Verifying Protocols by Model Checking: A Case Study of the Wireless Application Protocol and the Model Checker SPIN

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

Abstract

This paper deals with a formal verification of protocols, where the model checking approach is applied. As a case example, the Wireless Application Protocol (WAP Version 2.0) and the SPIN model checker are used. The paper concentrates on the Transaction Layer, in which some defects have been uncovered and proper corrections have been proposed.

1 Introduction

There is great advantage in being able to verify the correctness of computer systems. The most obvious is the case of safety-critical systems, but also applies to those that are commercially critical. Communication protocols belong to both categories. Formal verification methods have recently become usable by industry and the most popular method seems to be model checking [1]. The technique of model checking is intended to be used for concurrent systems, of which concurrency bugs are among the most difficult to be find by testing, since they tend to be non-reproducible.

The model checker SPIN [8, 9], along with its input modelling language PROMELA, has been successfully applied to analyze a range of protocols in different layers of the OSI Reference Model. For example, Stahl et al. [19] proved the safety of a sliding-window protocol, which is widely used in the data link layer; Obradovic [15] verified the Routing Information Protocol used in the network layer; and Josang [13] analyzed X.509 Authentication Protocol which operates in the presentation layer.

In many cases, however, modelling the protocol procedure looks more like programming with PROMELA (due to its closeness to ANSI-C) than formal reasoning. Moreover, verification of the service refinement is usually limited, partly because a formal description of the service definition is seldom made, especially when the protocol model is abstracted from an implementation (as for instance in [2]).

Gordon's verification [4] of the Wireless Transaction Protocol (WTP) is (to our knowledge) one of the few publications which adopts a formal protocol engineering methodology. He verified an old version of the WTP design [22] against some requirements like: faithful refinement of the service definition, successful termination and absence of livelocks. He modelled the system with Coloured Petri Nets (CPN), but his graphical model was restricted to have only lossless channels, which is not the general case in wireless applications. On the other hand, to obtain a correct verification result from a model with (for example) half a million states, it required almost 3.5 hours for a single verification run, which was time-consuming.

In this paper, we present our work of verifying a newly approved version of the WTP design [23]. We reformulate both the service definition and the protocol procedure with Finite State Automata (FSA), from which the
PROMELA models are derived. Using the SPIN tool, which supports Linear Temporal Logic (LTL) model checking, we specify correctness requirements with more flexibility. This enables us to find a deadlock error which was not reported in [4]. In addition, we are able to use considerably less runtime to verify the protocol model with bigger state spaces. The bigger state spaces result from the protocol configurations which are more complex and practical in the general protocol environment (e.g., which allows lossy channels).

The remainder of this paper is organized as follows: Section 2 gives a brief introduction of SPIN; Section 3 discusses the formalization of the WTP design, which provides for verification the (service and protocol) models and the requirement specifications; Section 4 describes the errors we found in the protocol design; Section 5 proposes some methodological issues in applying SPIN/PROMELA, and then shows the benefit of our method; Section 6 concludes with a summary and the future work.

This paper is a revised and extended version of [6].

2 Why to Choose SPIN/PROMELA

We first summarize some features of Holzmann’s SPIN/PROMELA system to show why it is chosen for our application.

PROMELA has been specifically designed to have the expressiveness of modelling a protocol precisely. A communication protocol is a distributed algorithm that co-ordinates two or more entities to exchange messages via (a)synchronous channels, where messages are defined with message format and their conditional sequences. Correspondingly, PROMELA supports three types of objects: processes, variables and channels. Independent processes are used to represent the distributed “entities”. Assignments in PROMELA update the variables; this evolution of system states is guarded by boolean expressions, which are employed to “co-ordinate” processes as “conditional sequences”. The actions to “exchange” messages are abbreviated as ? (receiving) and ! (sending) through channels, where a channel itself is an array of ordered sets of (message) variables. Moreover, the semantics of PROMELA preserve the concurrency and non-deterministic character of protocols.

Therefore, PROMELA is essentially a description language for extended Finite State Automata. By “extended”, we mean each state of FSA is a function mapping a set of variables to their ranges, while the variables consist of both the basic data types (e.g., booleans and integers) and the extended types (e.g., ordered sets of variables). We will elaborate the relationship between PROMELA and FSA in Section 3.2.

The SPIN tool facilitates simulation and verification of a model M specified in PROMELA. Random simulation runs provide the results of non-deterministic choices on the automaton’s transitions; they can be used to simulate the protocol execution. If an error is found in one verification run, a guided simulation plays back the error trail. This visualized counterexample helps to find the source of the error.

When used in verification, SPIN performs model checking on the problem M ⊨ ψ, where ψ is a property specified in Linear Temporal Logic. When ψ is translated into a Büchi automaton Aψ and the model M is represented as automaton AM, it is equivalent to check

\[ \mathcal{L}(A_M) \cap \overline{\mathcal{L}(A_\psi)} = \emptyset, \]

where \( \overline{\mathcal{L}(A)} \) is the complement of the language of automaton A.

Holzmann [8] implemented an algorithm of constructing both AM and Aψ on-the-fly to check the emptiness of their language intersection, so that during exploration of the reachable states, the successors of a state in Aψ are constructed only when they match the current states of the model AM. In another word, it avoids constructing (and storing) the entire state space of the synchronous product A¬ψ × AM if ψ is not satisfied in AM.

On the other hand, if no error is found after the entire state space of A¬ψ × AM has been explored, then M ⊨ ψ is verified completely. If a given memory space has been used up before an error can be found, SPIN’s bit-state-hashing search can provide a partial result saying M ⊨ ψ holds in a certain percentage of the actual state space. This state coverage can be
close to 1 on average, but its confidence level is measured by the hash factor (the ratio of the hash table size to the number of reached states).

Protocol development is a process of incrementally refining the service requirements until a target implementation is obtained. Applying formal methods throughout this process involves formalizing both the service definition and the refined protocol design, verifying that the protocol is a faithful refinement and (ideally) generating an implementation from the protocol model which has been verified against a set of correctness properties.

Our effort is to show 1) how to formalize the protocol design with both mathematical rigour and regular representations, since regularity is desirable for automating the process of system modelling; 2) the benefit of applying SPIN in verification. Therefore, as long as the computer platform has a C compiler (e.g., Gcc) installed, the SPIN/PROMELA system can be incorporated as part of the modelling and verification tool at the early stages of the protocol development process, while its verified product facilitates future implementation.

3 Formalization of the Wireless Transaction Protocol

This section briefly introduces the WTP design first. Then we present a formal model architecture using the finite state automaton (FSA) formalism, along with its equivalent in PROMELA syntax. This model architecture is used to formalize the WTP design at two levels of abstraction, namely the transaction service (TR-Service) and the transaction protocol (TR-Protocol). Finally, we specify a set of correctness properties to be verified.

3.1 Description of the Protocol Design

Detailed descriptions of the WTP design can be found in [23] and [4]. Here we summarize the design in line with the five elements of a protocol definition [7].

1. The assumptions about the protocol environment. In the layered WAP architecture, the Transaction Layer sits above the Transport Layer, in which a connectionless Wireless Datagram Protocol [21] is used. So, the WTP design assumes that messages exchanged between peer protocol entities (PEs) can be lost, re-ordered or duplicated.

2. The service to be provided by the protocol. The Class 2 transaction service defined in WTP intends to provide reliable transaction service for the users (TR-Users) in the upper Session Layer, by using a reliable invoke message (sent from the Initiator) with exactly one reliable result message (sent from the Responder). This TR-Service is defined abstractly by the service primitives and their sequences (called the service language).

Three kinds of primitives are of interest: Invoke, Result and Abort. Each can be affixed with a pair of primitive types (i.e., request req and indication ind, or response res and confirm cnf) to form a “submit-deliver pair”, e.g., Invoke.req and Invoke.ind.

The service language consists of both local and global primitive sequences. The WTP design, however, only defines the local sequences that are allowed on one TR-User side (see [22, Table 6]). For the global sequences of primitives exchanged between peer TR-Users, we adopt the end-to-end principle [10, 4], which requires the primitive types occur in the order (req, ind, res, cnf) and in a submit-deliver pair.

The primitive parameter Ack_Type is of special significance. If it is turned on, each indication primitive must be acknowledged with a response primitive, so it causes different primitive sequences to be generated.

3. The vocabulary of messages used to implement the protocol. It is specified as Protocol Data Units (PDUs) that are transmitted. We are interested in four PDU types: INVOKE, RESULT, ACK and ABORT, which correspond to the service primitives.
Table 1: A state table entry for the Responder in the TIDok_Wait state

<table>
<thead>
<tr>
<th>Event</th>
<th>Condition</th>
<th>Action</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>RcvAck</td>
<td>Class = 2</td>
<td>Generate TR-Invoke.ind</td>
<td>Invoke_Resp_Wait</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Start timer, A</td>
<td></td>
</tr>
</tbody>
</table>

4. The format of each message in the vocabulary, i.e., the encoding of PDU data. A PDU is structured as an integer number of octets, among which we focus on several bits in the header field that control the protocol operations. They include bit RID to indicate a re-transmitted PDU, bit U/P to indicate the parameter Ack_Type turned on, and bit TIDve/TIDok used for Transaction Identifier (TID) verification.

5. The procedure rules guarding the consistency of message exchanges. In the WTP design, the rules are specified with a set of state tables, which define how peer protocol entities change their states (referred to as PE_state) as a result of acting on incoming events. As an example shown in Table 1 [23, p. 55], each state table entry can be referred to as a tuple of (event, condition, action, next state). When an event occurs and its conditions are met, a certain set of actions is taken and the next state updated.

There are 10 state tables defined to realize the Class 2 transaction service. To be a faithful refinement of the TR-Service, the TR-Protocol should not enable any primitive sequences that may violate the predefined local sequences, or the end-to-end principle.

3.2 Formalization with FSA and PROMELA

In this section, we show how to incorporate the above five elements in our formalization.

First, our system modelling is based on the assumptions that we focus on a single transaction which involves only one pair of the Initiator and the Responder, and that the messages exchanged between the two sides are buffered in the asynchronous channels which allows re-ordering and loss. For the detailed justification of these assumptions, please refer to [5].

To generate the service language of the TR-Service and compare it with that of the TR-Protocol, we build a PROMELA model for each abstraction. Both models share a similar architecture as illustrated in Figure 1. The core of the model architecture consists of two concurrent processes: one on the Initiator side and the other on the Responder side, both of which function to generate the service language by communicating through two asynchronous channels (R2I and I2R) and the shared control variables. Accordingly, we give the FSA definition of each process as follows, serving as a formal basis; and it can be regularly translated to a proctype template in PROMELA.

Each process in Figure 1 is defined as an automaton $A_i = (S_i, s_0, \Sigma_i, \delta_i, F_i)$, where in the TR-Protocol model, $i = PI$ stands for the Initiator and $PR$ for the Responder. Each automaton has the following 5 components:

1. The set of states $S_i : V_i \rightarrow D_i$ maps the set of variables $V_i$ to the set of related ranges $D_i$. (This mapping is shown in Figure 1 as the open arrows inscribed with “w”, i.e., write.)

   $V_i = (Global_i, Local_i)$ is a tuple consisting of global variables and local ones. For example, the global variable set $Global_i = (channels, toggles)$ is a tuple, where

   - Channels comprise of two extended variables to model the asynchronous communication channels. Each channel is a tuple $(\text{chan\_name}, \text{chan\_size}, \text{message\_stored})$, i.e., an array of $\text{message\_stored}$ which has a maximum dimensionality of constant $\text{chan\_size}$ (the receiving buffer size). $\text{Message\_stored}$ is actually an ordered (finite) set of variables typed differently, $\text{type}_1 \times \text{type}_2 \times \ldots \times \text{type}_n$, with...
which can be used to model a PDU and its header field.

Figure 2 shows declaration of the I2R channel in PROMELA. The second line defines \(\text{chan.name} = \text{I2R}, \text{chan.size} = 4, \text{message.stored} = \text{mtype} \times \text{bit} \times \text{bit}\). The symbolic type declaration \(\text{mtype}\) enumerates the PDU types in the first line, while the next two bits can be used to model the control bits in the header field of a PDU.

\[
\begin{align*}
\text{mtype} &= \{\text{INVOKE, ACK, RESULT, ABORT}\}; \\
\text{chan.I2R} &= \{4\} \text{ of } (\text{mtype}, \text{bit}, \text{bit});
\end{align*}
\]

Figure 2: Declaration of the I2R channel in the TR-Protocol model

- **Toggles** = \((\text{Primitive.tgls}, \text{DLock})\) is a tuple, consisting of variables used to specify correctness requirements. \(\text{Primitive.tgls}\) are \text{bit} variables to record the occurrence of service primitive, e.g., 
  \(|\text{inv_{tgls}}| = 0\) when \text{Invoke.ind} is submitted. \(\text{DLock}\) is one bit to indicate deadlock in the process execution.

2. The initial state \(s_0\) defines the initialization of the variables in \(V_i\).

3. 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. (In Figure 1, open arrows inscribed with “r” represent reading the evaluation results.)

   Given the set of atomic propositions \(V = \{v = d \mid v \in V_i, d \in D_i\}\), the label function \(L : S_i \rightarrow 2^V\) defines a symbol \(a\) as a conjunction of the propositions. If any of the propositions is not true in state \(s\), the transitions guarded by \(a\) are disabled.

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

5. \(F_i\) consists of all the possible final states. \((\forall s_{F_i} \in F_i) s_{F_i}(\text{Local}_i) = s_{0_i}(\text{Local}_i)\), i.e., in a complete transaction, peer protocol entities finally return to their initial states after clearing all the state information (e.g., resetting counters and timers).

Then the basic operation of the TR-Protocol
Table 2: Sample entries in a Horizontal Condition Table

<table>
<thead>
<tr>
<th>entry</th>
<th>Guard</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>current $PE_{state}$ ∧ channel ∧ Local$<em>i$ ∧ Aflag ∧ next $PE</em>{state}$</td>
<td>$I2R - {ACK, RID, 1}$ $\rightarrow$ $I2R$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{Invoke_Resp_Wait}$</td>
</tr>
</tbody>
</table>

Note: 1. Variables not specified in the “Guard” column (e.g., RID in entry 1) can take any value within their respective ranges to make the guard true.
2. Variables not defined in the “Action” column remain unchanged.

can be modelled as an interleaving product automaton $A_M = A_{PI} \times A_{PR}$, where $A_M = (S_M, s_{0M}, \Sigma_M, \delta_M, F_M)$ is defined as follows:

- $S_M : V_{PI} \times V_{PR} \rightarrow D_{PI} \times D_{PR}$
- $s_{0M} = (s_{0PI}, s_{0PR})$
- $\Sigma_M = \Sigma_{PI} \cup \Sigma_{PR}$ is the union of each component’s guard alphabet.
- $\delta_M$ is defined by $(s'_i, s'_j) \in \delta_M((s_i, s_j), c)$ iff
  - $s'_i \in \delta_i(s_i, c)$ for each $i$ ($i = PI, PR$) such that $c \in \Sigma_i$ and
  - $s'_j = s_j$ for each $j$ such that $c \notin \Sigma_j$.

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

- $F_M = F_{PI} \times F_{PR}$ is the cartesian product of each component’s final states.

Before translating the above FSA definition into the PROMELA code, we create a set of Horizontal Condition Tables (HCTs), as an intermediate step, to specify the state transitions and their guards. The HCTs formalize the WTP state tables using the notations in set theory.

Table 2 shows an example of HCT that corresponds to Figure 1. The “Guard” column combines the “Event” and the “Condition” columns in the original state table. The guard defined in the first entry, for example, corresponds to the symbol

$$a = (PE_{state} = TIDok\_Wait) \wedge (ACK \in I2R) \wedge (TIDok = 1)$$

which is a conjunction of atomic propositions.

The term $(ACK \in I2R) \wedge (TIDok = 1)$ can be modelled in PROMELA by using the *channel poll* operation like $I2R??[ACK, RID, 1]$, which tests whether an $ACK$ PDU is received from the $I2R$ channel.

The “Action” column defines the new evaluations of variables, i.e., the successors of the current state. Variable updating includes:

- receiving messages, which is modelled as clearing messages from the incoming channel, e.g., $I2R - \{ACK\}$. As stated in the modelling assumptions, the messages in the channels can be out of order. We treat the re-ordering as that the PDUs can be taken out of the channel array at any position; this can be modelled by using PROMELA’s *random receive* operation, like $I2R??ACK(RID, 1)$.
- sending messages, modelled as adding messages into the outgoing channel, e.g., $I2R \cup \{INVOKE\}$. The PROMELA statement $I2R!INVOKE(1,0)$ just models re-transmitting an INVOKE PDU with $RID = 1$.
- submitting/delivering service primitives, modelled as incrementing the Primitive$_{tgls}$ variables, e.g., $iind\_tgl + 1$.
- assigning the next $PE_{state}$.  

6
There is no requirement of disjointness between the guards, so the same guard can lead to different actions. For completeness, the last “else” entry in the table says that the evaluations of variables other than those defined in the “Guard” column can make any symbol a true, while the current state remains unchanged. It is noted that the original WTP design lacks this completeness.

Each HCT entry, therefore, corresponds to a state transition, which actually defines a critical section for updating the shared variables (e.g., channels). To model a single transition, we use PROMELA’s atomic structure. But if any statement within one atomic sequence is blocked due an error (e.g., trying to send a new message to a full channel), PROMELA allows other atomic sequences to proceed. In this case, we cannot find which transition loses its atomicity. To ensure the exclusiveness of each critical section, we introduce a global control Aflag in each guard, as shown in Figure 3. It acts like a switch, opened at the entrance to the critical section and closed immediately before leaving. Thus in case one sequence loses its atomicity, other sequences cannot proceed.

```pascal
inline Entry_i ( ) {
  atomic {
    if (Guard == TRUE AND Aflag == 1) {
      Actions of updating variables;
      Aflag = 1;
    }
  }
}
```

Figure 3: Pseudo-code for a state transition corresponding to an HCT entry

The transition relations of the whole Initiator (or Responder) process can be implemented using a repetition construct with non-deterministic options of atomic sequences, as shown in Figure 4. The do-loop is broken under any one of the following conditions:

- Either the process reaches a state s which satisfies the condition \( L(s) = L(s_{F_i}) \), 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 submitted).
- Or the whole structure is blocked in an unexpected state \( s \notin F_i \) so that \( \forall a \in \Sigma_i \) \( \delta_i(s, a) = \emptyset \), which leads to a deadlock. Then we may use PROMELA’s predefined condition `timeout` as an escape from that system hanging state.

```pascal
1 proctype Initiator ( ) {
2 progress:
3 do :: Entry_1 ( ) // as defined in Figure 3
4 :: ... :: Entry_n ( )
5 :: atomic {
6   if (final state conditions == TRUE) {
7     Indicate the transaction completed;
8     break
9   }
10 }
11 :: atomic {
12   if (timeout) {
13     DLock = 1;
14     break
15   }
16 }
17 od
```

Figure 4: Pseudo-code for the Initiator process

In summary, to formalize a protocol design, we start with FSA as the most abstract model, then refine it into the PROMELA model through the following steps:

- The message vocabulary and its format, as well as the assumptions on channels, can be declared with a set of corresponding variables in PROMELA.
- The mapping of variables to their ranges in each FSM state is realized by assignments in PROMELA.
- Procedure rules of message exchanging, usually defined by a set of state tables in the protocol design, are abstracted with FSM alphabet and transition relations. But when the rules are formalized with HCT, each guarded transition can be translated into a PROMELA inline template as shown in Figure 3. Each protocol
entity with defined transition relations can be modelled with a POMELA proctype template as shown in Figure 4.

• The regularity shown in the structures of the above templates facilitates the automatic generation of a POMELA model from the tabular expression of a protocol design. In our case, we used GNU m4 [16], a macro processor, to generate those tens of inline transitions and the two communicating processes.

• The interleaving product of FSA formalizes the co-ordination in protocol execution, which is also implemented by the PROMELA semantic engine [18] that runs the protocol model.

Finally, as a side point, we show how to use a separate Stealing process to model the assumption of message loss with PROMELA. Figure 5 shows the structure of the Stealing process. It runs in parallel with the product automaton $A_M$ in an interleaved way. In any state $s \in S_M$, as long as either $I2R$ or $R2I$ channel is not empty, a PDU may be stolen by executing the random receive operation, e.g., $I2R??$, $R2I??$ (where ‘?’ is a PROMELA’s write-only variable used to store the lost PDU).

```proctype Stealing( ) { 
  do 
    I2R??, 
    R2I??, 
  od 
}
```

Figure 5: Promela code for the Stealing process

If the desired properties are proved with the existence of lossy channels and message re-ordering, we gain confidence that the WTP protocol is indeed designed to be able to recover from the errors and preserve the correctness requirements.

### 3.3 Specification of the Correctness Requirements

Holzmann [7] described some properties expected for a well-formed protocol design. With respect to the WTP design, we classify these properties into three categories: safety properties $P_S$, liveness properties $P_L$ and temporal behaviours $P_T$. Here we give the formal specifications of some safety and temporal properties as follows:

#### 3.3.1 Safety Properties.

A safety property generally says “nothing bad happens”. Formally, given the protocol automaton $A_M$, $(\forall s \in S_M)(\forall v \in V_M) s(v) \in D_M$, which means each variable cannot overflow its range. For example, given a local retransmission counter $PE.data.RCR$ with maximum $= RCR_{MAX}$, we have a safety property as $p_{S_1} = (PE.data.RCR \leq RCR_{MAX})$.

(2)

• Pure atomicity, i.e., updating of the shared channels/variables must be mutually exclusive. Referring to Figure 3, the global control $Aflag$ should always satisfy the safety property $p_{S_2} = (Aflag == 1)$.

(3)

2. For the progress of execution, safety requires:

• The protocol be bounded. Given $S_M : V_M \rightarrow D_M$, then $(\forall v \in V_M) (\forall s \in S_M) s(v) \in D_M$, which means each variable cannot overflow its range. For example, given a local retransmission counter $PE.data.RCR$ with maximum $= RCR_{MAX}$, we have a safety property as $p_{S_1} = (PE.data.RCR \leq RCR_{MAX})$.

(2)

• Pure atomicity, i.e., updating of the shared channels/variables must be mutually exclusive. Referring to Figure 3, the global control $Aflag$ should always satisfy the safety property $p_{S_2} = (Aflag == 1)$.

(3)

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

(4)
This property is guaranteed by the definition we have made that if a PDU is buffered in the incoming channel, but not used in any guard conditions, then it is ignored and the current state remains unchanged. This has been implicitly stated in the “else” entry of our HCTs (Table 2).

Actually, each \( p_S \) defined above is an invariant of the model \( A_M \), which can be expressed in an LTL formula as \( A_M \models \varphi_{p_S} \), where \( i = 1..3 \) and \( \varphi \) is the always operators. It is equivalent (and in some cases, more efficient) to verify \( A_M \models \neg \varphi_{\neg p_S} \). So instead of checking each formula \( \varphi_{p_S} \) separately, we can define a monitor automaton \( A_m \) (i.e., the property automaton shown in Figure 1), which runs interleavingly with the model \( A_M \) and reports any violations of these invariants. As described in Section 2, any violation of an invariant is found before the whole state space of the product \( A_m \times A_M \) has been constructed.

Assuming \( V_m = \{ \text{PE}_r, \text{data}, \text{RCR}, \text{Aflag}, \text{DLock} \} \) and \( D_m = \{ d_m | d_m = S_M(v_m) \} \), \( A_m = (S_m, s_0_m, \Sigma_m, \delta_m, F_m) \) is defined as follows:

- \( S_m = S'_m \times \{ \text{good}, \text{bad} \} \), where \( S'_m : V_m \rightarrow D_m \).
- \( s_0_m = s_0_m \times \{ \text{good} \} \).
- \( \Sigma_m = \{ \neg p_S, i = 1..3 \} \).
- \( \delta_m(s_m, a) = s_{F_m} \), where \( a \in \Sigma_m \) and \( s_{F_m} \in F_m \).
- \( F_m = S'_m \times \{ \text{bad} \} \), i.e., each final state is attached with a sign \( \text{bad} \), but the current evaluations of variables are not changed.

The above property automaton \( A_m \) can be implemented as a PROMELA process, shown in Figure 6. A PROMELA statement \( \text{assert}(p_S) \) can be used to report whether the assertion \( p_S \) is violated and then to locate a bad state. In the guarded sequence, the \( \text{assert}(p_S) \) statement is only executable when \( p_S \) is not true. This is efficient because \( p_S \) is expected true in most states.

### Figure 6: Pseudo-code for the Monitor process

```plaintext
1 proc
2   types Monitor( ) {
3     atomic {
4       if (p_{\neg S} == FALSE) { assert(p_S) }
5       ... 
6       if (p_S == FALSE) { assert(p_{\neg S}) }
7     fi
8   }
9 }
```

### 3.3.2 Temporal Behaviours.

Temporal behaviours mainly deal with the ordering of the service primitives. They are also used to verify whether the service language generated by the TR-Protocol is a subset of that of the TR-Service, so that the TR-Protocol is a faithful refinement of the TR-Service.

Here we give one of the seven temporal requirements that are to be verified. The end-to-end behaviour in the global sequence can be expressed as a precedence pattern in the global scope as defined in Dwyer’s Specification Pattern System [3]. Using the strong until operator \( \mathcal{U} \) and the future operator \( \mathcal{F} \), or equivalently using the weak until operator \( \mathcal{W} \) only, the following formula

\[
\text{pr}_1 = \varphi((\mathcal{F}\text{ICNF}_{\text{tgl}}) \Rightarrow \neg(\text{ICNF}_{\text{tgl}} \cup \text{ires}_{\text{tgl}})))
\]

\[
= \varphi(\neg(\text{ICNF}_{\text{tgl}} \mathcal{W} \text{ires}_{\text{tgl}})).
\]

specifies that the occurrence of the Invoke.res primitive (toggle bit \( \text{ires}_{\text{tgl}} = 1 \)) is a necessary pre-condition for an occurrence of the Invoke.cnf primitive (bit \( \text{ICNF}_{\text{tgl}} = 1 \)).

### 4 Summary of the Verification Results

In this section, we present the results of verifying the PROMELA models of both the TR-Service and the TR-Protocol against the properties specified in Section 3.3.

The TR-Service model is firstly proved to satisfy all those properties. This means the control flow defined within the Initiator and
the Responder processes, as well as other model structures, can be safely extended for use in the TR-Protocol model. Afterwards, it is reasonable for us to attribute the errors found in the TR-Protocol model to the defects in the original WTP design.

When verifying the TR-Protocol model, a deadlock error is revealed, as illustrated in Figure 7. The figure is a Message Sequence Chart (MSC) presentation of the two deadlock trails provided by SPIN’s guided simulation.

In Trail 1, the Responder receives the first INVOKE PDU with an invalid TID, so it sends the first ACK PDU for TID verification. But for some reason (e.g., the channel’s condition being poor), the Initiator does not get the ACK and sends the second INVOKE PDU when the re-transmission interval (R) expires. This transaction keeps on until the re-transmission counter reaches its maximum (in this case, $RCR_{MAX}\_Init = 2$), and the Initiator aborts the transaction by sending an Abort.ind primitive (AINDP). This leaves the Responder blocked in the $TIDok\_Wait$ (TW) state, since there is no new event to enable it to update its $PE\_state$.

It is noted that this deadlock scenario was not uncovered in Gordon’s verification [4], because a transition $aindp\_TSP$ (as shown in a dashed box in Trail 1) was enabled unconditionally in every Responder’s $PE\_state$. This transition was meant to model a possible abort initiated by the Transport-Service-Provider (TSP) in case of errors occurring in the lower protocol layers. But if improperly used, it could become an arbitrary exit from a hanging state in the protocol execution, even though no actual TSP errors have occurred, while the actual deadlock error cannot be revealed. Of course, that is not a correct way of modelling.

Tail 2 shows another deadlock scenario, in which the Responder receives the first INVOKE PDU with a valid TID, but finally aborts the transaction after it re-transmits the RESULT PDU for the maximum times (say, $RCR_{MAX}\_Resp = 1$). However, the Initiator sends the second INVOKE PDU when the time interval R expires before the first RESULT reaches. This delayed (and re-transmitted) INVOKE PDU triggers the Responder to request a TID verification, which again traps the Responder in the $TIDok\_Wait$ state, since no proper transition has been defined in this case for update (e.g., how to act on receiving the final ACK PDU from the Initiator).

Figure 7: Deadlock trails of the TR-Protocol
As can be seen, both deadlock scenarios are possible as long as the re-transmission interval \( R \) is shorter than the PDU round-trip-time (RTT, the time spent before receiving the response to the last PDU sent). This is a little more strict than the normal situation, when \( R \) is usually set to a fixed value that is longer than the average RTT. But the average RTT can only be obtained through statistics and it varies a lot especially in wireless networks, so a fixed \( R \) cannot guarantee to eliminate the deadlock possibility.

To correct this error, we suggest modifying the TR-Protocol design as follows: (1) the Responder start the timer for an interval \( W \), whenever it requests a TID verification in the TIDok.Wait state; (2) a new transition (shown in a dashed box in Trail 2) enables the Responder to return to its initial Listen (LIS) state when the interval expires, as does for the Initiator to return to its initial NUL state; (3) the time interval \( W \) is set long enough so that any ACK PDU sent from the Initiator (to respond to the TID verification) can reach the Responder side within the average response time.

After the above correction, there are still several errors found, which indicate that the TR-Protocol design is not a faithful refinement of the TR-Service definition. They include:

- an erroneous restart of a primitive sequence, which may cause the channel buffers overflow and violates Formula (3) described in Section 3.3;

- the erroneous end-to-end behaviours, which violate property \( p_{T1} \) (Formula (5)) and the property of Result.res preceding Result.cnf;

- an erroneous primitive ordering when \( Ack_Type = 1 \), violating the property of Invoke.res preceding Result.req.

These errors were reported in Gordon’s verification [4] of an older version of the WTP design, but they still exist in the latest approved version [23].

After we have modified the TR-Protocol state tables (see [5] for details), we verify, in SPIN’s full-state-space-search mode, that our PROMELA model of the TR-Protocol preserve all the properties specified in Section 3.3, even when messages may get lost or reordered in the lower layer service that includes a GSM-Short-Message-Service bearer. So the revised TR-Protocol is considered to be well-formed.

5 Discussion

In this section, we first summarize some methodological issues in applying SPIN for protocol verification. Then we show the benefit of our application by comparing it with the Coloured Petri Net Method.

5.1 Methodological Issues

In our application of SPIN/PROMELA, we have considered some methodological issues as follows:

Firstly, an incremental approach is desirable in the protocol analysis, because

- A protocol design normally has two levels of abstraction, the service and its refined protocol. It allows us to build models for these two abstraction with a similar model structure. We start with the service model, which is more abstract but simple and easy to fix modelling errors. When the control flow adopted has passes the sanity check and the correctness requirements have been verified against the service model, it is safe to extend the model structure for use in the refined protocol model.

- A protocol design is complex with different features and modelling assumptions, but they can be added into the model bit by bit. For example, we can disable the “Transaction Abort” feature until we have investigated the basic “Message Transfer” function with fewer state transactions. Similarly, we can first disable the lossy property of channels and begin with the model of a smaller state space. Both can be done by using PROMELA’s preprocessor function (like in C).

- To verify the protocol design with different configurations, we need to parameterize the PROMELA model. As for
the WTP design, the state space of its TR-Protocol model grows when the maximum of the local re-transmission counter ($RCR_{MAX\text{ Init}}$ or $RCR_{MAX\text{ Resp}}$) increases. We prefer to start with a small value of both $RCR_{MAX}$s when verifying the model against all the modelling assumptions and most protocol features. Then we increment them to the configurations for WTP practically used in a GSM SMS network ($RCR_{MAX}$s = 4) or an IP network ($RCR_{MAX}$s = 8). To generate the PROMELA model with different configurations, we use GNU m4 to parameterize the variables $RCR_{MAX\text{ Init}}$ and $RCR_{MAX\text{ Resp}}$.

Secondly, when applying the model checker SPIN, we aim to perform a full-state-space search (if applicable) with fewer memory resources, or gain more confidence in the average coverage of a partial search using bit-state hashing. This requires the model have fewer number of states and a small state vector (the number of bits required to distinguish each state for state comparison). Here we list some recommendations of efficient modelling [20] we have adopted to cope with the problem size of our application.

- Integer variables whose values are always smaller than 16 should be declared as unsigned variables, rather than int. This can result in a smaller state vector.
- Use local variables for those that are only used within a single process, because the local scope of local variables gives SPIN much more opportunities to remove states. Variables used in LTL formulas have to be global, but they can be declared with the local designator if used locally.
- A process that may terminate should not be created last (as the youngest process) in a PROMELA model, otherwise its -end-transition will always be executable when the process is the youngest to disappear from the system in a stack-order, which can result in a bigger state space. In our TR-Protocol model, the Stealing process is the second to the last.

5.2 Comparison with the Coloured Petri Net Method

We make a comparison between Gordon’s verification and ours, showing the benefit we get from using SPIN, with respect to the efficiency in the memory space and time consumption.

Gordon uses Coloured Petri Nets to model both the TR-Service and the TR-Protocol, uses the Design/CPN tool (version 4.0.5) to check the absence of deadlocks, livelocks and dead transitions, and uses AT&T’s FSM tool (version 3.6) to compare the service languages between the TR-Service and the TR-Protocol. The verification is run on a computer equipped with 512MB RAM and a 366MHz CPU. Using the sweep-line method, the maximum configuration of the model being checked is (denoted as) $Config. F-4-4-0$, where $F$ stands for $Ack\text{ Type} = 0$, 4-4 stands for $RCR_{MAX\text{ Init}} = RCR_{MAX\text{ Resp}} = 4$, and the last 0 stands for lossless channels. Correspondingly, its state space has 513,653 states.

We use Finite State Automata and its equivalent PROMELA language to model the WTP design, use SPIN (version 3.5.3) to check a set of safety, liveness properties and temporal behaviours. Our verification is run on a computer equipped with 256MB RAM and a 864MHz CPU. Using SPIN’s full state space checking mode, the maximum configuration checked is $Config. F-6-6-0$ with 2,997,940 states; while using the bit-hashing mode, the model with $Config. F-8-8-1$ (used in an IP network that allows lossy channels) is verified with a 98% coverage of the actual state space, i.e., having searched 30,755,600 states.

Table 3 presents our trials on some protocol configurations, where we limit the size of the hash table to 134.22 Megabytes (i.e., 128MB or $2^{30}$ bits). We can see SPIN’s state coding techniques are efficient: they enable us to use fewer memory space (128MB v.s. 512MB) to store much more states ($3 \times 10^7$ v.s. $5 \times 10^5$) for the model with complex configurations.

Table 3 also shows the extra computation time that is needed for treating the problem of message loss (while proving the safety proper-
Table 3: Verification results of the safety properties for the revised TR-Protocol model

<table>
<thead>
<tr>
<th>Config.</th>
<th>State-vector (bytes)</th>
<th>No. of States stored</th>
<th>Memory usage for states (Megabytes)</th>
<th>Compression ratio / Hash factor</th>
<th>User time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-5-5-0</td>
<td>152</td>
<td>1,058,540</td>
<td>59.65</td>
<td>34.36% / -</td>
<td>28.36</td>
</tr>
<tr>
<td>F-6-6-0</td>
<td>168</td>
<td>2,997,940</td>
<td>173.35</td>
<td>32.12% / -</td>
<td>90.16</td>
</tr>
<tr>
<td>F-8-8-0</td>
<td>200</td>
<td>18,458,600</td>
<td>134.22</td>
<td>- / 29.08</td>
<td>627.84</td>
</tr>
<tr>
<td>F-5-5-1</td>
<td>156</td>
<td>1,937,380</td>
<td>120.90</td>
<td>37.15% / -</td>
<td>114.44</td>
</tr>
<tr>
<td>F-6-6-1</td>
<td>172</td>
<td>5,280,140</td>
<td>134.22</td>
<td>- / 101.68</td>
<td>329.99</td>
</tr>
<tr>
<td>F-8-8-1</td>
<td>204</td>
<td>30,755,600</td>
<td>134.22</td>
<td>- / 17.46</td>
<td>2208.37</td>
</tr>
</tbody>
</table>

ties). By comparing the “User times” between Config. F-5-5-0 and Config. F-5-5-1, and other pairs of configurations likewise, we can see the time needed for one verification run of a “lossy channel” configuration is about 3.5 to 4 times that for a “lossless” one.

Table 4 compares the time consumed in verifying the same deadlock-free property of the TR-Protocol model with different configurations. We can see again SPIN significantly outperforms Design/CPN. As the state space grows, the time consumed in verifying the CPN model increases from about 100 times to 880 times that spent on our PROMELA model.

Table 4: Comparison of time consumption for verification of safety (time in seconds)

<table>
<thead>
<tr>
<th>Config.</th>
<th>CPN Time</th>
<th>Adjusted CPN Time</th>
<th>PROMELA Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-4-0-0</td>
<td>317</td>
<td>134</td>
<td>1.26</td>
</tr>
<tr>
<td>F-4-1-0</td>
<td>1799</td>
<td>762</td>
<td>2.83</td>
</tr>
<tr>
<td>F-4-2-0</td>
<td>6004</td>
<td>2544</td>
<td>5.03</td>
</tr>
<tr>
<td>F-4-3-0</td>
<td>10242</td>
<td>6881</td>
<td>7.79</td>
</tr>
<tr>
<td>F-4-4-0</td>
<td>12289</td>
<td>5296</td>
<td>11.35</td>
</tr>
</tbody>
</table>

Note: 1. Every CPN Time is scaled down to an Adjusted one by a factor of 2.36, which equals the CPU clock ratio (864/366).
2. The CPN time of Config. F-4-4-0 results from the sweep-line method, so it is less than that of Config. F-4-3-0.

To estimate SPIN’s checking time of all the properties (including the temporal behaviours), one has to sum up all the time spent on every single property, since SPIN builds a new state space for every verification run. Even that, the total run time of a PROMELA model is still much less than that of a CPN model for the same configuration. For example to check Config. F-4-4-0, the total verification time for a PROMELA model is about 270 seconds, 1/20 of that (5206 seconds) for a CPN model.

As the last comment with respect to the problem size, one may be more impressed by the model checker SMV (or its extension NuSMV), which claims to be able to explore a state space having as many as $10^{20}$ states [1]. Unlike SPIN, however, SMV has been designed for verifying synchronous systems (e.g., logic circuits) and has no explicit support for asynchronous communication channels. From the experience gained in [2], 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 memory space) and costs significantly more verification time, partly because extra maneuvers (consequently, more language components) are needed when modelling with SMV’s input language.

6 Conclusions

Through our work, we exemplify a procedure of formalizing and verifying a real-world protocol design, the Class 2 Service and Protocol design of WTP Version 2.0. Starting from building the mathematical model with Finite State Automata (FSA), we then establish a relationship between the FSA formalism and SPIN’s modelling language PROMELA, by illustrating how PROMELA can be applied to formalize a protocol design into a regular format, instead of
being used like a programming language in an ad-hoc way. The regularity also enables the future development of automatically generating a PROMELA model from a protocol design in tabular representations (e.g., Horizontal Condition Tables), so that the PROMELA model can inherit the mathematical rigour of tabular expressions [12].

During the verification of the TR-Protocol model, we uncover new deadlock scenarios which were not reported in previous literatures, as well as all those erroneous primitive sequences reported but yet uncorrected. Since these errors are found using two formalisms with different syntaxes and semantics, we are confident that our PROMELA models are correct. SPIN’s superior model checking performance also enables us to verify the correctness of a revised TR-Protocol with more complex configurations, which can be used in GSM SMS or IP networks with lossy communication channels.

As a future work inspired by some early attempts in [14], we expect that after a PROMELA model has been verified to be error free, the protocol implementation could be generated from the “verifier”, which has been a set of ANSI-C files translated from the PROMELA model by SPIN and already been used to simulate the protocol execution as a running prototype.

The model of concurrency used in this paper is principle an interleaving model with Linear Temporal Logic. An extension to more advance models of concurrency [11] is an open and non-trivial problem.

Acknowledgements

Drs. Steven Gordon and Gerard Holzmann are thanked for their help in understanding the subtlety of WAP and SPIN, respectively. The authors shall thank the anonymous referees for their precious comments as well.

About the Authors

Yu-Tong He is currently a Ph.D. candidate in the Department of Computing and Software at McMaster University. He served as a design engineer in China Telecom for 4 years, responsible for GSM/CDMA network projects. He holds a B.Eng. (Hon.) in Electronic Engineering and Management from Shanghai Jiao Tong University and an M.S. in Computer Science from McMaster University. His research interests include Markov Decision Processes and applications of formal methods in Software Engineering. His email address is hey3@mcmaster.ca.

Ryszard Janicki is a Professor in the Department of Computing and Software at McMaster University and a member of the Editorial Board of Fundamenta Informatiae. He worked at Warsaw University of Technology, Poland, Aalborg University, Denmark, and had visiting posts in University of Newcastle Upon Tyne, U.K. and Bordeaux University, France. His research interests include concurrency theory, fundamentals of software engineering, mereology and ranking theory. His email address is janicki@cas.mcmaster.ca.

References


