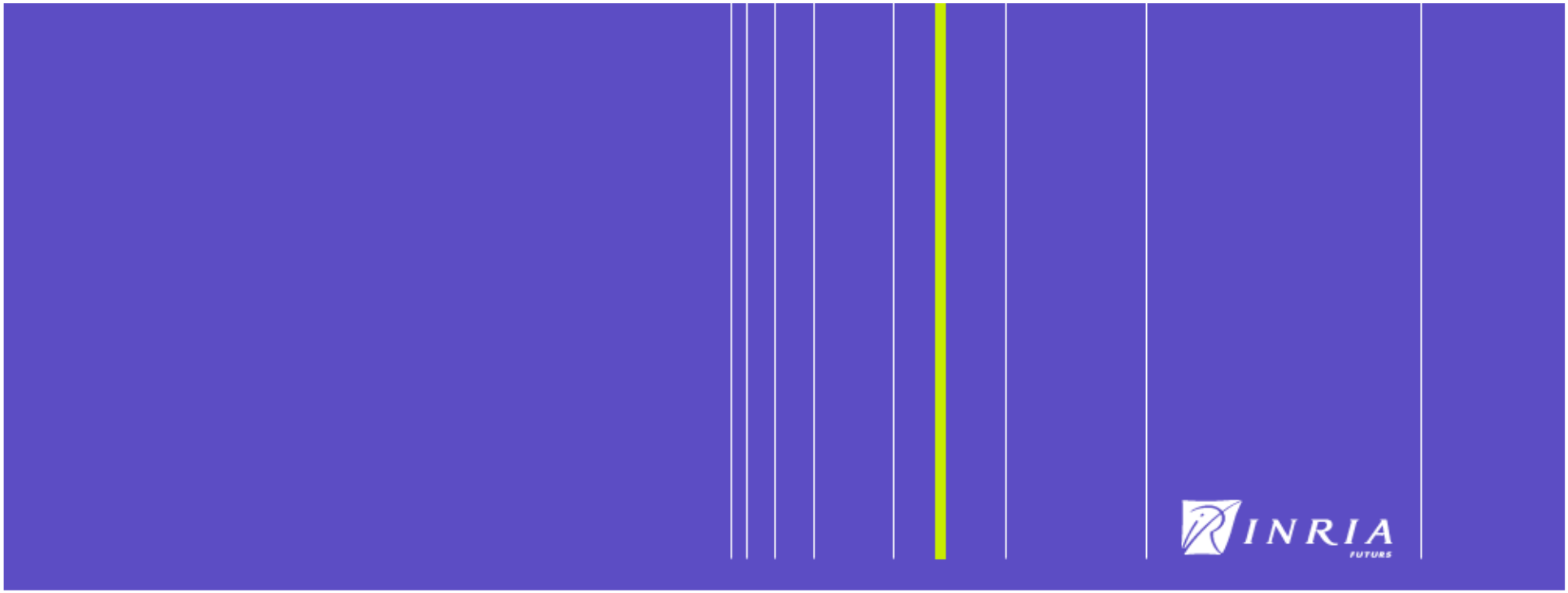
A Stepwise Approach to Developing Languages for SIP Telephony Service Creation

Nicolas Palix*, Charles Consel*, Laurent Réveillère*, Julia Lawall†

* Phoenix Group
LaBRI – INRIA, France
{palix, consel, reveillere}@labri.fr

† DIKU
University of Copenhagen, Denmark
julia@diku.dk



Introduction

- SIP telephony services
- Enriched telephony services
  - Address book
  - Calendar
  - Email
  - CRM
- Legacy technologies
  - General-purpose languages
    - Large and complex low-level platform APIs
    - Error-prone
  - Scripting languages
    - High-level
    - Dedicated to the telephony domain

Issues

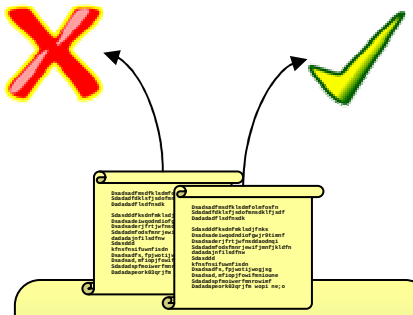
- Legacy technologies
  - Formal semantics unavailable
  - Reference implementation may be unavailable
- Service developers
  - Must study reference implementation
  - Must not corrupt the underlying platform
- Platform developers
  - Difficult to port a language to another run-time system
  - Difficult to provide another implementation of a run-time system

How to design and develop a scripting language dedicated to the development of robust telephony services?

Our Approach

Static semantics

- Type checker
- Service analyses



SPL

SIP Virtual Machine

SIP Platform

Domain Specific Language

- Simple and expressive
- Reflects the SIP VM abstractions
- Formally defined
  - Static semantics
  - Dynamic semantics

Domain Specific SIP Virtual Machine

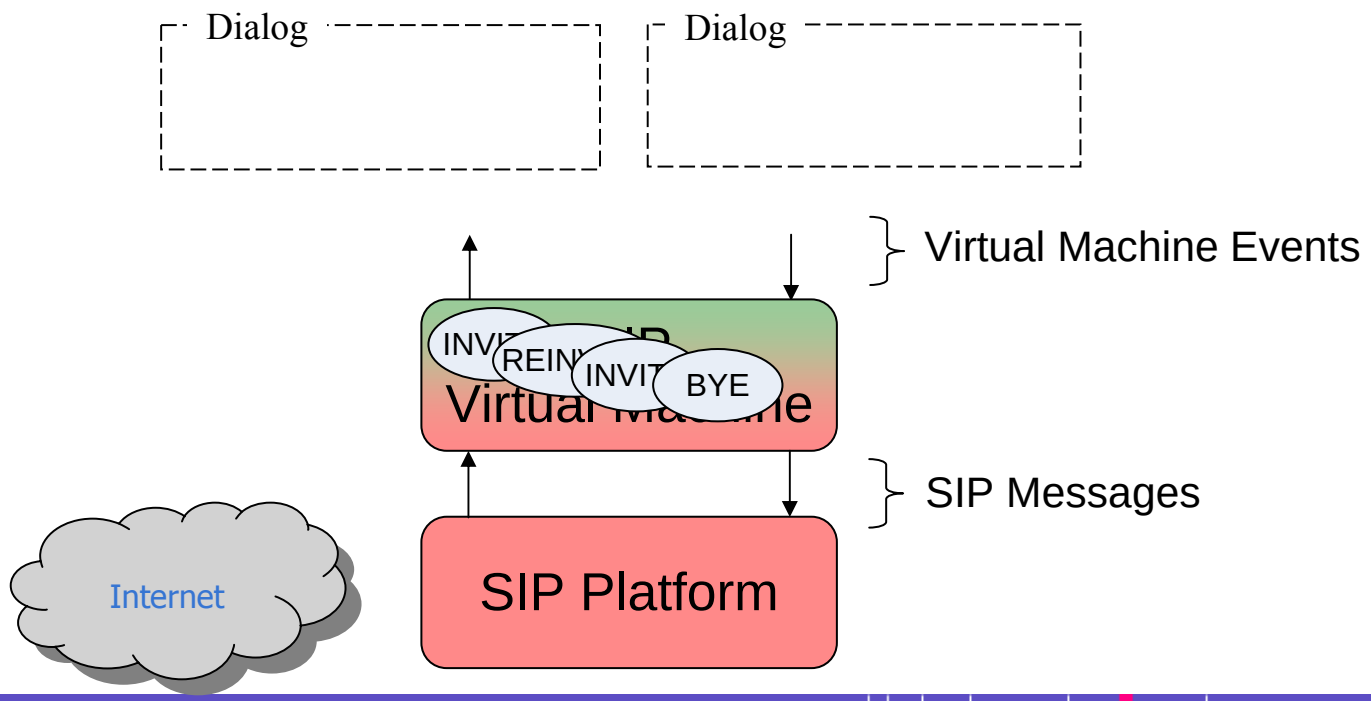
- Raises the abstraction level
- Consists of a set of events and a state
- Focuses on sessions

Dynamic semantics

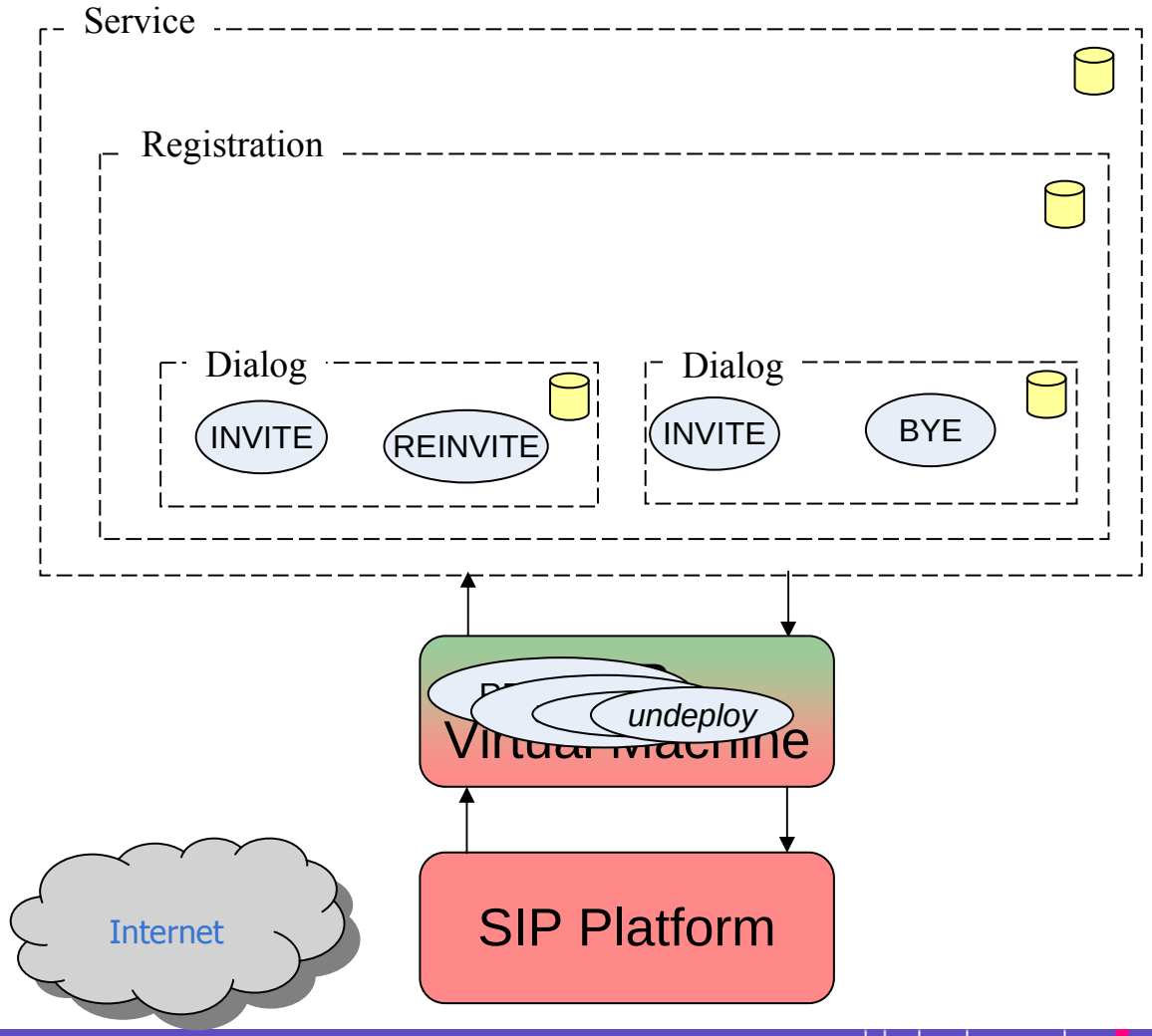
- Defines SPL runtime behavior
- Eases implementation
- Verifies runtime properties

1st Step: A SIP Virtual Machine

- Refines SIP messages in events
- Manages sessions



1st Step: A SIP Virtual Machine



- Refines SIP messages in events
- Manages sessions
- Organizes sessions hierarchically
- Introduces SIP VM events
- Stores and restores service state

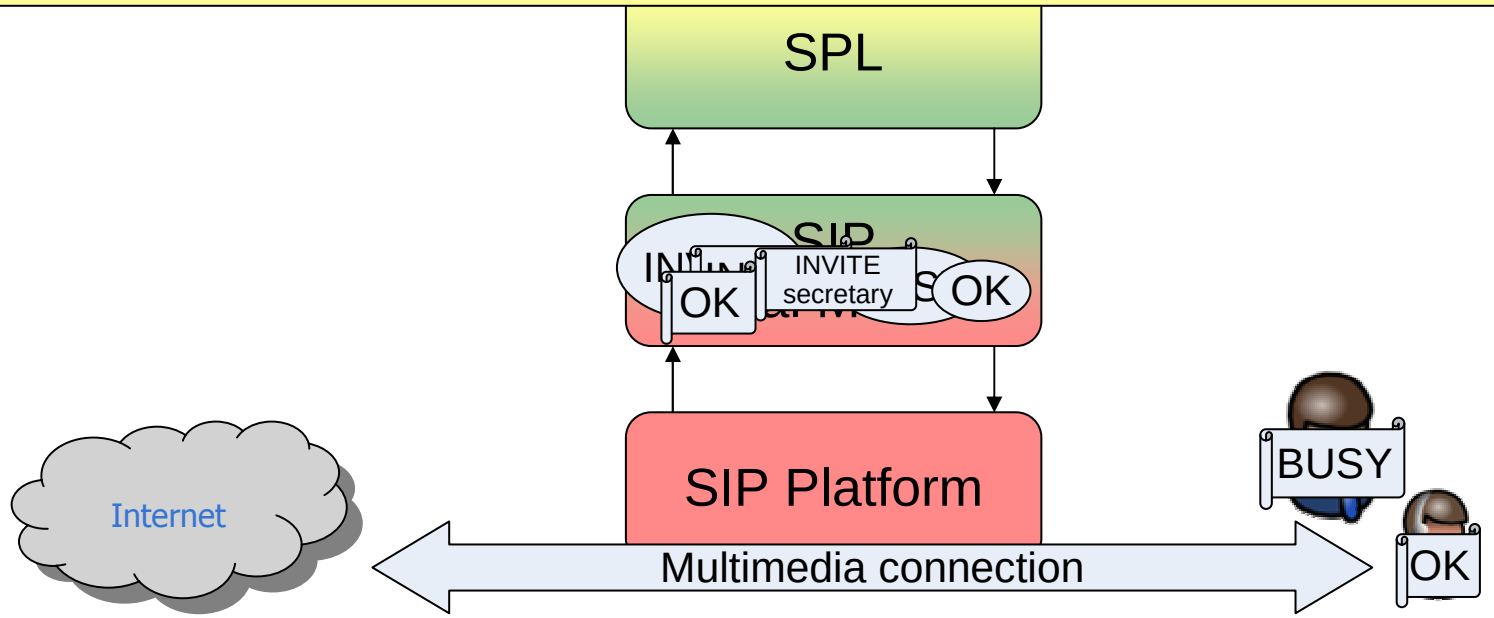
2nd Step: The *Session Processing Language*

```
response incoming INVITE() {
    […]
    response resp = f
    if (resp == /ERROR) {
     resp = f                'sip:phoenix.secretary@inria.fr';
    }
    r n resp;
}
```

INVITE

INVITE secretary

OK

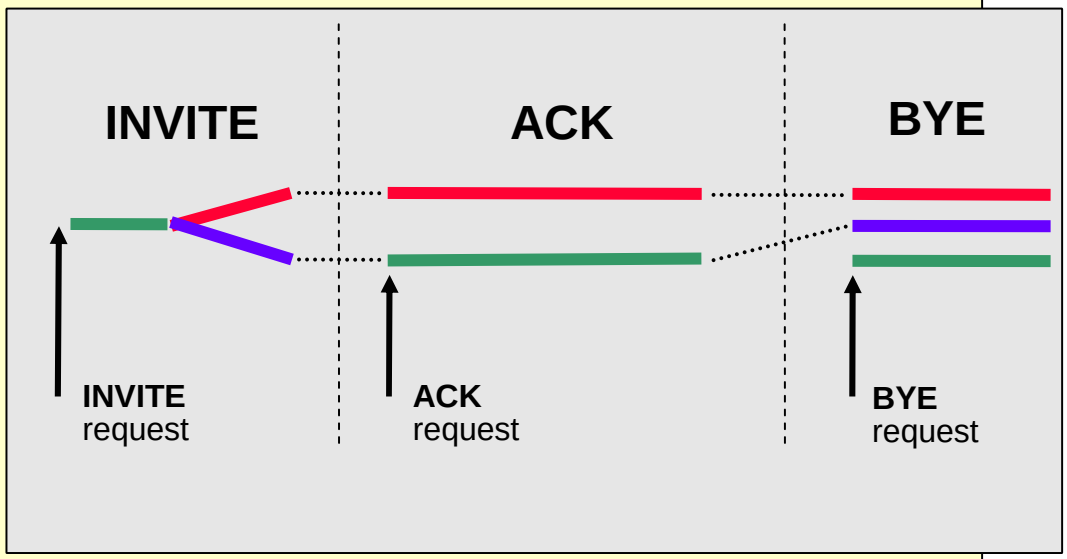


2nd Step: The *Session Processing Language*

```
service hotline {
    ...
    processing {
        uri<100> employees = <>;

        void deploy() {...}
        void undeploy() {...}

        registration {...

            response outgoing REGISTER() {
                uri employee = FROM;
                push employees employee;
                return forward;
            }
            ...

            dialog { ...
                response incoming INVITE() {
                    return forward employees;
                }
            }
        }
    }
}
```

Advanced features: Inter-Event Control Flow

```
dialog {
  response incoming INVITE() {
    response r;
    ...
    if (...) {
      ...
      return r branch hotline;
    }
    else {
      ...
      return r branch personal;
    }
  }

  void incoming ACK(){
    branch hotline {... }
    branch default {... }
  }

  response BYE() {
    branch hotline  {... }
    branch personal {... }
    branch default  {... }
  }
}
```

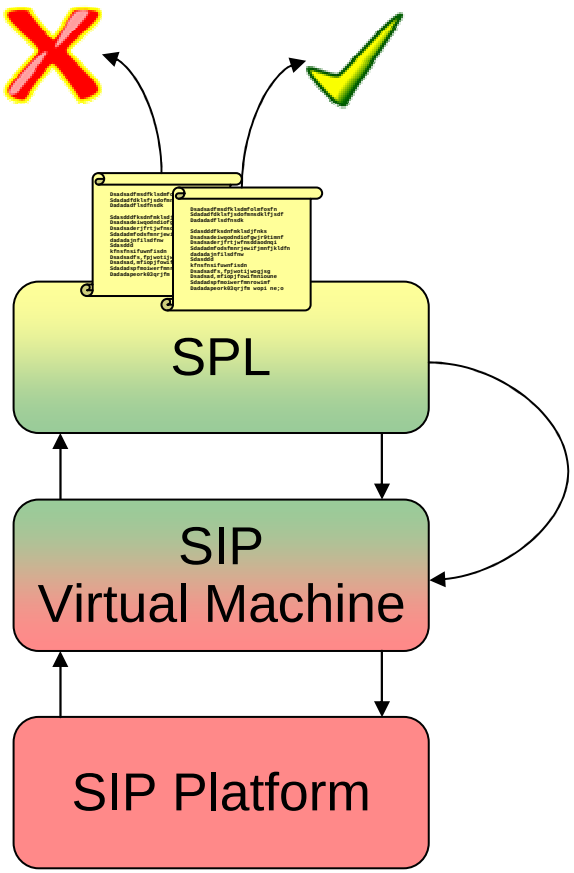


3rd Step: Formal Semantics

Static semantics

How to verify properties which are critical for the telephony domain?

- Type-checking
- Service analyses
  e.g. `forward` use



Dynamic semantics

- Defines SPL runtime behavior
- Eases implementation
- Verifies runtime properties

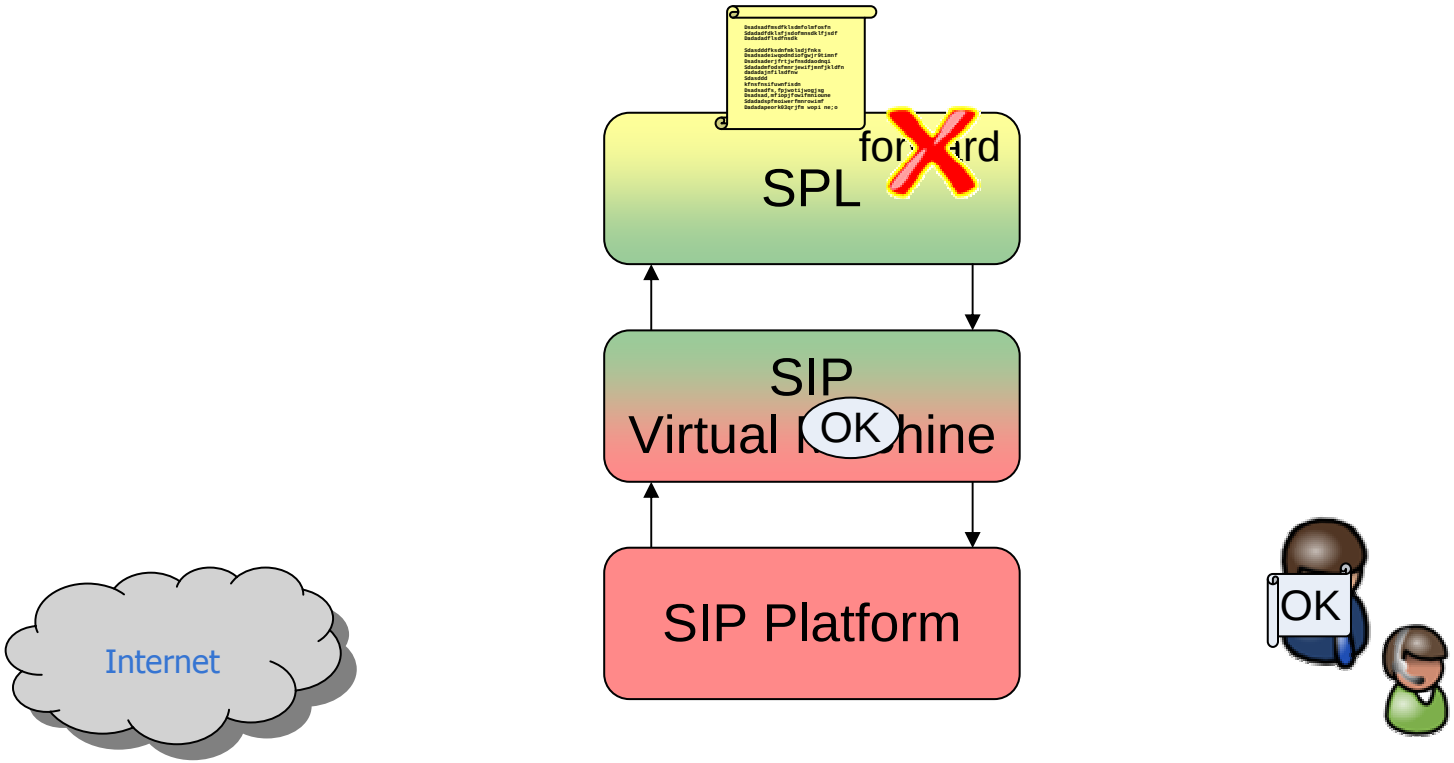
Static semantics: Type-Checking

```
dialog {
    uri caller;
    time start;

    response incoming INVITE() {
        caller = FROM;
        return forward;
    }

    void incoming ACK(){
        if(caller == 'sip:my.wife@home.fr')
            log(“Personal call”);
        start = getTime();
    }

    response BYE() {
        string duration = time_to_string(getTime() – start);
        log(“Call: ”+ duration +” ”+uri_to_string(caller));
        return forward;
    }
}
```

Static semantics: Service Analyses

Example of property to ensure: *forward* use

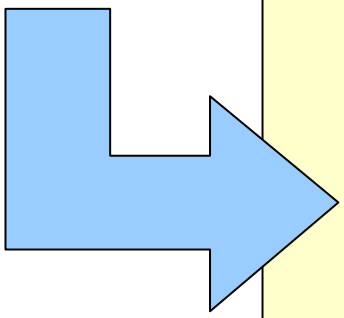
When a positive response is received, no additional forwarding operation is allowed



Service Analyses - `forward` Use Example

Concrete syntax

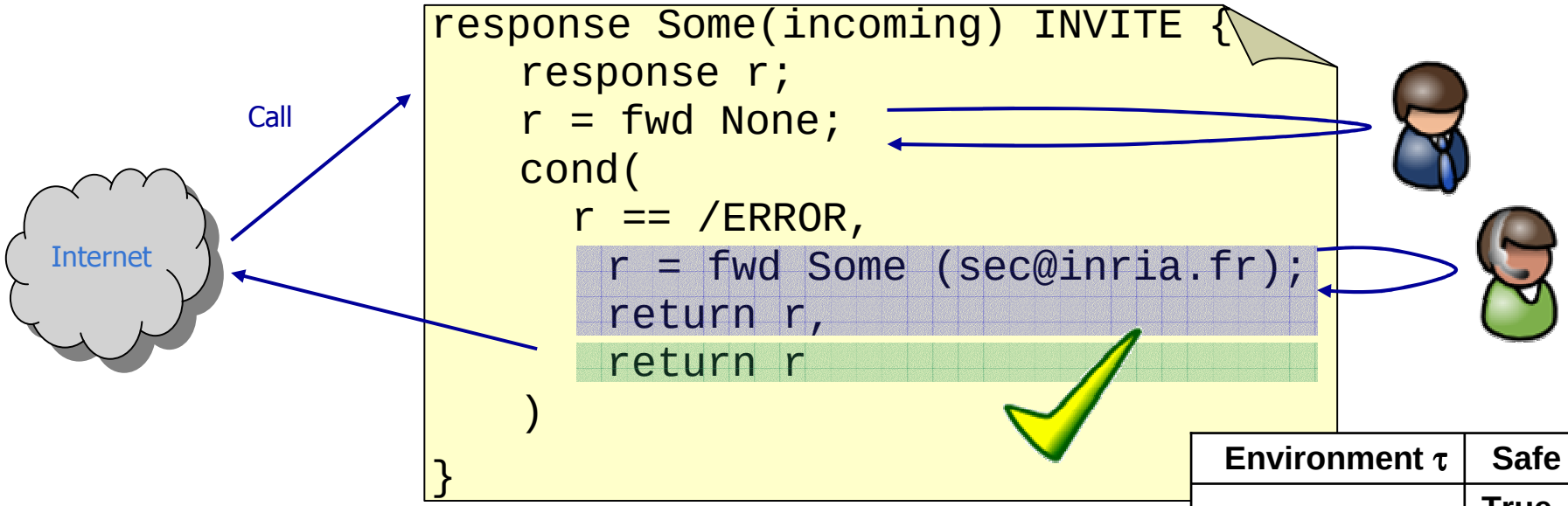
```
response incoming INVITE() {
    […]
    response r = forward;
    if (r == /ERROR) {
      r = forward 'sip:phoenix.secretary@inria.fr';
      return r;
    }else {
      return r;
    }
}
```



```
response Some(incoming) INVITE {
    response r;
    r = fwd None;
    cond(r == /ERROR,
        r = fwd Some (sip:phoenix.secretary@inria.fr);
        return r,
        return r)
}
```

Abstract syntax

`forward` Use Analysis for a Valid Service



Environment τ	Safe
	True
(r, error)	True
(r, ⊥)	True
(r, error)	True
(r, success)	True
(r, ⊥)	True
(r, ⊥)	True
(r, success)	True
(r, ⊥)	True

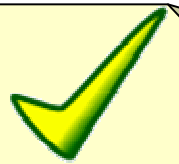
$$\frac{}{\tau \vdash^{D} \texttt{response}\ \ id : \tau[id \mapsto \text{error}]}$$

$$\frac{\forall (x, \sigma) \in \tau.\ \sigma = \text{error} \qquad \tau' = \tau[id \mapsto \bot]}{\tau \vdash^{S} id\ \texttt{=}\ \texttt{fwd}\ \ URI^{?} : \langle \tau', \text{true} \rangle}$$

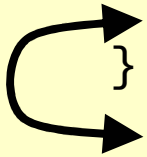
$$\frac{\tau \vdash^{E_B} E_B : (id, \sigma) \qquad \tau[id \mapsto \sigma] \vdash^{S} S_1 : \langle \tau_1, fwd_1 \rangle \qquad \tau[id \mapsto \neg\sigma] \vdash^{S} S_2 : \langle \tau_2, fwd_2 \rangle}{\tau \vdash^{S} \texttt{cond}\ (E_B,\ S_1,\ S_2) : \langle \tau_1 \uplus \tau_2, fwd_1 \wedge fwd_2 \rangle}$$

Service Analyses - **`forward`** Use Example

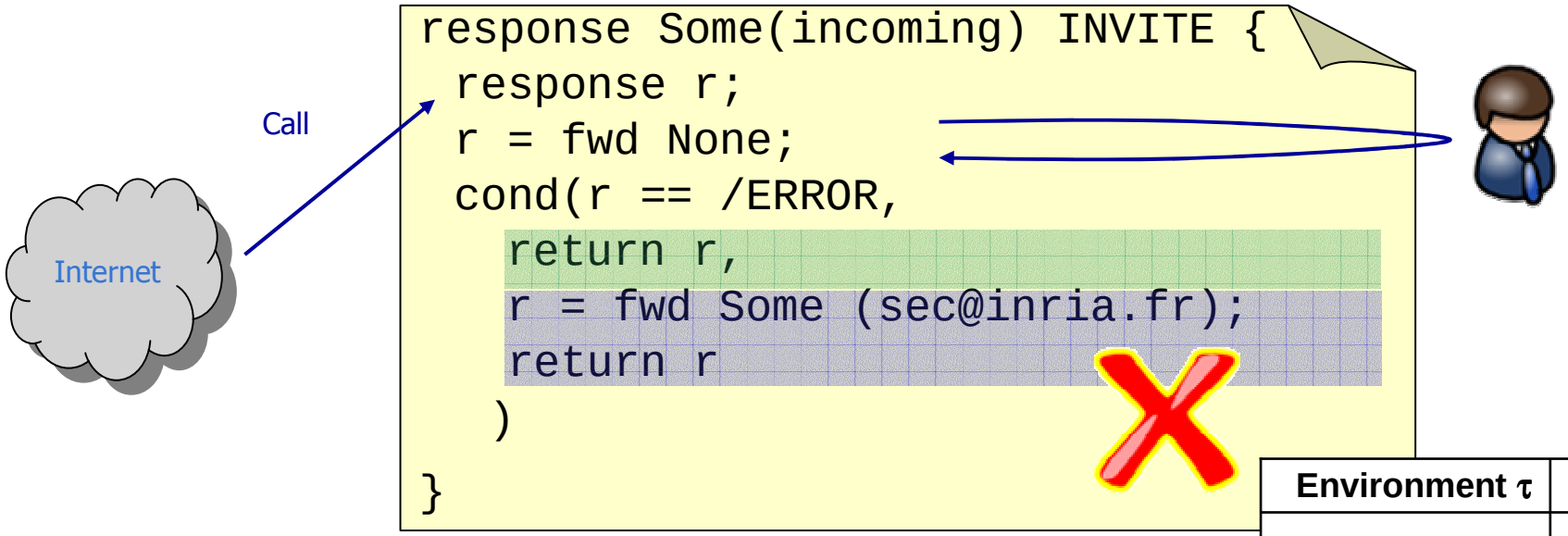
```
response incoming INVITE() {
    […]
    response r = forward;
    if (r == /ERROR) {
      r = forward 'sip:phoenix.secretary@inria.fr';
      return r;
    } else {
      return r;
    }
}
```



```
response incoming INVITE() {
    […]
    response r = forward;
    if (r == /ERROR) {
        return r;
    } else {
      r = forward 'sip:phoenix.secretary@inria.fr';
      return r;
    }
}
```



`forward` Use Analysis for an Invalid Service



$$\frac{\forall(x,\sigma) \in \tau.\ \sigma = \text{error} \qquad \tau' = \tau[id \mapsto \bot]}{\tau \vdash^{S} id \texttt{ = fwd } URI^{?} : \langle \tau', \text{true} \rangle}$$

$$\frac{\exists(x,\sigma) \in \tau.\ \sigma \neq \text{error}}{\tau \vdash^{S} id \texttt{ = fwd } URI^{?} : \langle \tau, \text{false} \rangle}$$

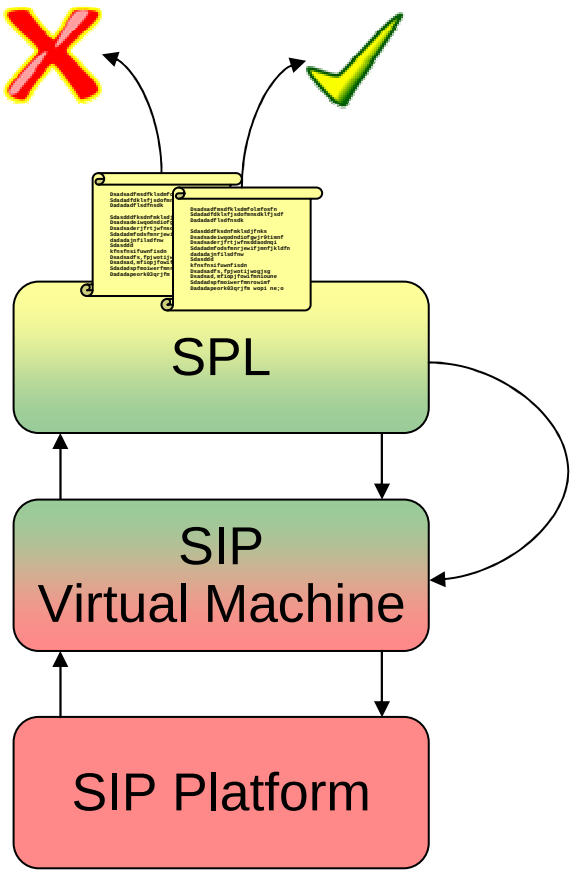
Environment τ	Safe
	True
(r, error)	True
(r, ⊥)	True
(r, error)	True
(r, success)	True
(r, error)	True
(r, success)	False
(r, success)	False
(r, ⊥)	False

3rd Step: Formal Semantics

Static semantics

How to verify properties which are critical for the telephony domain?

- Type-checking
- Service analyses
  e.g. `forward` use



Dynamic semantics

- Defines SPL runtime behavior
- Eases implementation
- Verifies runtime properties

A Direct Interpreter Implementation From The Dynamic Semantics

$$\frac{\begin{array}{c} \mathit{address} = \langle \mathit{service}, \mathit{rid}, \mathit{did} \rangle \\ \mathsf{lookup\_branches}(\sigma, \mathsf{parent}(\mathit{address})) = \langle \mathit{branch} \rangle \\ \mathsf{create\_session}(\phi, \sigma, \mathit{address}, \langle \mathit{branch} \rangle, \texttt{undialog}) = \sigma' \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma', \mathit{address}, \mathit{direction}, \texttt{initial INVITE}) = \langle m, \mathit{decls}, \mathit{stmts}, \mathit{envs}, \sigma'' \rangle \\ \tau = \langle \sigma'', \mathit{address} \rangle \qquad r = \langle \mathit{envs}, \langle m, \langle \mathit{rq}, \mathit{headers} \rangle \rangle \rangle \end{array}}{\begin{array}{r} \langle \texttt{initial INVITE}(\mathit{rid}, \mathit{did}), \mathit{direction}, \mathit{rq}, \mathit{headers} \rangle, \phi, \sigma \models \mathit{service} \\ \Rightarrow \tau, r, \langle \texttt{INITIAL\_INVITE } \phi \rangle \models \mathit{decls}, \mathit{stmts} \end{array}}$$

```
let interpret message phi sigma service =
  match message with
   (I_INVITE(rid, did), direction, rqid) ->
     let address = [service;(Reg, rid);(Dial, did)] in
     let branch = lookup_branches(sigma, parent(address)) in
     let sigma' =
        create_session(phi, sigma, address, branch, Some(If.UNINVITE)) in
     let (m_par, decls, stmts, envs, sigma'') =
        prepare_handler(phi, sigma', address, direction, If.INVITE) in
     let tau = (sigma'', address) in
     let rho = (envs, (m_par, rqid), []) in
       spl_handler_body tau rho ([]::[[T_INITIAL_INVITE(phi)]]) (decls, stmts)
```

Dynamic Semantics

- About 100 semantic rules
- Documentation
  - To understand program behavior
  - To implement the SPL interpreter
  - Language independent
- Straightforward implementation
  - 1 week in OCaml
  - 2 weeks in JAVA

Conclusion

- Stepwise approach
  - Domain-Specific virtual machine
  - Domain-Specific Language
  - Formal semantics
- Benefits
  - High-level language
  - Robust and verifiable services
  - Straightforward implementation
- The Session Processing Language
  - A call queuing service in about 100 lines
  - A SIP application server based on JAIN SIP API

Ongoing Work

- Other SPL properties
  - Feature Interactions
    - Multiple users
    - Multiple services
  - Inter-event control flow reachability
  - Optimizations
- Παντάχου (Pantachou)
  - Composition language for ubiquitous services
  - Rely on SIP network

Thank You For Your Attention !

http://phoenix.labri.fr/software/spl/