FACS'13, Jiangxi Normal University
Nanchang, China

Compatibility Checking for Asynchronously Communicating Software

Meriem Ouederni **Gwen Salaun** **Tevfik Bultan**

October 29th, 2013

Assoc. Prof. @ Toulouse INP, IRIT, ACADIE team

Communication is Everywhere!

- Software applications and users are widely distributed (via networks)
- Web applications are adopted instead of desktop applications
- Complex client requests are fulfilled thanks to software composition to better meet costs and deadline constraints
- Technology evolution led to SOC, Cyber-Physical and Pervasive Systems

  $\Longrightarrow$ A lot of heterogeneous physical and computational devices or components (peers) are integrated together to implement common functionalities

  $\Longrightarrow$ Checking the interaction of distributed peers is mandatory to ensure correct composition, *i.e.* compatibility checking

Can Peers be REUSED directly?

- Peers are accessible through their public interfaces (signature and interaction protocol)
- Peers interfaces often present mismatches, *e.g.* missing messages, missing parameters, or deadlocks $\Rightarrow$ the reused peers are said incompatible
- Peer incompatibility forbids the successful communication and the built system cannot meet its requirements
- The compatibility is crucial to ensure the correct peer reuse and composition
- But, how software compatibility can be defined and how peer interfaces can be checked?

Some Compatibility Definitions

Two peers are considered compatible if:

- They can at least engage one communication sequence until reaching a global final state (One-Path)
- Their interaction does not deadlock (Deadlock-Freeness)
- All reachable emissions can be received (Unspecified-Receptions)
- They have opposite behaviors (Opposite-Behaviors)
- ...

A Brief Look at Existing Work

- Brand and Zafiropulo claimed the undecidability issue for FSMs interacting asynchronously over unbounded buffers
- De Alfaro and Henzinger introduce an optimistic and pessimistic compatibility for open and closed systems modelled with I/O automata
- Bauer *el al.* came up with undecidable result in the general case due to the buffering mechanism for I/O automata
- Petri Net community yielded solutions for k-bounded systems
- There are restrictions on the upper buffer bound and the number of participants or communication cycles (Bouajjani and Emmi)
- . . .

Contribution

- The compatibility is checked for distributed systems communicating asynchronously over unbounded FIFO buffers
- There is no restriction on the number of participants and communication cycles
- A class of branching synchronizable systems (BSSs) is defined
- A sufficient condition enables to check BSSs
- A generic framework for verifying the compatibility of BSSs (different notions are considered)
- Implementation using process algebra-based tool-box (CADP)

Outline

Labelled Transition System (LTS)

Definition

A peer LTS is a tuple $\mathcal{P} = (S, s^0, \Sigma, T)$ consisting of:

- S: a finite set of states
- $s^0 \in S$: the initial state
- $\Sigma = \Sigma^! \cup \Sigma^? \cup \{\tau\}$: finite alphabet partitioned into a set of send messages, receive messages, and the internal action τ
- $T \subseteq S \times \Sigma \times S$: a transition relation.

- The LTS is very convenient for formal description and verification of service behaviors
- This model can be easily be derived from existing platform languages, *e.g.*, WF, BPEL for Web services

Labelled Transition System (LTS)

Definition

A peer LTS is a tuple $\mathcal{P} = (S, s^0, \Sigma, T)$ consisting of:

- S: a finite set of states
- $s^0 \in S$: the initial state
- $\Sigma = \Sigma^! \cup \Sigma^? \cup \{\tau\}$: finite alphabet partitioned into a set of send messages, receive messages, and the internal action τ
- $T \subseteq S \times \Sigma \times S$: a transition relation.

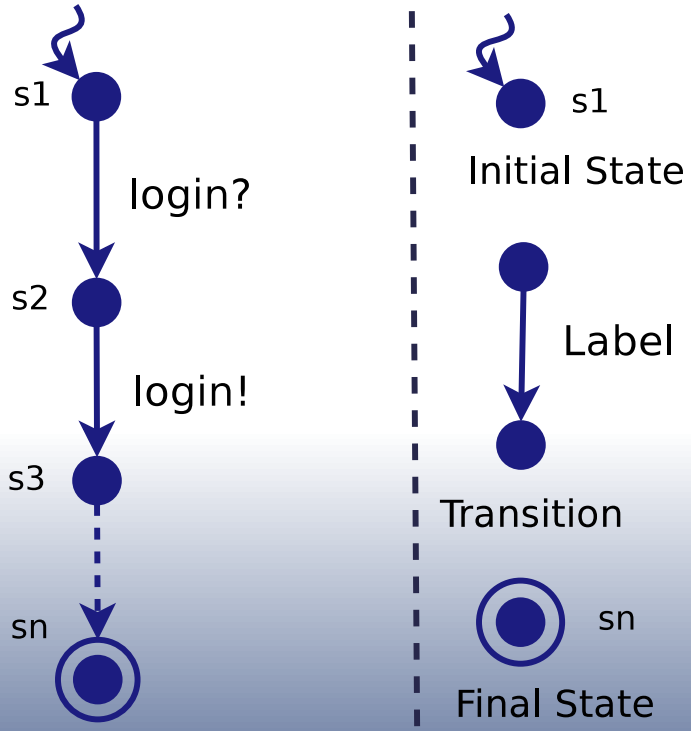
- The LTS is very convenient for formal description and verification of service behaviors
- This model can be easily be derived from existing platform languages, *e.g.*, WF, BPEL for Web services

Labelled Transition System (LTS)

Definition

A peer LTS is a tuple $\mathcal{P} = (S, s^0, \Sigma, T)$

Graphical Presentation



Outline

(A)synchronous Communication

Synchronous Composition

- Communication based on hand shaking
- If one peer is able to send a message there must be a peer in a state expecting that message
- Communicating peers can evolve independently through τ
- $LTS_s = (S_s, s_s^0, \Sigma_s, T_s)$

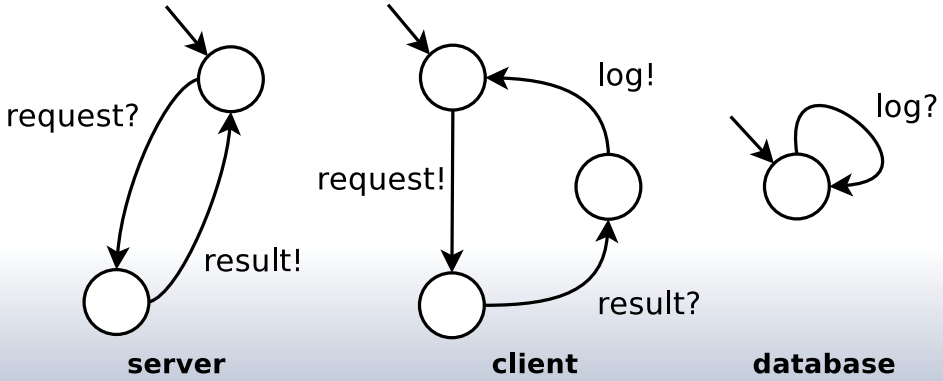
Asynchronous Composition

- Communication based on mailing box
- A peer can send a message to receiving buffer whenever this peer is in a sending state
- A peer $\mathcal{P}_i$ can consume a message if that message is received in its buffer Q_i
- Communicating peers can evolve independently through τ
- $LTS_a = (S_a, s_a^0, \Sigma_a, T_a)$ where $\forall \mathcal{P}_i, Q_i$ is unbounded

Branching Synchronizability

Definition

- A system is synchronizable if and only if its behaviors are identical for asynchronous and synchronous communication
- An **equivalence** relation can be used to check the synchronizability
- In peer composition, the LTS_a might hold branches of τ actions $\Longrightarrow$ Branching equivalence $\equiv_{br}$ is considered



Theorem

$$LTS_s \equiv_{br} LTS_a^1 \Leftrightarrow LTS_s \equiv_{br} LTS_a$$

Well-Formedness

Definition

- A system, LTS_a, is well-formed if and only if every send message is eventually received

Theorems

- LTS_a is well-formed $\Leftrightarrow$ LTS_a^1 is well-formed
- $\forall LTS_a$ composed of $\mathcal{P}_1, \ldots, \mathcal{P}_n$ where LTS_a is synchronizable and $\mathcal{P}_1, \ldots, \mathcal{P}_n$ are deterministic, LTS_a is well-formed

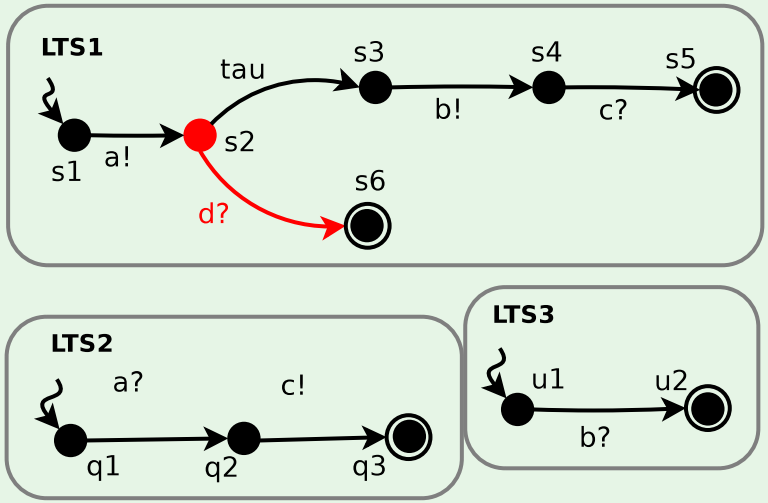
Outline

Synchronous Definition

Unspecified Receptions

Each reachable emission must be received by a peer, and the system must be deadlock-free

Illustration Example



Compatibility Verification

Sufficient Condition

- LTS_a composed of $\mathcal{P}_1, \ldots, \mathcal{P}_n$
- LTS_a is branching synchronizable
- LTS_a is well-formed
- LTS_s is compatible

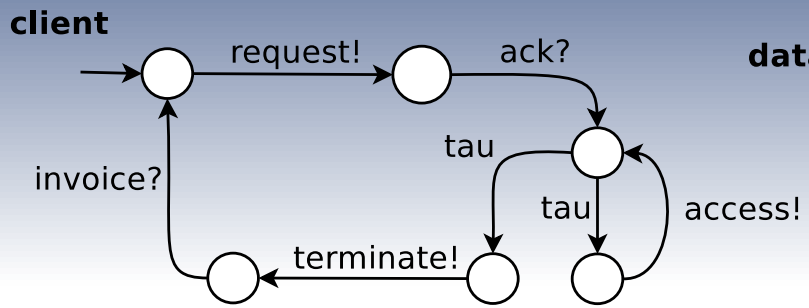
Result

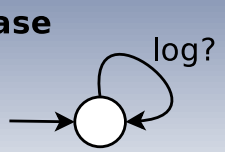
$\Longrightarrow$ LTS_a is asynchronously compatible

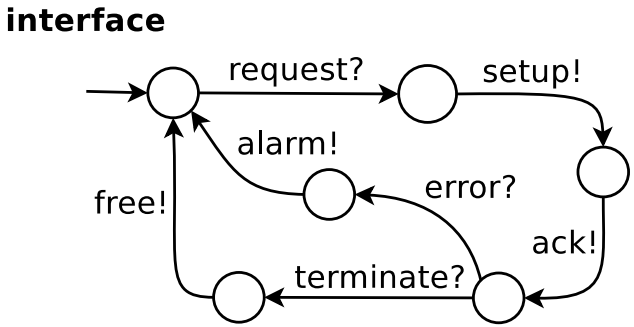
Theorem

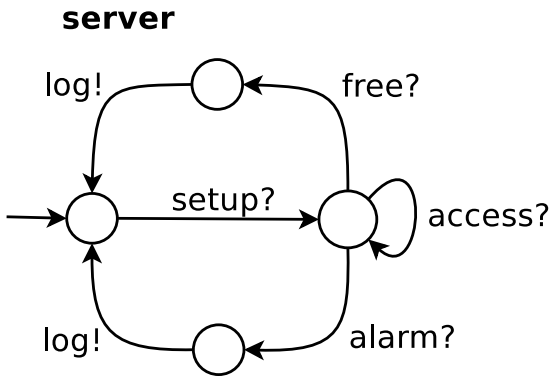
LTS_a is branching synchronizable, well-formed, and synchronously compatible $\Rightarrow$ LTS_a is asynchronously compatible

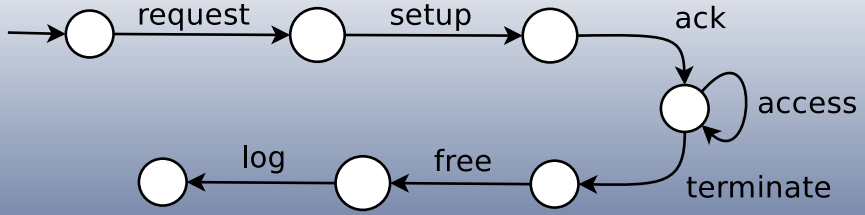
Example

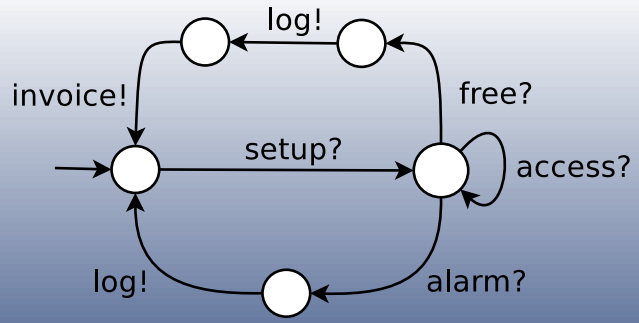

client
request!
ack?
tau
invoice?
tau
access!
terminate!


database
log?


interface
request?
setup!
alarm!
free!
error?
ack!
terminate?


server
log!
free?
setup?
access?
log!
alarm?


request
setup
ack
access
log
free
terminate


log!
invoice!
free?
setup?
access?
log!
alarm?

Outline

Implementation

Automated Verification

- Automatic generation of peers and buffers LTSs in LOTOS (abstract formal language) with python script
- The verification is performed with CADP tool-box
  - LTS_s and LTS_a^1 are automatically computed using LOTOS parallel composition
  - The synchronizability is automatically checked with the branching equivalence between LTS_s and LTS_a^1
  - Automatic checking of compatibility using deadlock-freedom or model checking of MCL properties (LTL, CTL)

Experiments

$\Longrightarrow$ 160 communicating systems (real world and handcrafted examples)
$\Longrightarrow$ 5 out of 96 real-world examples are NOT branching synchronizable

Outline

Conclusions and Perspectives

Model and Semantics

- Distributed peers are described with LTSs and interact asynchronously through unbounded FIFO buffers

Verification Assumptions

- There is no restriction on the number of participants and communication cycles

Verification Result

- Branching synchronizable systems can be analysed under a sufficient condition
- Asynchronous compatibility can be checked using tools and techniques existing for synchronous models

Forward Challenges

Short Term Perspectives

- Formal (mathematical) proofs of branching synchronizability and asynchronous compatibility
- Benchmark and
- Applications of branching synchronizability to realizability issues

Long Term Perspectives

- Enforcement of asynchronous compatibility
- Can the compatibility condition be necessary?
- Is there a class of equivalences for which the compatibility can be decidable?
- Is there a larger class for which the compatibility of asynchronous systems can be checked?

Thank you

- Software engineers want to be real engineers
- Real engineers use mathematics
- Formal methods are the mathematics of software engineering
- Therefore, software engineers should use formal methods

Mike Holloway, NASA