Execution and Planning of Distributed Systems Reconfigurations

ICE Workshop @DisCoTec, 18th of June 2021

Hélène Coullon (Maverick Chardet, Simon Robillard, Dimitri Pertin, Christian Perez, Claude Jard, Didier Lime, Charlène Servantie)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Associate professor at IMT Atlantique, France

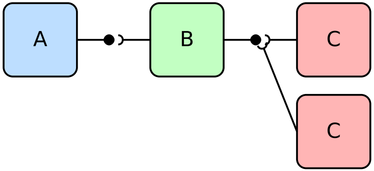
Inria researcher, France

Adjunct professor at UiT, Tromsø, Norway

Introduction and motivations

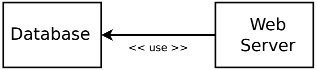
General definition

- Non monolithic code,
- modular units of code - **components**,
- software system = **architectural assembly** of component instances,
- interactions between components through **communications**.



- Master/workers,
- microservices,
- service-oriented,
- layered,
- etc.

Deployment example



Machine 1 **[WHERE]**

Database (DB) [WHAT]

[HOW] [LIFECYCLE]

1. Install
2. Configure
3. Start the service
4. Prepare the service

Machine 2 **[WHERE]**

Web-server (WS) [WHAT]

[HOW] [LIFECYCLE]

1. Install
2. Configure firewall
3. Download
4. Configure parameters
5. Start the service

[WHEN][DEPENDENCIES]: DB $\ll$ WS (component granularity)

[DEPENDENCIES]: DB(3) $\ll$ WS(4), DB(4) $\ll$ WS(5) (lifecycle granularity)

Management and reconfiguration

Ever-running and long-running distributed systems

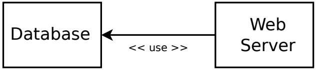
What is a reconfiguration?

- Reconfiguration through time
  - need to add/remove components and/or connections
  - need to change internal configurations
- A set of instructions to move from one state of the system to another.

Examples of reconfiguration reasons

- Faults or errors on services or hardware (e.g., re-deploy),
- dynamic energy or security constraints (e.g., change the set of components),
- dynamic improvement of performance (e.g., scaling),
- dynamic upgrade of some modules.

Reconfiguration example



Database (DB) [WHAT]

[HOW] [LIFECYCLE]

1. Backup data
2. Stop the service
3. Download update
4. Configure parameters
5. Start the service
6. Restore data

Web-server (WS) [WHAT]

[HOW] [LIFECYCLE]

1. Pause the service
2. Configure parameters
3. Start the service

[DEPENDENCIES]: WS(1) $\ll$ DB(2), DB(5) $\ll$ WS(2), DB(6) $\ll$ WS(3) (lifecycle granularity)

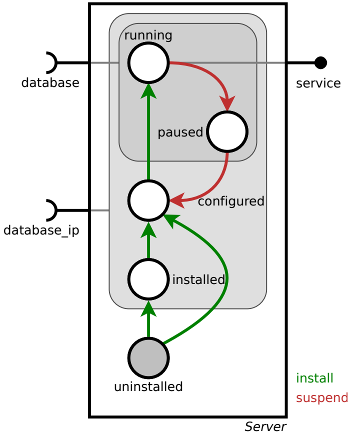
Our goals

1. **Efficiency of reconfigurations**
    - reach quickly a targeted configuration,
    - reduce disruption time.
2. **Execution time prediction**
3. **Safety of reconfigurations**

Table of Contents

The Concerto reconfiguration model

Control components



Written by the component developers

Internal net [LIFECYCLE]

- places = milestones
- transitions = concrete actions to perform

Interfaces [DEPENDENCIES]

- data or service ports
  - use ports = requirements
  - provide ports = provisions
- behaviors
  - subset of transitions
- during execution: active/inactive

Reconfiguration language

1. Create assemblies of components (software system)
2. Make this assembly evolve at runtime
3. Interact with the lifecycle of components

Add/remove
Add/remove a component instance to the current assembly

Connect/disconnect
Connect/disconnect two component instances with compatible ports

Push behavior
Push a behavior to the behavior queue on a component instance

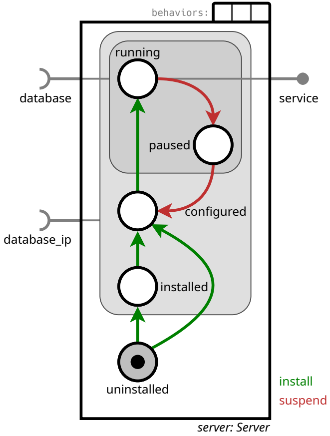
Wait
Wait for a given component instance or wait all components

Reconfiguration example - deployment

Written by the reconfiguration developer

Deployment program:

```
add(server: Server)
add(db: Database)
con(server.database_ip,db.ip)
con(server.database,db.service)
pushB(server, install)
pushB(db, deploy)
wait(server)
```

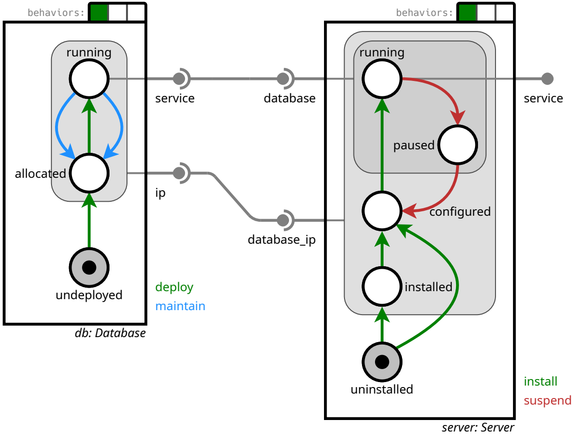


Reconfiguration example - deployment

Written by the reconfiguration developer

Deployment program:

```
add(server: Server)
add(db: Database)
con(server.database_ip,db.ip)
con(server.database,db.service)
pushB(server, install)
pushB(db, deploy)
wait(server)
```

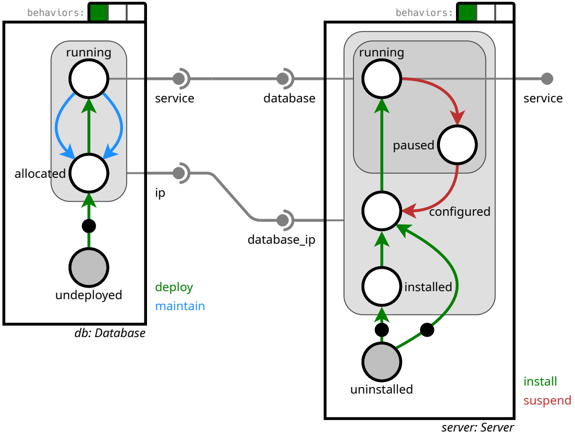


Reconfiguration example - deployment

Written by the reconfiguration developer

Deployment program:

```
add(server: Server)
add(db: Database)
con(server.database_ip,db.ip)
con(server.database,db.service)
pushB(server, install)
pushB(db, deploy)
wait(server)
```

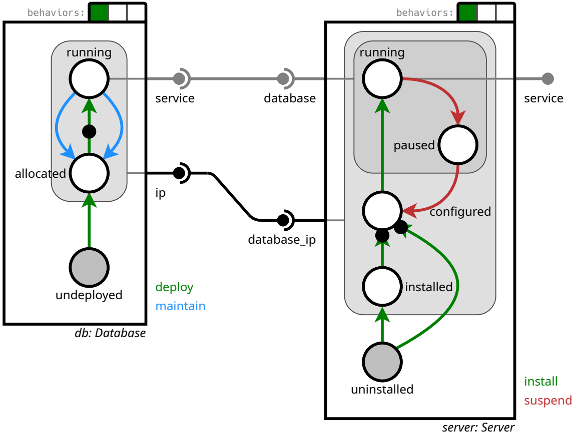


Reconfiguration example - deployment

Written by the reconfiguration developer

Deployment program:

```
add(server: Server)
add(db: Database)
con(server.database_ip,db.ip)
con(server.database,db.service)
pushB(server, install)
pushB(db, deploy)
wait(server)
```

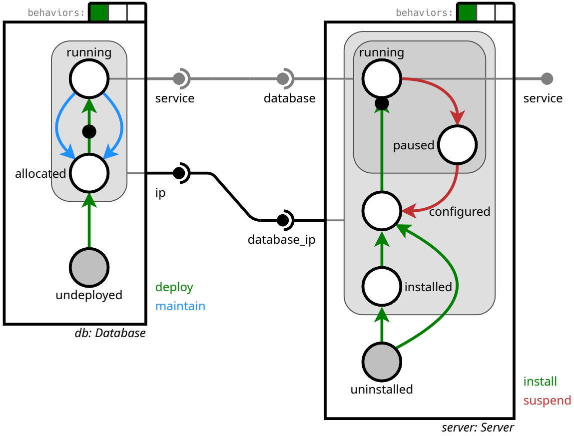


Reconfiguration example - deployment

Written by the reconfiguration developer

Deployment program:

```
add(server: Server)
add(db: Database)
con(server.database_ip,db.ip)
con(server.database,db.service)
pushB(server, install)
pushB(db, deploy)
wait(server)
```

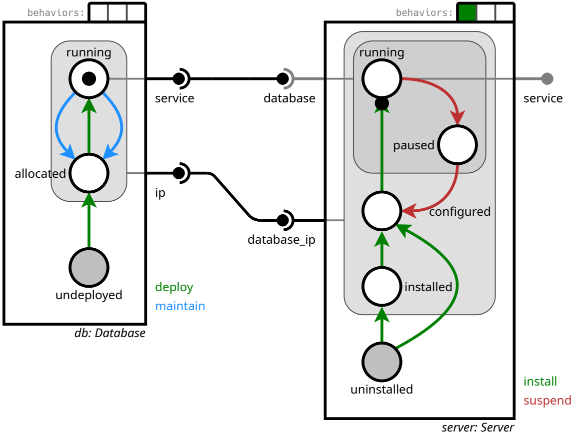


Reconfiguration example - deployment

Written by the reconfiguration developer

Deployment program:

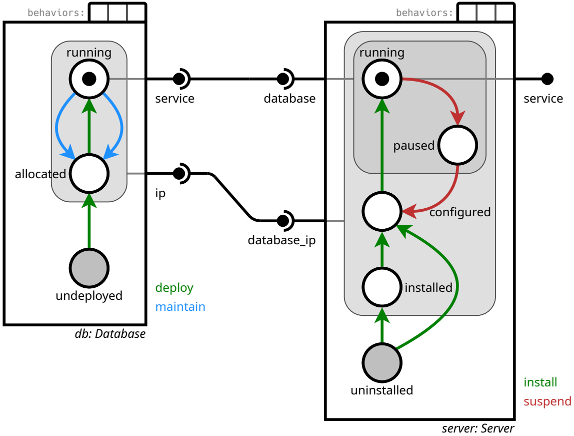
```
add(server: Server)
add(db: Database)
con(server.database_ip,db.ip)
con(server.database,db.service)
pushB(server, install)
pushB(db, deploy)
wait(server)
```



Written by the reconfiguration developer

Deployment program:

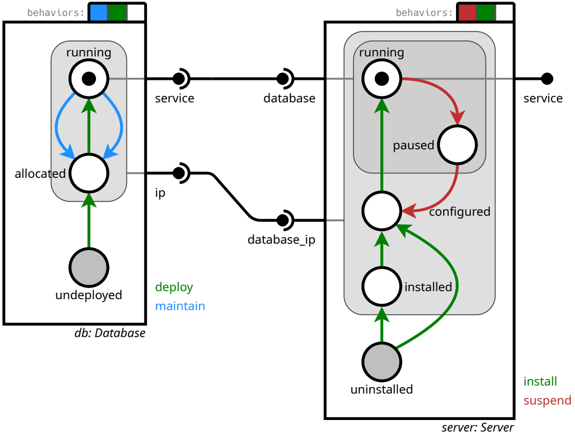
```
add(server: Server)
add(db: Database)
con(server.database_ip,db.ip)
con(server.database,db.service)
pushB(server, install)
pushB(db, deploy)
wait(server)
```



Written by the reconfiguration developer

Maintenance program:

```
pushB(db, maintain)
pushB(db, deploy)
pushB(server, suspend)
pushB(server, install)
wait(server)
```

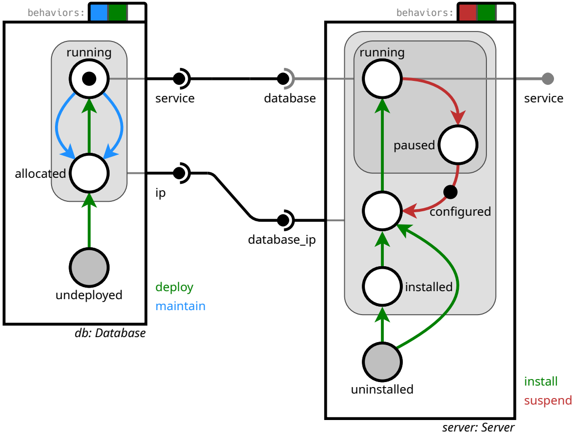


Reconfiguration example - maintenance

Written by the reconfiguration developer

Maintenance program:

```
pushB(db, maintain)
pushB(db, deploy)
pushB(server, suspend)
pushB(server, install)
wait(server)
```

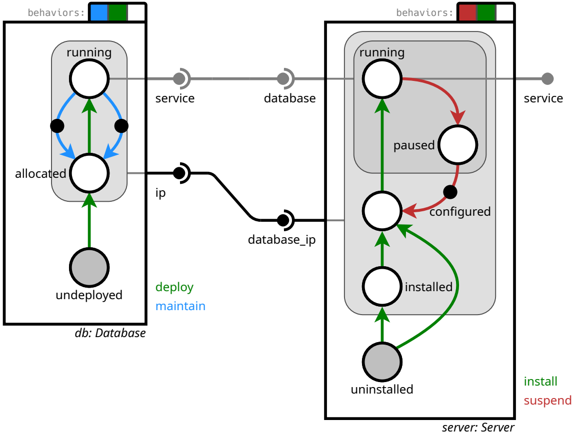


Reconfiguration example - maintenance

Written by the reconfiguration developer

Maintenance program:

```
pushB(db, maintain)
pushB(db, deploy)
pushB(server, suspend)
pushB(server, install)
wait(server)
```



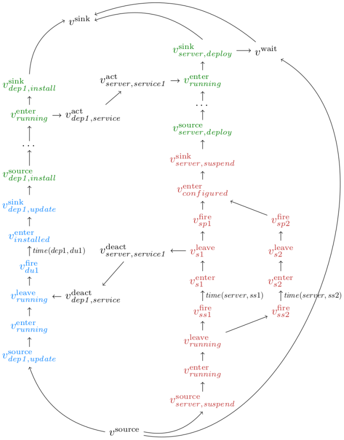
Inputs:

- reconfiguration program
- time estimations for transitions

Output: execution time prediction

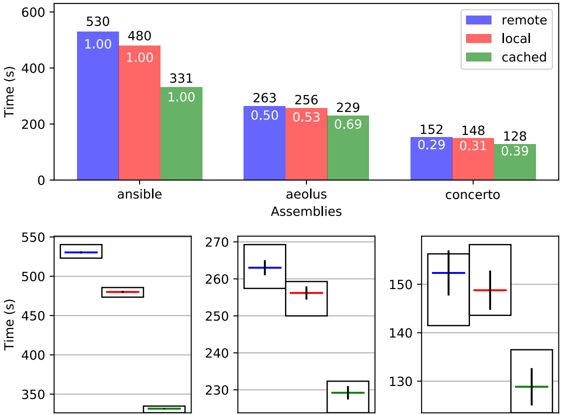
Solution:

1. Dependency graph Generalization
2. Critical path computation (longest path)



Evaluation on the deployment of OpenStack

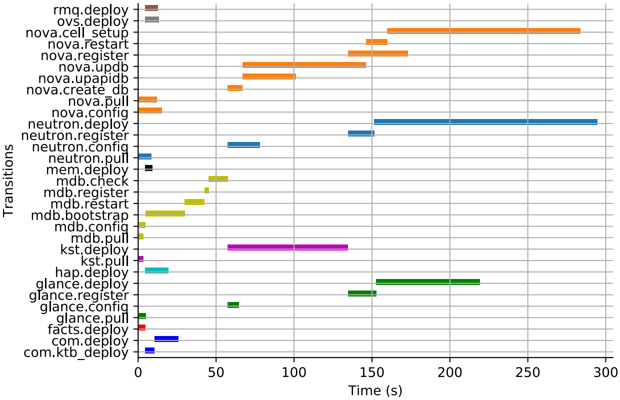
Deployment of a minimal OpenStack: 11 components, 36 services in total



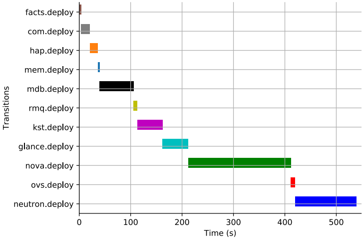
- Results on three nodes Ecotype (Nantes) of Grid'5000
- Comparison to Kolla-Ansible (production tool), and Aeolus (literature)
- Reproducible experiments on Grid'5000

Evaluation on the deployment of OpenStack

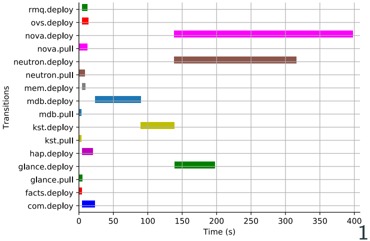
[concerto]



[ansible]



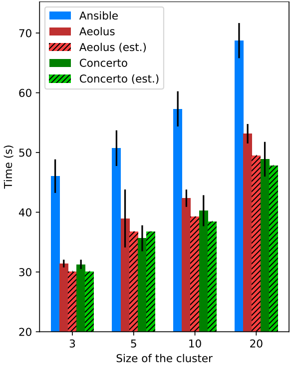
[aeolus]



Evaluation on the reconfiguration of MariaDB

Real use-case extracted from the OpenStack Summit 2018

Initial state: centralized MariaDB running



decentralization

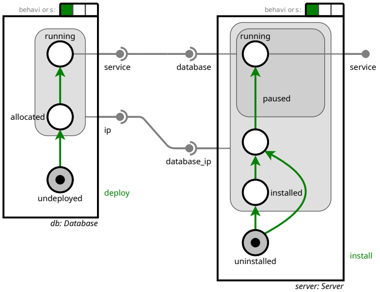
- replaces centralized DB with a Galera cluster
- requires a backup of the data, and a restart of the master node

Results on nodes of UvB (Sophia) of Grid'5000 (2×6-core Intel Xeon X5670 CPUs, 96 GB RAM, 250 GB HDD, internal 40 Gbps InfiniBand, external 1 Gbps)
Reproducible experiments

Model checking on a Concerto deployment program

Verification of deployments

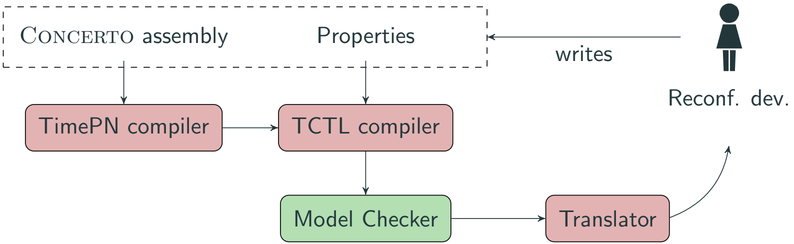
Hypothesis: *the deployment of a distributed software system is written in* CONCERTO *and the developer wants to verify its safety and enhance its efficiency*



1. how to verify some safety properties on the deployment assembly?
2. how to enhance the efficiency without running the deployment many times?

Goal: Study the use of model checking to help solving the challenges.

Verification of deployments



Qualitative properties

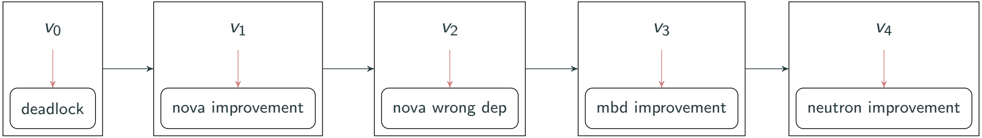
- `deployability` (inevitability)
- `sequentiality` (observer subnet)
- `forbidden` (observer subnet)

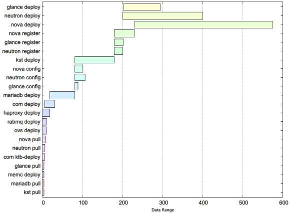
Quantitative properties

- `parallelism`
  $\max(\sum(\text{reachable markings}))$
- `gantt boundaries`
  min/max costs + causality

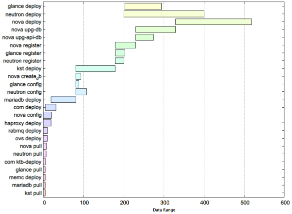
Evaluation (1/2)

5 versions of the OpenStack deployment successively enhanced with the tool





(a) 1-naive with critical path: *nova deploy, nova register, kst deploy, mariadb deploy, haproxy deploy*



(b) 2-nova with critical path: *nova deploy, nova upg-db, nova register, kst deploy, mariadb deploy, haproxy deploy*

Evaluation (2/2)

Experiments conducted with the model checker Romeo

	0-deadlock	1-naive	2-nova	3-nova	4-nova-mdb
CONCERTO places	27	27	28	28	29
CONCERTO transitions	22	22	25	25	28
CONCERTO connections	30	30	30	30	30
Petri net places	113	113	124	124	134
Petri net transitions	75	75	84	84	92
Transformation time (ms)	1.6	1.6	1.8	1.7	1.5
Deployability	False	True	True	True	True
Resolution time (s)	0	41.6	78.7	88.7	152.6
Parallelism nova	-	1	2	2	2
Resolution time (s)	-	42.1	82.7	93.6	154.3
Parallelism full	-	10	11	11	12
Resolution time (s)	-	43.2	86.1	98.4	162.9
Gantt & critical path	-	Fig	Fig	Fig	Fig
Resolution time (s)	-	130.1	266.9	275.4	588.1
Boundaries	-	[575,615]	[518,554]	[400,423]	[377,398]
Resolution time (s)	-	130.1, 128.8	266.9, 269.7	275.4, 267.6	588.1, 580.8

Generalization of the approach to Concerto?

- difficult to generalize to any reconfiguration program because of the state explosion when handling the combination of multiple behaviors per components;
- other kinds of reconfiguration patterns may be studied such as scaling, rolling upgrade etc., but may not offer an acceptable solving time.

Synthesis

Instead of verifying a CONCERTO reconfiguration program, we synthesize a correct reconfiguration program.

Synthesis of Concerto reconfiguration programs

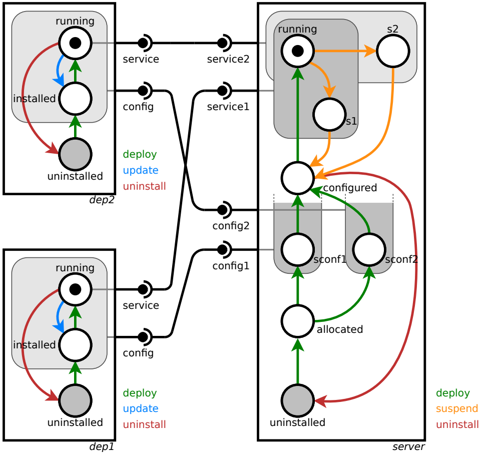
Work of Simon Robillard, postdoc (associate professor in Montpellier starting in September)

Reconfiguration synthesis

- **input:** a reconfiguration goal
  - set of behaviors to execute on designated components
  - constraints on the final state of ports
- **output:** a correct reconfiguration script
  - `pushB` requests
  - `wait` commands
- out of scope: component creations/deletions/(dis)connections
  - usual for safety reasons to handle topological changes before and after behaviors requests and synchronization (e.g., deployment)

Solution: a three-phases algorithm

Example: updating components



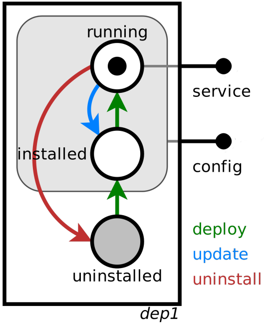
Reconfiguration goal

1. dep1 and dep2 must execute `update`
2. all ports should be active at the end of the reconfiguration

incomplete list of behaviors, a partial specification, is enough

1. Correct and optimal individual component behaviors

- find a sequence of behaviors that satisfies the goal
- enumerate and analyze possible sequences
- no solution $\implies$ failure of the procedure
- multiple solutions $\implies$ pick one according to some optimization criterion
  - shortest sequence
  - shortest estimated execution time
  - fewest port requirements



solution for this example

[update, deploy]

2. Correct global schedule of behaviors

- **Goal**: find a global schedule
- **assumption**: reconfiguration plan with *steps*
  - at most 1 behavior per component/step
  - each step followed by global synchronization
- **problem: assign for each required behavior a given step to schedule it**

```
pushB(server,suspend)
waitAll()
pushB(dep1,update)
pushB(dep2,update)
waitAll()
pushB(dep1,deploy)
pushB(dep2,deploy)
waitAll()
pushB(server,deploy)
waitAll()
```

What is expected in the second phase with 3 steps.

SMT encoding

constraints extracted from the internal nets on sequentiality of behaviors, port requirements and status, incompatible behaviors etc.

Missing behaviors and missing port requirements are detected during this phase

3. Correct reduction of the number of synchronizations

Solution get from phase 2

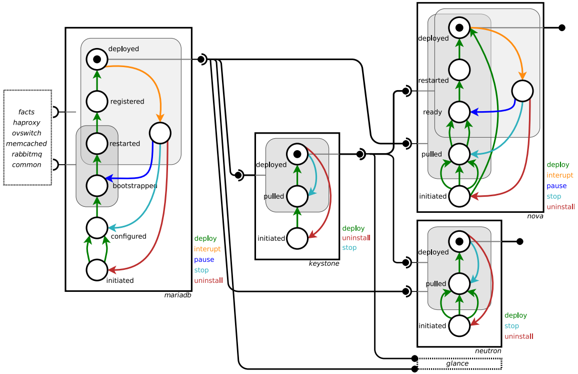
```
pushB(server,suspend)
waitAll()
pushB(dep1,update)
pushB(dep2,update)
waitAll()
pushB(dep1,deploy)
pushB(dep2,deploy)
waitAll()
pushB(server,deploy)
waitAll()
```

Final solution expected from phase 3

```
pushB(server,suspend)
pushB(dep1,update)
pushB(dep2,update)
pushB(dep1,deploy)
pushB(dep2,deploy)
wait(dep1)
wait(dep2)
pushB(server,deploy)
wait(server)
```

1. replace global synchronization barriers by targeted ones
2. delay barriers whenever possible by following correct rules

OpenStack use case



Reconfiguration scenario
update database & restore system to working state

Result

- correct reconfiguration script generated in 2.49 seconds
- contains only synchronization barriers that are needed
- 12 behaviors on 5 components

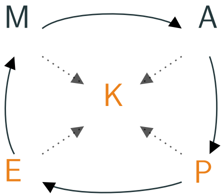
Conclusion and perspectives

Conclusion

- Complexity of the reconfiguration coordination problem
- Goals: efficiency, safety
- Concerto reconfiguration model
- Model-checking on Concerto deployments
- Concerto reconfiguration program synthesis
- Running use-case on OpenStack

Perspectives

- Reconfiguration patterns to improve the scalability of verification and synthesis approaches
  - deployment, scaling, rolling upgrade, substitution etc.,
  - to combine with component patterns (e.g., Docker containerized component).
- Other aspects of self-adaptation in a safe and efficient manner



Thank you!

[1] *Predictable Efficiency for Reconfiguration of Service-Oriented Systems with Concerto*. Maverick Chardet, Hélène Coullon, Christian Perez. In CCGrid 2020.

[2] *Toward Safe and Efficient Reconfiguration with Concerto*. Maverick Chardet, Hélène Coullon, Simon Robillard. In journal SCP, 2020.

[3] *Enhancing Separation of Concerns, Parallelism, and Formalism in Distributed Software Deployment with Madeus*. Maverick Chardet, Hélène Coullon, Christian Perez, Dimitri Pertin, Charlène Servantie, Simon Robillard. [preprint]

[4] *Integrated Model-checking for the Design of Safe and Efficient Distributed Software Commissioning*. Hélène Coullon, Didier Lime, Claude Jard. In iFM 2019, Bergen, Norway.

[5] *Madeus: A formal deployment model*. Maverick Chardet, Hélène Coullon, Christian Perez and Dimitri Pertin. In 4PAD 2018 (hosted at HPCS 2018).

Backup

Related work

Lifecycle

- fixed lifecycle: Tosca, Deployware, SmartFrog, Engage
    - easier to use, less flexible
- programmable lifecycle: Aeolus, Ansible (DevOps configuration tool)
    - more difficult to use, more flexible

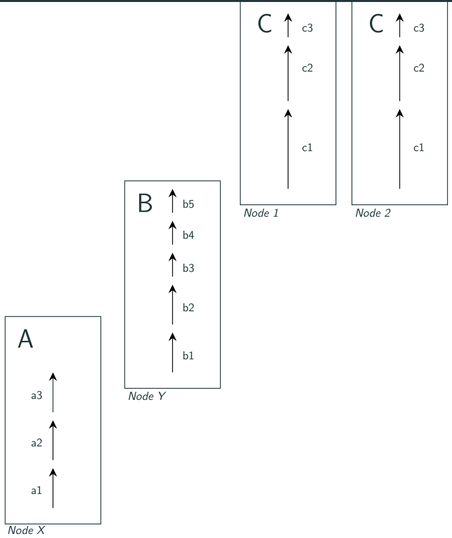
Dependencies

1. same component level: Ansible
2. component level: Tosca, Deployware (DB $\ll$ WS)
3. lifecycle level: Tosca, Engage, Aeolus (DB(3) $\ll$ WS(4))
4. intra-lifecycle level: Concerto

Performance through parallelism and dependencies

level1: multiple nodes, same action

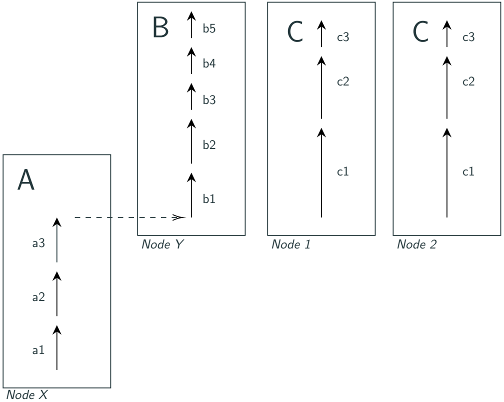
- no dependencies declared
- procedural execution order
- parallelism for the same component
- ANSIBLE



Performance through parallelism and dependencies

level2: level1+non-dependent components

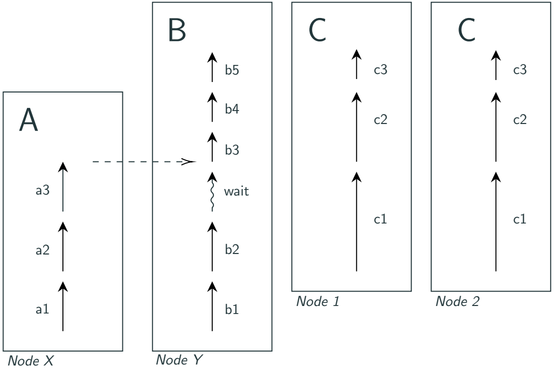
- dependencies at the component level (A $\ll$ B)
- DEPLOYWARE, (basic) TOSCA, ENGAGE



Performance through parallelism and dependencies

level3: level1 + level2 + inter-component

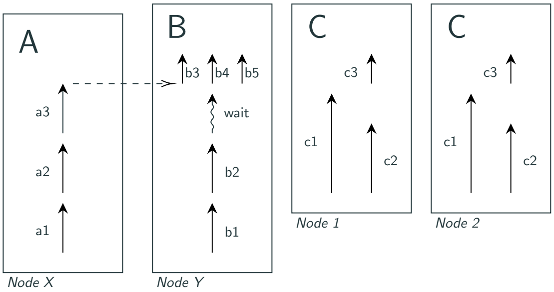
- dependencies at the lifecycle level between components (a3 $\ll$ b3)
- (advanced) Tosca, Aeolus



Performance through parallelism and dependencies

level 4: level1 + level2 + level3 + intra-component

- parallelism within the lifecycle of one component (b1 $\ll$ b2)
- CONCERTO



The finer the dependencies granularity is, the better is the efficiency

Control components in practice

Written by the component developers

```python
class Server(Component):
    def create(self):
        self.places = ['uninstalled','installed','configured','running','paused']

        self.initial_place = 'uninstalled'

        self.behaviors = ['b_install','b_suspend']

        self.transitions = {
            'install1': ('uninstalled','installed','b_install',self.install1),
            'install2': ('uninstalled','configured','b_install',self.install2),
            'configure': ('installed','configured','b_install',self.configure),
            'start': ('configured','running','b_install',self.start),
            'suspend1': ('running','paused','b_suspend',self.suspend1),
            'suspend2': ('paused','configured','b_suspend', self.suspend2)
        }
```

Control components in practice

Written by the component developers

```
class Server(Component):
    def create(self):
        ...

        self.dependencies = {
            'database_ip': (DepType.USE, ['installed','configured','running','paused']),
            'database': (DepType.USE, ['running','paused']),
            'service': (DepType.PROVIDE, ['running'])
        }

    # Definition of the actions
    def install1(self):
        remote = SSHClient()
        remote.connect(host, user, pwd)
        remote.exec_command(cmd)
        ...
```

Deployments pattern with Concerto

Assumptions

- for each component there is one behavior that leads from an uninstalled place to a running place, namely `deploy`;

```
for i in [1,n]
    add(i)
```

- the final assembly is specified by the user;

```
connect as specified by the user
```

Semantics

Simply apply the CONCERTO semantics on a single behavior per component

```
for i in [1,n]
    pushB(i, deploy)
waitall
```

Properties (2/2)

HALP automatically transformed to TCTL (Time Computational Tree Logic) formulae

Qualitative properties

- deployability $\longrightarrow$ inevitability
- sequentiality $\longrightarrow$ observer subnet + invariant
- forbidden $\longrightarrow$ observer subnet

Quantitative properties

- parallelism $\longrightarrow$ max($\sum$(reachable markings))
- gantt boundaries: min/max costs + causality in the trace to get the critical path

SMT encoding of the scheduling problem 2.

Constraints added to the problem:

sequentiality of behaviors
from step 1. $int(schedule(dep1.update)) < int(schedule(dep1.deploy))$

port requirements at the beginning of behaviors
$\neg active_u(schedule(dep1.update))$

separation of behaviors with incompatible port effects
$schedule(server.deploy) \neq schedule(dep1.update)$

port status after behaviors
$active_p(succ(schedule(dep1.deploy)))$

Missing port requirements in 2.

Problem: some unsatisfied ports requirements may make scheduling impossible

Example

- we determined that `dep1` and `dep2` should execute `[update, deploy]`
- but the updates can't be executed while the server is relying on the provide ports

Solution:

1. deduce new individual component goals
   - go to state that satisfies missing port requirement
   - go to state that satisfies final port constraints
2. extend the set of behaviors to schedule and go back to phase 1