entry), ‘I_NULL’)

define(‘State’, 2)
define(‘Entry’, 12)
define(ActionNameI(State,Entry), ‘RIND1’)
 ifelse(MODIFY(3),1,‘define(GuardDataI(State,Entry),
                            ‘idata.Ucnf && Resp2Init??[RESULT] || Resp2Init??[RESULT‘,result.RID,0,’0]’
                     )’,
                    ‘define(GuardDataI(State,Entry), ‘idata.HoldOn && Resp2Init??[RESULT]’)’
 )
ifelse(CNT_TGL1,1,‘define(TglI(State,Entry), ‘RIND_tgl’)’,
                    ‘define(TglI(State,Entry), ‘RIND1_tgl’)’
 )
define(ActionsI(State,Entry),
    ‘ifelse(MODIFY(3),1,
        ‘Resp2Init??RESULT(result.RID,0,result.CNF);
        ResetUcnf(idata) ->’,
        ‘Resp2Init??RESULT(result.RID,0,pidtmp) -> ’ ‘’dnl
```

```
    )
        StartTimer(idata) ->‘’dnl
’)
 define(NextStateI(State,Entry), ‘I_RESULT_RESP_WAIT’)

define(‘State’, 2)
define(‘Entry’, 13)
define(ActionNameI(State,Entry), ‘ICNF_RIND’)
 ifelse(MODIFY(3),1,‘define(GuardDataI(State,Entry), ‘!idata.Ucnf && Resp2Init??[RESULT‘,result.RID,0,’1]’)’,
                    ‘define(GuardDataI(State,Entry), ‘!idata.HoldOn && Resp2Init??[RESULT]’)’
 )
 ifelse(CNT_TGL1,1,‘define(TglI(State,Entry), ‘ICNF_tgl++; RIND_tgl’)’,
                    ‘define(TglI(State,Entry), ‘ICNF2_tgl++; RIND2_tgl’)’
 )
define(ActionsI(State,Entry),
    ‘ifelse(MODIFY(3),1,
        ‘Resp2Init??RESULT(result.RID,0,1);
        ResetUcnf(idata) ->’,
        ‘Resp2Init??RESULT(result.RID,0,pidtmp) -> ’ ‘’dnl
    )
        StartTimer(idata) ->‘’dnl
’)
 define(NextStateI(State,Entry), ‘I_RESULT_RESP_WAIT’)

ifelse(MODIFY(3),0, ,‘
 define(‘State’, 2)
 define(‘Entry’, 15)/*Line 2a*/
 define(ActionNameI(State,Entry), ‘RcvAck_StopT’)
 define(GuardDataI(State,Entry), ‘
        idata.Ucnf && Resp2Init??[ACK‘,ack.RID,’0] || Resp2Init??[ACK‘,ack.RID,0,’0]
 ’)
 define(ActionsI(State,Entry),
        ‘Resp2Init??ACK(ack.RID,0,ack.CNF); /*CNF could be zero*/
        StopTimer(idata) ->  ‘’dnl
 ’)
 define(NextStateI(State,Entry), ‘I_RESULT_WAIT’)
’)

/*****I_RESULT_RESP_WAIT*********************************/

define(‘State’, 3)
define(‘Entry’, 1)
define(ActionNameI(State,Entry), ‘RRES’)
define(GuardDataI(State,Entry), 1) define(TglI(State,Entry),
‘RRES_tgl’) define(ActionsI(State,Entry),
 ‘ifelse(MODIFY(3a),1,
        ‘Init2Resp!ACK(0,0,1);
        SetUcnf(idata) ->’,
        ‘Init2Resp!ACK(0,0,pidtmp) -> ’ ‘’dnl
  )
        StartTimer(idata) ->‘’dnl
’)
 define(NextStateI(State,Entry), ‘I_WAIT_TIMEOUT’)


ifelse(ABORTallowed,0, ,‘
 define(‘State’, 3)
 define(‘Entry’, 7)
 define(ActionNameI(State,Entry), ‘AINDP_T2A’)
 define(GuardDataI(State,Entry), ‘idata.Uack && idata.Timer’)
 ifelse(MODIFY(4),0,‘’,
        ‘ifelse(CNT_TGL1,1,‘define(TglI(State,Entry), ‘AINDP_tgl’)’,
                           ‘define(TglI(State,Entry), ‘AINDP_T2A_tgl’)’
        )
```

```
 ')
 define(ActionsI(State,Entry),
        `Init2Resp!ABORT(0,0,pidtmp);
        ClearI(idata) ->  ``dnl
 ')
 define(NextStateI(State,Entry), `I_NULL')

')  /*end of ABORTallowed*/

define(`State', 3)
define(`Entry', 9)
define(ActionNameI(State,Entry), `T2A_LastAck_I')
define(GuardDataI(State,Entry), `!idata.Uack && idata.Timer')
define(ActionsI(State,Entry),
    `ifelse(MODIFY(3),1,
        `Init2Resp!ACK(0,0,0);',
        `Init2Resp!ACK(0,0,pidtmp);' ``dnl
    )
        StartTimer(idata) ->``dnl
')
 define(NextStateI(State,Entry), `I_WAIT_TIMEOUT')

/*****I_WAIT_TIMEOUT***********************************/

define(`State', 4)
define(`Entry', 2)
define(ActionNameI(State,Entry), `RcvResult_2ndLastAck')
define(GuardDataI(State,Entry), `Resp2Init??[RESULT`,'1]')
define(ActionsI(State,Entry),
    `ifelse(MODIFY(3a),1,
        `Init2Resp!ACK(1,0,idata.Ucnf);',
        `Init2Resp!ACK(1,0,pidtmp);' ``dnl
    )
        Resp2Init??RESULT(1,0,pidtmp) ->``dnl
')
 define(NextStateI(State,Entry), `I_WAIT_TIMEOUT')

define(`State', 4)
define(`Entry', 6)
define(ActionNameI(State,Entry), `Timer2W_I')
define(GuardDataI(State,Entry), `idata.Timer')
define(ActionsI(State,Entry),
        `ClearI(idata) ->``dnl
')
 define(NextStateI(State,Entry), `I_NULL')

/*****Abort*********************************************/

ifelse(ABORTallowed,0, ,`
 define(`State', 5) define(`Entry', 1)
 define(ActionNameI(State,Entry), `AREQ')    /*I_RW_1, I_RRW_4, I_WT_7*/
 define(GuardDataI(State,Entry), `1')
 ifelse(CNT_TGL1,1,`define(TglI(State,Entry), `AREQ_tgl')',
                   `define(TglI(State,Entry), `AREQ2_tgl')'
 )
 define(ActionsI(State,Entry),
    `ifelse(MODIFY(7),1,
        `if
        :: (istate == I_WAIT_TIMEOUT) -> skip
        :: else -> Init2Resp!ABORT(0,0,pidtmp)
        fi;',
        `Init2Resp!ABORT(0,0,pidtmp) -> ' ``dnl
    )
        ClearI(idata) ->``dnl
```

```
 ')
 define(NextStateI(State,Entry), `I_NULL')

 define(`State', 5)
 define(`Entry', 2)
 define(ActionNameI(State,Entry), `AIND')    /*I_RW_7, I_RRW_3, I_WT_4*/
 define(GuardDataI(State,Entry), `Resp2Init??[ABORT]')
 define(TglI(State,Entry), `AIND_tgl')
 define(ActionsI(State,Entry),
        `Resp2Init??ABORT(0,0,pidtmp);
        ClearI(idata)->``'dnl
 ')
 define(NextStateI(State,Entry), `I_NULL')

')  /*end of ABORTallowed*/

ifelse(TSP_ABORT,0,  ,`
 define(`State', 5)
 define(`Entry', 3)
 define(ActionNameI(State,Entry), `AINDP_TSP')
 define(GuardDataI(State,Entry), `1')
 ifelse(CNT_TGL1,1,`define(TglI(State,Entry), `AINDP_tgl')',
                   `define(TglI(State,Entry), `AINDP_TSP_tgl')'
 )
 define(ActionsI(State,Entry),
        `ClearI(idata)->``'dnl
 ')
 define(NextStateI(State,Entry), `I_NULL')
')

 /*****Generating Promela code in batch*************/
divert``'dnl

m4_forloop(`i', 1, NumberOfStatesI, ` ``'dnl
 m4_forloop(`j', 1, TableEntryI(i), ` ``'dnl
 ifelse(substr(``'ActionNameI(i,j),0,6),`Action', ```'dnl',`
inline ActionNameI(i,j)`_'``'j`()' {
    atomic {
        ((GuardStateI(i)) &&
         (GuardDataI(i,j))
        ) ->
        Aflag=0;
        ``'ActionsI(i,j)
        istate=``'NextStateI(i,j);
 ifelse(substr(TglI(i,j),0,3),`Tgl',  , ``'dnl /*calculate only primitve tgls*/
 `ifelse(W,0, , ``'dnl /*if count tgls*/
  `ifelse(i,5,
    `ifelse(j,3, ``'dnl /*TSP_ABORT tgl*/
        `if
        :: (istate == I_WAIT_TIMEOUT) -> skip
        :: else -> TglI(i,j)++
        fi;', ``'dnl
      `ifelse(MODIFY(6),0, , ``'dnl /*AREQ tgl*/
        `if
        :: (istate == I_WAIT_TIMEOUT) -> skip
        :: else -> TglI(i,j)++
        fi;' ``'dnl
        )' ``'dnl
      )', ``'dnl
        `TglI(i,j)++;' ``'dnl /*other tgls than ABORT*/
    ) ``'dnl
')')
        Aflag=1;
 ifelse(i,NumberOfStatesI, ` ``'dnl
```

```
        printf("MSC: ‘’ActionNameI(i,j)@\n"); ’,‘ ‘’dnl   /*print abort primitives*/
        printf("MSC: (‘’StateMsgI(i)‘’j)_‘’ActionNameI(i,j)@\n"); ‘’dnl   /*print other primitives*/
’)
    }
} ‘’dnl /*end of inline*/
’)
’)’)dnl
```

C.2.4 RTransitions.m4

```
/*****************************************************\
* TR-Resp-PE State Table Structure                    *
\*****************************************************/

divert(-1)

define(‘NumberOfStatesR’, 6)
 define(‘StateR’, ‘StateR$1’)
 define(‘TableEntryR’, ‘TableEntryR$1’)
 define(‘ActionNameR’, ‘ActionNameR$1$2’)
 define(‘GuardStateR’, ‘GuardStateR$1’)
 define(‘GuardDataR’, ‘GuardDataR$1$2’)
 define(‘TglR’, ‘TglR$1$2’)
 define(‘ActionsR’, ‘ActionsR$1$2’)
 define(‘NextStateR’, ‘NextStateR$1$2’)
 define(‘StateMsgR’, ‘StateMsgR$1’)

define(‘StateR1’, ‘R_LISTEN’)
 define(‘StateR2’, ‘R_TIDOK_WAIT’)
 define(‘StateR3’, ‘R_INVOKE_RESP_WAIT’)
 define(‘StateR4’, ‘R_RESULT_WAIT’)
 define(‘StateR5’, ‘R_RESULT_RESP_WAIT’)

define(‘TableEntryR1’, 4)
 define(‘TableEntryR2’, 6)
 define(‘TableEntryR3’, 12)
 define(‘TableEntryR4’, 8)
 define(‘TableEntryR5’, 7)
 define(‘TableEntryR6’, 3)

m4_forloop(‘i’, 1, NumberOfStatesR, ‘ ‘’dnl
    ifelse(i, NumberOfStatesR,‘define(GuardStateR(i), ‘Aflag && !(rstate == ‘’StateR(1))’)’,
                               ‘define(GuardStateR(i), ‘Aflag && (rstate == ‘’StateR(i))’)’
    )
’)

define(‘StateMsgR1’, ‘R_LIS’)
 define(‘StateMsgR2’, ‘R_TW’)
 define(‘StateMsgR3’, ‘R_IRW’)
 define(‘StateMsgR4’, ‘R_RW’)
 define(‘StateMsgR5’, ‘R_RRW’)

/*****************************************************\
* Definition of each table entry                      *
\*****************************************************/

/*****R_LISTEN*****************************************/

define(‘State’, 1)
define(‘Entry’, 1)
define(ActionNameR(State,Entry), ‘iind_validTID’)
 define(GuardDataR(State,Entry), ‘Init2Resp??[INVOKE] && First_Invoke && !rdata.Timer’)
ifelse(CNT_TGL1,1,‘define(TglR(State,Entry), ‘iind_tgl’)’,
```

```
                    ‘define(TglR(State,Entry), ‘iind_validTID_tgl’)’
)
 define(ActionsR(State,Entry),
        ‘Init2Resp??INVOKE(invoke.RID, invoke.UPack, pidtmp);
        First_Invoke=0;
        SetUack(rdata,invoke.UPack) -> StartTimer(rdata) -> ‘’dnl
’)
 define(NextStateR(State,Entry), ‘R_INVOKE_RESP_WAIT’)

define(‘State’, 1)
define(‘Entry’, 4)
define(ActionNameR(State,Entry), ‘RcvInvoke_TIDve1’)
 define(GuardDataR(State,Entry), ‘Init2Resp??[INVOKE] && !rdata.Timer’)
define(ActionsR(State,Entry),
        ‘Init2Resp??INVOKE(invoke.RID, invoke.UPack, pidtmp);
        First_Invoke=0;
    ifelse(MODIFY(3),1,‘‘’dnl
        Resp2Init!ACK(0, 1, 0);’,‘‘’dnl
        Resp2Init!ACK(0, 1, pidtmp);’
    )
    ifelse(MODIFY(2),1,‘ ‘’dnl
        StartTimer(rdata) ->’,
        ‘‘’dnl’
    )
        SetUack(rdata,invoke.UPack) -> ‘’dnl
’)
 define(NextStateR(State,Entry), ‘R_TIDOK_WAIT’)

/*****R_TIDOK_WAIT*************************************/

define(‘State’, 2)
define(‘Entry’, 1)
define(ActionNameR(State,Entry), ‘iind_TIDok’)
define(GuardDataR(State,Entry), ‘Init2Resp??[ACK‘,ack.RID,’1]’)
ifelse(CNT_TGL1,1,‘define(TglR(State,Entry), ‘iind_tgl’)’,
                    ‘define(TglR(State,Entry), ‘iind_TIDok_tgl’)’
) define(ActionsR(State,Entry),
        ‘Init2Resp??ACK(ack.RID, 1, pidtmp);
        StartTimer(rdata) -> ‘’dnl
’)
 define(NextStateR(State,Entry), ‘R_INVOKE_RESP_WAIT’)

define(‘State’, 2)
define(‘Entry’, 5)
define(ActionNameR(State,Entry), ‘RcvInvoke_TIDve2’)
define(GuardDataR(State,Entry), ‘Init2Resp??[INVOKE‘,’1]’)
define(ActionsR(State,Entry),
        ‘Init2Resp??INVOKE(1, invoke.UPack, pidtmp);
    ifelse(MODIFY(3),1,
        ‘Resp2Init!ACK(1, 1, 0);’,
        ‘Resp2Init!ACK(1, 1, pidtmp);’ ‘’dnl
    )
    ifelse(MODIFY(2),0, , ‘ ‘’dnl
        StartTimer(rdata);’ ‘’dnl
    ) ‘’dnl
’)
 define(NextStateR(State,Entry), ‘R_TIDOK_WAIT’)

ifelse(MODIFY(2),0, ,‘
 define(‘State’, 2)
 define(‘Entry’, 6)
 define(ActionNameR(State,Entry), ‘Timer2W_R’)
 define(GuardDataR(State,Entry), ‘rdata.Timer’)
 define(ActionsR(State,Entry),
```

```
        ‘ClearR(rdata) -> ‘’dnl
 ’)
 define(NextStateR(State,Entry), ‘R_LISTEN’)
’)

/*****R_INVOKE_RESP_WAIT*********************************/

define(‘State’, 3)
define(‘Entry’, 3)
define(ActionNameR(State,Entry), ‘ires’)
define(GuardDataR(State,Entry), ‘1’)
 define(TglR(State,Entry), ‘ires_tgl’)
define(ActionsR(State,Entry),
    ‘ifelse(MODIFY(3),1,
        ‘SetUcnf(rdata) ->’,
        ‘’ ‘’dnl
    )
        StartTimer(rdata) -> ‘’dnl
’)
 define(NextStateR(State,Entry), ‘R_RESULT_WAIT’)

define(‘State’, 3)
define(‘Entry’, 4)
define(ActionNameR(State,Entry), ‘rreq_UackF’)
 ifelse(MODIFY(5),1,‘define(GuardDataR(State,Entry), ‘!rdata.Uack’)’,
                   ‘define(GuardDataR(State,Entry), ‘1’)’
 )
 ifelse(CNT_TGL1,1,‘define(TglR(State,Entry), ‘rreq_tgl’)’,
                  ‘define(TglR(State,Entry), ‘rreq_UackF_tgl’)’
 )
 define(ActionsR(State,Entry),
    ‘ifelse(MODIFY(3),1,
        ‘Resp2Init!RESULT(0,0,0);’,
        ‘Resp2Init!RESULT(0,0,pidtmp);’ ‘’dnl
    )
        ResetRCR(rdata) -> StartTimer(rdata) -> ‘’dnl
’)
 define(NextStateR(State,Entry), ‘R_RESULT_RESP_WAIT’)

ifelse(ABORTallowed,0, ,‘
 define(‘State’, 3)
 define(‘Entry’, 9)
 define(ActionNameR(State,Entry), ‘aindp_T2A’)
 define(GuardDataR(State,Entry), ‘rdata.Uack && rdata.Timer’)
 ifelse(CNT_TGL1,1,‘define(TglR(State,Entry), ‘aindp_tgl’)’,
                   ‘define(TglR(State,Entry), ‘aindp_T2A_tgl’)’
 )
 define(ActionsR(State,Entry),
        ‘Resp2Init!ABORT(0,0,pidtmp);
        ClearR(rdata) -> ‘’dnl
 ’)
 define(NextStateR(State,Entry), ‘R_LISTEN’)
’)

define(‘State’, 3)
define(‘Entry’, 12)
define(ActionNameR(State,Entry), ‘T2A_HoldAck’)
define(GuardDataR(State,Entry), ‘!rdata.Uack && rdata.Timer’)
define(ActionsR(State,Entry),
    ‘ifelse(MODIFY(3),1,
        ‘Resp2Init!ACK(0,0,0);’,
        ‘Resp2Init!ACK(0,0,pidtmp);’ ‘’dnl
    )
        StopTimer(rdata) -> SetAckSent(rdata) -> ‘’dnl
```

```
')
 define(NextStateR(State,Entry), `R_RESULT_WAIT')

/*****R_RESULT_WAIT************************************/

define(`State', 4)
define(`Entry', 1)
define(ActionNameR(State,Entry), `rreq_late')
define(GuardDataR(State,Entry), `1')
ifelse(CNT_TGL1,1,`define(TglR(State,Entry), `rreq_tgl')',
                  `define(TglR(State,Entry), `rreq_late_tgl')'
)
 define(ActionsR(State,Entry),
    `ifelse(MODIFY(3),1,
        `Resp2Init!RESULT(0,0,rdata.Ucnf);',
        `Resp2Init!RESULT(0,0,pidtmp);' ``dnl
    )
        ResetRCR(rdata) -> StartTimer(rdata) -> ``dnl
')
 define(NextStateR(State,Entry), `R_RESULT_RESP_WAIT')

define(`State', 4)
define(`Entry', 4)
define(ActionNameR(State,Entry), `RcvInvoke_2ndACK')
 define(GuardDataR(State,Entry), `rdata.AckSent && Init2Resp??[INVOKE,1]')
define(ActionsR(State,Entry),
    `ifelse(MODIFY(3),1,
        `Resp2Init!ACK(1,0,rdata.Ucnf); ',
        `Resp2Init!ACK(1,0,pidtmp);' ``dnl
    )
        Init2Resp??INVOKE(1,invoke.UPack, pidtmp) -> ``dnl
')
 define(NextStateR(State,Entry), `R_RESULT_WAIT')

define(`State', 4)
define(`Entry', 8)
define(ActionNameR(State,Entry), `TimerTO_A')
define(GuardDataR(State,Entry), `rdata.Timer')
define(ActionsR(State,Entry),
    `ifelse(MODIFY(3),1,
        `Resp2Init!ACK(0,0,rdata.Ucnf); ',
        `Resp2Init!ACK(0,0,pidtmp);' ``dnl
    )
        StopTimer(rdata) -> SetAckSent(rdata) -> ``dnl
')
 define(NextStateR(State,Entry), `R_RESULT_WAIT')

/*****R_RESULT_RESP_WAIT*********************************/

define(`State', 5)
define(`Entry', 4)
define(ActionNameR(State,Entry), `rcnf')
 ifelse(MODIFY(3a),1,`define(GuardDataR(State,Entry), `Init2Resp??[ACK`,ack.RID,0,'1]')',
                     `define(GuardDataR(State,Entry), `Init2Resp??[ACK`,ack.RID,'0]')'
 )
define(TglR(State,Entry), `rcnf_tgl')
define(ActionsR(State,Entry),
    `ifelse(MODIFY(3a),1,
        `Init2Resp??ACK(ack.RID, 0, 1);',
        `Init2Resp??ACK(ack.RID, 0, pidtmp);' ``dnl
    )
        ClearR(rdata) -> ``dnl
')
 define(NextStateR(State,Entry), `R_LISTEN')
```

```
ifelse(MODIFY(3a),0, ,‘
 define(‘State’, 5)
 define(‘Entry’, 5) /*Line 4a*/
 define(ActionNameR(State,Entry), ‘no_rcnf’)
 define(GuardDataR(State,Entry), ‘Init2Resp??[ACK‘,ack.RID,0,’0]’)
 define(ActionsR(State,Entry),
        ‘Init2Resp??ACK(ack.RID, 0, 0);
         ClearR(rdata) -> ‘’dnl
 ’)
 define(NextStateR(State,Entry), ‘R_LISTEN’)
’)

define(‘State’, 5)
define(‘Entry’, 6)
define(ActionNameR(State,Entry), ‘T2R_2ndRESULT’)
 define(GuardDataR(State,Entry), ‘(rdata.RCR < RCR_MAX_R) && rdata.Timer’)
define(ActionsR(State,Entry),
    ‘ifelse(MODIFY(3),1,
        ‘Resp2Init!RESULT(1,0,rdata.Ucnf);’,
        ‘Resp2Init!RESULT(1,0, pidtmp);’ ‘’dnl
    )
        IncRCR(rdata) -> StartTimer(rdata)-> ‘’dnl
’)
 define(NextStateR(State,Entry), ‘R_RESULT_RESP_WAIT’)

define(‘State’, 5)
define(‘Entry’, 7)
define(ActionNameR(State,Entry), ‘aindp_T2R’)
 define(GuardDataR(State,Entry), ‘(rdata.RCR == RCR_MAX_R) && rdata.Timer’)
ifelse(CNT_TGL1,1,‘define(TglR(State,Entry), ‘aindp_tgl’)’,
                  ‘define(TglR(State,Entry), ‘aindp_T2R_tgl’)’
)
 define(ActionsR(State,Entry),
        ‘ClearR(rdata) -> ‘’dnl
’)
 define(NextStateR(State,Entry), ‘R_LISTEN’) ’)

/*****Abort**********************************************/

ifelse(ABORTallowed,0, ,‘
 define(‘State’, 6)
 define(‘Entry’, 1)
 define(ActionNameR(State,Entry), ‘areq’)    /*R_IRW_5, R_RW_6, R_RRW_1*/
 define(GuardDataR(State,Entry), ‘!(rstate == R_TIDOK_WAIT)’)
 define(TglR(State,Entry), ‘areq_tgl’)
 define(ActionsR(State,Entry),
        ‘Resp2Init!ABORT(0,0,pidtmp);
         ClearR(rdata) -> ‘’dnl
 ’)
 define(NextStateR(State,Entry), ‘R_LISTEN’)

 define(‘State’, 6)
 define(‘Entry’, 2)
 define(ActionNameR(State,Entry), ‘aind’)    /*R_TW_3, R_IRW_6, R_RW_7, R_RRW_2*/
 define(GuardDataR(State,Entry), ‘Init2Resp??[ABORT]’)
 define(TglR(State,Entry), ‘aind_tgl’)
 define(ActionsR(State,Entry),
        ‘Init2Resp??ABORT(0, 0, pidtmp);
        ClearR(rdata) -> ‘’dnl
 ’)
 define(NextStateR(State,Entry), ‘R_LISTEN’)

’) /*end of ABORTallowed*/
```

```
ifelse(TSP_ABORT,0, ,‘
 define(‘State’, 6)
 define(‘Entry’, 3)
 define(ActionNameR(State,Entry), ‘aindp_TSP’)
 define(GuardDataR(State,Entry), ‘1’)
 ifelse(CNT_TGL1,1,‘define(TglR(State,Entry), ‘aindp_tgl’)’,
                   ‘define(TglR(State,Entry), ‘aindp_TSP_tgl’)’
 )
 define(ActionsR(State,Entry),
        ‘ClearR(rdata)->‘’dnl
 ’)
 define(NextStateR(State,Entry), ‘R_LISTEN’)
’)

 /*****Generating Promela code in batch*******************/
divert‘’dnl

m4_forloop(‘i’, 1, NumberOfStatesR, ‘ ‘’dnl
 m4_forloop(‘j’, 1, TableEntryR(i), ‘ ‘’dnl
 ifelse(substr(‘’ActionNameR(i,j),0,6),‘Action’, ‘‘’dnl’,‘
inline ActionNameR(i,j)‘_’‘’j‘()’ {
    atomic {
        ((GuardStateR(i)) &&
         (GuardDataR(i,j))
        ) ->
        Aflag=0;
        ‘’ActionsR(i,j)
    ifelse(NextStateR(i,j), ‘R_LISTEN’,
      ‘ifelse(MODIFY(1), 0, ‘’, ‘ ‘’dnl
        StartTimer(rdata) ->’)’,
    ‘ ’ ‘’dnl
    )
        rstate=‘’NextStateR(i,j);
    ifelse(i,NumberOfStatesR, ‘ ‘’dnl
        printf("MSC: ‘’ActionNameR(i,j)@\n"); ’,‘ ‘’dnl
        printf("MSC: (‘’StateMsgR(i)‘’j)_‘’ActionNameR(i,j)@\n"); ‘’dnl
    ’)
    ifelse(substr(TglR(i,j),0,3),‘Tgl’,  , ‘’dnl
      ‘ifelse(W,0, , ‘’dnl /*if count tgls*/
       ‘ifelse(i,6,
        ‘ifelse(j,1, ‘’dnl
        ‘TglR(i,j)++;’, ‘’dnl
        ‘if
        :: (rstate == R_TIDOK_WAIT)  -> skip
        :: else -> TglR(i,j)++
        fi;’ ‘’dnl
        )’, ‘’dnl
        ‘TglR(i,j)++;’ ‘’dnl
       ) ‘’dnl
    ’)’)
        Aflag=1;
    ifelse(i,NumberOfStatesR, ‘ ‘’dnl
        printf("MSC: ‘’ActionNameR(i,j)@\n"); ’,‘ ‘’dnl
        printf("MSC: (‘’StateMsgR(i)‘’j)_‘’ActionNameR(i,j)@\n"); ‘’dnl
    ’)
    }
} ‘’dnl
’)
’)’)dnl
```

C.2.5 Lossy.m4

```
active proctype Stealing() {
```

```
 endS:  do
        :: atomic {
            !(Init2Resp??[INVOKE,0]) && (UserAck==!ACK_TYPE) && nempty(Init2Resp) -> Init2Resp??_,_,_,_
           }
        :: Resp2Init??_,_,_,_
        od
}
```

C.2.6 Monitor.m4

```
 #define BOUNDNESS1 (‘len(Resp2Init)’<=R2ISIZE && ‘len(Init2Resp)’<=I2RSIZE)
 #define BOUNDNESS2 ((idata.RCR <= RCR\_MAX\_I) && (rdata.RCR <= RCR\_MAX\_R))

active proctype monitor() {
 endM: atomic {
        if
        :: !BOUNDNESS1 -> assert(BOUNDNESS1);
        :: !BOUNDNESS2 -> assert(BOUNDNESS2);
        :: !Aflag -> assert(Aflag);
        :: (DLock_R || DLock_I) -> assert(!DLock_R && !DLock_I);
        fi
    }
}
```

C.2.7 TR-PE-Process.m4

```
 /**************Beginning of Code**********************/

include(‘SetConfig.m4’)
 include(‘DeclareVar.m4’)
 include(‘ITransitions.m4’)
 include(‘RTransitions.m4’)

/*****************************************************\
* PROCESS TYPE                                        *
\*****************************************************/

active proctype TR_Init_PE() {
 progressI:  do
             :: atomic {
                    timeout ->
                    DLock_I=1 -> printf("MSC: DEADLOCK-I\n")-> break
                }
             :: atomic {
                    ABORTED_I  ->
                    printf("MSC: ABORTED_I\n") -> break
                }
             :: atomic {
                    SUCCESS_I ->
                    printf("MSC: SUCCESSFUL-I\n") -> break
                }

m4_forloop(‘i’, 1, NumberOfStatesI,
 ‘m4_forloop(‘j’, 1, TableEntryI(i),
 ‘ifelse(substr(‘’ActionNameI(i,j),0,6),‘Action’, ‘‘’dnl’,   ‘‘’format(‘
             :: %s_%d() ‘’dnl’, ActionNameI(i,j), j)’)
’)’)dnl
             od
}


active proctype TR_Resp_PE() {
 progressR:  do
             :: atomic {
```

```
                    timeout ->
                    DLock_R=1 -> printf("MSC: DEADLOCK-R\n") -> break
                }
             :: atomic {
                    NO_INVOKE_R ->
                    printf("MSC: No-Invoke-R\n") -> break
                }
             :: atomic {
                    SUCCESS_R ->
                    printf("MSC: SUCCESSFUL-R\n") -> break
                }

m4_forloop(‘i’, 1, NumberOfStatesR,
 ‘m4_forloop(‘j’, 1, TableEntryR(i),
 ‘ifelse(substr(‘’ActionNameR(i,j),0,6),‘Action’, ‘‘’dnl’, ‘‘’format(‘
             :: %s_%d() ‘’dnl’, ActionNameR(i,j), j)’)
’)’)dnl
             od
}

ifelse(LOSSY, 1, ‘include(‘Lossy.m4’)’, ‘’) dnl

ifelse(MONITOR, 1, ‘include(‘Monitor.m4’)’, ‘’) dnl

 /*********************End of Code**********************/
```