Abstraction Preservation and Secure Sessions in Distributed Languages

PhD defense of Pierre-Malo Deniélo

MOSCOVA Project (INRIA) MSR-INRIA Joint Centre

Advisors: Jean-Jacques Lévy and James Leifer

25/01/2009

Pierre-Malo.Denielou@inria.fr
http://moscova.inria.fr/~denielou/these/
Distributed systems

A distributed system
Independent programs that realise a global task through network interactions
Distributed systems

A distributed system

Independent programs that realise a global task through network interactions

They need to agree
- on data semantics
  - Misunderstanding
- on protocols
  - Miscommunication

Alice

Bob

Charlie
Distributed systems

Alice

Independent programs that realise a global task through network interactions

Bob

They need to agree
- on data semantics
  Misunderstanding
- on protocols
  Miscommunication

Charlie
Distributed systems

A distributed system
Independent programs that realise a global task through network interactions

They need to agree
- on data semantics
- Misunderstanding
- on protocols
- Miscommunication

There is little trust
- Errors (Safety)
- Typing system
- Corruption (Security)
- Cryptographic protocol
Distributed systems

Alice

A distributed system
Independent programs that realise a global task through network interactions

Bob

They need to agree
- on data semantics
  - Misunderstanding
- on protocols
  - Miscommunication

Charlie

There is little trust
- Errors (Safety)
  - Typing system
- Corruption (Security)
  - Cryptographic protocol
Distributed systems

Alice

Independent programs that realise a global task through network interactions

Bob

They need to agree
- on data semantics
  Misunderstanding
- on protocols
  Miscommunication

Charlie

There is little trust
- Errors (Safety)
  Typing system
- Corruption (Security)
  Cryptographic protocol
Different from sequential programming

- Independent programs need to cooperate: **safety**.
- Complicated interactive software: easier to **generate/prove** than to program/debug.
- No control over the execution environment (peers, network): **security**.
Improving Distributed Programming

Different from sequential programming

- Independent programs need to cooperate: **safety**.
- Complicated interactive software: easier to **generate/prove** than to program/debug.
- No control over the execution environment (peers, network): **security**.

Most existing tools are not well-suited

- Compilers and type systems are local.
- Security and networking libraries are low-level, binary.
Different from sequential programming

- Independent programs need to cooperate: **safety**.
- Complicated interactive software: easier to **generate/prove** than to program/debug.
- No control over the execution environment (peers, network): **security**.

Most existing tools are not well-suited

- Compilers and type systems are local.
- Security and networking libraries are low-level, binary.

Contribution I: Abstract Type Safety

- How to enforce local semantics in a distributed environment
Improving Distributed Programming

Different from sequential programming

- Independent programs need to cooperate: safety.
- Complicated interactive software: easier to generate/prove than to program/debug.
- No control over the execution environment (peers, network): security.

Most existing tools are not well-suited

- Compilers and type systems are local.
- Security and networking libraries are low-level, binary.

Contribution I: Abstract Type Safety

- How to enforce local semantics in a distributed environment

Contribution II: Session Security

- How to secure a distributed execution despite compromised parties
Improving Distributed Programming

Different from sequential programming
- Independent programs need to cooperate: safety.
- Complicated interactive software: easier to generate/prove than to program/debug.
- No control over the execution environment (peers, network): security.

Most existing tools are not well-suited
- Compilers and type systems are local.
- Security and networking libraries are low-level, binary.

Contribution I: Abstract Type Safety
- How to enforce local semantics in a distributed environment

Contribution II: Session Security
- How to secure a distributed execution despite compromised parties

Computer science = Engineering ∩ Mathematics
- Industrial objects: prototyping
- Experiments and measures:
  experimental method
- Logical objects: mathematical definition
- Theorems and proofs:
  formal method
Part I
Abstraction preservation and subtyping
Abstract type preservation

Alice’s counter

module Counter =
struct
  sig
    type t = int
    let init = 0 : val init : t
    let incr x = x+1 : val incr : t → t
    let value x = x : val value : t → int
  end
end

Bob’s counter

module Counter =
struct
  sig
    type t = int
    let init = 0 : val init : t
    let incr x = x+1 : val incr : t → t
    let decr x = x-1 : val decr : t → t
    let value x = x : val value : t → int
  end
end

Alice ↔ Bob

1. Alice sends Counter.init 0:Counter.t  Bob
Abstract type preservation

Alice's counter

```ocaml
module Counter =  
  struct
      type t = int
      let init = 0 : val init : t
      let incr x = x+1 : val incr : t \rightarrow t
      let value x = x : val value : t \rightarrow int
  end
```

Bob's counter

```ocaml
module Counter =  
  struct
      type t = int
      let init = 0 : val init : t
      let incr x = x+1 : val incr : t \rightarrow t
      let decr x = x-1 : val decr : t \rightarrow t
      let value x = x : val value : t \rightarrow int
  end
```

Alice \(\leftrightarrow\) Bob

1. Alice sends \texttt{Counter.init} 0:Counter.t \(\rightarrow\) Bob
Abstract type preservation

Alice’s counter

module Counter =
struct
  type t = int
  let init = 0 :
  let incr x = x+1
  let value x = x
end

Bob’s counter

module Counter =
struct
  type t = int
  let init = 0 :
  let incr x = x+1
  let decr x = x-1
  let value x = x
end

Alice ↔ Bob

1. Alice sends Counter.init 0:Counter.t → Bob
2. Bob applies Counter.decr
Abstract type preservation

Alice’s counter

```ocaml
module Counter =
  struct
    type t = int
    let init = 0 : val init : t
    let incr x = x+1 : val incr : t → t
    let value x = x : val value : t → int
  end
end
```

Bob’s counter

```ocaml
module Counter =
  struct
    type t = int
    let init = 0 : val init : t
    let incr x = x+1 : val incr : t → t
    let decr x = x-1 : val decr : t → t
    let value x = x : val value : t → int
  end
end
```

Alice ↔ Bob

1. Alice sends `Counter.init 0` → Bob
2. Bob applies `Counter.decr`
3. Alice ← Bob sends the result
Abstract type preservation

Alice’s counter

```ocaml
module Counter = struct
  type t = int
  let init = 0 : val init : t
  let incr x = x+1 : val incr : t → t
  let value x = x : val value : t → int
end
```

Bob’s counter

```ocaml
module Counter = struct
  type t = int
  let init = 0 : val init : t
  let incr x = x+1 : val incr : t → t
  let decr x = x-1 : val decr : t → t
  let value x = x : val value : t → int
end
```

Alice ↔ Bob

1. Alice sends `Counter.init 0` → Bob
2. Bob applies `Counter.decr`
3. Alice ← `1:Counter.t` → Bob sends the result
4. Alice applies `Counter.value`
Abstract type preservation

Alice’s counter

```ocaml
module Counter = struct
  type t = int
  let init = 0 : val init : t
  let incr x = x+1 : val incr : t -> t
  let value x = x : val value : t -> int
end
```

Bob’s counter

```ocaml
module Counter = struct
  type t = int
  let init = 0 : val init : t
  let incr x = x+1 : val incr : t -> t
  let decr x = x-1 : val decr : t -> t
  let value x = x : val value : t -> int
end
```

Alice ↔ Bob

1. Alice sends `Counter.init 0 : Counter.t` → Bob
2. Bob applies `Counter.decr`
3. Alice ← `1:Counter.t` → Bob sends the result
4. Alice applies `Counter.value`
5. Alice fails! (broken invariant)

Abstract types refer to local modules. Type safety requires more than comparing names. Different internal invariants, different concrete types, different dependencies.
Abstract type preservation

Alice's counter

```ocaml
module Counter = struct
  type t = int
  let init = 0 : val init : t
  let incr x = x+1 : val incr : t -> t
  let value x = x : val value : t -> int
end
```

Bob's counter

```ocaml
module Counter = struct
  type t = int
  let init = 0 : val init : t
  let incr x = x+1 : val incr : t -> t
  let decr x = x-1 : val decr : t -> t
  let value x = x : val value : t -> int
end
```

Alice ↔ Bob

1. Alice sends `Counter.init 0 : Counter.t` to Bob
2. Bob applies `Counter.decr` to the received `Counter.t` value
3. Alice receives the result `-1 : Counter.t`
4. Alice applies `Counter.value` to the received `Counter.t` value
5. Alice fails! (broken invariant)

Abstract types refer to local modules.

Type safety requires more than comparing names.

- different internal invariants
- different concrete types
- different dependencies
A solution using hashes and colour brackets

- ... used in Acute (ICFP 2005) and HashCaml (`ML 2006).

Idea: hash the source code of modules

- We use the hash as a unique identifier for each abstract type.
- Thus, the compiler replaces the local type name `Counter.t` by the global `h.t` where `h` is the hash of `Counter` (recursively dealing with dependencies).
- Each change yields a new hash.
- We can easily compare abstract types dynamically at unmarshall time by a simple equality check on hashes. Thus, type errors are detected at the earliest possible moment.
- Coloured brackets are used to track abstract values during evaluation.
Present contributions

Motivation: More flexibility

- We want to exchange values between executables running different versions of modules (upgrades, bug fixes, ...).
- Compatibility after a module upgrade is not necessarily symmetric!

⇒ We model this by a subtyping relation.

Our contributions:

We give a sound semantics for subtyping with hashing, coloured brackets and marshalling.

1. Records and structural subtyping for concrete types
2. User-declared subtyping between abstract types
3. Partial abstract types (bounded existentials)
The invariants of CounterA.t and CounterB.t are different but they are compatible in one direction.

Problem: No way in general to infer the invariant compatibility, thus preventing potentially useful and safe communications. Solution:

Then we’ll only be able to use CounterA.t <: CounterB.t.
Summary (1/2): final semantics

Type system (85 rules)
- Singleton kinds (à la Harper & Lillibridge) and bounded kinds
- Subtyping
- Type equivalence
- ...

Operational semantics (30 rules)
- Machines (compilation): $H, m \rightarrow_c H', m'$ (2 rules)
- Expressions (run-time execution): $H, e \rightarrow_c H', e$ (21 rules)
- Networks (communication): $n \rightarrow n'$ (7 rules)
Summary (2/2): Theorems

Abstraction preservation is a combination of two results.

<table>
<thead>
<tr>
<th>Type Preservation</th>
</tr>
</thead>
<tbody>
<tr>
<td>If $\vdash^H_e \ e : T$ and $H, e \rightarrow^c H', e'$ then $\vdash^H_{e'} e' : T$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Typing Unicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>If $\vdash^H_e \ e : T_0$ and $\vdash^H_e \ e : T_1$, then $\vdash^H_c T_0 =\equiv T_1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Progress</th>
</tr>
</thead>
<tbody>
<tr>
<td>If $\vdash^H_e \ e : T$ then one of the following holds:</td>
</tr>
<tr>
<td>- $e$ is a value in the colour $c$, blocked on I/O, or an exception.</td>
</tr>
<tr>
<td>- $e$ reduces, i.e. there exist $e'$ and $H'$ such that $H, e \rightarrow^c H', e'$.</td>
</tr>
</tbody>
</table>
Part II

Compiler for secure sessions
Securing distributed languages

Uncertainty over the execution environment

The programmer has little control over:

- the network
- the remote peers
Securing distributed languages

Uncertainty over the execution environment

The programmer has little control over:
- the network
- the remote peers

⇒

Only realistic security assumption

Everyone is potentially malicious.
Securing distributed languages

Uncertainty over the execution environment
The programmer has little control over:
- the network
- the remote peers

⇒ Only realistic security assumption
Everyone is potentially malicious.

Designing a (correct) security protocol is hard
- Involves low-level, error-prone coding below communication abstractions.
- Depends on global message choreography.
- Should handle compromised peers.
Securing distributed languages

Uncertainty over the execution environment
The programmer has little control over:
- the network
- the remote peers

⇒ Only realistic security assumption
Everyone is potentially malicious.

Designing a (correct) security protocol is hard
- Involves low-level, error-prone coding below communication abstractions.
- Depends on global message choreography.
- Should handle compromised peers.

Our goal
- To automatically generate taylored cryptographic protocols protecting against the network and compromised peers;
- To hide implementation details with a clear semantics and proofs of correctness
Sessions (protocols, contracts, conversations, workflows, ...)

How do we specify a message flow between several roles?

- They can be represented as global graphs;

![Diagram](image)

- or as local processes (our concrete syntax).

```
session Rpc =
  role c : int =
    send Request : string ;
    recv Reply : int
  role w : unit =
    recv Request : string →
    send Reply : int
```

Active area of research

- Pi-calculus, web services, operating systems
- Common strategy: type systems enforce protocol compliance
  if every site program is well-typed, sessions follow their specification
Secure compilation of session abstractions

Contributions
- Design of a high-level session language
- Automated generation of a secure implementation from the specification

Results
- **Functional result**: Well-typed programs play their role
- **Security theorem**: A role using our generated implementation can assume that remote peers play their role without having to trust them.

Outline:
1. Session programming & examples
2. Security threats
3. Generated protocol
4. Theorem
5. Performance evaluation
S2ml, A secure session compiler

ML Application code

Session declarations

Session implementation

An extension of ML with sessions

Networking & Cryptography

Concrete

ML compiler

Concrete Model

Executable
An extension of ML with sessions

S2ml, A secure session compiler

Networking & Cryptography

Symbolic

Session declaration
ML Application code

Session implementation
ML Application code

Symbolic Model

formally verified code

ML compiler
An extension of ML with sessions

S2ml, a secure session compiler

Networking & Cryptography

Session implementation

ML Application code

Session declarations

ML Application code

ML compiler

Concrete Model

Symbolic Model

formally verified code

Executable

Concrete Model

Symbolic Model

Concrete
Session expressiveness

- **Ws**: 2 roles, 3 messages, 1 choice
  
  ![Ws Diagram](Diagram)

- **Wsn**: 2 roles, 4 messages, 1 choice, 1 loop
  
  ![Wsn Diagram](Diagram)

- **Shopping**: 3 roles, 8 messages, 1 choice, 1 loop
  
  ![Shopping Diagram](Diagram)
Programming with continuations

File **Rpc.mli**

```ml

(* Function for role w *)
type result_w = unit

type msg3 = {
    hRequest : (prins * string → msg4)
}

and msg4 =
    Reply of (int * result_w)

val w : principal → msg3 → result_w

[...]
```

Arbitrary ML code can be used to run the session and produce the message content.

Sample user file to play w’s role

```ml
Rpc.w "Bob"
    {hRequest = function (_,x) → match x with "Cheese" → Reply(24,()) |
    "Wine" → Reply(53,())}
```

Pierre-Malo Deniéloü (PhD Defense)
Threats against session integrity

Powerful Attacker model
- can spy on transmitted messages
- can join a session as any role
- can initiate sessions
- can access the libraries (networking, crypto)
- cannot forge signatures

Attacks against an unsecure implementation
- Message integrity (Offer by Reject)
- Message replay (Offer triggers a new iteration)
- Control integrity (from Reject to Change)
- Sender authentication (c could send Confirm to o)
Protocol outline

Principles of our protocol generation

1. Each edge is implemented by a unique concrete message.
2. We want static message handling for efficiency.

Against replay attacks

- between session executions: session nonces
- between loop iterations: time stamps
- at session initialisations: anti-replay caches

Against session flow attacks

- Signatures of the entire message history (optimisations possible ...)

Pierre-Malo Deniérou (PhD Defense)
Optimising the protocol

Signing and countersigning the full history

1. Using time stamps to avoid countersigning
2. Using local states to remember past achievements

Execution paths: which signatures to convince the receiver?

- Request-Contract-**Reject-Abort**
- Request-Contract-Offer-Change-Offer-Change
- Request-Contract-(Offer-Change)^n-**Reject-Abort**

Visibility: at most one signature from each of the previous roles is enough.
Session integrity

Our formalism:
- $F+S$ is our high-level language where sessions are primitive;
- $F$ is our low-level language without sessions (ie ML);
- $F \subseteq F+S$.

Theorem (Session integrity)

If $L \ M \tilde{S} \ U \ O'$ may fail in $F$ then $L \tilde{S} \ U \ O$ may fail in $F+S$.

Intuition
- $L$ is the set of libraries.
- $\tilde{S}$ is a set of session declarations and $M_{\tilde{S}}$ their generated session implementation.
- Failure is a barb raised by the user code $U$.
- $U$ is the same code in $F+S$ and $F$.
- $O$ cannot make $U$ see an observable difference between $F+S$ and $F$. 
### Performance of the code generation

<table>
<thead>
<tr>
<th>Session $S$</th>
<th>Rôles</th>
<th>Fichier (loc)</th>
<th>Appli-cation (loc)</th>
<th>Graphe (loc)</th>
<th>Graphes Locaux (loc)</th>
<th>S.mli (loc)</th>
<th>S.ml (loc)</th>
<th>Compilation (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>2</td>
<td>5</td>
<td>21</td>
<td>8</td>
<td>12</td>
<td>19</td>
<td>247</td>
<td>1.26</td>
</tr>
<tr>
<td>Rpc</td>
<td>2</td>
<td>7</td>
<td>25</td>
<td>10</td>
<td>18</td>
<td>23</td>
<td>377</td>
<td>1.35</td>
</tr>
<tr>
<td>Forward</td>
<td>3</td>
<td>10</td>
<td>33</td>
<td>12</td>
<td>25</td>
<td>34</td>
<td>632</td>
<td>1.66</td>
</tr>
<tr>
<td>Auth</td>
<td>4</td>
<td>15</td>
<td>45</td>
<td>16</td>
<td>38</td>
<td>49</td>
<td>1070</td>
<td>1.86</td>
</tr>
<tr>
<td>Ws</td>
<td>2</td>
<td>7</td>
<td>33</td>
<td>12</td>
<td>24</td>
<td>25</td>
<td>481</td>
<td>1.36</td>
</tr>
<tr>
<td>Wsn</td>
<td>2</td>
<td>15</td>
<td>44</td>
<td>13</td>
<td>42</td>
<td>29</td>
<td>782</td>
<td>1.50</td>
</tr>
<tr>
<td>Wsne</td>
<td>2</td>
<td>19</td>
<td>45</td>
<td>15</td>
<td>48</td>
<td>31</td>
<td>881</td>
<td>1.90</td>
</tr>
<tr>
<td>Shopping</td>
<td>3</td>
<td>29</td>
<td>70</td>
<td>21</td>
<td>85</td>
<td>49</td>
<td>1780</td>
<td>2.43</td>
</tr>
<tr>
<td>Conf</td>
<td>3</td>
<td>48</td>
<td>86</td>
<td>37</td>
<td>181</td>
<td>78</td>
<td>3451</td>
<td>3.32</td>
</tr>
<tr>
<td>Loi</td>
<td>6</td>
<td>101</td>
<td>189</td>
<td>57</td>
<td>310</td>
<td>141</td>
<td>7267</td>
<td>6.29</td>
</tr>
</tbody>
</table>

### Performance of the generated code (10000 messages)

<table>
<thead>
<tr>
<th>Authentication using</th>
<th>signatures</th>
<th>MACs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total execution time</td>
<td>93.92 s</td>
<td>1.77 s</td>
</tr>
<tr>
<td>Without verification</td>
<td>90.80 s</td>
<td>1.66 s</td>
</tr>
<tr>
<td>Without cryptography</td>
<td></td>
<td>1.43 s</td>
</tr>
<tr>
<td>Unprotected</td>
<td></td>
<td>1.31 s</td>
</tr>
</tbody>
</table>
Conclusion

I. Abstraction preservation
- Design of a distributed language with abstract data types and subtyping.
- Semantics to ensure abstract type safety.
- Soundness, typing unicity and progress proofs.

II. Compiler for secure session
- Design of a high-level session language
- Automated generation of a secure implementation from the specification
- Generic proof of the security protocol correctness
Conclusion

I. Abstraction preservation
- Design of a distributed language with abstract data types and subtyping.
- Semantics to ensure abstract type safety.
- Soundness, typing unicity and progress proofs.

II. Compiler for secure session
- Design of a high-level session language
- Automated generation of a secure implementation from the specification
- Generic proof of the security protocol correctness

Thank you!