A Simple Model of Communication APIs – Application to Dynamic Partial-order Reduction

Cristian Rosa  Stephan Merz  Martin Quinson

AVOCS 2010
22/09/2010
Motivation

- Distributed Algorithms are hard to get right:
  - lack of a shared clock
  - lack of a global view of the state

- Errors are hard to find and reproduce

- State-exploration techniques
  - exhaustive
  - counter-examples
  - reproducible
Model-checking Distributed Systems

We define a Distributed System (DS) as:

- processes running distributed across several networked hosts
- no shared clock
- no shared memory
- communication by message exchange
Model-checking Distributed Systems

We define a Distributed System (DS) as:

- processes running distributed across several networked hosts
- no shared clock
- no shared memory
- communication by message exchange

We verify the real C implementation of the programs:

- no models, no abstraction, the processes are really executed
- from a given initial state
- state space generated by the interleaving of the messages
- exploration is depth bounded (useful for debugging anyway)

It its mandatory to apply partial-order techniques!
Happened-before Relation

Consider the following processes:

A

\[ a_0 \xrightarrow{s_a} a_1 \xrightarrow{l_a} a_2 \]

B

\[ b_0 \xrightarrow{s_b} b_1 \xrightarrow{l_b} b_2 \]

Execution traces:

C

\[ c_0 \xrightarrow{r} c_1 \xrightarrow{r} c_2 \]
Happened-before Relation

Consider the following processes:

A  
\[ a_0 \xrightarrow{s_a} a_1 \xrightarrow{l_a} a_2 \]

B  
\[ b_0 \xrightarrow{s_b} b_1 \xrightarrow{l_b} b_2 \]

C  
\[ c_0 \xrightarrow{r} c_1 \xrightarrow{r} c_2 \]

Execution traces:

A  
\[ \square \xrightarrow{a_0} \square \xrightarrow{s_a} \bullet \xrightarrow{a_1} \square \xrightarrow{l_a} \bullet \xrightarrow{a_2} \]

B  
\[ \square \xrightarrow{b_0} \bullet \xrightarrow{s_b} \square \xrightarrow{b_1} \bullet \xrightarrow{l_b} \square \xrightarrow{b_2} \]

C  
\[ \square \xrightarrow{c_0} \bullet \xrightarrow{r} \square \xrightarrow{c_1} \bullet \xrightarrow{r} \square \xrightarrow{c_2} \]
Consider the following processes:

A
\[ \begin{array}{c}
\text{a}_0 \\
\text{s}_a \\
\text{a}_1 \\
\text{l}_a \\
\text{a}_2
\end{array} \]

B
\[ \begin{array}{c}
\text{b}_0 \\
\text{s}_b \\
\text{b}_1 \\
\text{l}_b \\
\text{b}_2
\end{array} \]

C
\[ \begin{array}{c}
\text{c}_0 \quad \text{r} \quad \text{c}_1 \quad \text{r} \\
\text{c}_2
\end{array} \]

Execution traces:

A
\[ \begin{array}{c}
\text{a}_0 \\
\text{s}_a \\
\text{a}_1 \\
\text{l}_a \\
\text{a}_2
\end{array} \]

B
\[ \begin{array}{c}
\text{b}_0 \\
\text{s}_b \\
\text{b}_1 \\
\text{l}_b \\
\text{b}_2
\end{array} \]

C
\[ \begin{array}{c}
\text{c}_0 \quad \text{r} \\
\text{c}_1 \quad \text{r} \\
\text{c}_2
\end{array} \]
Depth-first Search Exploration

A: \( a_0 \rightarrow s_a \rightarrow a_1 \rightarrow l_a \rightarrow a_2 \)

B: \( b_0 \rightarrow s_b \rightarrow b_1 \rightarrow l_b \rightarrow b_2 \)

C: \( c_0 \rightarrow r \rightarrow c_1 \rightarrow r \rightarrow c_2 \rightarrow r \rightarrow l_a \)

It's another serialization of the same partial-order!
Depth-first Search Exploration

Depth-first search state exploration algorithm:

It's another serialization of the same partial-order!
Depth-first Search Exploration

Depth-first search state exploration algorithm:

It’s another serialization of the same partial-order!
Dynamic Partial-order Reductions by Example

What are the transitions that we should interleave?
or equivalently ...
How do we generate a serialization of a different partial-order?
Dynamic Partial-order Reductions by Example

What are the transitions that we should interleave? or equivalently ...

How do we generate a serialization of a different partial-order?

Interleaving dependent transitions!

\[ D(t_i, t_j) = \neg l(t_i, t_j) \]
Computing $D$ Efficiently

How do we get the predicate $D$?

- Using the semantics of the transitions
- Proof of "independence theorems" for each pair of transitions
- The $I$ predicate is the disjunction of these cases

$$I(t_i, t_j) = (t_i = A \land t_j = B) \lor \ldots$$
Computing D Efficiently

How do we get the predicate $D$?

- Using the semantics of the transitions
- Proof of "independence theorems" for each pair of transitions
- The $I$ predicate is the disjunction of these cases

$$I(t_i, t_j) = (t_i = A \land t_j = B) \lor \ldots$$

- It can be overapproximated by a $D'$ such that

$$D(A, B) \Rightarrow D'(A, B)$$

but the converse might not be true

$$D'(A, B) \nRightarrow D(A, B)$$

If we don’t know if $I(t_i, t_j)$ we assume $D'(t_i, t_j)$ (for soundness).
API Functions as Transitions

- The transitions are the calls to the communication API
- The problem: almost no API has a formal specification of their semantics
- Manual specification is required based on:
  - informal API references
  - experiments
  - user experience
- It is a tedious and time consuming job (i.e. +100 pages for a subset of MPI [PGK+07])
- It should be done for each API
Contributions

The contributions of this work are:

- A core set of five networking primitives
- Formal specification in $\text{TLA}^+$ of it's semantics
- Theorems of independence between certain primitives
- Implementation of the networking core inside of SimGrid simulator [CLQ08]
- A DPOR exploration algorithm using the SimGrid simulation framework
The communication model is based on mailboxes:

- processes post send/receive request into mailboxes
- requests are queued and matched in FIFO order
The Communication Model

The communication model is based on mailboxes:

- processes post send/receive request into mailboxes
- requests are queued and matched in FIFO order

There are four primitives:

- **Send** – asynchronous send
- **Recv** – asynchronous receive
- **WaitAny** – block until completion of a communication
- **Test** – test for completion without blocking
The Communication Model

A

Network

B

Send(&x);

Recv(&y);

WaitAny({id});
y:=x;
The Communication Model

Network

A

Send(&x);

B

WaitAny({id});
y:=x;
The Communication Model

A

id

Send(&x);

Network

{id,"send",A,_,&x,_}

B

Recv(&y);

WaitAny({id});
y:=x;
The Communication Model

A

id

Send(&x);

Network

{id,"send",A,_,&x,_}

B

Recv(&y);

WaitAny({id});
y:=x;
The Communication Model

Network

{[id,"ready",A,B,&x,&y]}

Send(&x);

Recv(&y);

WaitAny({id});
y:=x;
The Communication Model

A

id

Send(&x);

Network

{[id,"send",A,_,&x,_]}

B

id

Recv(&y);

Send(&x);

Recv(&y);

{[id,"done",A,B,&x,&y]}

WaitAny({id});

y:=x;
(In)Dependence Theorems

Theorem

Any two Send and Recv transitions are independent.

\[ \forall p_1, p_2 \in Proc, rdv_1, rdv_2 \in RdV, d_1, d_2 \in Addr, c_1, c_2 \in Addr : \]
\[ I(Send(p_1, rdv_1, d_1, c_1), Recv(p_2, rdv_2, d_2, c_2)) \]
(In)Dependence Theorems

Theorem

Any two Send and Recv transitions are independent.

\[ \forall p_1, p_2 \in Proc, rdv_1, rdv_2 \in RdV, d_1, d_2 \in Addr, c_1, c_2 \in Addr : \]
\[ I(Send(p_1, rdv_1, d_1, c_1), Recv(p_2, rdv_2, d_2, c_2)) \]
(In)Dependence Theorems

Theorem

Any two Send and Recv transitions are independent.

∀p_1, p_2 ∈ Proc, rdv_1, rdv_2 ∈ RdV, d_1, d_2 ∈ Addr, c_1, c_2 ∈ Addr :
I(Send(p_1, rdv_1, d_1, c_1), Recv(p_2, rdv_2, d_2, c_2))
(In)Dependence Theorems

Theorem

Any two Send andRecv transitions are independent.

∀p₁, p₂ ∈ Proc, rdv₁, rdv₂ ∈ RdV, d₁, d₂ ∈ Addr, c₁, c₂ ∈ Addr :
I(Send(p₁, rdv₁, d₁, c₁), Recv(p₂, rdv₂, d₂, c₂))
(In)Dependence Theorems

Theorem

Any two Send andRecv transitions are independent.

∀p₁, p₂ ∈ Proc, rdv₁, rdv₂ ∈ RdV, d₁, d₂ ∈ Addr, c₁, c₂ ∈ Addr :
I(Send(p₁, rdv₁, d₁, c₁),Recv(p₂, rdv₂, d₂, c₂))
(In)Dependence Theorems

Theorem

Any two Send andRecv transitions are independent.

\[ \forall p_1, p_2 \in \text{Proc}, rdv_1, rdv_2 \in \text{Rdv}, d_1, d_2 \in \text{Addr}, c_1, c_2 \in \text{Addr} : \]

\[ I(\text{Send}(p_1, rdv_1, d_1, c_1), \text{Recv}(p_2, rdv_2, d_2, c_2)) \]
(In)Dependence Theorems

Theorem
Any two Send and Recv transitions are independent.

\[ \forall p_1, p_2 \in Proc, rdv_1, rdv_2 \in RdV, d_1, d_2 \in Addr, c_1, c_2 \in Addr : \]
\[ I(\text{Send}(p_1, rdv_1, d_1, c_1), \text{Recv}(p_2, rdv_2, d_2, c_2)) \]
(In)Dependence Theorems

Theorem

Any two Send and Recv transitions are independent.

∀p₁, p₂ ∈ Proc, rdv₁, rdv₂ ∈ RdV, d₁, d₂ ∈ Addr, c₁, c₂ ∈ Addr :
I(Send(p₁, rdv₁, d₁, c₁), Recv(p₂, rdv₂, d₂, c₂))
(In)Dependence Theorems

Theorem

Any two Send andRecv transitions are independent.

\[
\forall p_1, p_2 \in Proc, rdv_1, rdv_2 \in RdV, d_1, d_2 \in Addr, c_1, c_2 \in Addr : \\
I(Send(p_1, rdv_1, d_1, c_1), Recv(p_2, rdv_2, d_2, c_2))
\]
The Predicate $I$

$$I(t_i, t_j) \triangleq \begin{align*}
&\lor \quad t_i = \text{Send}(-, -, -, -) \land t_j = \text{Recv}(-, -, -, -) \\
&\lor \quad t_i = \text{Send}(p_1, rdv_1, -, -) \land t_j = \text{Send}(p_2, rdv_2, -, -) \\
&\quad \quad \land p_1 \neq p_2 \land rdv_1 \neq rdv_2 \\
&\lor \quad t_i = \text{Recv}(p_1, rdv_1, -, -) \land t_j = \text{Recv}(p_2, rdv_2, -, -) \\
&\quad \quad \land p_1 \neq p_2 \land rdv_1 \neq rdv_2 \\
&\lor \quad t_i = \text{WaitAny}(-, \{c\}) \land t_j = \text{WaitAny}(-, \{c\}) \\
&\lor \quad t_i = \text{Test}(-, c, -) \land t_j = \text{Test}(-, c, -) \\
&\lor \quad t_i = \text{WaitAny}(-, \{c\}) \land t_j = \text{Test}(-, c, -) \\
&\lor \quad t_i = \text{Local}(\_) \land t_j = \text{Local}(\_)
\end{align*}$$
SimGrid’s Internals

SimDag

MSG

SMPI

SMURF
SimIX network proxy

SimIX
"POSIX–like" API on a virtual platform

SURF
virtual platform simulator

XBT

GRAS in real world
Experiment 1

Server code

```c
count = 0
while (count<3) {
    value = receive_from("M")
    count++
}
assert( value == 3 )
```

Code of clients 1..3

```c
client(int ID) {
    // do something
    if(done){
        send_to(ID, "M")
    }
}
```

Number of explored traces:
- Classic DFS: 20064
- With DPOR: 14778

First Counter-example

```
[server]  Recv
[client1]  Send
[client3]  Send
[client2]  Send
[server]  Wait
[client1]  Wait
[server]  Recv
[client3]  Wait
[server]  Wait
[server]  Recv
[client2]  Wait
[server]  Wait
```

Experiment 2

--- Server code ---

```java
// step 1
val1 = receive_from("M")
val2 = receive_from("M")
assert(min(val1, val2) == 1)
// step 2
val1 = receive_from("M")
val2 = receive_from("M")
assert(min(val1, val2) == 1)
```

--- Code of client 1 ---

```java
send_to(1,"M") // step 1
send_to(1,"M") // step 2
```

--- Code of client 2 ---

```java
send_to(2,"M") // step 1
send_to(2,"M") // step 2
```

--- Second Counter-example ---

```
[server] Recv
[client1] Send
[server] Wait
[client1] Wait
[server] Recv
[client1] Send
[client2] Send
[server] Wait
[client1] Wait
[server] Recv
[client2] Wait
[server] Wait
[client2] Send
...```

Number of explored traces:

- Classic DFS: 96420
- With DPOR: 32268
Conclusion and Future Work

Conclusion:

► Model-checking actual distributed programs is a current challenges in verification.
► It is feasible only if reduction algorithms such as DPOR are employed.
► Realistic communication APIs for distributed programming are large and complex.
► We propose instead to identify a small set of core primitives that are sufficiently expressive for encoding realistic APIs, with a formal semantic.
► Implementation in the SimGrid framework.

Future Work:

► Validate our approach over larger distributed programs
► Extend it to cover more complex properties, including liveness properties