Combining Sequential and Concurrent Verification – The SMTP Case Study –

Bruno Langenstein, Werner Stephan

German Research Center for Artificial Intelligence (DFKI GmbH)
Saarbrücken, Germany
VERISOFT

Planned for 4 + 4 years
Now: first half of first phase (2 years over)
VERISOFT: Overall Goals

• Pervasive Verification
  – complete verification of real systems tractable
    • “academic system”, automotive, biometric access control, hardware
  – general methods for the organization of stacked verification

• Integration of Verification Techniques (Tools)
  – Interactive Systems
  – Automatic Theorem Proving
  – Special Purpose Systems
Verification of Concurrent Applications

Second Phase

\[ p_1 \parallel … \parallel p_n \]

Spec

Temporal Logic

Dynamic Logic

Hoare Logic

First Phase

Formalization in TLA

Integration of sequential programming

Soundness of rules

Application-Level Concurrency

C0 - Semantics

Hierachy of models

Compiler correctness

System Model

OS

CVM

Compiler

German Research Center for Artificial Intelligence
Sequential Programs

- C0: subset of C in Verisoft
  - sequential flow of control
  - pointers
  - resource limitations (guards)
- Semantic definitions in Isabelle
- Hoare–Logic (VCG) in Isabelle
  - shallow embedding
- Dynamic Logic in VSE
  - \([\alpha]\phi : \text{"If } \alpha \text{ terminates } \phi \text{ holds afterwards."}\)
  - soundness of proof rules (for sequential programs)
  - specification of state transitions by E–Theories
A Hierarchy of Models

Work in progress

- SOS* + Com
- SOS*

UdS:
W. Paul
S. Bogan
E. Alkassar
S. Knapp

DFKI:
W. Stephan
B. Langenstein
A. Nonnengart
G. Rock

- VAMOS + SOS
- VAMOS*/C0
- VAMOS*

+(simple) operating system
C0 program execution
abstraction from scheduler
+kernel: interrupts, scheduler
communicating virtual machines

Application level concurrency
Architecture

C0

client

send-c

mapping

C0

server

send-r

receive-c

mapping

interface

resout[i]

resin[j]

OS

cin[i]

cout[j]

abstract „mediator“ functionality (of OS)
Internal Structure of C0–Components

**Instantiation:** Fixed communication schemes are programmed by using concrete low-level primitives

**Abstraction:** Communication is modelled by abstract „actions“.

- Communication
- $e$-call($\tau_1$, …, $\tau_n$, send–c)
- Correctness proof
- Send-call
- Receive-result
- Wait-answer(x, receive–r)
- Local C0-state
- Shared data
- C0–statements

Institution: German Research Center for Artificial Intelligence
Communication Mechanisms

- External (procedure) calls
  - client-server
- Remote Procedure Calls
  - based on ipc
  - used in the example
- Communication over pipes
- File system
- Other solutions can be added
C0 Statements

- Local state from C0 semantics
  - transformation into small-step semantics

\[
\begin{align*}
\text{eval}(\tau, s_{loc}) = a & \implies \langle \Pi, x := \tau, s_{loc}, s_{ife} \rangle \xrightarrow{1} \langle \Pi, \emptyset, s'_{loc}, s_{ife} \rangle \\
\text{eval}(\epsilon, s_{loc}) = \text{true} & \implies \langle \Pi, \text{if } \epsilon \text{ then } \alpha_1 \text{ else } \alpha_2 \text{ fi } , s_{loc}, s_{ife} \rangle \xrightarrow{1} \\
& \quad \langle \Pi, \alpha_1 , s_{loc}, s_{ife} \rangle ,
\end{align*}
\]

where \( s_{loc}(x/a)s'_{loc} \).
Generic Communication Statements

- Interface state $s_{ife}$ for communication
  - shared variables
  - manipulated by communication statements
  - data type of interface: lists of “messages”
  - empty list means “free”.

\[
\text{enabl} \text{com}(s_{loc}, s_{ife}) \implies \langle < \Pi, \text{com}[\tau, x, u] >, s_{loc}, s_{ife} \rangle \stackrel{1}{\rightarrow} \langle < \Pi, \emptyset >, s'_{loc}, s'_{ife} \rangle;
\]

where

\[
(s_{loc} \sim x \quad s'_{loc} \wedge s_{ife} \sim u \quad s'_{ife} \wedge

(s_{ife}(x), s_{loc}(\bar{y})) \langle \text{evul}(\bar{\tau}, s_{loc}) \rangle s'_{ife}(x), s'_{loc}(\bar{y}) \rangle.
\]
System of Processes for the MTA

- Processes: eMail Queue, SMTP-Server, SMTP-Client
Tasks of the Message Transfer Agent

• Interaction with the eMail client
  – getmail
  – sendmail

• Local storage of eMails

• Communication over the net via SMTP
  – receiving
  – sending
Processes

- Mail Queue:
  - Storage
  - Interface to eMail–Client
- SMTP–Server: Reception of eMails via TCP/IP
- SMTP–Client: Sending eMails via TCP/IP

Why several processes?
- SMTP–Server forks
  ⇒ can handle several incoming connections
- Separation of RPC–Servers and RPC–Clients
- RPC and several TCP/IP connections run concurrently
Communication in the eMail Server

- RPC „Remote Procedure Calls“
  - Similar to local procedure calls
    - procedure name
    - parameters
  - Executed by foreign process
- RPC-Servers
  - eMail Queue
  - TCP/IP
- RPC-Clients
  - SMTP Server
  - SMTP Client
  - eMail Client
Communication in the eMail Server

- **RPC „Remote Procedure Calls“**
  - Similar to local procedure calls
    - procedure name
    - parameters
  - Executed by foreign process
  - **RPC-Servers**
    - eMail Queue
    - TCP/IP
  - **RPC-Clients**
    - SMTP Server
    - SMTP Client
    - eMail Client
Example SMTP-Server

```
Process SMTPServer
    tcpip.listen(smtpPort);
    WHILE true DO
        tcpip.accept();
        FORK
            smtpServerProtocol:init();
            WHILE smtpServerProtocol:running DO
                answer = smtpProtocol:generateAnswer();
                tcpip.send(answer);
                cmd = tcpip.read();
                smtpServerProtocol:handleCommand(cmd);
                IF smtpServerProtocol:mailAvailable() THEN
                    mail = smtpServerProtocol:getMail();
                    eMailQueue.sendMail'(mail)
                FI
            OD;
        OD;
    KROF
ProcessEnd
```
Sequential Parts

- No communication statements
- Internal computations of components (processes)
- State of the sequential part is local
  - local state variables
  - specified as (atomic) operations by E-Theories
    - including pointer structures
    - refined into C0 (sequential) programs
    - thereby simplifying reasoning about “communication free” computations

- Examples in the SMTP-Server
  - parsing SMTP commands incl. e-mail addresses
  - handling of incoming (from TCP/IP) “commands”
Communication

- RPC calls to other components
  - send – wait scheme (in general)
  - local proofs use assumptions about the environment

- Examples
  - initializing the communication (with TCP/IP)
  - requests/answers (from/to TCP/IP)
  - delivering mails to queue (sendmail’
Complete Components

- Sequential flow of control
- But: mixture of sequential computations and communication statements
- Specification in TLA
  - actions for communications
  - E–Theory transitions for sequential parts
Example SMTP-Server

```
Process SMTPServer
    tcpip.listen(smtpPort);
    WHILE true DO
        tcpip.accept();
        FORK
            smtpServerProtocol: init();
            WHILE smtpServerProtocol: running DO
                answer = smtpProtocol: generateAnswer();
                tcpip.send(answer);
                cmd = tcpip.read();
                smtpServerProtocol: handleCommand(cmd);
                IF smtpServerProtocol: mailAvailable() THEN
                    mail = smtpServerProtocol: getMail();
                    eMailBag.send(mail)
                FI
            OD;
        KROF
    OD
ProcessEnd
```
Formalization in VSE

• TLA
  - Temporal Logic
    □ φ: „always φ“; ◊ φ „eventually φ“
  - Systems specified by actions (state transitions)
    • primed and unprimed variables
    • interface to E–Theory refinements by certain actions
  - structured into components in VSE
  - modeling the (sequential) flow of control
    • special form of program counters
Formalization in VSE

• ADT/E–Theory refinement
  - Specification of sequential computations by a combination of Abstract Data Types and Dynamic Logic (DL)
  - $[\alpha]\phi$: „If $\alpha$ terminates $\phi$ holds afterwards.“
  - $\langle\alpha\rangle\phi$: „$\alpha$ terminates with $\phi$.“
  - Refinements lead to proof obligations in DL
    Common subset First Order Logic
    • correctness proofs for sequential programs
**SMTP Server ETheory**

```
ETYPEORY smtpServer
USING ThSsmtpServer

DATA
stage, stage0: tstage = idle; sender, sender0: taddress = addressInvalid ...

PREDICATES
handleCommand: stage, sender, recipients, body, answer, IN chars.list ...

PROCEDURES
PROCEDURE handleCommand
PARAMS stage; sender; recipients; body; cmdLine: IN chars.list
BODY
   handleCommand0(parse(cmdLine))
PROCEDUREEND ...

AXIOMS
stage = start ->
   <handleCommand0(cmdEhlo(host))> stage = ehlo AND answer = ok;
stage = ehlo ->
   <handleCommand0(cmdMail(r))> stage = mail AND sender = r AND answer = ok;
   ...
ETYPEORYEND
```
Processes in TLA

- Abstract program counter
- Initialize program counter (in general for families)

\[ pc = \text{here}; \alpha \]

- TLA actions for each kind of program counter
  Example:

\[
\begin{align*}
\text{while} \\
\left( \begin{array}{l}
\text{pc} = \alpha (\text{while } \varepsilon \text{ do } \beta; \text{ here od}) \\
\varepsilon \to \text{pc}' = \alpha (\text{while } \varepsilon \text{ do here; } \beta \text{ od}) \\
\land \neg \varepsilon \to \text{pc}' = \alpha (\text{while } \varepsilon \text{ do } \beta \text{ od; here})
\end{array} \right)
\end{align*}
\]
Processes in TLA (Another Example)

- RPC call
  - Send call to server
    \[
    \begin{aligned}
    p_c &= \alpha(\text{here}; p.f(x; y)) \\
    &\rightarrow \\
    p'_c &= \alpha(\text{blocked}; y = p.f(x)) \\
    \land cout'[pid] &= msg.p.f(x)
    \end{aligned}
    \]
  
  - Get result
    \[
    \begin{aligned}
    p_c &= \alpha(\text{blocked}; y = p.f(x)) \\
    \land res[pid] &= \langle res.p.f(v) \rangle \\
    p'_c &= \alpha(y = p.f(x); \text{here}) \\
    &\rightarrow \\
    res'[pid] &= \langle \rangle \\
    \land y' &= v
    \end{aligned}
    \]
Call to Sequential Procedures in TLA

- E–Theory operations as specified in DL are exported as first order predicates.
  - with additional places for the primed variables (results)
  - example: handling of commands

\[ \square pc[pid] = \alpha(here; z = p : f(x)) \rightarrow \left( \begin{array}{c}
  pc'[pid] = (y = p : f(x); here) \\
  \land \\
  f_p(s, s', x[pid], z)
\end{array} \right) \]

where \( f_p \) is the predicate belonging to \( p: f \) and \( s \) are the state variables of the E–Theory \( p \).
Properties of SMTP Server and Client

- The SMTP Client only sends mails it has received via getmail:

\[
\exists \ cch, crh : \\
H(\text{cout}[\text{smtpclient.pid}], cch) \land H(\text{resin}[\text{smtpclient.pid}], crh) \\
\land \ \Box \text{mailsent}(cch, srh, m) \rightarrow \text{res.fMailQueue.getmail}(m) \in crh
\]

- If the SMTP Server receives a mail via SMTP it will eventually forward this mail:

\[
\exists \ sch, srh : \\
H(\text{cout}[\text{smtpserver.pid}], sch) \land H(\text{resin}[\text{smtpserver.pid}], srh) \\
\land \ \Box( \text{mailsent}(<\text{cinvert}(sch, srh), \text{resinvert}(scr, srh), m) \\
\rightarrow \Diamond \text{cout}[\text{smtpserver}] = \text{msg.fMailQueue.sendmail}(m))
\]
Conclusion

The example SMTP exhibits the need for
• structured formal system model on different levels of abstraction
• integration of different deductive approaches grounded in the system model.

The SMTP case study is based on verification techniques developed in Verisoft and VSE:

• C0 verification (including pointer verification)
• verification of concurrent programs