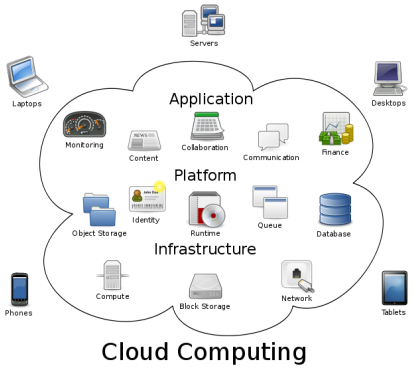
Verification of a Self-configuration Protocol for Distributed Applications in the Cloud

Gwen Salaün
Grenoble INP, INRIA, France

joint work with
Xavier Etchevers[1], Noel de Palma[2], Fabienne Boyer[2], Thierry Coupaye[1]
[1]Orange Labs, France
[2]UJF-Grenoble 1, INRIA, France

Introduction

- Cloud computing aims at delivering resources and applications as a service over a network (*e.g.*, the Internet)



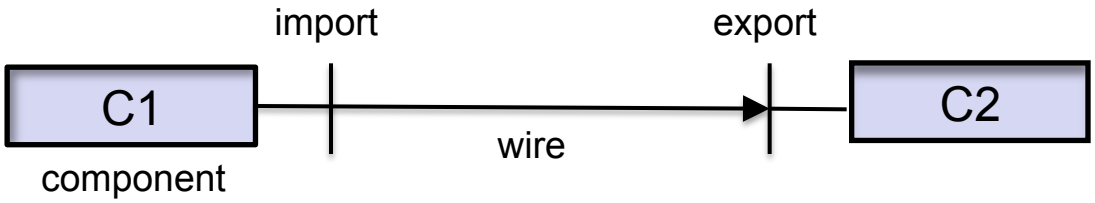
- Cloud applications are often complex distributed applications composed of multiple software running on separate virtual machines
- Setting up, (re)configuring, and monitoring these applications is a burden, and involves complex management protocols
- In this talk, we present the verification of an innovative self-configuration protocol which automates the configuration of distributed applications consisting of interacting virtual machines

Outline

1. The Self-configuration Protocol
2. LNT and CADP
3. Specification in LNT
4. Verification with CADP
5. Concluding Remarks

Application Model

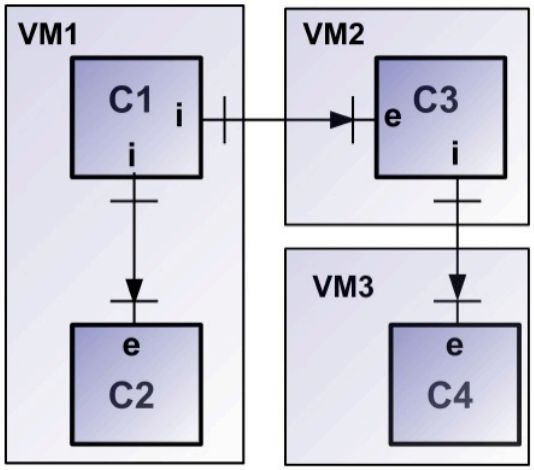
- An application model consists of a set of components and a set of bindings connecting these components together
- A component is composed of input and output ports, namely imports and exports
- An import can be either optional or mandatory
- A binding connects an import of one component to an export of another component

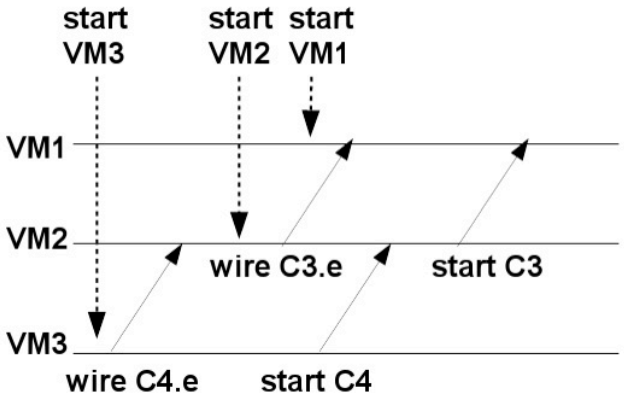


- Components are distributed over virtual machines, which are in charge of their administration (local and remote bindings)

Self-configuration Protocol

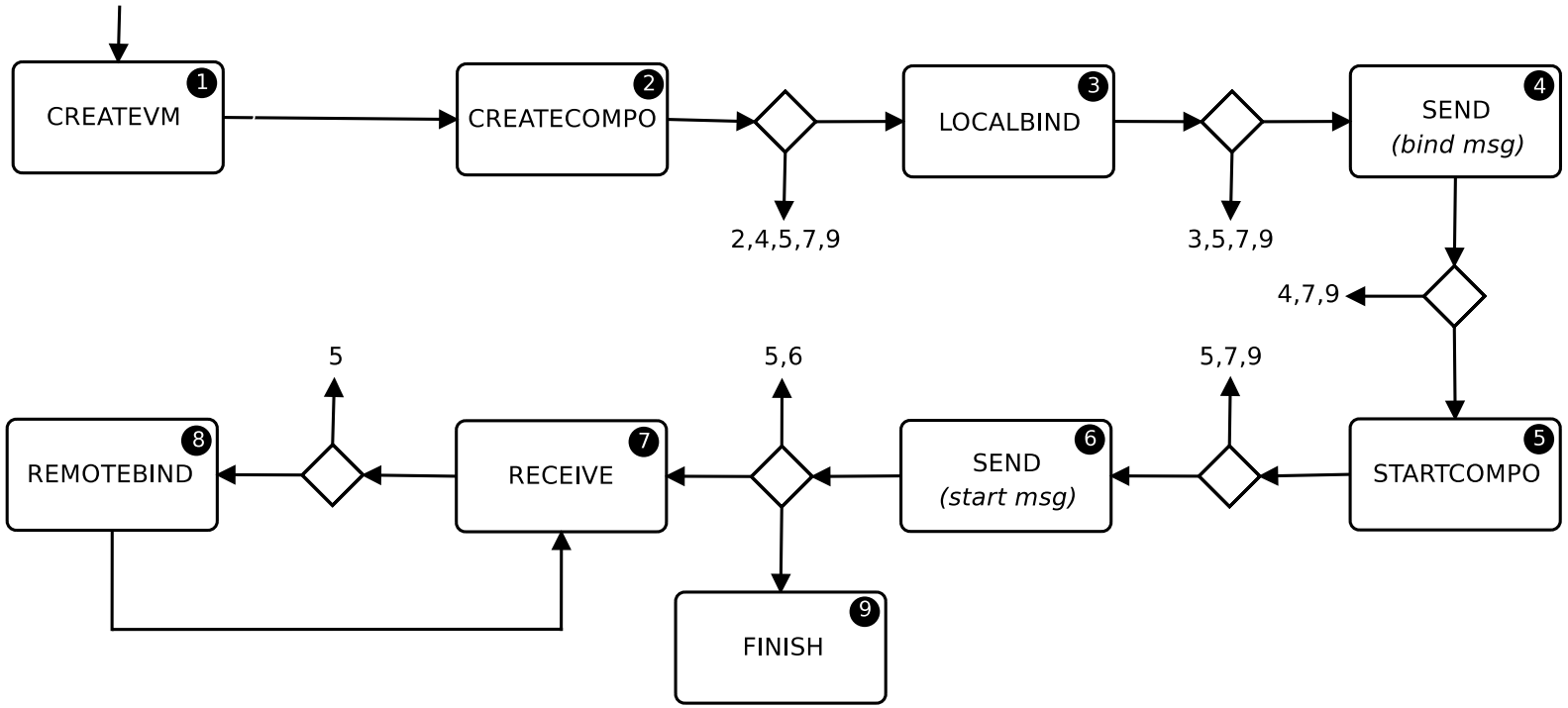
- This protocol (developed at Orange labs) for configuring distributed applications is decentralized and loosely-coupled
- Each virtual machine (VM) embeds the application model and a configurator that manages the component binding configuration and the application start-up

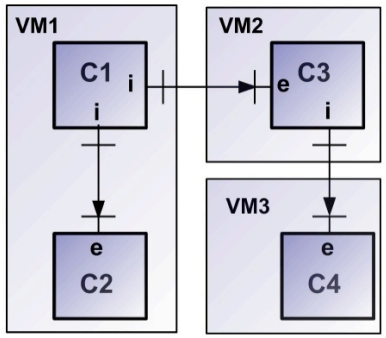


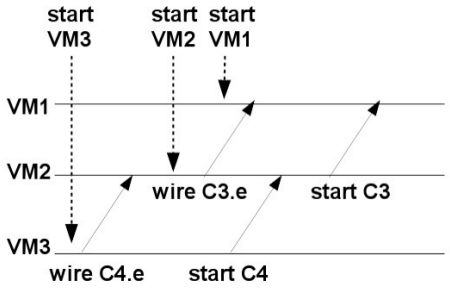


- Configurators interact together through a Message Oriented Middleware (MOM), which relies on message buffers

Configurator Workflow







Outline

1. The Self-configuration Protocol
2. LNT and CADP
3. Specification in LNT
4. Verification with CADP
5. Concluding Remarks

LOTOS NT

- LOTOS NT (LNT) is a value-passing process algebra with user-friendly syntax and operational semantics

- LNT is an imperative-like language where you can specify data types, functions (pattern matching and recursion), and processes

- Excerpt of the LNT process grammar:

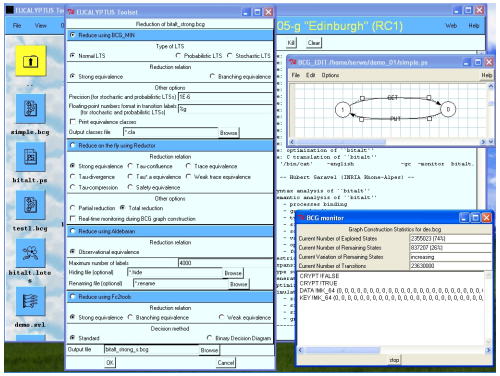
B ::= **stop** | G(!E, ?X) **where** E' | **if** E **then** B1 **else** B2 **end if**
| x:=E | **hide** G **in** B **end hide** | P [G1,...,Gm] (E1,...,En)
| **select** B1 **[]** … **[]** Bn **end select** | B1 **;** B2
| **par** G **in** B1 **||** … **||** Bn **end par**

- Verification using CADP through an automated translation to LOTOS

Construction and Analysis of Distributed Processes (CADP)

- Design of asynchronous systems
  - Concurrent processes
  - Message-passing communication
  - Nondeterministic behaviours

CADP
(INRIA/VASY)



- Formal approach rooted in concurrency theory: process calculi, Labeled Transition Systems, bisimulations, branching temporal logics

- Many verification techniques: simulation, model and equivalence checking, compositional verification, test case generation, performance evaluation, etc.

- Numerous real-world applications: avionics, embedded systems, hardware design, middleware and software architectures, etc.

Outline

1. The Self-configuration Protocol
2. LNT and CADP
3. Specification in LNT
4. Verification with CADP
5. Concluding Remarks

Specification in LNT (1/2)

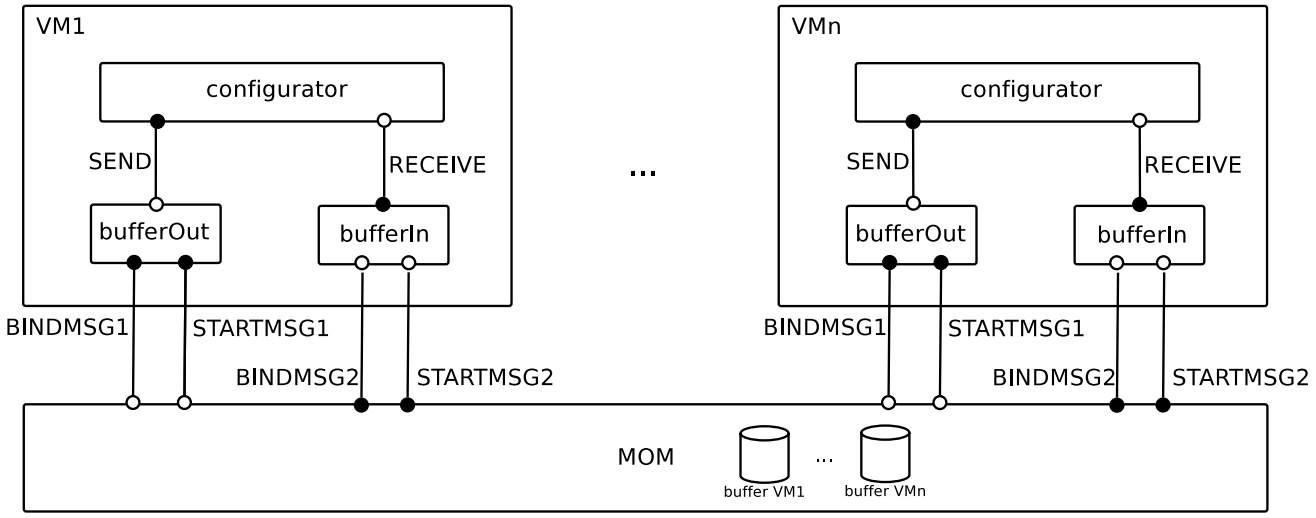
- The specification consists of three parts: data types (300 lines), functions (1000 lines), processes (700 lines)
- Data types describe the application model (components, ports, bindings, buffers, etc.)
- Functions are necessary for:
  - extracting information from the application model
  - describing buffers and basic operations on them
  - keeping track of the started components to know when components can be started

```
function add (m: TMessage, q: TBuffer): TBuffer is
        case q in
        var hd: TMessage, tl: TBuffer in
                nil -> return cons(m,nil)
        |       cons(hd,tl) -> return cons(hd,add(m,tl))
        end case
end function
```

Specification in LNT (2/2)

- Processes specify VMs (configurator, input and output buffer), the communication layer (MOM), and the system architecture consisting of VMs interacting through the MOM

- Labels in transition systems generated from this specification correspond to local interactions (configurator-buffers), remote interactions between VMs through the MOM, and some information we need for verification



Main LNT Process (Excerpt)

```
process SELFCONFIG [CREATEVM: any, SEND: any, ..] is
    var appli: TApplication in
      appli:=appli();
      CHECKCYCLE (!check_cycle_mandatory(appli));
      CHECKMANDATORY (!check_mandatory_connected(..));
      par BINDMSG1, BINDMSG2, STARTMSG1, STARTMSG2, .. in
          MOM [..] (vmbuffer(VM1,nil),vmbuffer(VM2,nil))
      ||
          par CREATEVM in
              par   (* virtual machine deployer *)
                 CREATEVM (!VM1) || CREATEVM (!VM2)
              end par
          ||
              par SEND, RECEIVE in   (* first machine, VM1 *)
                 configurator [..] (VM1,appli)
                 || bufferOut [SEND,BINDMSG1,..](nil) || bufferIn[RECEIVE,BINDMSG2,..](..)
              end par
          ||    ...  (* second virtual machine, VM2 *)
end par end par end var end process
```

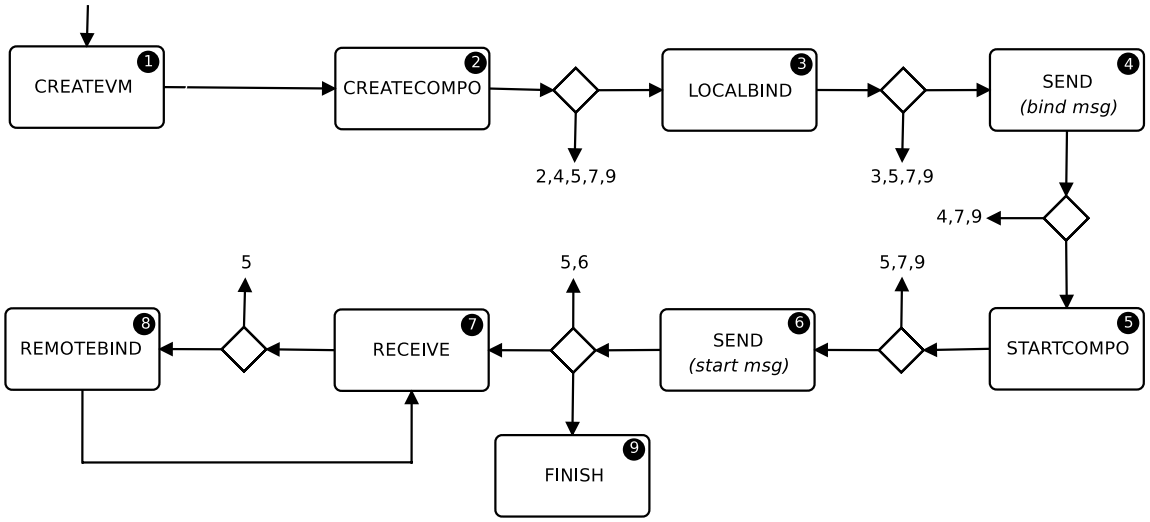
Application model involving two virtual machines (VM1 and VM2)

Outline

1. The Self-configuration Protocol
2. LNT and CADP
3. Specification in LNT
4. Verification with CADP
5. Concluding Remarks

Three Kinds of Verification (1/2)

- We verify that each input application respects a few structural properties, such as:
  - No cycle of bindings through mandatory imports
  - All mandatory imports are connected
- We check that each VM behaviour for a given input application respects the correct ordering of actions (pre-order)



Three Kinds of Verification (2/2)

- We identify and check 14 safety and liveness properties that must be preserved by the protocol, such as:
  - A STARTMSG2 message cannot appear before a STARTMSG1 message with the same parameters

```
[ true*.STARTMSG2 ?vm:String ?cx:String ?cy:String.
        true*.STARTMSG1 !vm !cx !cy ] false
```

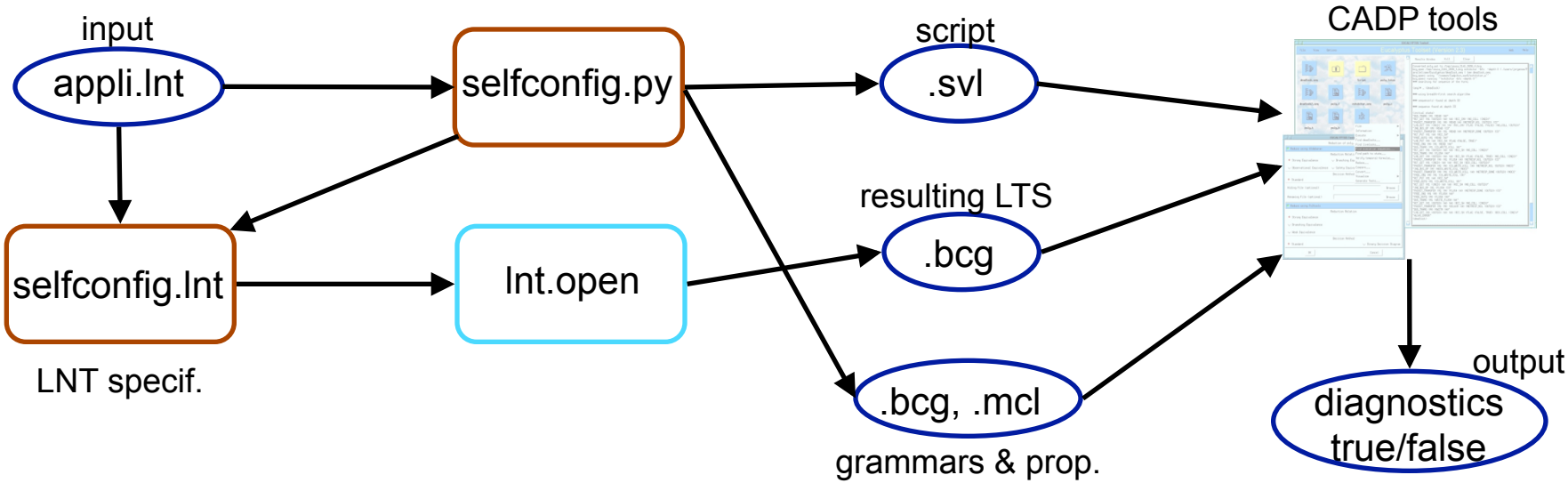
  - A component cannot be started before the components it depends on

```
[ true* . 'STARTCOMPO !.* !C1' . true* .
      'STARTCOMPO !.* !C2' ] false
```

  - All components are eventually started

```
( mu X . ( <true> true and [ not 'STARTCOMPO !.* !C1' ] X ) )
                              and
( mu X . ( < true > true and [ not 'STARTCOMPO !.* !C2' ] X ) )
                              and ...
```

Tool Support



- Experiments were conducted on more than 100 application models
- Issues identified:
  - A bug was found in the configurator start-up part of the protocol ⇒ it was corrected in the Java implementation (Orange Labs)
  - Experiments on how communication among VMs can be implemented ⇒ using a single buffer in the MOM may generate deadlocks

Outline

1. The Self-configuration Protocol
2. LNT and CADP
3. Specification in LNT
4. Verification with CADP
5. Concluding Remarks

Concluding Remarks

- We have presented the specification and verification of a self-configuration protocol involving components distributed over several VMs
- The experience was successful because we have detected a major bug which was corrected in the corresponding Java implementation

Main Perspective:

- Extending the protocol to take component failures into account
- A component failure may impact the whole application, yet we want our protocol to keep on starting and configuring as many VMs and components as possible
- The extended protocol will be extensively validated using analysis tools to check some new properties, *e.g.*, "*a component connected through a mandatory import to a failed component will never be started*"