A P2P Protocol Verification using Murϕ

Valerio Grossi

Dept. of Computer Science
University of Pisa

March 13, 2009
Outline

1. **Introduction**
   - Model Checking
   - Model Checker

2. **Murphy**
   - A Quick Look

3. **A Peer-to-Peer Protocol Verification using Murphy**
   - The Sigma Protocol
   - Creating the Model
   - State Description
   - Automata Descriptions

4. **Results**
   - Running Experiments

5. **Final Remarks**
Model Checking

From Wikipedia

Model checking is the process of checking whether a given structure is a model of a given logical formula. The concept is general and applies to all kinds of logics and suitable structures.

A simple model-checking problem is verifying whether a given formula in the propositional logic is satisfied by a given structure.
State-space Exploration

Model checking tools face a combinatorial blow up of the state-space, commonly known as the state explosion problem, that must be addressed to solve most real-world problems.

There are several approaches to define a model and to combat the state explosion problem.
Overview

- The Murϕ description language is based on a collection of guarded commands repeatedly executed in an infinite loop.
- Murϕ data structures and commands are written in a high-level language that is familiar to programmers.
- Murϕ is essentially a formal verifier based on explicit state enumeration.

Current Version

- Originally Developed at Stanford, now maintained at University of Utah.
- Latest version is Murphi3.1 of year 1998, but there are several more recent versions available.
- Murϕ works on various versions of Unix and Linux. It can work on WindowsXP using Cygnus’s Cygwin environment.
- Home site: http://sprout.stanford.edu/dill/murphi.html
Applications of Murphy

- Verification of the cache coherence protocols in Stanford's DASH and FLASH multiprocessors.
- Verification of link-level protocol and cache coherence protocol in Sun’s S3.mp multiprocessor.
- Verification of the cache coherence algorithm in Sun’s UltraSparc-1.
- Executable specification, analyzer, and verifier for Sparc V9 memory models: TSO, PSO, and RMO.
- Incorporated into U. of Wisconsin’s Tempest customizable cache coherence protocol system.
- Verification of part of SCI ("Scalable Coherent Interface"), IEEE Std 1596-1992. Some bugs were discovered.
- Analysis of cryptographic and security-related protocols.
- Verification of proprietary protocols at several companies, including Fujitsu, HAL Computer Systems, HP, and IBM.
The System

From a Conceptual point of view

Murphy takes as input a Finite State System $S$ and checks that a given invariant property $\varphi$ for $S$ is satisfied.

Definition

1. A Finite State System $S$ is a 4-tuple $(S, I, A, R)$
   - $S$ is a finite set of states.
   - $I \in S$ initial state.
   - $A$ is a finite set of transition labels.
   - $R$ is a transition relation on $S \times A \times S$.

2. There is a transition from $s$ to $s'$ iff there exists $a \in A$ s.t $R(s, a, s')$.

3. $\text{Reach}(S)$ is the set of states reachable in zero or more steps from $I$. 

The Language

Pascal-Like Language

A Murϕ description consists of:

- constant and type declarations
- variable declarations
- functions and procedures definitions
- rule definitions
- a collection of invariants
- a description of a start state
The Language

**Rule**
Each rule is a guarded command consisting of a *condition* and an *action*.
- The condition is a boolean expression consisting of constants, declared variables and operators.
- The action is a sequence of statements

**Invariant**
An invariant is a boolean expression that references the defined variables

**State**
A state is a function that gives values to all the variables
Usage

The Execution

An execution of the system is a finite (or infinite) sequences of states $s_0, s_1, \ldots$, where:

- $s_0$ is determined by the Mur$\varphi$ description.
- If $s_i$ is any state in the sequence, $s_{i+1}$ can be obtained by applying some rules whose condition is true in $s_i$ and whose actions transform $s_i$ in $s_{i+1}$.
  - $s_i$ can satisfy several conditions, the verifier must cover all the possibilities.
- The invariants are applied whenever a state is explored.
  - if any invariant is violated, an error is reported
Usage

From the description to verifier

The Mur$\varphi$ compiler takes a source description and generates a C++ program, which is compiled together with the verifier code.

Not only invariant violation

The Mur$\varphi$ verifier checks for:

- invariant violation.
- error statement.
- assertion violation.
- deadlock.

The verifier attempts to enumerate all the states of the system, checking for error conditions as it proceeds.
State Exploring

Internal Data Structures

Every state is encoded using the minimum number of bits in two structures:

- Double hashing *Hash Table* that stores reached states is used to decide if:
  - newly-reached state is *old*
  - has not been reached already

- New states are stored in a *queue*\(^a\) of *active states*

\(^a\)Depending on the organization of this queue, the verifier does a breadth-first (or depth) search.
State Exploring

Symmetry

1. Detection of structural symmetries in the system.
2. On-the-fly detection of symmetrically-equivalent states during verification, so that the full state space does not need to be constructed.
3. New data types introduced to facilitate detection of symmetries (and testing equivalent states).
   - **Scalarset** for describing elements that can be freely permuted.
   - **Multiset** for describing a bounded set of values whose order is irrelevant.

Error-Tracking

Every state in the hash table has a pointer to a predecessor state that can be used to generate an error trace if a problem is detected.
A Quick Look

cMurφ

Caching Murφ

Developed by Enrico Tronci et. al. at University of Roma, Caching Murφ has been obtained from Murphi 3.1 release by replacing Murφ Hash Table with a Cache and Murφ RAM queue with a disk queue.


64 bit Version

CMurphi 3.4.64bits developed at University of Utah, is able to overcome the limitation of the hash compaction implementation of Murφ 3.1, thus working also on 64-bit hardwares. Moreover, disk swapping mechanism for the BF consumption queue may be used.
A Little Bit of Syntax

Constants, Types, Variables, Functions declaration, and so on

The syntax is quite similar to Pascal, we will see an example during our case study

Rule

Rule rule_name \(\Rightarrow\)

condition

Begin

sequence_of_commands;

End;

Ruleset

Ruleset formal : range Do

ruleSet

End;
A Little Bit of Syntax

Startstate

Startstate
Begin
initial_values
End;

Invariant

Invariant
set_of_conditions\(^a\)
End;

\(^a\)This set of conditions can be defined using also quantifiers
A Little Bit of Syntax

**Universal Quantifier**

**Forall** formal : range Do

conditions

End;

**Existential Quantifier**

**Exists** formal : range Do

conditions

End;
A Simple Example: Ping Pong

Type player_t : 0..1;
Var Players : Array[ player_t ] of Record
  hasball, gotball: boolean
End;

Ruleset p : player_t Do
  Alias ping: Players[p];
    pong: Players[ 1 - p ] Do
  Rule “Pass ball”
    ping.hasball ==> Begin
      ping.hasball := false;
      pong.gotball := true;
    End;
  Rule “Get ball” ...
  Rule “Keep ball” ...
A Simple Example: Ping Pong

Startstate

Begin

ping.hasball := true;
ping.gotball := false;
pong.hasball := false;
pong.gotball := false;

End;

Invariant “Only one ball in play”

Forall p : player_t Do

!(Players[p].hasball & Players[p].gotball) &
(Players[p].hasball | Players[p].gotball) ->

Forall q : player_t Do

(Players[q].hasball | Players[q].gotball) -> p = q

End

End;
Execution: Ping Pong

Protocol: ../ex/toy/pingpong
Algorithm:
  Verification by breadth first search.

==========================================
Status:
  No error found.
State Space Explored:
  4 states, 6 rules fired in 0.10s.
Rules Information:
  Fired 1 times - Rule "Pass ball, p:0"
  Fired 1 times - Rule "Pass ball, p:1"
  Fired 1 times - Rule "Keep ball, p:0"
  Fired 1 times - Rule "Keep ball, p:1"
  Fired 1 times - Rule "Get ball, p:0"
  Fired 1 times - Rule "Get ball, p:1"
Execution: some of the available options

Verification Strategy: (default: -vbfs)
- vbfs verify with breadth-first search.
- vdfs verify with depth-first search.
- ndl do not check for deadlock.

Others Options: (default: -m8, -p3, -loop1000)
- m<n> amount of memory for closed hash table in Mb.
- k<n> same, but in Kb.
- loop<n> allow loops to be executed at most n times.
- p<n> report progress every 10^n events, n in 1..5.
- pn print no progress reports.
- pr print out rule information.

Error Trace Handling: (default: -tn)
- tv write a violating trace.
- td write only state differences from the previous states.
- tf write full states in trace.
- ta write all generated states at least once.
- tn write no trace (default).
Execution: Ping Pong with \(-ta\)

Protocol: ../ex/toy/pingpong

Algorithm:
Verification by breadth first search.

State 1: State 3:
Players[0].hasball:true Players[0].hasball:false
Players[0].gotball:false Players[0].gotball:false
Players[1].gotball:false Players[1].gotball:true

State 2: State 4:
Players[0].hasball:false Players[0].hasball:false
Players[0].gotball:false Players[0].gotball:true
Players[1].hasball:true Players[1].hasball:false

Status:
No error found.

State Space Explored:
4 states, 6 rules fired in 0.10s.
Inserting an Error

Rule “Pass ball”

\[
\text{ping.hasball} \Rightarrow
\]

Begin

\[
\text{ping.hasball} := \text{true};
\]

\[
\text{pong.gotball} := \text{true};
\]

End;
Error Trace: Ping Pong

The following is the error trace for the error:

Invariant "Only one ball in play." failed.

Startstate 0, p:0 fired.
Players[0].hasball:true
Players[0].gotball:false
Players[1].hasball:false
Players[1].gotball:false

Rule Pass ball, p:0 fired.
The last state of the trace (in full) is:
Players[0].hasball:true
Players[0].gotball:false
Players[1].hasball:false
Players[1].gotball:true

End of the error trace.

Result:
Invariant "Only one ball in play." failed.
Protocol Verification

Formal verification of a protocol proceeds by describing the protocol in some language, and then comparing the behavior of this description with a specification of the desired behavior.
Protocol Verification

Formal verification of a protocol proceeds by describing the protocol in some language, and then comparing the behavior of this description with a specification of the desired behavior.

A verifier generates states from the description, comparing them with the specification as it goes.

- If the verifier detects an inconsistency, this fact is reported, with an example of sequence states that illustrates how the problem can occur.
Protocol Verification

Formal verification of a protocol proceeds by describing the protocol in some language, and then comparing the behavior of this description with a specification of the desired behavior.

A verifier generates states from the description, comparing them with the specification as it goes.

- If the verifier detects an inconsistency, this fact is reported, with an example of sequence states that illustrates how the problem can occur.

The usual assumption about the role of formal verification is to provide a guarantee of design correctness, also potentially reducing the cost and time of product development.
Mutual Exclusion on P2P Environment

Distributed Mutual Exclusion

To manage the access to a single, indivisible shared resource by at most a process at any time in a distributed environment.

Peer-To-Peer (P2P) Environment

We are in presence of P2P systems with Distributed Hash Table (DHT), and individual clients work with the servers to coordinate mutual exclusion, rather than working with other clients directly.
Asynchronous Message-passing Distributed System

Processes are logically separated into clients and servers:

- Clients request to enter their mutually exclusive critical sections.
- Servers help coordinate the client access to the critical sections.

The system is dynamic:

- New clients may join the system and make new requests at any time.
- Servers may crash, then recover or be replaced by a new server.
The Sigma Protocol

The Protocol Description

Any client that wants to enter a critical section fires request messages to each server and waits for responses. A server grants a lease, if it is not owned by anyone, while otherwise rejects the request. The client which obtains $m$ out of $S$ servers ($m > \frac{S}{2}$) wins the round. Losers release acquired permission, backoff and retry the access.

As the $client_A$ receives enough responses, three alternatives may be introduced:

1. $client_A$ is the winner by quorum consensus. It succeeds and obtains permission to enter the critical section.

2. A client other than $client_A$ is the winner. $client_A$ does nothing, because it knows it has been enqueued on the server already.

3. Nobody has won. $client_A$ then sends out a YIELD message to each of the acquired servers.
Service policy

The use of logical clock and the First-Come First-Serve policy at server does not guarantee FCFS, since client requests can take arbitrarily long to arrive. Thus, Sigma can be best described as quasi-FCFS.

Safety

Sigma guarantees safety with high probability. No known protocol can ensure 100% correctness under failure. The probability of violating safety can be lower, by setting appropriate parameters. However, these parameters do not affect our Mur$\varphi$ model, which has a limited number of clients and servers and does not consider some network features present in the original model, such as the network latency and the channel or the peer fault.

Liveness

Progress is ensured by using lease.
# Preliminary Assumptions

Our model differs from the Sigma protocol in some details:

- The resource $R$ is shared by all servers.
  - A client has to send a request message indifferently to each server.
- The server response does not contain the ID of the resource owner, but only (Yes, No).
- The server sends a Yes message to the client $i$, if the latter wins the round to reach the resource.
- The Communication Channel is safe
  - No message can be lost.
  - No message can be forged\(^a\).

\(^a\)This feature is also assumed in the original specification of the protocol.
Communication relies on a bi-directional channel implemented by two matrices.
Creating the Model

RQS Matrix

The RGS assumes the following values

\[
RQS[s][c] = \begin{cases} 
\text{INIT}_\text{REQ} & \text{initial value} \\
\text{REQUEST, Timestamp} & \text{client } c \text{ sends a request to } s \\
\text{RELEASE} & \text{client } c \text{ releases the server } s \text{ resource} \\
\text{YIELD} & \text{client } c \text{ sends YIELD to server } s 
\end{cases}
\]

where \( s \) and \( c \) are respectively the server and the client index

The \textit{REQUEST} message contains also a timestamp value, in order to guarantee the request ordering on the server side
Creating the Model

Operations on RQS Matrix

Only clients can modify RQS status with:

- `ClearRQS(s,c)`: resets the RQS array to the initial value.
- `SendRequest(s,c,timestamp)`: sends a request message from client `c` to server `s`.
- `SendRelease(s,c)`: sends a release message from client `c` to server `s`.
- `SendYield(c)`: sends a `YIELD` message from client `c` to every acquired server.
Creating the Model

Operations on RQS Matrix

Only clients can modify RQS status with:

- **ClearRQS(s,c)**: resets the RQS array to the initial value.
- **SendRequest(s,c,timestamp)**: sends a request message from client c to server s.
- **SendRelease(s,c)**: sends a release message from client c to server s.
- **SendYield(c)**: sends a YIELD message from client c to every acquired server.

Servers can only read RQS status with:

- **OnRequest(s)**: checks if server s has received a request from some client.
- **IsReq(s,c)**: checks if server s has received a request from client c.
- **OnRelease(s,c)**: checks if client c wants to release server s.
- **OnYield(s,c)**: checks if client c has sent a YIELD message to server s.
- **GetTimestamp(s,c)**: returns the timestamp value of client c.
RSP Matrix

The RSP assumes the following values

\[
RSP[s][c] = \begin{cases} 
\text{INIT\_RESP} & \text{initial value} \\
\text{YES} & \text{client } c \text{ is the owner of server } s \\
\text{NO} & \text{client } c \text{ does not win the round for server } s
\end{cases}
\]
RSP Matrix

The RSP assumes the following values

\[
\text{RSP}[s][c] = \begin{cases} 
\text{INIT\_RESP} & \text{initial value} \\
\text{YES} & \text{client } c \text{ is the owner of server } s \\
\text{NO} & \text{client } c \text{ does not win the round for server } s 
\end{cases}
\]

Only servers can modify RSP status with:

- \text{ClearRSP}(s,c): resets the RSP array to the initial value.
- \text{SendResponse}(s,c,msg): sends a message (msg) from server s to client c.
# RSP Matrix

The RSP assumes the following values

$$RSP[s][c] = \begin{cases} 
\text{INIT\_RESP} & \text{initial value} \\
\text{YES} & \text{client c is the owner of server s} \\
\text{NO} & \text{client c does not win the round for server s} 
\end{cases}$$

Only servers can modify RSP status with:

- \text{ClearRSP}(s,c): resets the RSP array to the initial value.
- \text{SendResponse}(s,c,msg): sends a message (msg) from server s to client c.

Clients can only read RSP status with:

- \text{Release}(c): checks if every server has replied to the release message.
Creating the Model

Voting

The Sigma protocol adopts a quorum consensus schema to select the round winner. Our model employs a simple constant value $M$.

The Voting Mechanism: three functions

- `EnoughResponsesReceived(c)`: checks if client $c$ has received enough responses from the servers.
- `IsWinner(c)`: checks if client $c$ is the winner of the round.
- `OtherWinner(c)`: checks if another client (different from $c$) is the winner of the round.
In order to avoid starvation, the protocol provides for a logical clock (timestamp) which allows the servers for requests ordering.

**The Logical Clock**

The logical clock is implemented in our model by the variable `logical_clock` and the following functions:

- `init Logical Clock()`: initializes the logical clock.
- `get Logical Clock()`: returns the new value of the Lamport’s clock.
In the Sigma protocol, a queue is used by servers to collect the client requests, while a server grants permission to a different client.

### Implementation

```
s_queue : Record
    element : Array [time] of ind_C;
End;
```

### Operations

- `queue_Init(s)`
- `queue_Front(s)`
- `queue_Empty(s)`
- `queue_Insert(s,c,t)`
- `queue_Remove(s)`
State Description

Client State

```
client_state : Record
    state : label_C;
    t : time;
End;

label_C : Enum{INIT_C, WAIT_RESP, CS};
```
Server State

Description

server_state : Record
state : label_S;
baby : boolean;
owner : ind_C;
queue : s_queue;
End;

label_S : Enum{INIT_S, BUSY};
The Client Automata
The Server Automata

OnInit:
- OnRequest
  grants permission to the winner & enqueue additional requests

OnBusy:
- OnRelease & not queue_Empty
  grants permission to the front queue client

OnYield:
- grants permission to the front queue client
Running Experiments

Experimental Setup

System Running

AMD Athlon™ 64 X2 3800 with 2Gb of RAM, Fedora 4 (kernel 2.6.11).

Model Checker

- Murϕ ver. 3.1.
- cMurϕ ver. 4.25.

Two different experiments

- Every server has to grant permission to the same client \( (M = S) \).
- Majority constant \( M \) equal to \( \frac{S}{2} + 1 \).

Options Used

- \(-b\) Bit compacted states.
- \(-c\) Hash compaction.
- \(-\text{disk}\) Enables hash table diskswapping only in cMurϕ 4.25.
Invariant Expression

**Definition**

**Invariant**

\[ \neg \exists i : \text{ind} \ C \ \text{Do} \]

\[ \exists j : \text{ind} \ C \ \text{Do} \]

\[ (\text{client}[i].\text{state} = \text{CS} \ & \ \text{client}[j].\text{state} = \text{CS} \ & \ i \neq j) \]

\[ \text{End} \]

\[ \text{End;} \]
## Explored States

<table>
<thead>
<tr>
<th>C</th>
<th>S</th>
<th>Murφ 3.1$^2$</th>
<th>Murφ 3.1 ($M = \frac{S}{2} + 1$)</th>
<th>cMurφ 4.25$^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$M = S$</td>
<td>$M = \frac{S}{2} + 1$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>51</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>745</td>
<td>745</td>
<td>745</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1.482</td>
<td>14.893</td>
<td>14.893</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>10.162</td>
<td>252.932</td>
<td>252.932</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>60.337</td>
<td>5.775.618$^4$</td>
<td>5.775.618$^4$</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>327.327</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>857.615</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

$^2$No options are used during compilation.

$^3$Model verified with $M = \frac{S}{2} + 1$.

$^4$Model executed with `-m1024` option (that is the amount of memory for closed hash table in Mb).
## Fired Rules

<table>
<thead>
<tr>
<th>C</th>
<th>S</th>
<th>\text{Mur}\phi 3.1^2</th>
<th>\text{Mur}\phi 3.1 (M = \frac{S}{2} + 1)</th>
<th>\text{cMur}\phi 4.25^3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(M = S)</td>
<td>(M = \frac{S}{2} + 1)</td>
<td>(-b) option</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>91</td>
<td>91</td>
<td>91</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2.076</td>
<td>2.076</td>
<td>2.076</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4.672</td>
<td>47.625</td>
<td>47.714</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>39.552</td>
<td>929.212</td>
<td>922.428</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>281.748</td>
<td>24.556.566^4</td>
<td>24.329.349^4</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1.793.738</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>4.967.808</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

\(^2\)No options are used during compilation.

\(^3\)Model verified with \(M = \frac{S}{2} + 1\).

\(^4\)Model executed with \texttt{-m1024} option (that is the amount of memory for closed hash table in Mb).
Observations

This is only an example:

- Assumptions on communication channel are too strong.
  - No message can be lost.

- There are no failures in the system in particular:
  - Random Crash: Statistical independent faults.
  - Non-random Crash: Using a statistical model to simulate crashes.
  - Byzantine Crash: Malicious attacks.
References


