# Wildfire Verification Challenge

Final Report

CS4271, AY2004-2005, Semester 2

Li Jia (U018225X), Chen Yan (U018221B)
{lijia1,chenyan1}@comp.nus.edu.sg

April 11, 2005

# Contents

# Chapter 1

# Introduction

This report documents our effort and positive results obtained in tackling the Wildfire Verification Challenge Problem posed by [Lamport et al., 2001] [1]. The problem contains a simplified specification of the cache-coherence protocol used in Wildfire, code name for the Compaq AlphaServer GS series, a family of multiprocessor computers containing up to 32 Alpha processors. A bug has been deliberately inserted into this specification, and the goal is to locate the bug via formal verification methods.

The problem has already been solved by Georges Gonthier. In addition to the error the authors had planted, he found another bug that the authors were not aware of - a bug in the specification, not in the actual protocol. Later Abdelwaheb Ayari found that other bug, but not the one that was deliberately inserted. A group of students from Utah also found the unintentional bug. These are the results reported so far at this point of time writing this report.

The specification is written in TLA$^+$ , a language for writing TLA, or Temporal Logic of Actions, specifications. Both TLA and TLA$^+$ were invented by Leslie

[1] The description of the problem is currently hosted at
`http://research.microsoft.com/users/lamport/tla/wildfire-challenge.html`

Lamport, who also published a book [Lamport, 2002] on them, which formed the primary learning material for our project. Though functionally similar to other specification languages such as SMV, TLA and TLA$^+$ are different from the rest because it is a robust language for writing mathematics rather than one that resembles a computer programming language.

In this project, the Wildfire specification is checked by TLC version 2.0, a model checker for TLA$^+$ , primarily designed by Yuan Yu. It is freely available from `http://research.microsoft.com/users/lamport/tla/tools.html`.

We did the bulk of our project on the Tembusu-Linux cluster in the School of Computing, which each node PC having dual Pentium-4 2.8GHz CPUs and 2.5GB RAM, running on Fedora Core 2.

The rest of the report is organised as follows. Chapter 2 describes briefly the structure of a TLA specification in general, and that of the Wildfire specification files in particular. Chapter 3 reports our steps in verifying the system. Chapter 4 presents and discusses our findings. Chapter 5 concludes the report.

*Note: If you have read our Mid-term report, the significant changes we have made since then are mainly in Section 4.2 Bug 2. Our previously proposed solution to Bug 2 did not solve the problem and we managed to find a new solution that worked.*

# Chapter 2

# Understanding The Specification

## 2.1 Structure of an TLA specification

TLA allows specification of a complex system to be written in several smaller, self-contained MODULE . Modules can reference each other by using the EXTENDS keyword. Usually, there is a main module which EXTENDS other modules, and in which the initial state, the next state, and the liveness formulae are defined to form the top-most level specification. In TLA, the top-most level specification has the form:

$$Spec \triangleq \wedge Init \wedge \Box[Next]_{variable} \wedge Liveness$$

In the above formula, the symbol $\Box$ is equivalent to the $AG$ operator in CTL$^+$, i.e. $\Box P$ asserts that $P$ is true for every step in the behaviour. The symbols [ and $]_{variable}$ around $Next$ means that the next state of the system allows a *stuttering step*, in which *variable* remains unchanged.

Usually, a complex TLA specification such as one for a cache-coherence protocol defines many CONSTANTS . Quite contrary to its name, a CONSTANT 's value is left unspecified in the specification until model-checking time. For instance, a TLA specification for an cache-coherence protocol may define a CONSTANT $Procs$,

denoting the set of CPUs used in the system. It may also define a CONSTANT *DataLen* denoting the size of a data word addressable in the memory or cache. These CONSTANTS are only given values in the $configurationfile$, which is fed into the TLC model checker together with the main TLA module of the specification.

TLA$^+$ uses a similar syntax to that of LaTeX with added syntax (i.e. shortcuts to produce TLA symbols), so the TLA specifications can be typesetted into PostScript files by using the accompanied tool called `tlatex`. During model checking using the TLC model checker, only the textual format of the specification is used, but in this report, we will use the typesetted format to present TLA expressions and formulae.

## 2.2 The Wildfire specification

### 2.2.1 Files

The Wildfire specification is made up of several smaller TLA modules :

**Alpha.tla** This is the complete specification of the Alpha memory module.

**AlphaConstants.tla** This module declares the constant parameters of the specification and defines some useful constants and constant operators.

**AlphaInterface.tla** This module describes the interface between the Alpha Memory System and its environment, which is a collection of processors. The environment interacts with the system by sending requests to which the system responds.

**InnerAlpha.tla** This module defines the "internal" specification of the Alpha memory model–that is, the specification with internal variables visible.

| Constant | Description |
|---|---|
| $Proc$ | Set of CPUs used in the system |
| $Adr$ | Set of all memory addresses used in the system |
| $LS$ | Set of Local Switches used in the system |
| $DataLen$ | The size of a data word addressable in the memory |
| $AdrLS$ | The function that maps a given memory address to a Local Switch. This function is needed because the logical memory space of Wildfire is physically distributed into smaller memory modules at each Local Switch |
| $ProcLS$ | The function that maps a given CPU to a Local Switch |
| $InitMem$ | The initial content of the memory |
| $ResponseToEnv$ | |
| $ResponseFromEnv$ | |

Table 2.1: Constants in the Wildfire specification

**Wildfire.tla** This is the main module that references other modules to form the entire specification of the system. The initial state, next state and liveness formulae are defined in this module.

### 2.2.2 Constants

The list of CONSTANTS defined in the Wildfire specification is in Table 2.1.

As mentioned earlier that CONSTANTS have to be assigned value at model-checking time, we have given the above CONSTANTS the following values according to their definitions.

$$Proc \triangleq \{p1, p2\} \tag{2.1}$$

$$Adr \triangleq \{1, 2, 3, 4\} \tag{2.2}$$

$$LS \triangleq \{l1, l2\} \tag{2.3}$$

$$DataLen \triangleq 2 \tag{2.4}$$

$$AdrLS(a) \triangleq \text{CASE } a\%2 = 0 \rightarrow l1 \ \square \ a\%2 = 1 \rightarrow l2 \tag{2.5}$$

$$ProcLS(l) \triangleq \text{CASE } l = l1 \rightarrow p1 \ \square \ l = l2 \rightarrow p2 \tag{2.6}$$

$$ZeroData \triangleq [n \in 0..(DataLen - 1) \mapsto 0] \tag{2.7}$$

$$InitMem \triangleq [n \in Adr \mapsto ZeroData] \tag{2.8}$$

Equation 2.1 to Equation 2.4 are easy to understand.

Equation 2.5 uses a CASE construct to say that if the memory address is 0 *modulus* 2, then it belongs to the memory module at $l1$, otherwise it belongs to $l2$.

Equation 2.6 says that if the Local Switch is $l1$, then the CPU attached to it is $p1$; if the case of $l2$, the CPU attached to it is $p2$.

Equation 2.8 says that the intial content of the memory is such that each memory address maps to a *ZeroData*, which is a zero-filled word of length *DataLen*.

The actual assignment of CONSTANT values take place in the *config* file that goes into the TLC model checker, which will be covered in the next chapter.

# Chapter 3

# Model Checking using TLC

## 3.1 Reducing model checking complexity

### 3.1.1 Constraints

The TLC model checker expects two input files, one TLA$^+$ file with extension `.tla` and one *config* file which only differ from the former in its `.cfg` extension. The actual assignment of values to CONSTANTS takes place in the `.cfg` file, which also states the name of the specification and properties to be checked by the model checker. The `.tla` file can be the main module of the specification, but it is usually a good practice to define a new module which EXTENDS the main module, since there are several other things to be defined and it would be a good idea to leave the main module unchanged for modularity's sake.

One important thing to be added in the `.tla` file is the *constraint* formula, which is used to enforce a bound in the VARIABLES defined in the specification. For instance, the communication channel between a CPU and its Local Switch is modeled using a queue data structure. During model checking, random values will be added into and removed from the queue and the queue might expand and shrink

randomly. Without placing a bound on the length of the queue, the number of unique states will be infinite because the length of the queue can become infinitely large.

We defined our constraints for the Wildfire problem to be the following:

$$
\begin{aligned}
&SeqConstraint \triangleq \\
&\land Cardinality(fillQ[\text{CHOOSE } x \in Proc : x \in Proc]) \leq QLen \\
&\land Len(Q.ProcToLS[\text{CHOOSE } x \in Proc : x \in Proc]) \leq QLen \\
&\land Len(Q.LSToProc[\text{CHOOSE } x \in Proc : x \in Proc]) \leq QLen \\
&\land Len(Q.LSToGS[\text{CHOOSE } x \in LS : x \in LS]) \leq QLen \\
&\land Len(Q.GSToLS[\text{CHOOSE } x \in LS : x \in LS]) \leq QLen \\
&\land Len(reqQ[\text{CHOOSE } x \in Proc : x \in Proc]) \leq QLen \\
&\land Len(respQ[\text{CHOOSE } x \in Proc : x \in Proc]) \leq QLen
\end{aligned}
$$

In the above code segment, $QLen$ is a CONSTANT we defined, and later specified in the `.cfg` file. This way we can easily change the constraint by assigning different values to $QLen$.

### 3.1.2 Assigning small-sized values

In Wildfire, the processors maintain unbounded queues of unfulfilled requests, so the state spaces in infinite. Moreover, the model is so complex that the number of reachable states in the Wildfire problem grows very fast as a function of the number of requests [Lamport et al., 2001]. As such, our strategy is to start small and gradually increase the size of the problem.

As a start, we gave the CONSTANTS in Table 2.1 the following values.

$$Proc \triangleq \{p1, p2\} \tag{3.1}$$

$$Adr \triangleq \{1, 2\} \tag{3.2}$$

$$LS \triangleq \{l1, l2\} \tag{3.3}$$

$$DataLen \triangleq 1 \tag{3.4}$$

$$AdrLS(a) \triangleq \text{CASE } a\%2 = 0 \rightarrow l1 \ \square \ a\%2 = 1 \rightarrow l2 \tag{3.5}$$

$$ProcLS(l) \triangleq \text{CASE } l = l1 \rightarrow p1 \ \square \ l = l2 \rightarrow p2 \tag{3.6}$$

$$ZeroData \triangleq [n \in 0..(DataLen - 1) \mapsto 0] \tag{3.7}$$

$$InitMem \triangleq [n \in Adr \mapsto ZeroData] \tag{3.8}$$

## 3.2 Simulation mode

Besides running the normal model checking mode by exhaustively enumerating all possible reachable states at each time step, the TLC model checker also support *simulation mode*, in which the checker randomly computes a seed value, and picks the path indicated by the seed. This way, the checker "digs" deep into the state trace instead of spanning all reachable states of a single time step before moving on to the next time step. We found this feature can be very useful in checking Wildfire since we cannot afford the computational complexity to exhaustively check for errors that might only occur at the $n^{th}$ time step when $n$ is big. In fact, we found a bug at the time step where $n = 18$. Details of the bug will be discussed in the next chapter.

# Chapter 4

# Results and Analysis

Thus far, we have discovered two bugs by checking the provided Next Step and Liveness constraints provided by the supplied Wildfire TLA$^+$ model. We did not write any additional TLA$^+$ formulae for checking.

## 4.1 Bug 1

As of $3^{rd}$ of March, we believe we have found the unintentional bug. The bug was at Line 1722 of the `Wildfire.tla` file. The setup of the experiment when the bug was spotted is as follows:

1. TLC model checker was running in simulation mode
2. A configuration with 1-bit data words, 2 local switches each containing 1 processor and 1 memory address, and a fixed initial memory state.

### 4.1.1 Reported Error

We first list the error reported by the TLC model checker as well as the error trace. Since the trace is very long, we only report relevant part of the the last state (i.e.

State 18).

```
... Attempted to select nonexistent field "adr" from the record
[type |-> "MB"] line 1722, col 41 to line 1722, col 54 of module
Wildfire The behavior up to this point is: STATE 1: ... ... STATE
18:
 1 /\ reqQ = ( P1 :> << [type |-> "LL", adr |-> 2] >> @@
 2   P2 :>
 3       << [data |-> (0 :> 0), type |-> "Wr", adr |-> 1, mask |-> (0 :> 1)],
 4          [type |-> "LL", adr |-> 2],
 5          [data |-> (0 :> 1), type |-> "Wr", adr |-> 2, mask |-> (0 :> 0)],
 6          [type |-> "MB"],
 7          [data |-> (0 :> 0), type |-> "Wr", adr |-> 2, mask |-> (0 :> 1)],
 8          [type |-> "MB"],
 9          [type |-> "Rd", adr |-> 1],
10          [data |-> (0 :> 1), type |-> "Wr", adr |-> 2, mask |->
(0 :> 1)], 11          [data |-> (0 :> 1), type |-> "SC", adr |->
1, mask |-> (0 :> 1)], 12          [data |-> (0 :> 1), type |->
"SC", adr |-> 2, mask |-> (0 :> 1)] >> ) The number of states
generated: 6751845 Simulation using seed 7420508466826850145 and
aril 21720144
```

### 4.1.2 Source Code

We next list the source code of the procedure in which the offending line resides.

```
1704 ProcExecuteFromCache(p, idx) ==
1705   (*************************************************************************)
1706   (* Processor p executes the idx-th request in its request queue directly *)
1707   (* from the cache--that is, without having to issue a directory          *)
1708   (* operation.  For a read or write, this means that the data are already *)
1709   (* in its cache.                                                         *)
1710   (*************************************************************************)
1711   LET req == reqQ[p][idx]
1712   IN  /\ \/ /\ req.type = "MB"
1713                (*********************************************************)
1714                (* Case 1: the request is a MB.                          *)
1715                (*********************************************************)
1716             /\ \A i \in 1..(idx-1) :
1717                  /\ reqQ[p][i].type # "MB"
1718                     (*****************************************************)
1719                     (* This is the first MB in the request queue.        *)
1720                     (*****************************************************)
1721                  /\ DirOpInProgress(p, reqQ[p][i].adr)
1722                  /\ \A j \in 1..(i-1) : reqQ[p][j].adr # reqQ[p][i].adr
1723                     (*****************************************************)
1724                     (* All requests r ahead of the MB in the request queue *)
1725                     (* have generated directory commands.  This means that *)
1726                     (* there is a directory command in progress for r's    *)
1727                     (* address, and there is no other request to the same  *)
1728                     (* address ahead of r in the queue.  (It follows that  *)
1729                     (* r is the ONLY request to this address ahead of the  *)
1730                     (* MB.)                                                *)
1731                     (*****************************************************)
1732             /\ ~\E m \in msgsInTransit :
1733                     /\ m.type \in {"Comsig", "GetShared", "GetExclusive",
1734                                    "ChangeToExclusive"}
1735                     /\ m.cmdr = p
1736                     (*****************************************************)
1737                     (* p is not waiting for a Comsig, meaning there is no  *)
1738                     (* Comsig in transit to p, and no QO message from p    *)
1739                     (* that will generate a Comsig.                        *)
1740                     (*****************************************************)
1741             /\ reqQ' = [reqQ EXCEPT ![p] = SeqMinusItem(@, idx)]
1742             /\ UNCHANGED <<respQ, cache, locked>>
1743
1744          \/ /\ req.type # "MB"
1745                (*********************************************************)
1746                (* Case 2: the request is a Rd or Wr or LL or SC.        *)
1747                (*********************************************************)
1748             /\ CanExecuteFromCache(p, idx)
1749                (*********************************************************)
1750                (* The request can be fulfilled from the p's cache.      *)
1751                (*********************************************************)
1752             /\ ProcRetireRdOrWr(p, idx, cache)
1753
1754       /\ UNCHANGED <<memDir, Q, fillQ, aInt>>
```

### 4.1.3 Discussion

Line 1716-1722 in the the source code listing is the equivalent to a `for` loop in ordinary programming language. Its goal, as states in the nearby comment, is to ensure the request at index `idx` is the first MB (Memory Barrier) request, by checking that all previous requests are not MBs.

However, TLC does not execute this "loop" in the normal way like other programming languages, i.e. by enumerating the items in the list from the smallest index to the largest index. In fact, the author did not specify what order is used, according to page 232 of [Lamport, 2002]. In fact, at the point when the error was thrown, the variable `i` in Line 1716 was assigned 8 first, and is apparently going in descending order, according to our later debug print messages. Thus, passing the loop at `i=8` does not guarantee that all previous requests does not contain any MB request. This will thus fail the statement at Line 1722, which tries to access the `adr` field by assuming a non-MB request.

By looking at the trace of the last state, State 18, we can see that indeed the `reqQ` queue contains two MB requests (at Line 6 and 8).

We fixed this bug by changing Line 1716-1722 to the following:

```
/\ \A i \in 1..(idx-1) :
     /\ reqQ[p][i].type # "MB"
        (*****************************************************)
        (* This is the first MB in the request queue.        *)
        (*****************************************************)
/\ \A i \in 1..(idx-1) :
     /\ DirOpInProgress(p, reqQ[p][i].adr)
     /\ \A j \in 1..(i-1) : reqQ[p][j].adr # reqQ[p][i].adr
```

This will ensure that every previous request is checked, no matter in what order, before other operations are carried out.

On hindsight, we realized that error caused by this bug will only appear if the queue length is at least 4. This would require considerable time for exhaustive

model checking, and simulation mode provides us a much faster way to encounter this bug. We should also count ourselves lucky to have our first few simulations bumping into this bug.

## 4.2 Bug 2

Although we found the first bug, we believe there was still another bug purposely inserted by the designers of this problem. As we continued to carry out our model checking by modifying some constraints and constant definitions. On $10^{th}$ of March, our checking process reported another bug.

The setup of the experiment this time was as follows:

1. TLC model checker was running in exhaustive model checking mode with 2 simultaneous workers (threads) on tembusu2 cluster.

2. A configuration with 1-bit data words, 2 local switches each containing 1 processor and 1 memory address, and a fixed initial memory state.

### 4.2.1 Reported Error

We first list the error reported by the TLC model checker as well as the error trace. Since the trace is very long, we only report relevant part of the the last state (i.e. State 12).

```
Error: Attempted to apply tuple <<  >> to integer 0 which is out
of domain. The behavior up to this point is:

STATE 1:
...
...

STATE 12: /\ Q = [ ProcToLS |-> (P1 :> <<  >> @@ P2 :> <<
>>),
  LSToProc |->
      ( P1 :> << {[type |-> "Inval", cmdr |-> P2, adr |-> 2, dest |-> P1]} >> @@
        P2 :> <<  >> ),
  LSToGS |->
      ( L1 :>
            << { [type |-> "Comsig", cmdr |-> P2],
                 [type |-> "FillMarker", cmdr |-> P2, adr |-> 2],
                 [type |-> "ComsigClear", cmdr |-> P2, adr |-> 2] } >> @@
        L2 :> <<  >> ),
  GSToLS |-> (L1 :> <<  >> @@ L2 :> <<  >>) ]
/\ aInt = 2
/\ locked = (P1 :> unL @@ P2 :> unL)
/\ fillQ = (P1:>{} @@ P2 :> {[data |-> (0 :> 0), state |->
"Exclusive", adr |-> 2]})
/\ cache = ( P1 :>
      ( 1 :>
            [ state |-> "Invalid",
              fillOrCTEAckPending |-> FALSE,
              version |-> <<  >> ] @@
        2 :>
            [ state |-> "Invalid",
              fillOrCTEAckPending |-> FALSE,
              version |-> <<  >> ] ) @@
...

The error occurred when TLC was evaluating the nested expressions
at the following positions:

0. Line 1381, column 6 to line 1381, column 41 in Wildfire
1. Line 1387, column 6 to line 1652, column 79 in Wildfire
2. Line 1492, column 24 to line 1504, column 52 in Wildfire
3. Line 1492, column 27 to line 1492, column 39 in Wildfire
4. Line 1486, column 25 to line 1486, column 63 in Wildfire
5. Line 1486, column 25 to line 1486, column 45 in Wildfire
6. Line 1162, column 20 to line 1162, column 46 in Wildfire

1169421 states generated, 30978 distinct states found, 4989 states
left on queue. The depth of the complete state graph search is 13.
```

### 4.2.2 Source Code

We next list the source code of the procedure in which the offending line resides.

```
1159    OldestIdx(version) == 1
1160    NewestIdx(version)==Len(version)
1161    Oldest(version) == version[OldestIdx(version)]
1162    Newest(version) == version[NewestIdx(version)]
...
1376    ProcReceiveMsg(p, idx) ==
        (*************************************************************************)
        (* Processor p receives a message set, the idx-th set in the queue from  *)
        (* the local switch to p, and processes each of the messages in the set. *)
        (*************************************************************************)
1381        /\ CanDequeueMsgSet("LSToProc", p, idx)
            (*********************************************************************)
            (* This is the action's enabling condition, true iff the message    *)
            (* can be received now.                                              *)
            (*********************************************************************)
1387        /\ LET
                msgset == Q.LSToProc[p][idx]
                (*****************************************************************)
                (* The message set that processor p is receiving.  It contains    *)
                (* either a single message, or else a ChangeToExclusiveAck        *)
                (* accompanied by a Comsig.                                       *)
                (*****************************************************************)
            msg == CHOOSE m \in msgset : m.type # "Comsig"
                (*****************************************************************)
                (* The non-Comsig message in the set, if there is one.            *)
                (*****************************************************************)
            entry == cache[p][msg.adr]
            IN
1402            CASE \E m \in msgset : m \in ForwardedGet ->
            ...
1473            []   \E m \in msgset : m \in Inval ->
            (*************************************************************)
            (* Case 2: The message is an Inval.                          *)
            (*************************************************************)
1477            LET forOldVersion ==
                  (*************************************************)
                  (* True iff the Inval is for an earlier version of    *)
                  (* the data, and hence should be thrown away.         *)
                  (*************************************************)
1482                  \/ Len(entry.version) = 0
                     (**********************************************)
                     (* There is no current version.                *)
                     (**********************************************)
1486                 \/ Newest(entry.version).fillMarkerPending
                     (**********************************************)
                     (* An Inval for any version must follow that       *)
                     (* version's FillMarker, so in this case the Inval *)
                     (* isn't for the current version.                  *)
                     (**********************************************)
1492            IN  /\  \/ forOldVersion
                       (*********************************************)
                       (* Throw the Inval away, since the fillmarker is *)
                       (* for an earlier version.                     *)
                       (*********************************************)
   ...
```

### 4.2.3 Discussion

Starting the discussion by tracing the error from the nested expressions listed in the error report. The trace was from Line 1381, which was inside the function of `ProcReceiveMsg(p, idx)` from Line 1376. The function was defined to retrieve the messages from message queue of process P, and process those messages. In this case, state 12 shows that P1 received a Invalid message from Local Switch at address 2.

```
LSToProc |->
      ( P1 :> << {[type |-> "Inval", cmdr |-> P2, adr |-> 2, dest |-> P1]} >>
```

#### First Try

Our first attempt to fix the bug started by looking at the code from Line 1473 specifying the case for processing Invalidate message. We continued tracing down the line to Line 1486 which calling another function `Newest(entry.version)` in order to get the latest version of this particular cache entry. From Line 1162, we understood that: `Newest(version) == version[NewestIdx(version)] = version[Len(version)]`. However, the cache entry version here was empty, i.e. `Len(version) = 0`.

```
2 :>
            [ state |-> "Invalid",
              fillOrCTEAckPending |-> FALSE,
              version |-> <<  >> ] )
```

In the Wildfire Specification, the version index could not be 0, because the oldest version index was defined as 1. The problem occurs here due to this index "0".

We suggest to fix this bug by changing Line 1622 to the following:

```
1162    Newest(version) == IF Len(version) >0
                           THEN version[NewestIdx(version)]
                           ELSE version[1]
```

Unfortunately, the problem was not resolved after our modification. However, the error reports gave us more information so that we start the second try.

```
Error: Attempted to apply tuple <<  >> to integer 1 which is out of
domain. The behavior up to this point is:
STATE 1: ...
```

### Second Try

From the second error report, we realize that the reason triggering the bug is not just index "0", because index "1" will also make the program execution crash.

Take a close look at the expression from Line 1477 to Line 1486, the value of variable `forOldVersion` depends on 2 conditions here. In the other words, `forOldVersion` is TRUE if either version of the entry is empty or there is a new version indicating `fillMarkPending`.

```
1477 LET forOldVersion ==
                     (*****************************************************)
                     (* True iff the Inval is for an earlier version of   *)
                     (* the data, and hence should be thrown away.        *)
                     (*****************************************************)
1482                     \/ Len(entry.version) = 0
                        (**************************************************)
                        (* There is no current version.                   *)
                        (**************************************************)
1486                    \/ Newest(entry.version).fillMarkerPending
                        (**************************************************)
                        (* An Inval for any version must follow that      *)
                        (* version's FillMarker, so in this case the Inval *)
                        (* isn't for the current version.                 *)
                        (**************************************************)
```

In TLA$^+$ specification, the execution of the program will test both predicates that stated above, even if the first predicate is TRUE. When the program tests the second predicate, it will call function `Newest(entry.version)` which will eventually get `version[0]`. In this case here, there is no version in the entry. Hence, for expression `version[NewestIdx(version)]`, no matter what the index of the set `version` is, the problem will occur due to its emptiness.

```
2 :>
            [ state |-> "Invalid",
              fillOrCTEAckPending |-> FALSE,
              version |-> <<  >> ] )
```

This time, we suggest to fix this bug by changing the code from Line 1482 to Line 1486 to the following, so that the program will **only** test the second clause when there is an entry version.

```
1482      IF Len(entry.version) = 0
          THEN TRUE
...
1486      ELSE Newest(entry.version).fillMarkerPending
```

After testing, we prove that our solution is correct.

## 4.3 Successful Checking Result

After we have found, and more importantly, solved 2 bugs in the Wildfire Specification, we obtain a full-length TLC model checking result report.

```
TLC Version 2.0 of Jun 10, 2003
Model-checking

ParsingfileCheckWildfire.tla
...

Model checking completed.
No error has been found.

Estimates of the probability that TLC did not check all reachable
states
    because two distinct states had the same fingerprint:
    calculated (optimistic):  1.3642639229470913E-8
    based on the actual fingerprints:  2.758720326262068E-10
3439093 states generated, 74804 distinct states found, 0 states left
on queue.
The depth of the complete state graph search is 33
```

## 4.4 Summary

We have presented our checking of the Wildfire Model with two Local Wwitches each containing one processor and one memory address, and the checker has generated more than 3 million states. It shows that the Wildfire Challenge is really a very complicated problem for checking cache-coherency.

Our model checking program has successfully verified three key areas of the Wildfire Specification, namely Initial State, Next State and Liveness. The Initial State guarantees that the initialization parameters are of the correct types and values. The Next State governs how the system should change from the current time $t$ to $t + 1$. The Liveness of this model is to guarantee that good things will always happen, i.e. every request eventually generates a response.

To further prove our correction is right, we also changed the `QLen` constraint from 2 to 3, i.e. maximum message queue size was 3. This made the model checking so complex that it generated more than 10 million states (we only monitored the process intermittently). However, due to the huge intermediate files generated from the model checking, the process quitted unexpectedly due to Disk Full error.

# Chapter 5

# Conclusion

After a year the problem has been posted, it has been downloaded from over 275 sites, it is possible that many of them have seriously tried to solve the problem but backed off due to the enormous computational complexity.

Our effort spent on this project include:

- Reading the 300 odd pages of the official TLA$^+$ book, and understanding the syntax and semantic of the TLA$^+$ language, which was vastly different from other model checking languages such as SMV.

- Understanding the usage of the accompanying tools, such as the TLA$^+$ model typesetter and model checker

- Sufficient understanding of the WildFire protocol by reading through the source code.

  This enabled us to write the `CheckWildFire.tla` and `CheckWildFird.cfg` files to hook up the model with the model checker.

- Repeated tuning to get the parameters right to get a manageable and meaningful model.

- Understanding the mystic error trace
- Repeated attempts to correct the bug and re-test

The cache-coherent protocol has many potential problems awaiting to be verified. Although we have found two bugs of the specification in such a short time and do hope that these two bugs are the correct answers to this challenge, there is still ample room for us to do further research and verification on it.

# Bibliography

Leslie Lamport. *Specifying Systems*. Addison-Wesley, 2002.

Leslie Lamport, Madhu Sharma, Mark Tuttle, and Yuan Yu. The Wildfire Challenge Problem. *Compaq*, 2001.