

**FORMAL METHODS IN NETWORKING**  
**COMPUTER SCIENCE 598D, SPRING 2010**  
**PRINCETON UNIVERSITY**

**LIGHTWEIGHT MODELING**  
**IN PROMELA/SPIN AND ALLOY**

*Pamela Zave*

*AT&T Laboratories—Research*

*Florham Park, New Jersey, USA*

# SIP VERSION 6

```
bool user1mod = false;  
bool user2mod = false;
```

```
proctype user1 (chan in, out) {  
    ....  
    confirmed: do  
        :: in?invite; out!accept  
        :: in?bye; out!byeAck; goto end  
        :: out!invite; goto relnving  
        :: user1mod; .....  
        out!bye; goto Byeing  
    od;  
    relnving: do  
        :: in?invite; out!race  
        :: in?accept; user1mod = true;  
        goto confirmed  
        :: in?race; goto confirmed  
        :: in?bye; out!byeAck; goto end  
    od;  
    Byeing: do  
        :: in?invite  
        :: in?bye; out!byeAck  
        :: in?byeAck; goto end  
    od;  
end: assert(user1mod && user2mod) }
```

**DOMAIN ASSUMPTION:** a user process does not end the session until it has modified the session at least once

```
proctype user2 (chan in, out) {  
    ....  
    confirmed: do  
        :: in?invite; out!accept  
        :: in?bye; out!byeAck; goto end  
        :: out!invite; goto relnving  
        :: user2mod;  
        out!bye; goto Byeing  
    od;  
    relnving: do  
        :: in?invite; out!race  
        :: in?accept; user2mod = true;  
        goto confirmed  
        :: in?race; goto confirmed  
        :: in?bye; out!byeAck; goto end  
    od;  
    Byeing: do  
        :: in?invite  
        :: in?bye; out!byeAck  
        :: in?byeAck; goto end  
    od;  
end: assert(user1mod && user2mod) }
```

**REQUIREMENT:** in every end state, each user has modified the session at least once

# SIP VERSION 6

```
bool user1mod = false;
bool user2mod = false;
```

```
proctype user1 (chan in, out) {
    ...
    confirmed: do
        :: in?invite; out!accept
        :: in?bye; out!byeAck; goto end
        :: out!invite; goto relnving
        :: user1mod;
        out!bye; goto Byeing
    od;
    relnving: do
        :: in?invite; out!race
        :: in?accept; user1mod = true;
        goto confirmed
        :: in?race; goto confirmed
        :: in?bye; out!byeAck; goto end
    od;
    Byeing: do
        :: in?invite
        :: in?bye; out!byeAck
        :: in?byeAck; goto end
    od;
end: assert(user1mod && user2mod) }
```

**DOMAIN ASSUMPTION: a user process does not end the session until it has modified the session at least once**

```
proctype user2 (chan in, out) {
    ...
    confirmed: do
        :: in?invite; out!accept
        :: in?bye; out!byeAck; goto end
        :: out!invite; goto relnving
        :: user2mod;
        out!bye; goto Byeing
    od;
    relnving: do
        :: in?invite; out!race
        :: in?accept; user2mod = true;
        goto confirmed
        :: in?race; goto confirmed
        :: in?bye; out!byeAck; goto end
    od;
    Byeing: do
        :: in?invite
        :: in?bye; out!byeAck
        :: in?byeAck; goto end
    od;
end: assert(user1mod && user2mod) }
```

**the assumption is not sufficient, because either user can end the session unilaterally, and the other user may not have acted yet**

# SIP VERSION 7

```
bool user1mod = false;  
bool user2mod = false;
```

this version solves the  
problem by strengthening  
the domain assumption

they are used only to  
check that the  
specification satisfies a  
conditional requirement,  
so they will not be  
implemented!

these are global history variables—not  
easily implemented in a distributed system

```
proctype user1 (chan in, out) {  
    . . .  
    confirmed: do  
        :: in?invite; out!accept  
        :: in?bye; out!byeAck; goto end  
        :: out!invite; goto relnving  
        :: user1mod && user2mod;  
        out!bye; goto Byeing  
    od;  
    relnving: do  
        :: in?invite; out!race  
        :: in?accept; user1mod = true;  
        goto confirmed  
        :: in?race; goto confirmed  
        :: in?bye; out!byeAck; goto end  
    od;  
    Byeing: do  
        :: in?invite  
        :: in?bye; out!byeAck  
        :: in?byeAck; goto end  
    od;  
end:    assert(user1mod && user2mod) }
```

```
proctype user2 (chan in, out) {  
    . . .  
    confirmed: do  
        :: in?invite; out!accept  
        :: in?bye; out!byeAck; goto end  
        :: out!invite; goto relnving  
        :: user2mod && user1mod;  
        out!bye; goto Byeing  
    od;  
    relnving: do  
        :: in?invite; out!race  
        :: in?accept; user2mod = true;  
        goto confirmed  
        :: in?race; goto confirmed  
        :: in?bye; out!byeAck; goto end  
    od;  
    Byeing: do  
        :: in?invite  
        :: in?bye; out!byeAck  
        :: in?byeAck; goto end  
    od;  
end:    assert(user1mod && user2mod) }
```

# LIGHTNING OVERVIEW OF LINEAR-TIME TEMPORAL

## LOGIC (LTL)

LTL IS A LOGIC, I.E., A LANGUAGE OF TRUTH-VALUED FORMULAS

THE TRUTH OF AN LTL FORMULA IS DEFINED WITH RESPECT TO A STATE SEQUENCE (TRACE)

$P$  state predicate *not temporal*

$P ? \dots$  (P true of first state in trace)

$P U Q$  P until Q

$Q ? \dots, P \dots, P P P Q ? \dots$  (weak)  
 $Q ? \dots, P P P Q ? \dots$  (strong)

$\square P$  always P *invariance*

$P \dots$  (P is true of every state in trace)

$\diamond P$  eventually P *guarantee*

$??? P ? \dots, P ? \dots$  (P is true of at least one state)

$\square \diamond P$  always eventually P *recurrence*

$??? P ? P ? P ? \dots$  (in every state, eventually P)

$\diamond \square P$  eventually always P *stability*

$??? P \dots$  (eventually, P becomes invariantly true)

$$! \square P \Leftrightarrow \diamond ! P$$

$$! \diamond P \Leftrightarrow \square ! P$$

# LTL AND SPIN

(LTL IS THE UNDERLYING MATHEMATICS OF SPIN)

## "SAFETY" PROPERTY

- usually, an invariance
- falsifiable by a finite trace prefix

## "LIVENESS" OR "PROGRESS" PROPERTY

- contains a guarantee
- not falsifiable by a finite trace prefix

*note: all hard real-time properties  
are safety properties*

## DEFAULT CHECKING IN SPIN

- specific invariances
- invalid end state:  
 ! (terminal state &&  
process not in "end")
- assertion violation:  
 ! (program counter at  
assertion && assertion  
not true in current state)
- requirement in SIP Versions 6  
and 7 is a safety property, is not  
good enough because a user  
process could be starved forever

## LTL CHECKING IN SPIN

- can check any temporal formula,  
including progress properties
- the SIP requirements we really want  
are:  
 (user1tried  $\rightarrow$   $\diamond$  user1mod)  
 (user2tried  $\rightarrow$   $\diamond$  user2mod)

# SIP VERSION 8

SIP guarantees a response to the caller (user1) by giving caller static priority

```
proctype user1 (chan in, out) {  
    . . .  
    confirmed: do  
        :: in?invite; out!accept  
        :: in?bye; out!byeAck; goto end  
        :: out!invite; user1tried = true;  
        goto relnving  
        :: user1mod && user2mod;  
        out!bye; goto Byeing  
    od;  
    relnving: do  
        :: in?invite; out!race  
        :: in?accept; user1mod = true;  
        goto confirmed  
        :: in?race; goto confirmed  
        :: in?bye; out!byeAck; goto end  
    od;  
    Byeing: do  
        :: in?invite  
        :: in?bye; out!byeAck  
        :: in?byeAck; goto end  
    od;  
    end: skip }  
}
```

□ (user1tried → ◇ user1mod)  
now holds for all traces

```
proctype user2 (chan in, out) {  
    . . .  
    confirmed: do  
        :: in?invite; out!accept  
        :: in?bye; out!byeAck; goto end  
        :: out!invite; user2tried == true;  
        goto relnving  
        :: user2mod && user1mod;  
        out!bye; goto Byeing  
    od;  
    relnving: do  
        :: in?invite; out!accept  
        :: in?accept; user2mod = true;  
        goto confirmed  
        :: in?race; goto confirmed  
        :: in?bye; out!byeAck; goto end  
    od;  
    Byeing: do  
        :: in?invite  
        :: in?bye; out!byeAck  
        :: in?byeAck; goto end  
    od;  
    end: skip }  
}
```

# SIP VERSION 8

SIP guarantees a response to the caller (user1) by giving caller static priority

```
proctype user1 (chan in, out) {  
    ....  
    confirmed: do  
        :: in?invite; out!accept  
        :: in?bye; out!byeAck; goto end  
        :: out!invite; user1tried = true;  
        goto relnverting  
        :: user1mod && user2mod;  
        out!bye; goto Byeing  
    od;  
    relnverting: do  
        :: in?invite; out!race  
        :: in?accept; user1mod = true;  
        goto confirmed  
        :: in?race; goto confirmed  
        :: in?bye; out!byeAck; goto end  
    od;  
    Byeing: do  
        :: in?invite  
        :: in?bye; out!byeAck  
        :: in?byeAck; goto end  
    od;  
    end: skip }  
}
```

□ (user2tried → ◇ user2mod)  
is not true for all traces, detectable  
by means of a cycle in the  
reachability graph

```
proctype user2 (chan in, out) {  
    ....  
    confirmed: do  
        :: in?invite; out!accept  
        :: in?bye; out!byeAck; goto end  
        :: out!invite; user2tried == true;  
        goto relnverting  
        :: user2mod && user1mod;  
        out!bye; goto Byeing  
    od;  
    relnverting: do  
        :: in?invite; out!accept  
        :: in?accept; user2mod = true;  
        goto confirmed  
        :: in?race; goto confirmed  
        :: in?bye; out!byeAck; goto end  
    od;  
    Byeing: do  
        :: in?invite  
        :: in?bye; out!byeAck  
        :: in?byeAck; goto end  
    od;  
    end: skip }  
}
```

# SIP VERSION 9

SIP implementations  
use timers to achieve  
specified behavior

now both  
processes are  
guaranteed a  
response

```
proctype user1 (chan in, out) {  
    ...  
    confirmed: do  
        :: in?invite; out!accept;  
        user2accepted = true  
        :: in?bye; out!byeAck; goto end  
        :: !user2lost || user2accepted;  
        out!invite; user1tried = true;  
        goto relnving  
        :: user1mod && user2mod;  
        out!bye; goto Byeing  
    od;  
    relnving: do  
        :: in?invite; out!race;  
        user2lost = true  
        :: in?accept; user1mod = true;  
        goto confirmed  
        :: in?race; goto confirmed  
        :: in?bye; out!byeAck; goto end  
    od;  
    Byeing: do  
        :: in?invite  
        :: in?bye; out!byeAck  
        :: in?byeAck; goto end  
    od;  
end: skip }
```

now user1 lets  
user2 in if it has  
lost a race

```
proctype user2 (chan in, out) {  
    ...  
    confirmed: do  
        :: in?invite; out!accept  
        :: in?bye; out!byeAck; goto end  
        :: out!invite; user2tried == true;  
        goto relnving  
        :: user2mod && user1mod;  
        out!bye; goto Byeing  
    od;  
    relnving: do  
        :: in?invite; out!accept  
        :: in?accept; user2mod = true;  
        goto confirmed  
        :: in?race; goto confirmed  
        :: in?bye; out!byeAck; goto end  
    od;  
    Byeing: do  
        :: in?invite  
        :: in?bye; out!byeAck  
        :: in?byeAck; goto end  
    od;  
end: skip }
```

# OTHER SPIN OPTIONS

## SEARCH

- default search (traversal of reachability graph) is depth-first
- can search breadth-first
- can limit depth of search

*there is a default of 10K,  
so you may have to increase limit*

## MEMORY—USUALLY THE SCARCEST RESOURCE

- default is 128 Mb
- can increase it by factors of 2
- compression saves memory with modest cost in time
- supertrace saves a lot of memory, but search is no longer complete

*visited states are stored in a hash table, where multiple states may be indistinguishable*

## FEATURES I HAVE LITTLE USE FOR

- random or manual simulation mode (simulation guided by an error trail is essential!)
- turning off partial order reduction (an optimization that appears to have no disadvantages)
- weak fairness

*probably too weak to make your model run correctly*

*how does an implementor implement a system whose specification is only correct with fairness built in?*

*strong fairness might make your model run correctly, but it is too expensive for Spin to offer*

# TALES OF SIP (THE SESSION INITIATION PROTOCOL)

## SIP IS THE DOMINANT SIGNALING PROTOCOL FOR IP-BASED MULTIMEDIA APPLICATIONS

- telecommunications

*voice-over-IP  
video chat  
large-scale conferencing  
telemonitoring*

- computer-supported cooperative work

*embedded telecommunications  
distance learning  
emergency services  
virtual reality*

- computer-supported cooperative play

*multiplayer games  
collaborative television  
networked music performance*

## SIP IS STANDARDIZED BY THE INTERNET ENGINEERING TASK FORCE (IETF)

- IETF philosophy is to standardize based on "rough consensus and working code"

- in the IETF, a finite-state machine is exotic

- IETF culture supports ignoring "corner cases"

*a corner case is an unanticipated and undesirable situation, which is declared to be rare without evidence*

- the IETF is dominated by equipment manufacturers, who do not want standards

*they standardize only under pressure from their customers, and participate in the IETF as a highly competitive game*

# TALES OF SIP: THE PROTOCOL SPECIFICATION

SIP HAS BEEN, AND IS BEING, DEFINED BOTTOM-UP IN RESPONSE TO AN ENDLESS SERIES OF NEW USE CASES

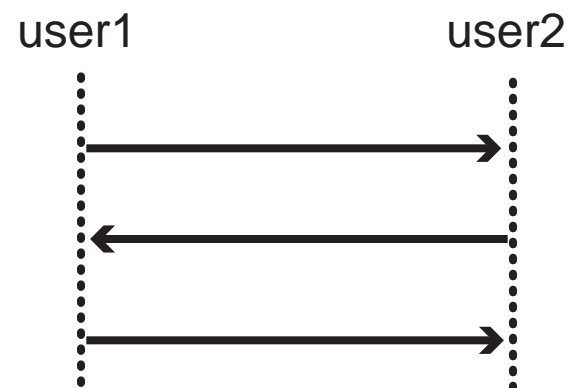
- the base document (IETF RFC 3261) is 268 pages
- "A Hitchhiker's Guide to SIP" is a snapshot of SIP RFCs and drafts . . . . .  
 . . . which lists 142 documents, totaling many thousands of pages
- everyone wants "simple SIP", and everyone has a different idea of what it should be
- opinions are based on a false opposition between generality and simplicity

***no conception  
that a protocol based on better  
abstractions could be both  
more general and simpler***

- message overhead is too high

## THE DOCUMENTS

- written in English, augmented only by message sequence charts that look like this (IETF macros):



***compare these to the charts  
generated by SpinÑ  
these are inviting, almost  
forcing, you to think that  
network communication  
is instantaneous!***

- not surprisingly , the standard is incomplete, inconsistent, or ambiguous in places

# TALES OF SIP: USING PROMELA/SPIN

## MODELING

- we have a collection of SIP models
- we are gradually increasing their scope (bigger subsets of protocol, endpoint/server configurations)

## UNDERSTANDING SIP

- models show what an endpoint must do to use and interpret the protocol correctly—this is far more complicated than previously understood
- on TCP vs. UDP: with non-FIFO communication, the reachability graph is 100 times the size of the FIFO reachability graph
- an RFC documents 7 race conditions—our model reveals those and 49 others of the same type

## DOCUMENTING SIP

- we annotate our models with pointers to the relevant sections of RFCs
- as documentation, our models are guaranteed to be complete, consistent, and unambiguous
- also, you know where to find the answer to your question!

## OTHER USES OF MODEL CHECKING

- we verify the algorithms in our tools for SIP service creation, e.g., showing that media channels are controlled correctly
- we have modified Spin to generate test cases automatically; then we subject SIP components to thousands of tests with guaranteed coverage

# EVALUATION OF PROMELA/SPIN

## SPIN

- a powerful, industrial-strength tool
- mostly easy to use, with a few bad spots (horrible parser, false negatives in reporting unreachable code)

## PROMELA

- great for temporal modeling and assertions
- great for message channels
- primitive data structures (bool, byte, mtype, int, array)
- primitive data assertions (==, <, <=, >, >= on values)

# A SUGGESTED CLASS PROJECT

## CREATE A MODEL OF TCP

- at approximately the same level of detail as the SIP examples
- for example, SYN, FIN, ACK are message types
- include DATA messages to show when each endpoint can send data
- include nondeterminism representing environment choice, system failure, and concurrency

## ANALYZE THE MODEL

- at approximately the same level of detail as the SIP examples