Cloud deployment of Service-Oriented Application

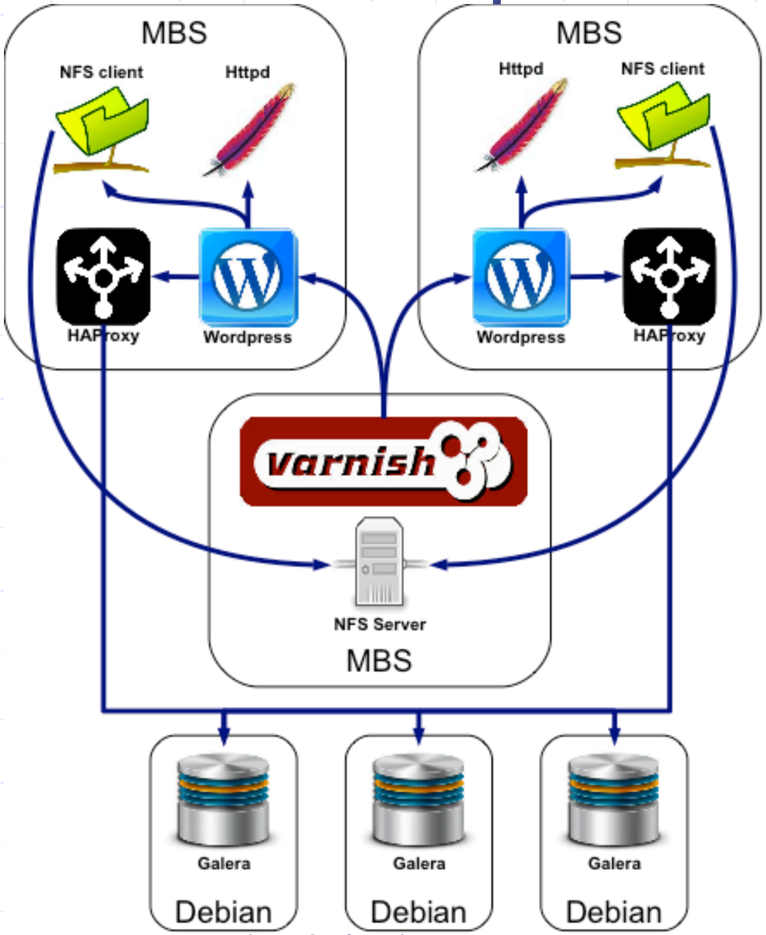
- Cloud computing offers the possibility to **easily/quickly/economically** realize service-oriented applications
  - Services are deployed on top of a **virtualized infrastructure**
- Also in this specific context **tools** and **languages** emerged to deal with automatic deployment

Juju: a tool for (semi)automatic deployment

- Juju is a language, developed by the Ubuntu community, for programming **deployment scripts**
- It also has a GUI for **graphical design** of service-oriented applications to be deployed in the cloud:
  - A **taste** of Juju: *https://demo.jujucharms.com*

The Wordpress running example

- An **optimised** Wordpress installation:



Pros and Cons of Juju

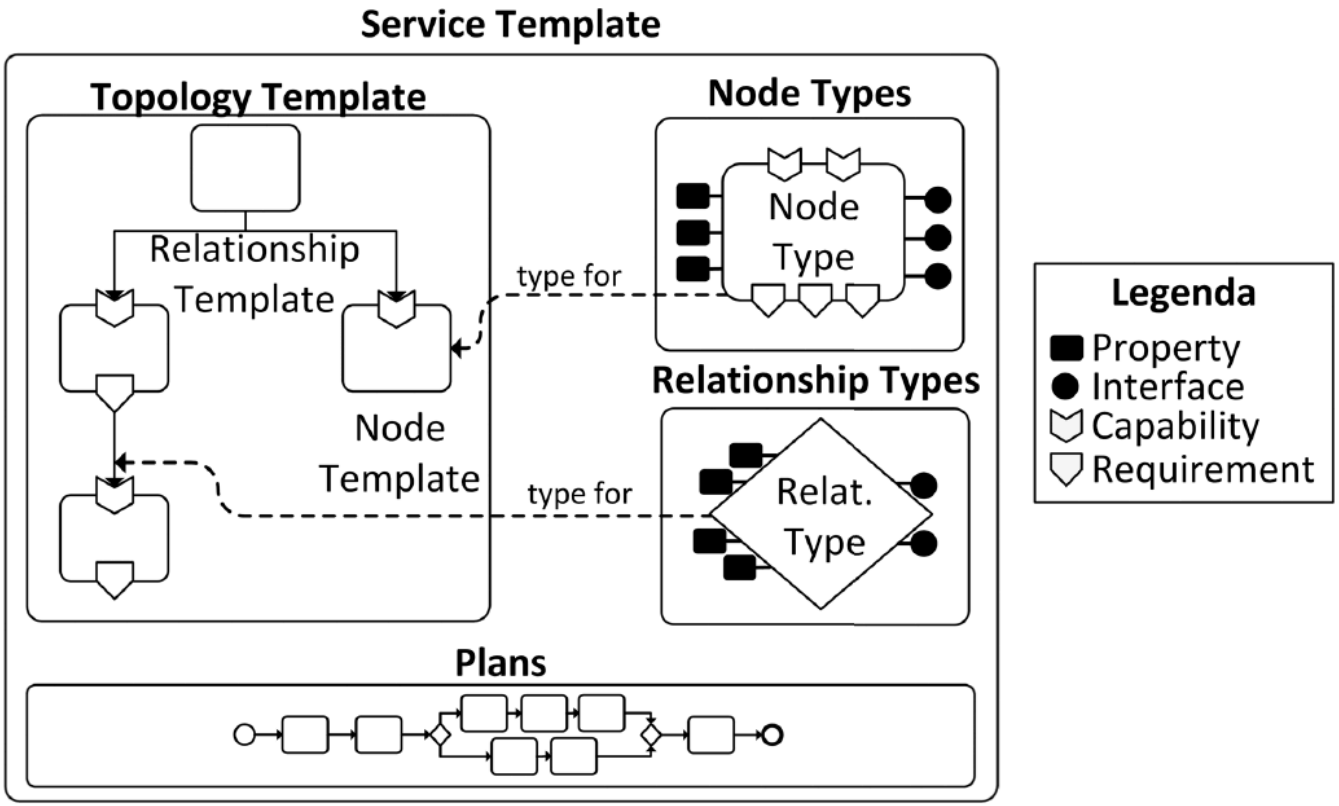
- Pros:
  - rich **library** of deployable services
  - GUI that can be used by **non-expert** users
- Cons:
  - No **correctness** check (all required functionalities connected to some provider)
  - No suggestion about the **deployment order**
  - No optimal **resource usage** (no minimization of the virtual machines costs)

Bottom-up vs Top-down

- Juju is an example of a tool for **bottom-up** deployment:
  - Services are singularly **selected** and then **connected** to form a topology
- There exists also **top-down** deployment:
  - The overall **topology** is first designed
  - Subsequently, a detailed corresponding **deployment plan** is specified

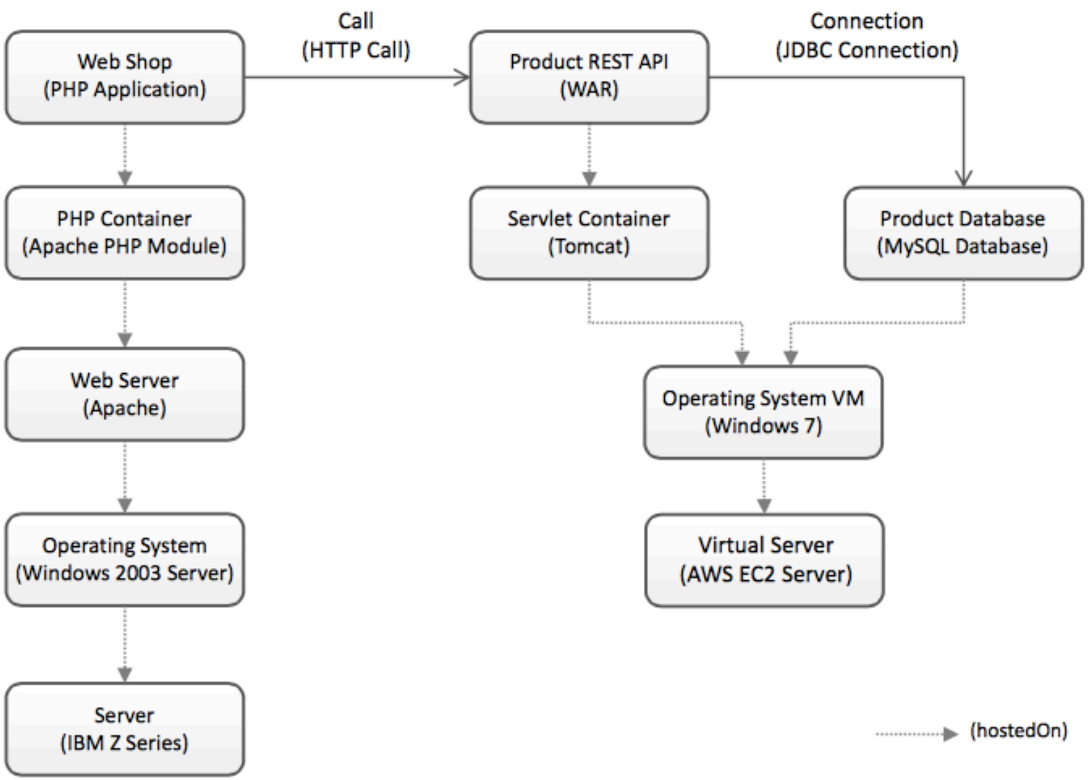
The reference language for top-down deployment

- **TOSCA**: Topology and Orchestration Specification for Cloud Applications



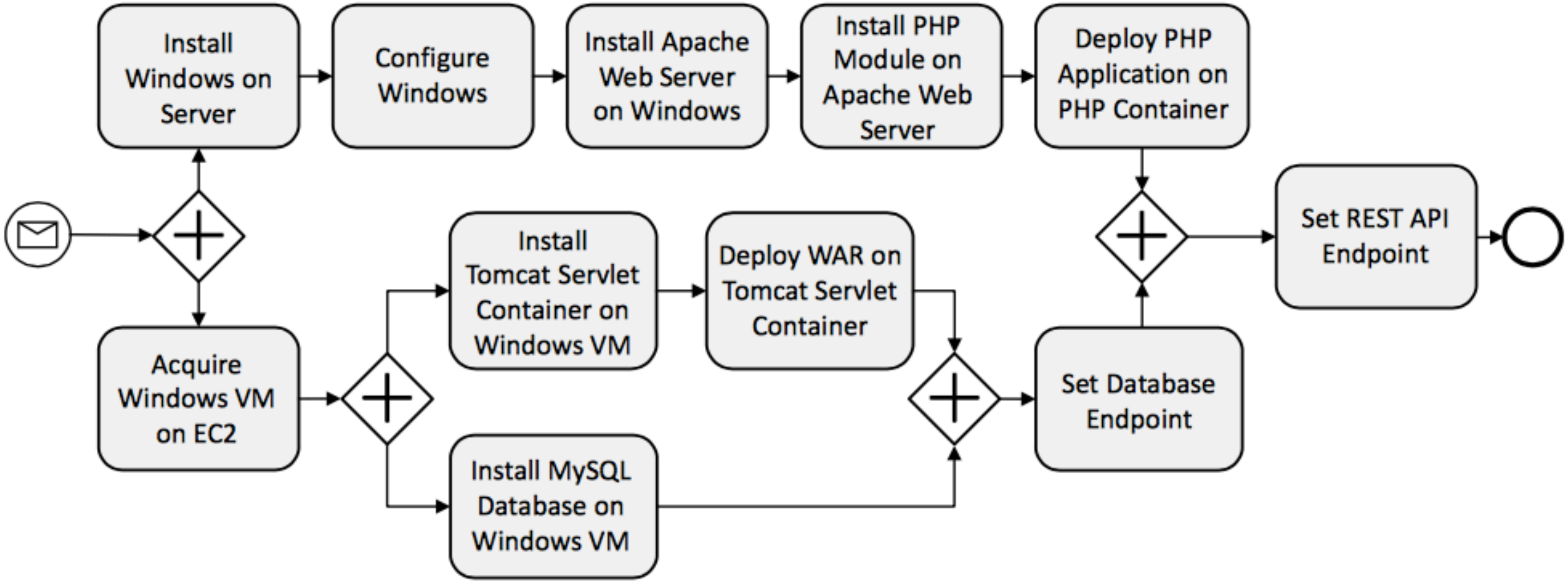
The reference language for top-down deployment

- TOSCA: **Topology** and Orchestration Specification for Cloud Applications



The reference language for top-down deployment

- TOSCA: Topology and **Orchestration** Specification for Cloud Applications



Pros and Cons of TOSCA

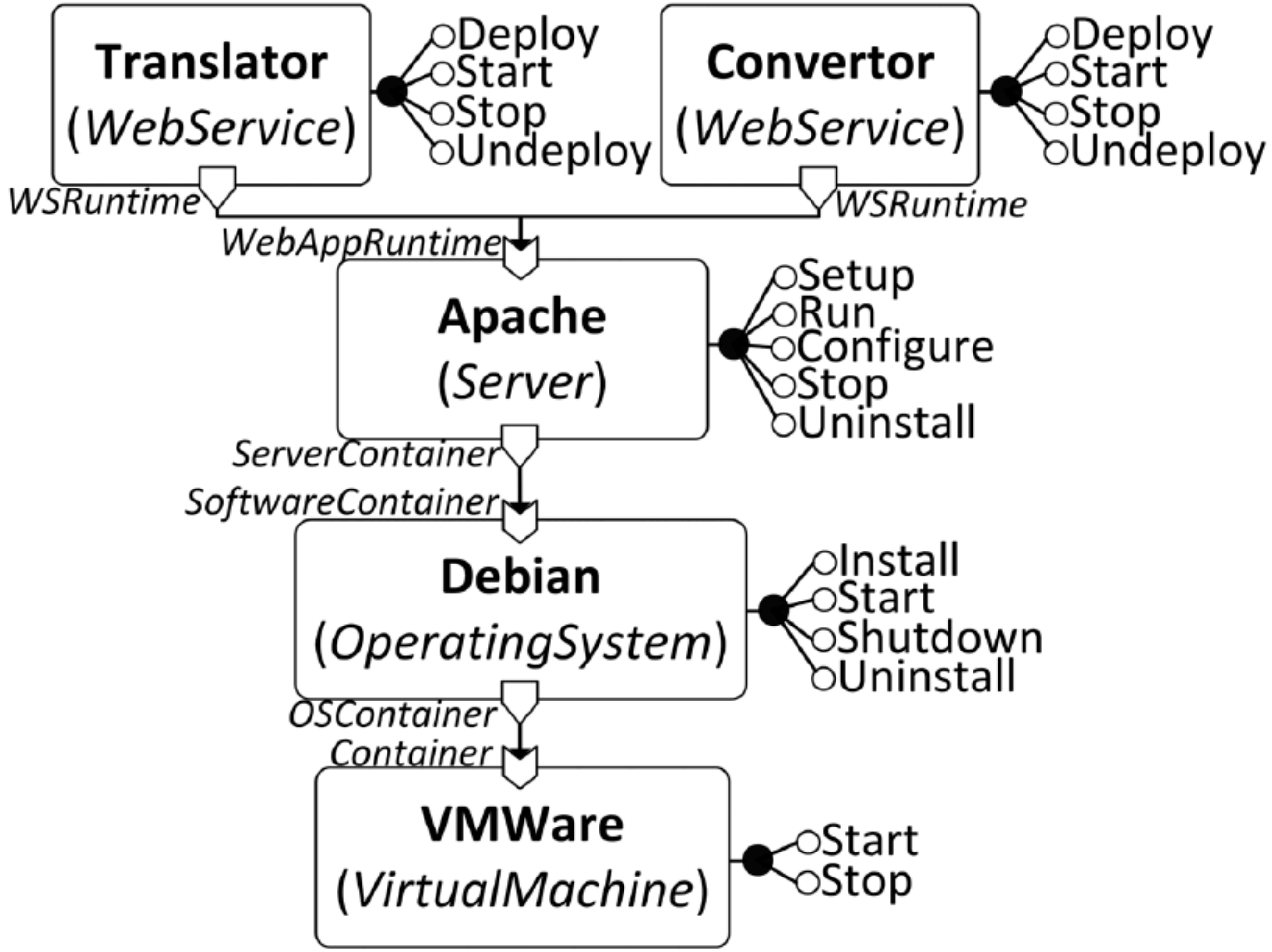
- Pros:
  - **Portability**: cloud-provider agnostic
  - **Standard**: cloud providers and software developers can expose their services / artefacts in a TOSCA compliant way
- Cons:
  - **Manual** design of both topology and deployment plan
  - No check of **correctness**

Barrel: automatic check of deployment plans

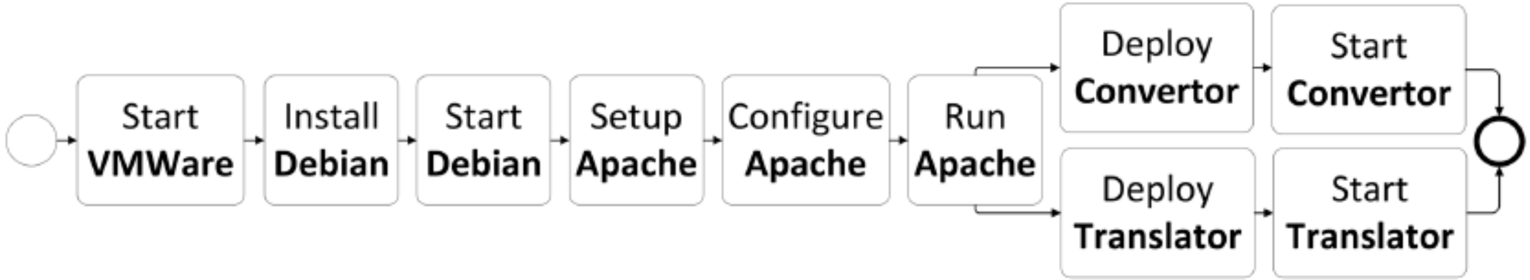
- The deployment actions should follow some precise **ordering** (see Juju demo)
  - The ordering depends on **local constraints**:
    - Wordpress must be connected to a NFS-server **before** being scaled-up
  - In TOSCA there is **no way to specify** such constraints (because it is top-down)
  - Barrel **extends** TOSCA with such specification

A. Brogi, A. Canciani, J. Soldani: *Modelling and Analysing Cloud Application Management.* In Proc. ESOCC 2015: 19-33

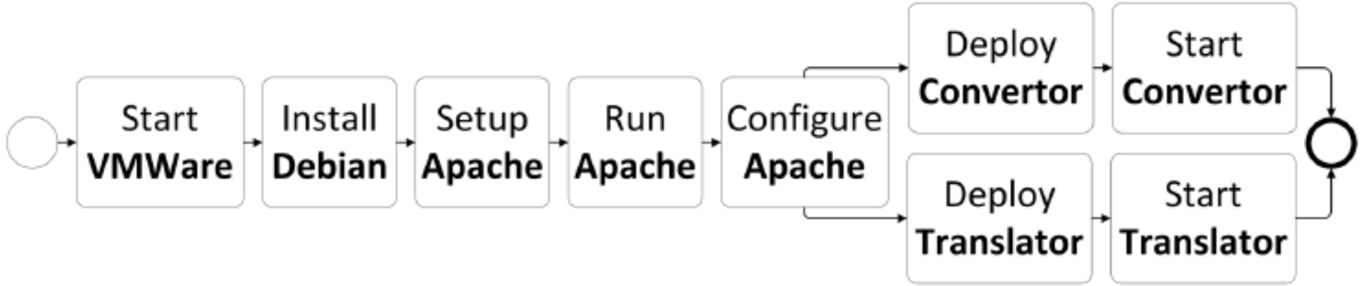
Example: a TOSCA topology



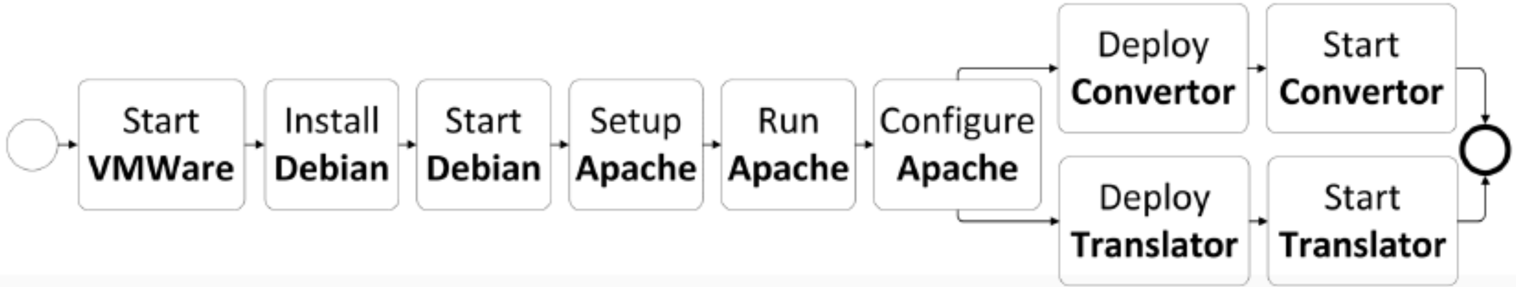
Example: corresponding TOSCA deployment plans



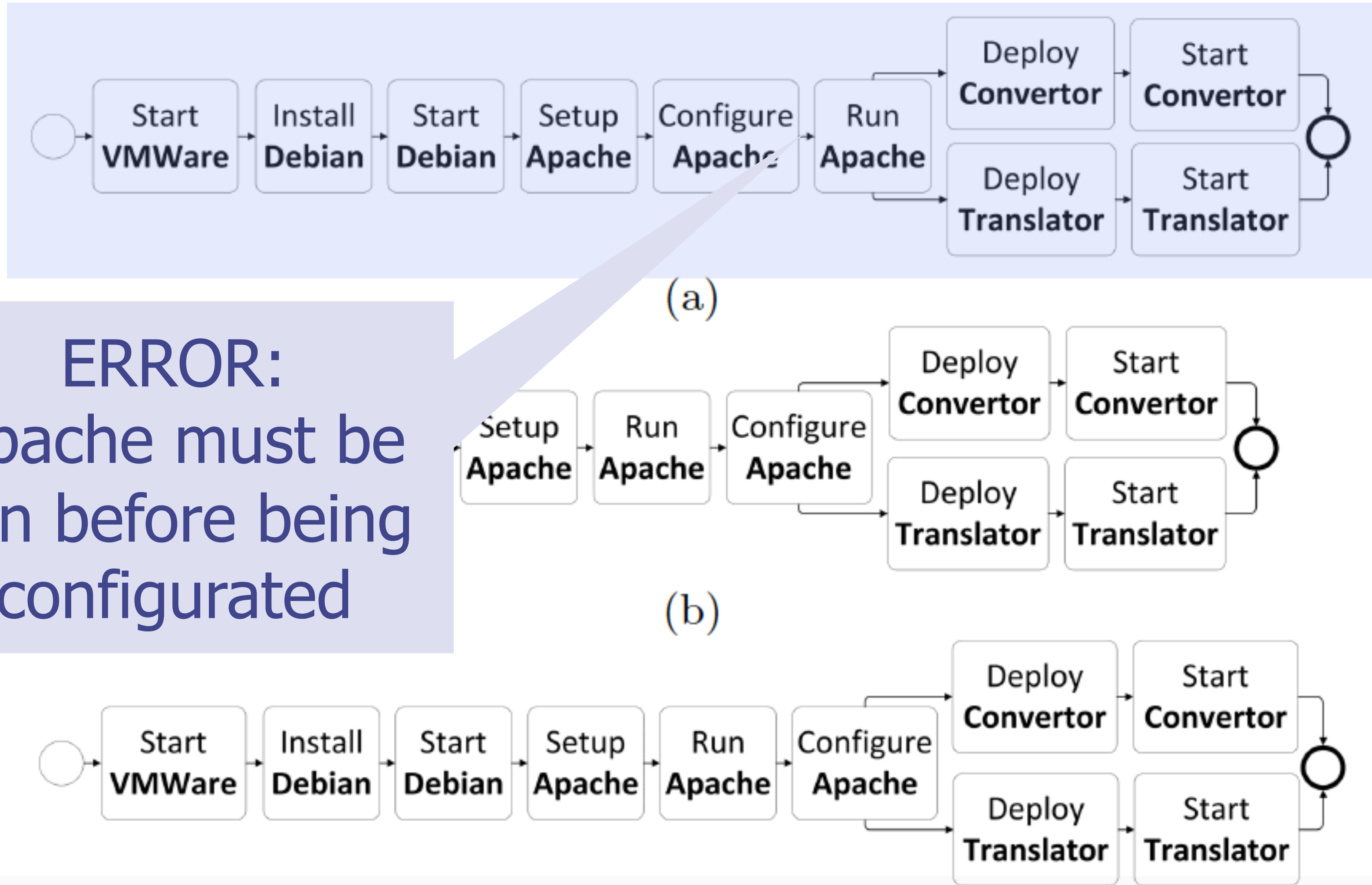
(a)



(b)

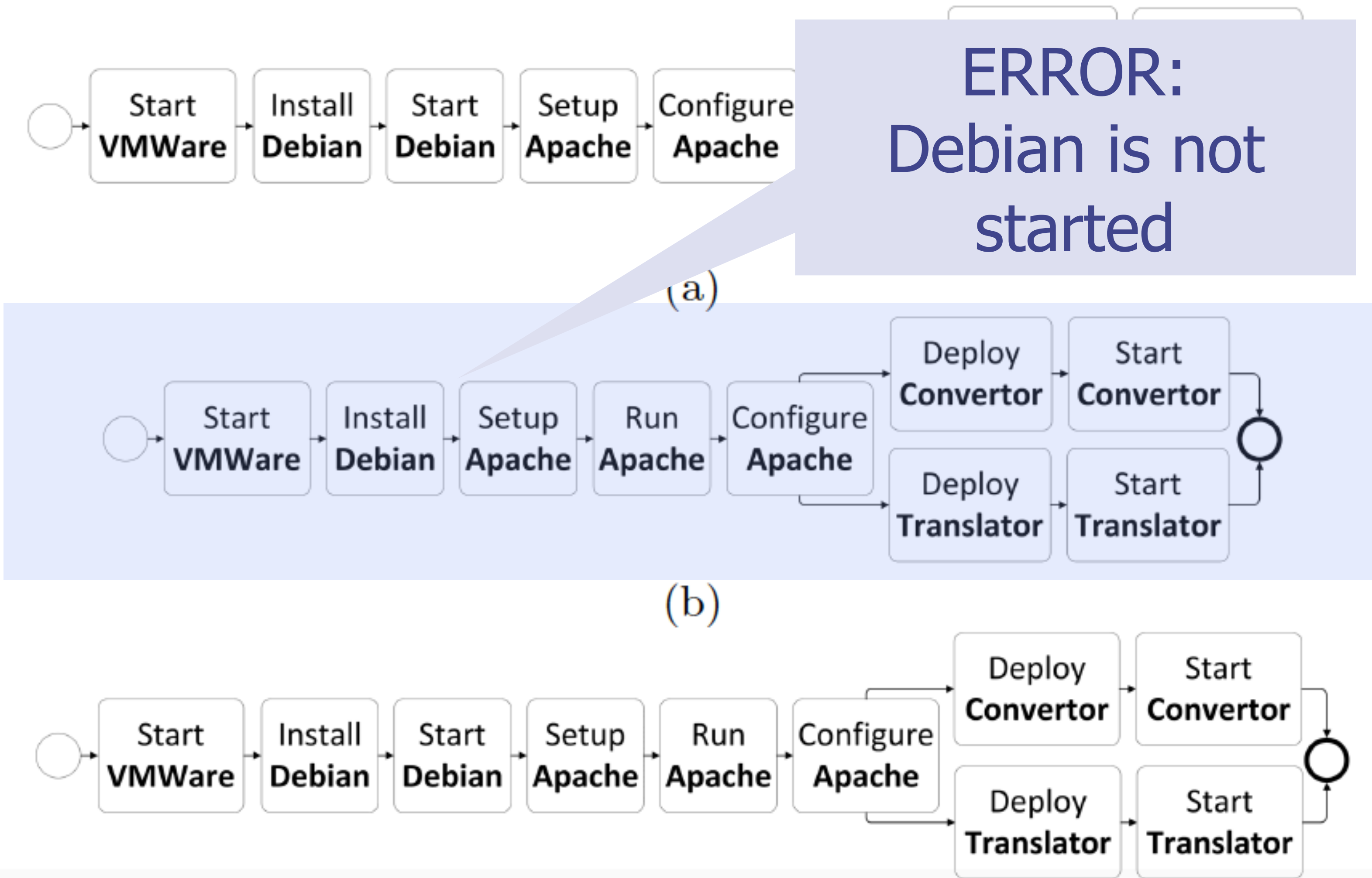


Example: corresponding TOSCA deployment plans

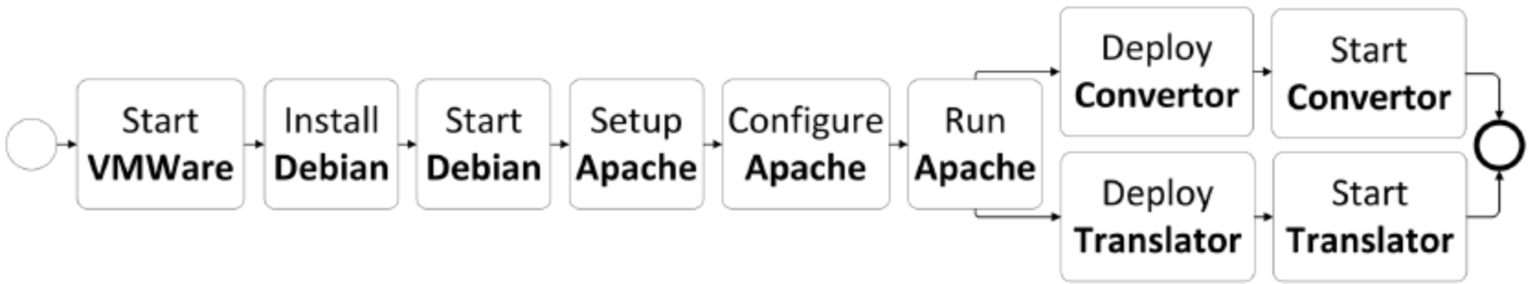


ERROR:
Apache must be run before being configurated

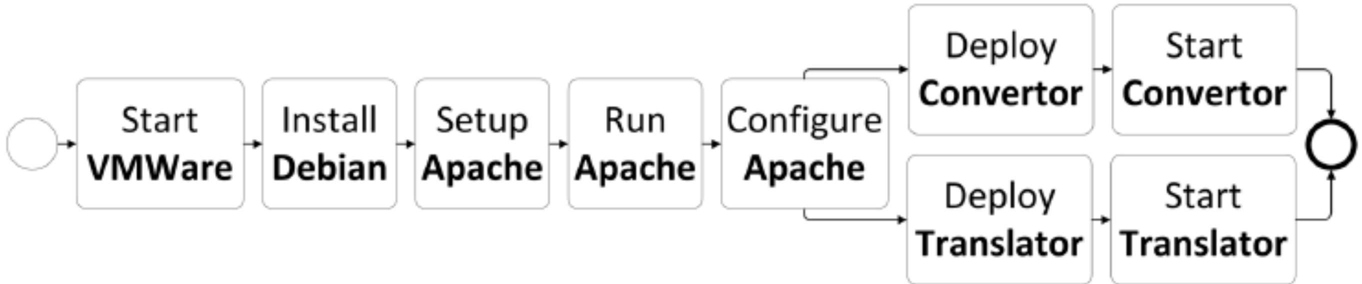
Example: corresponding TOSCA deployment plans



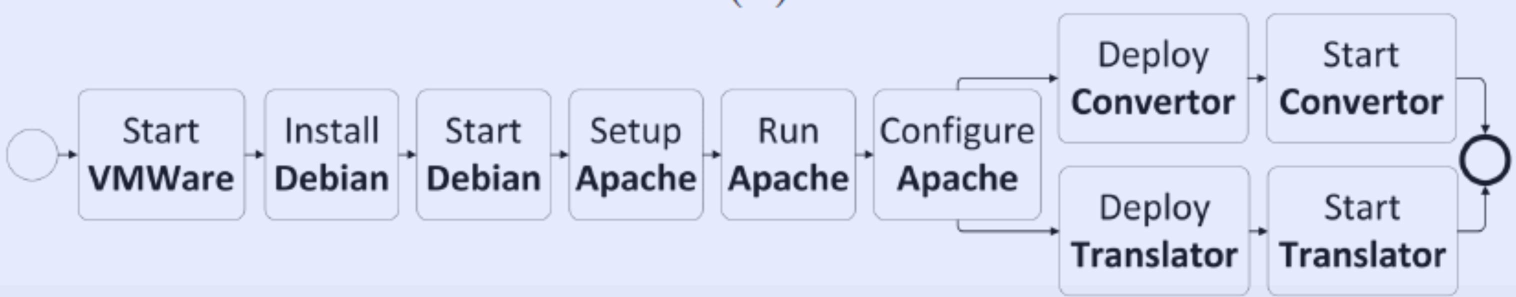
Example: corresponding TOSCA deployment plans



(a)

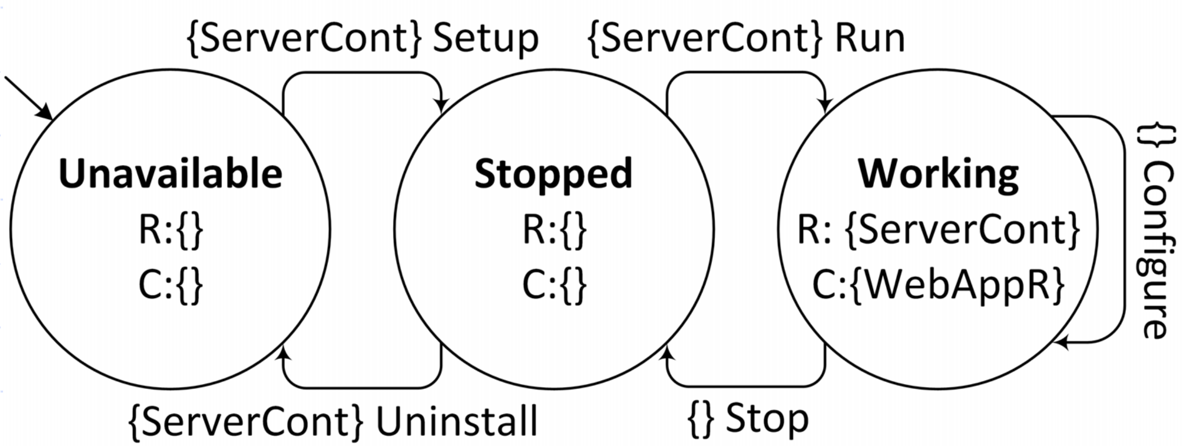


(b)



Automatic check of plans

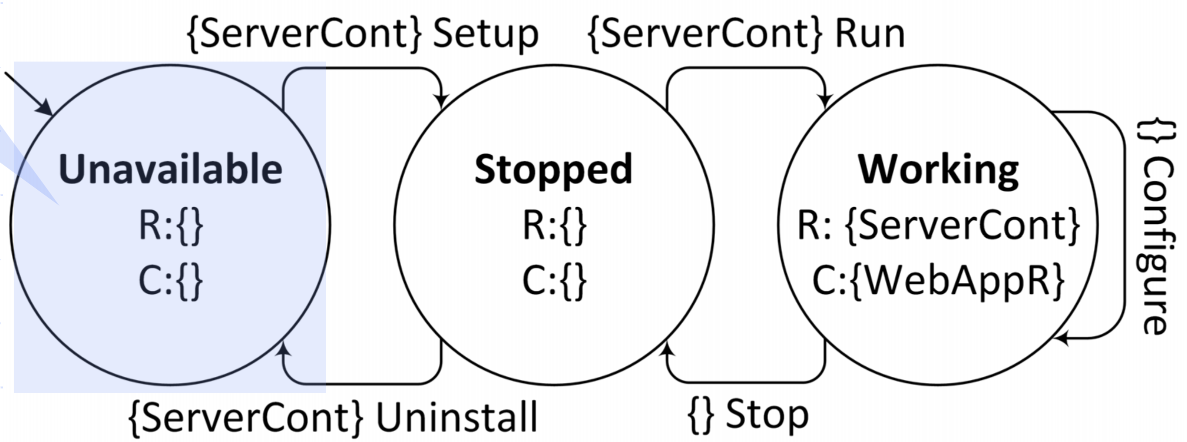
- There is a **lack of information**:
  - We can **intuitively** realise possible errors in the plans …
  - … but to automatically check them, we need to describe local **deployment protocols**



Automatic check of plans

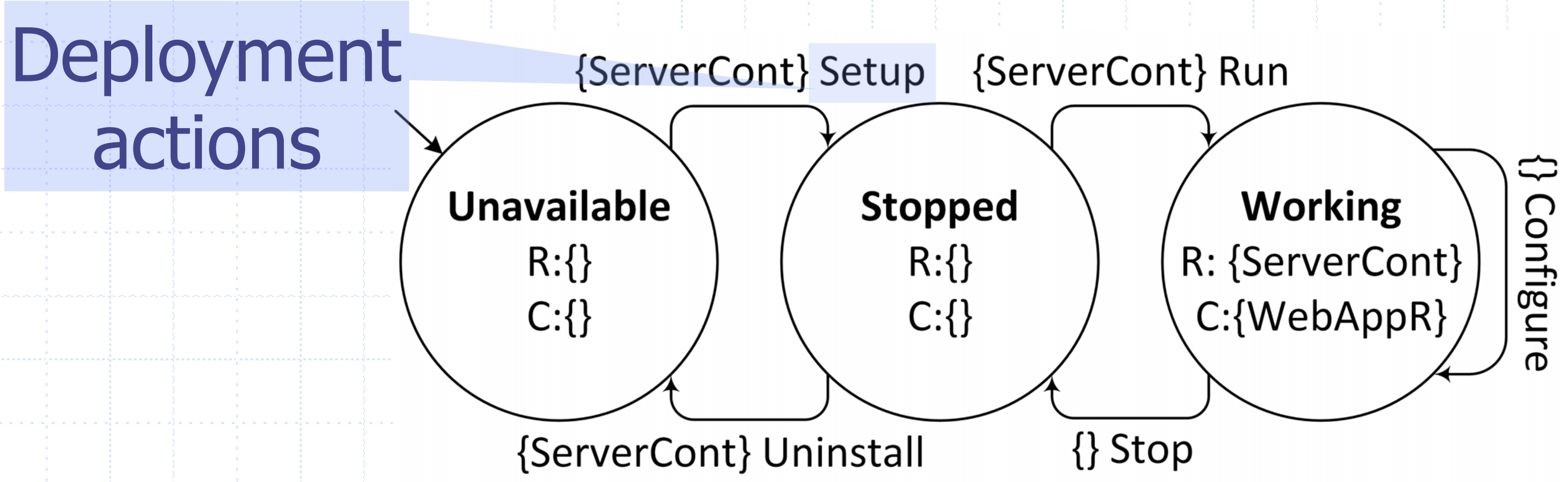
- There is a **lack of information**:
  - We can **intuitively** realise possible errors in the plans…
  - … but to automatically check them, we need to describe local **deployment protocols**

Internal states



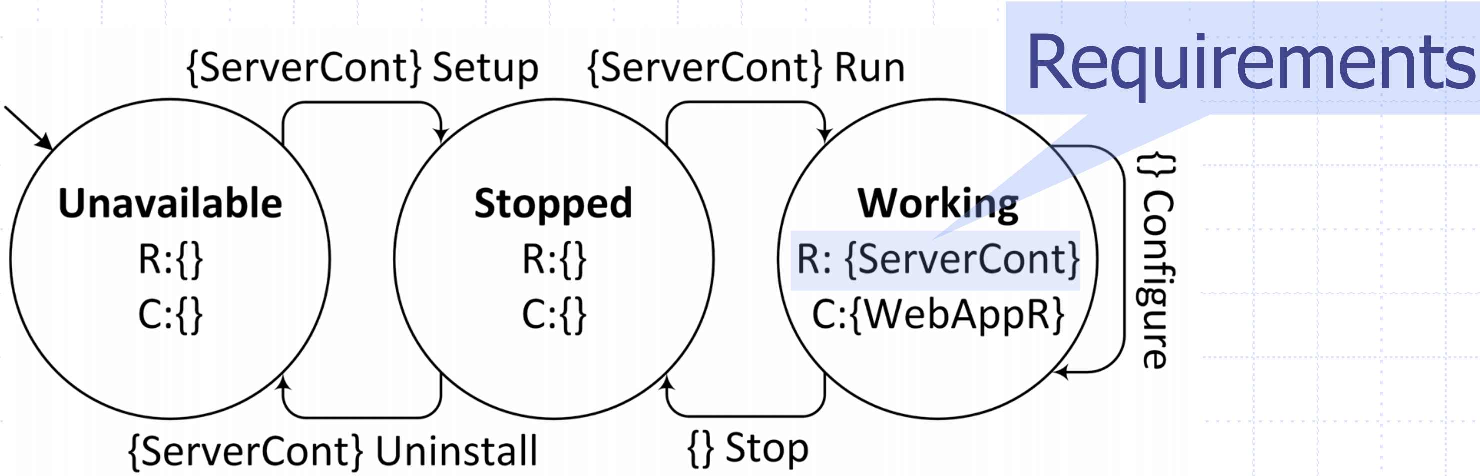
Automatic check of plans

- There is a **lack of information**:
  - We can **intuitively** realise possible errors in the plans…
  - … but to automatically check them, we need to describe local **deployment protocols**



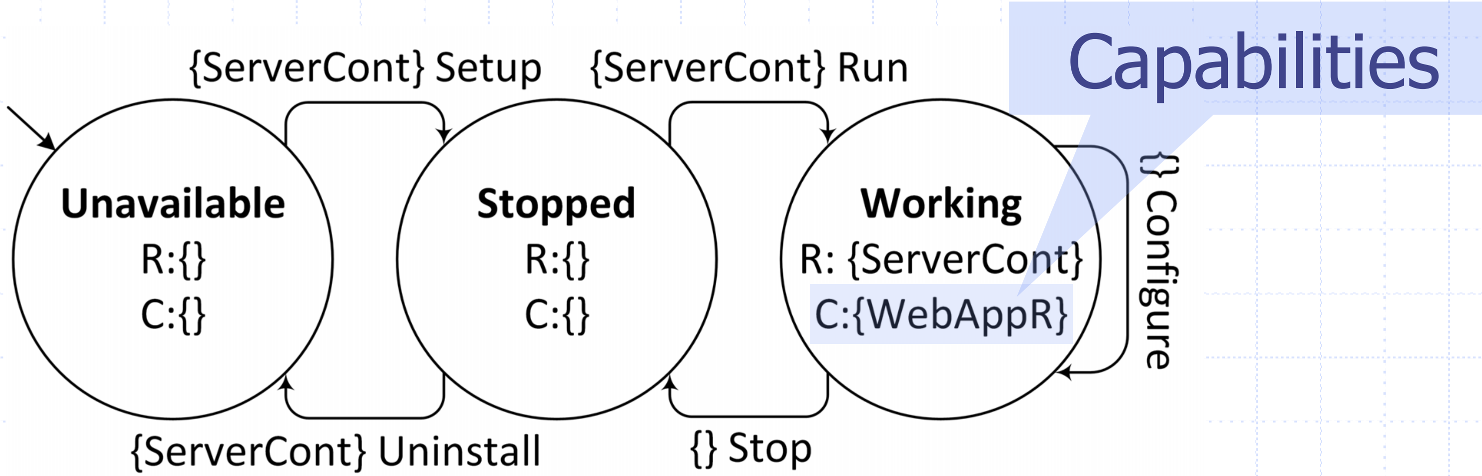
Automatic check of plans

- There is a **lack of information**:
  - We can **intuitively** realise possible errors in the plans…
  - … but to automatically check them, we need to describe local **deployment protocols**



Automatic check of plans

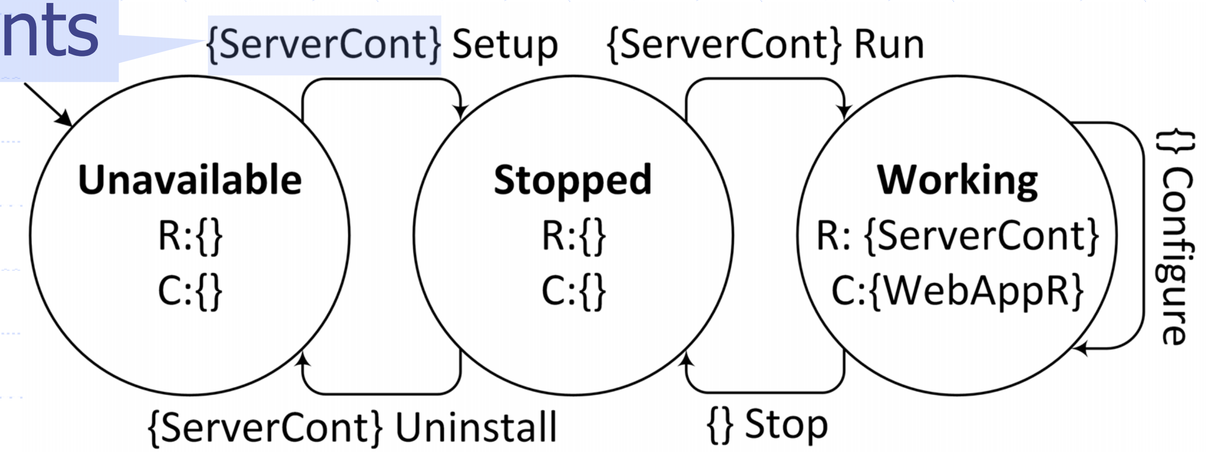
- There is a **lack of information**:
  - We can **intuitively** realise possible errors in the plans…
  - … but to automatically check them, we need to describe local **deployment protocols**



Automatic check of plans

- There is a **lack of information**:
  - We can **intuitively** realise possible errors in the plans…
  - … but to automatically check them, we need to describe local **deployment protocols**

Requirements



A taste of Barrel

- With Barrel you can:
  - Specify the **deployment protocols**
  - Check the **correctness** of a sequence of deployment actions
  - A plan is **not correct** if:
    - Perform an **action** with requirements unsatisfied
    - A component **state** has requirements unsatisfied
  - You can try it at: *http://ranma42.github.io/MProt/*

DEMO

Bottom-up and Top-down approaches

- In both the previously described approaches there are many decisions to be taken **manually**:
  - which software **components** to select
  - the overall application **architecture**
  - the order of the **configuration** actions
  - …

The challenge

- Understand how much of these manual activities can be **automatised**:
  - selection of the software components (selected from appropriate **repositories** like the Juju library)
  - **synthesis** of the overall architecture
  - **planning** of the configuration actions to be executed to realize the expected architecture
  - …

A foundational study of this deployment problem

- We have investigated this **problem**:
  - Defined a formal language for **single** service deployment protocols (similar to Barrel)
  - Formalised the **"automatic deployment"** problem
  - Studied its **complexity** (under several assumptions)

R. Di Cosmo, J. Mauro, S. Zacchiroli, G. Zavattaro:
Aeolus: A component model for the cloud.
Inf. Comput. 239: 100-121 (2014)

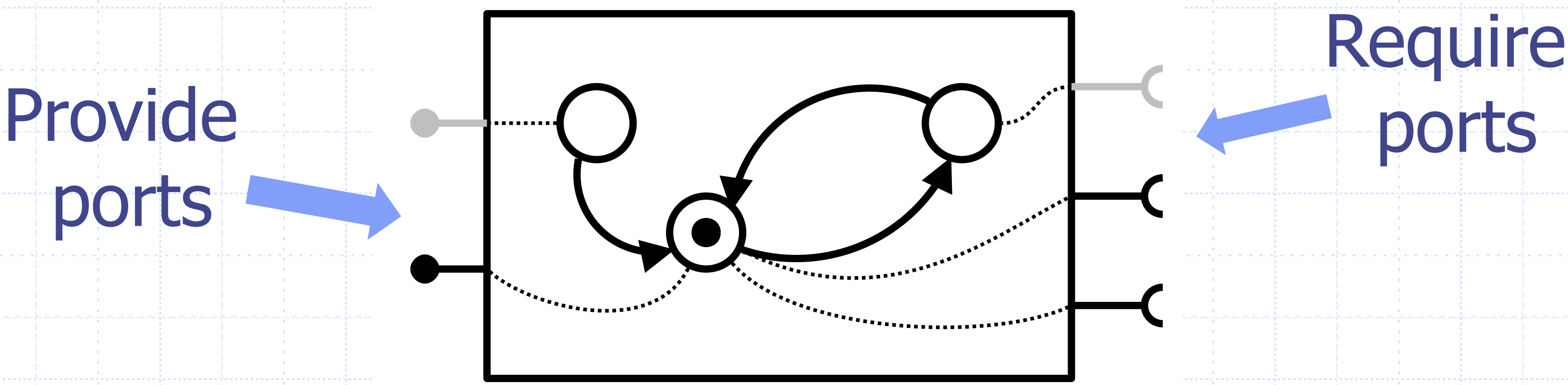
An anticipation about the final result of our research …

- We have implemented a tool that:
  - starting from a **library** of available services (equipped with a local deployment protocol)
  - and the indication of a **target component**
  - computes a **global** deployment plan
    - that reaches a **correct** configuration including at least an instance of the target component

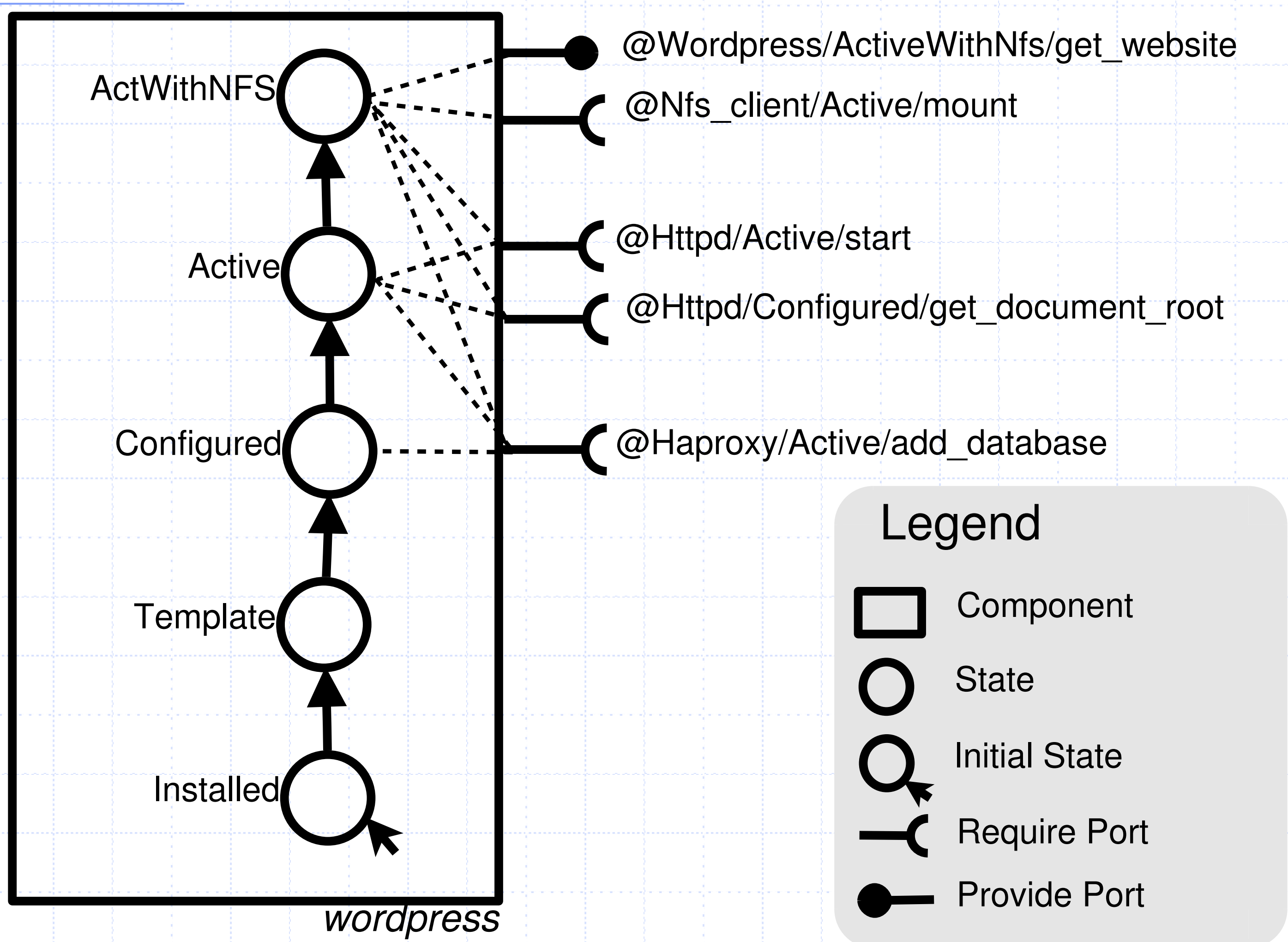
T.A. Lascu, J. Mauro, G. Zavattaro:
Automatic deployment of component-based applications.
Sci. Comput. Program. 113: 261-284 (2015)

Describing the Services: Component types

- A component has **provide** and **require** ports
- A component has an internal **state machine**
- Ports are **active** or **inactive** according to the current internal state

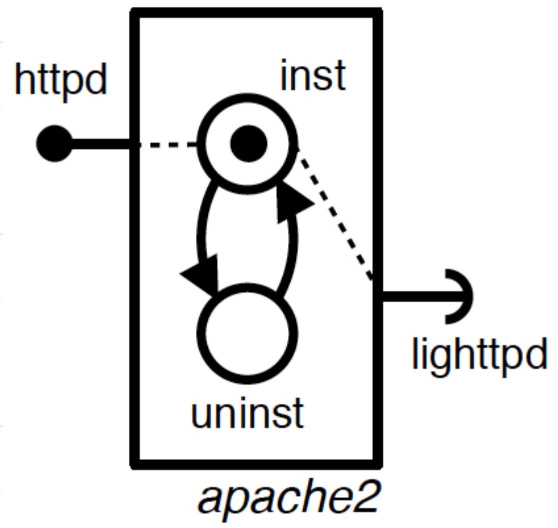


Example: the Wordpress component type



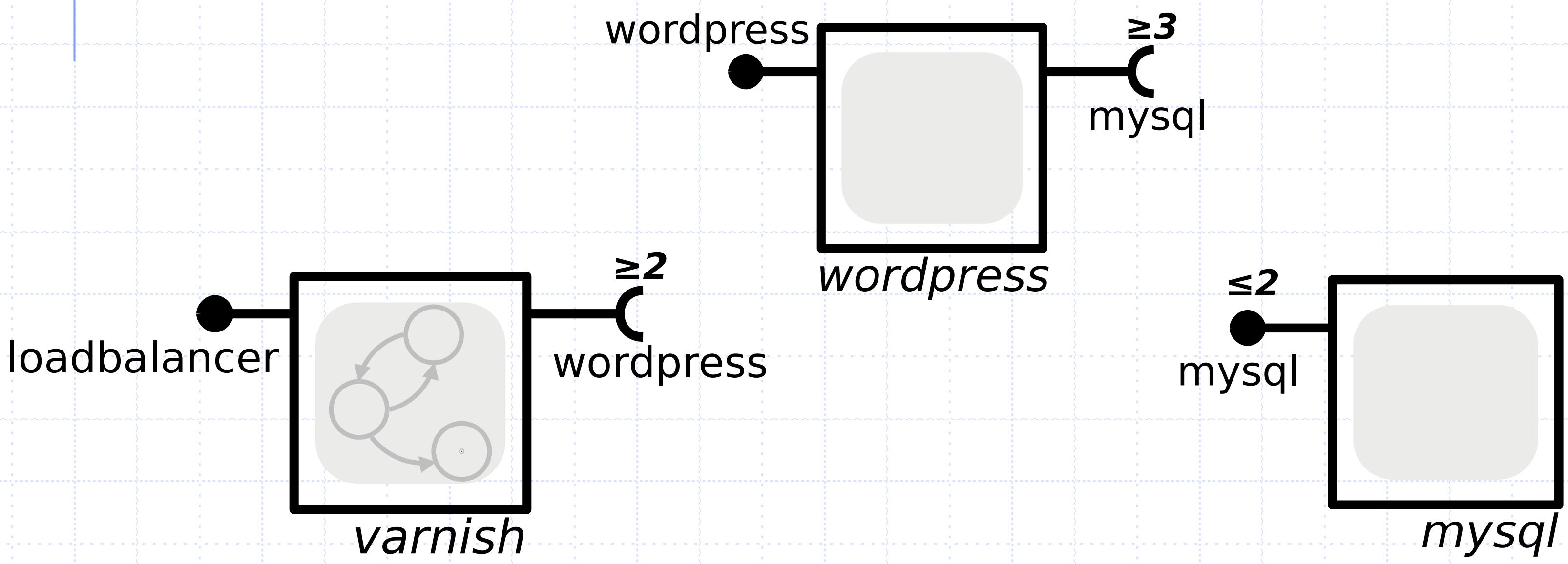
Conflicts

- Conflicts are expressed as **special** ports
  - The apache web server is in **conflict** with the lighttpd web server



Capacity constraints

- Provide (resp. require) ports could have an associated **upper** (resp. **lower**) **bound** to the number of connections

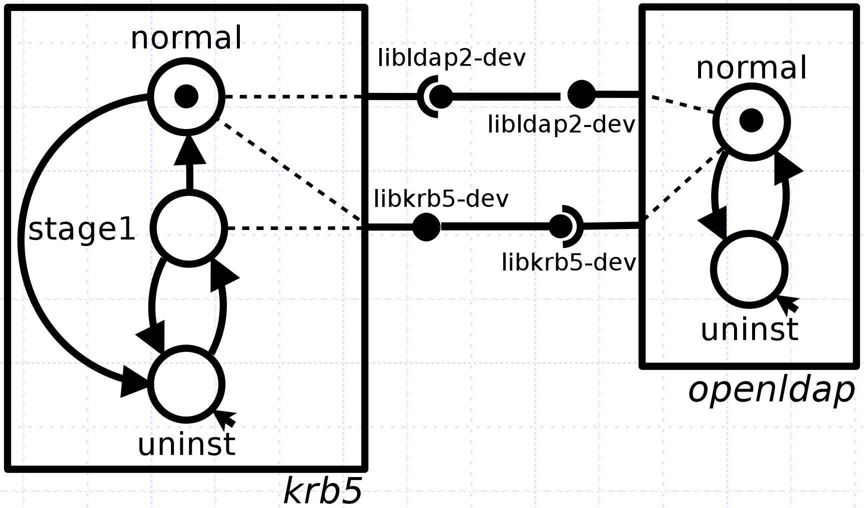


Configurations

- Component **instances**, with a current **state**, and complementary provide/require ports connected by **bindings**

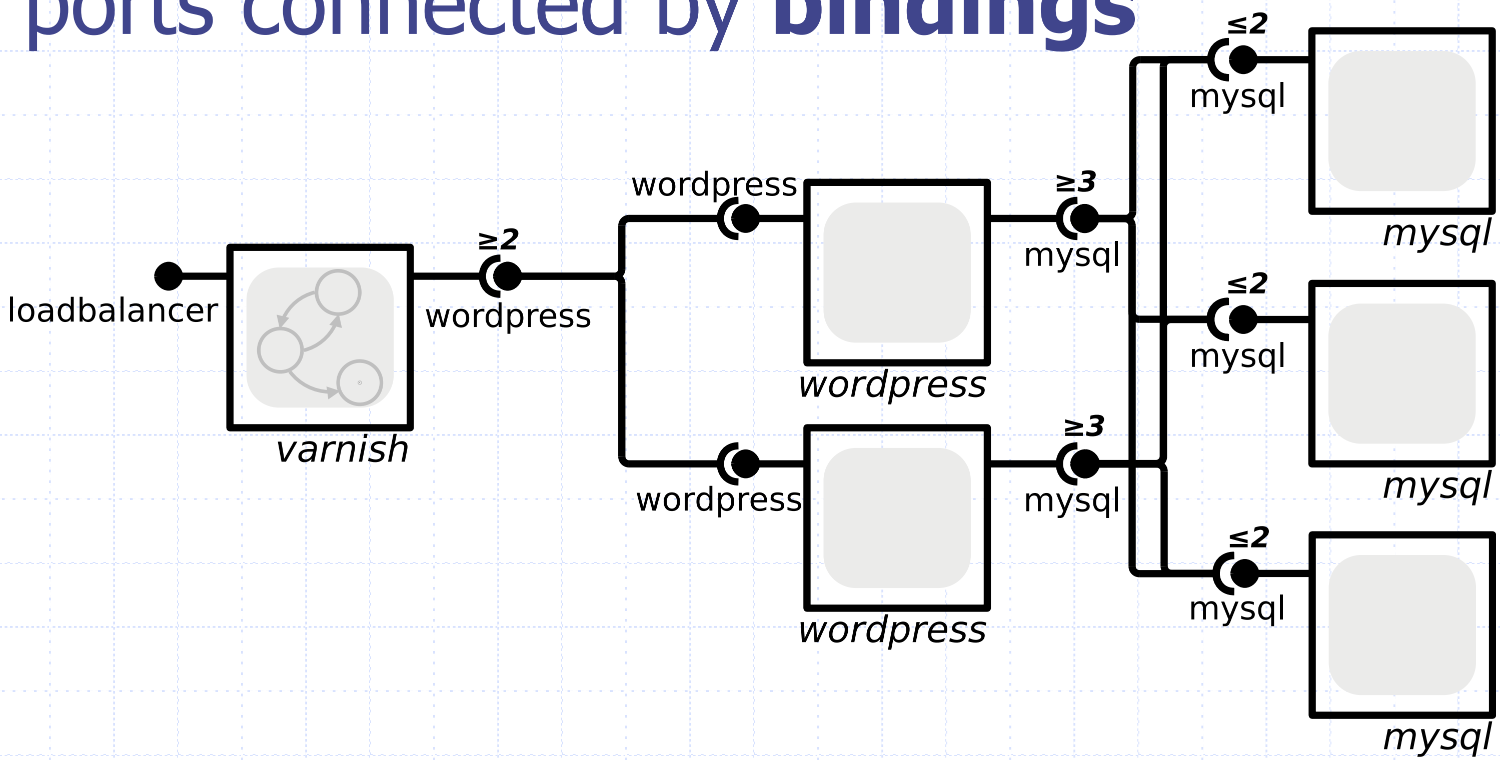
Configurations

- Component **instances**, with a current **state**, and complementary provide/require ports connected by **bindings**
  - Example: Kerberos with ldap support in Debian (example of **circular** dependency)



Configurations

- Component **instances**, with a current **state**, and complementary provide/require ports connected by **bindings**



Formalizing the "deployment" problem

▸ **Definition 1** (Component type). The set Γ of *component types* of the Aeolus model, ranged over by $\mathcal{T}_1, \mathcal{T}_2, \ldots$ contains 5-ple $\langle Q, q_0, T, P, D\rangle$ where:

- Q is a finite set of states;
- $q_0 \in Q$ is the initial state and $T \subseteq Q \times Q$ is the set of *transitions*;
- $P = \langle \mathbf{P}, \mathbf{R}\rangle$, with $\mathbf{P}, \mathbf{R} \subseteq \mathcal{I}$, is a pair composed of the set of *provide* and the set of *require*-ports, respectively;
- D is a function from Q to 2-ple in $(\mathbf{P} \nrightarrow \mathbb{N}^+_\infty) \times (\mathbf{R} \nrightarrow \mathbb{N})$.

Formalizing the "deployment" problem

▸ **Definition 1** (Component type). The set Γ of *component types* of the Aeolus model, ranged over by $\mathcal{T}_1, \mathcal{T}_2, \ldots$ contains 5-ple $\langle Q, q_0, T, P, D\rangle$ where:

- Q is a finite set of states;
- $q_0 \in Q$ is the initial state and $T \subseteq Q \times Q$ is the set of *transitions*;
- $P = \langle \mathbf{P}, \mathbf{R}\rangle$, with $\mathbf{P}, \mathbf{R} \subseteq \mathcal{I}$, is a pair composed of the set of *provide* and the set of *require*-ports, respectively;
- D is a function from Q to 2-ple in $(\mathbf{P} \rightarrow\!\!\!\!\!\rightarrow \mathbb{N}^+_\infty) \times (\mathbf{R} \rightarrow\!\!\!\!\!\rightarrow \mathbb{N})$.

▸ **Definition 2** (Configuration). A *configuration* $\mathcal{C}$ is a quadruple $\langle U, Z, S, B\rangle$ where:

- $U \subseteq \Gamma$ is the finite *universe* of all available component types;
- $Z \subseteq \mathcal{Z}$ is the set of the currently deployed *components*;
- S is the component *state description*, i.e., a function that associates to components in Z a pair $\langle \mathcal{T}, q\rangle$ where $\mathcal{T} \in U$ is a component type $\langle Q, q_0, T, P, D\rangle$, and $q \in Q$ is the current component state;
- $B \subseteq \mathcal{I} \times Z \times Z$ is the set of *bindings*, namely 3-ples composed by an interface, the component that requires that interface, and the component that provides it; we assume that the two components are distinct.

Formalizing the "deployment" problem

- **Definition 5** (Actions). The set $\mathcal{A}$ contains the following actions:
  - $stateChange(z, q_1, q_2)$ where $z \in \mathcal{Z}$: change the state of the component z from q_1 to q_2;
  - $bind(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$: add a binding between z_1 and z_2 on port r;
  - $unbind(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$: remove the specified binding;
  - $new(z : \mathcal{T})$ where $z \in \mathcal{Z}$ and $\mathcal{T}$ is a component type: add a new component z of type $\mathcal{T}$;
  - $del(z)$ where $z \in \mathcal{Z}$: remove the component z from the configuration.

Formalizing the "deployment" problem

▶ **Definition 6** (Reconfigurations). Reconfigurations are denoted by transitions $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ meaning that the execution of $\alpha \in \mathcal{A}$ on the configuration $\mathcal{C}$ produces a new configuration $\mathcal{C}'$. The transitions from a configuration $\mathcal{C} = \langle U, Z, S, B\rangle$ are defined as follows:

$$\mathcal{C} \xrightarrow{stateChange(z,q_1,q_2)} \langle U, Z, S', B\rangle$$
$$\text{if } \mathcal{C}[z].\mathtt{state} = q_1$$
$$\text{and } (q_1, q_2) \in \mathcal{C}[z].\mathtt{trans}$$
$$\text{and } S'(z') = \begin{cases} \langle \mathcal{C}[z].\mathtt{type}, q_2\rangle & \text{if } z' = z \\ \mathcal{C}[z'] & \text{otherwise} \end{cases}$$

$$\mathcal{C} \xrightarrow{new(z:\mathcal{T})} \langle U, Z \cup \{z\}, S', B\rangle$$
$$\text{if } z \notin Z, \mathcal{T} \in U$$
$$\text{and } S'(z') = \begin{cases} \langle \mathcal{T}, \mathcal{T}.\mathtt{init}\rangle & \text{if } z' = z \\ \mathcal{C}[z'] & \text{otherwise} \end{cases}$$

$$\mathcal{C} \xrightarrow{bind(r,z_1,z_2)} \langle U, Z, S, B \cup \langle r, z_1, z_2\rangle\rangle$$
$$\text{if } \langle r, z_1, z_2\rangle \notin B$$
$$\text{and } r \in \mathcal{C}[z_1].\mathtt{req} \cap \mathcal{C}[z_2].\mathtt{prov}$$

$$\mathcal{C} \xrightarrow{del(z)} \langle U, Z \setminus \{z\}, S', B'\rangle$$
$$\text{if } S'(z') = \begin{cases} \perp & \text{if } z' = z \\ \mathcal{C}[z'] & \text{otherwise} \end{cases}$$
$$\text{and } B' = \{\langle r, z_1, z_2\rangle \in B \mid z \notin \{z_1, z_2\}\}$$

$$\mathcal{C} \xrightarrow{unbind(r,z_1,z_2)} \langle U, Z, S, B \setminus \langle r, z_1, z_2\rangle\rangle \quad \text{if } \langle r, z_1, z_2\rangle \in B$$

"Deployment" problem

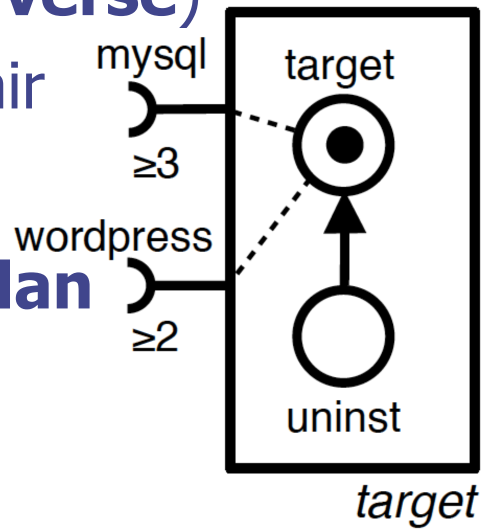
- Input:
  - A set of component types (called **Universe**)
  - One **target** component type-state pair
- Output:
  - **Yes**, if there exists a **deployment plan**
  - **No**, otherwise

Deployment plan:

a sequence of actions leading to a final configuration containing at least one component of the given target type, in the given target state

“Deployment” problem

- Input:
  - A set of component types (called **Universe**)
  - One **target** component type-state pair
- Output:
  - **Yes**, if there exists a **deployment plan**
  - **No**, otherwise

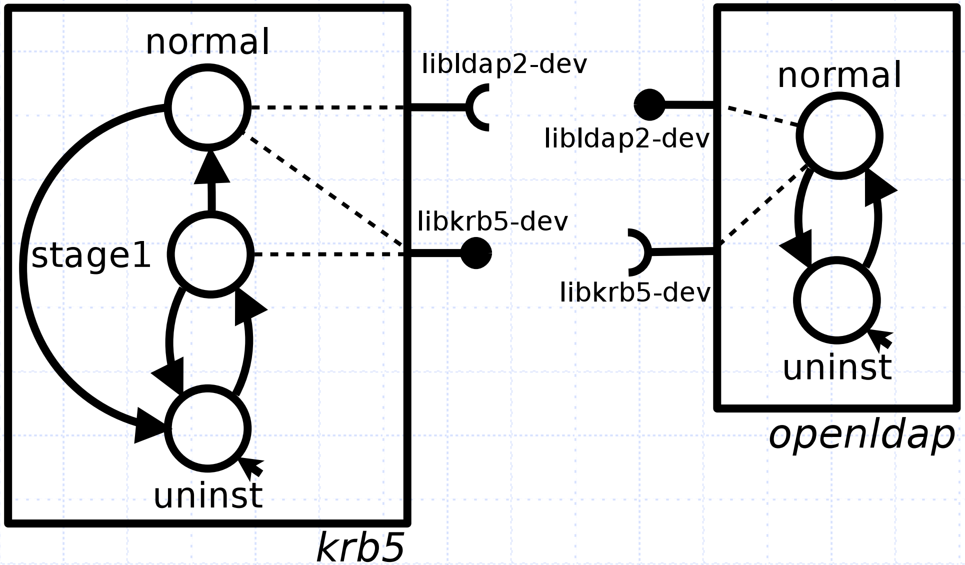


Deployment plan:

a sequence of actions leading to a final configuration containing at least one component of the given target type, in the given target state

Deployment problem: example

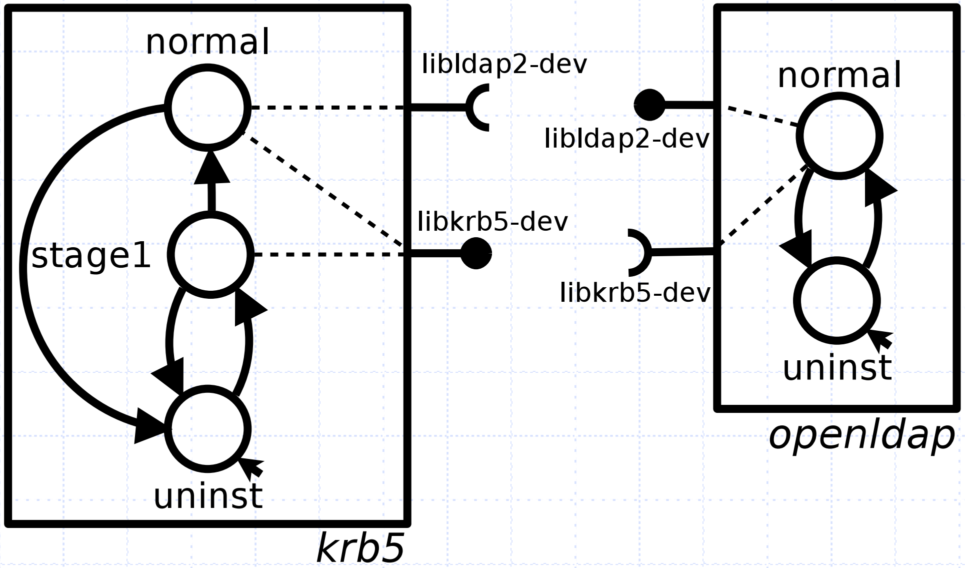
- Consider the problem of installing kerberos with ldap support in Debian
  - **Universe**: packages krb5 and openldap
  - **Target**: krb5 in normal state



Deployment problem: example

- Deployment plan:

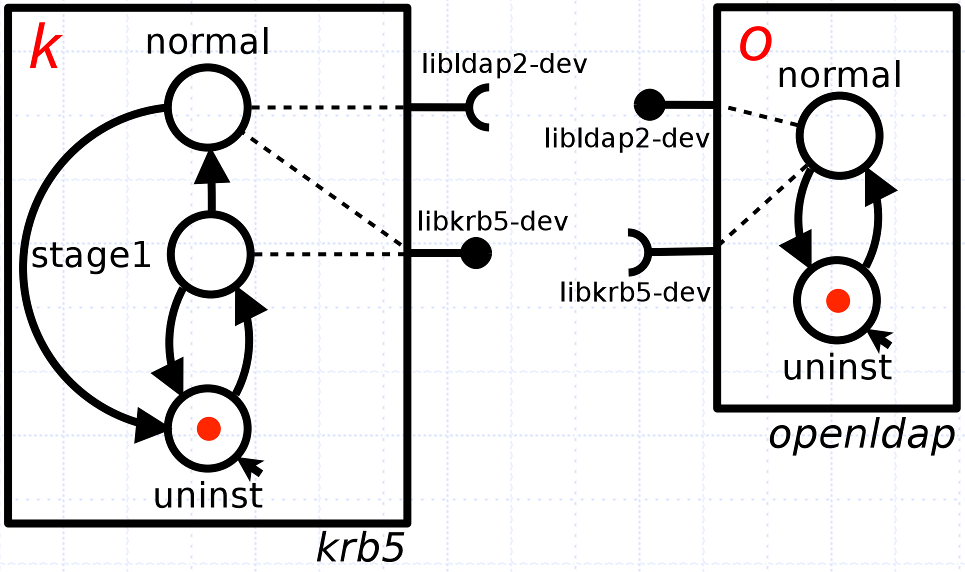
```
new(k:krb5),new(o:openldap),
stateChange(k,uninst,stage1),
bind(libkrb5-dev,o,k),stateChange(o,uninst,normal),
bind(libldap2-dev,k,o),
stateChange(k,stage1,normal)
```



Deployment problem: example

- Deployment plan:

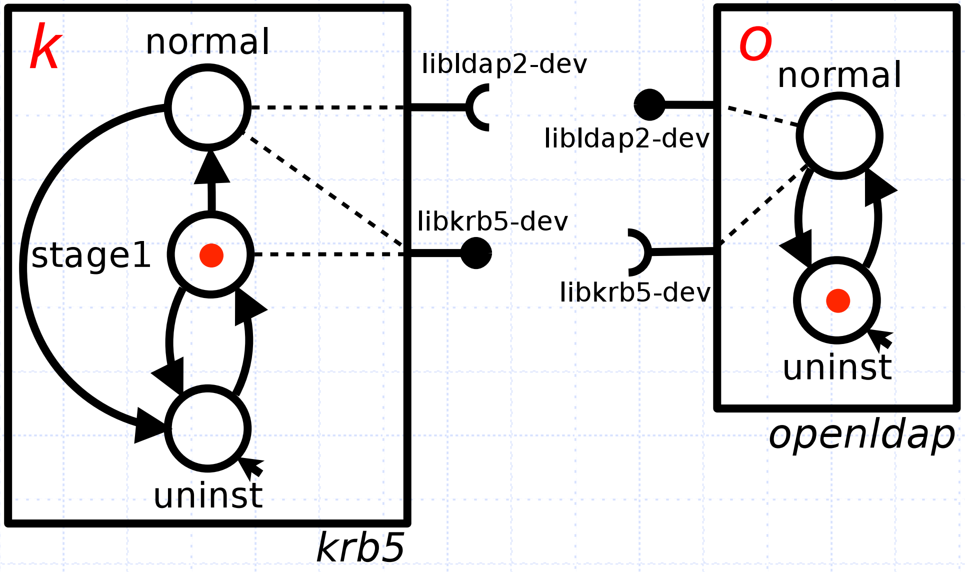
```
new(k:krb5),new(o:openldap),
stateChange(k,uninst,stage1),
bind(libkrb5-dev,o,k),stateChange(o,uninst,normal),
bind(libldap2-dev,k,o),
stateChange(k,stage1,normal)
```



Deployment problem: example

- Deployment plan:

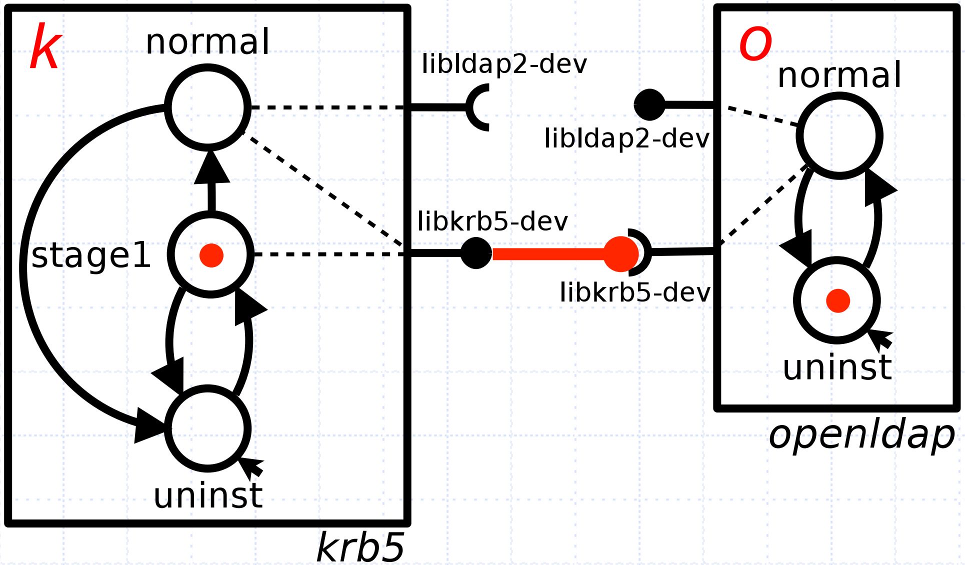
```
new(k:krb5),new(o:openldap),
stateChange(k,uninst,stage1),
bind(libkrb5-dev,o,k),stateChange(o,uninst,normal),
bind(libldap2-dev,k,o),
stateChange(k,stage1,normal)
```



Deployment problem: example

- Deployment plan:

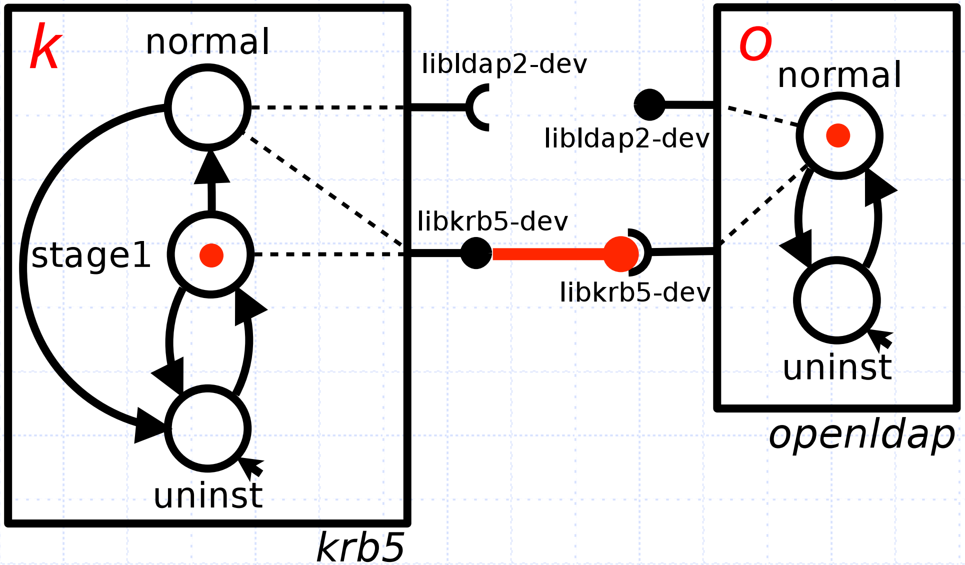
  new(k:krb5),new(o:openldap),
  stateChange(k,uninst,stage1),
  bind(libkrb5-dev,o,k),stateChange(o,uninst,normal),
  bind(libldap2-dev,k,o),
  stateChange(k,stage1,normal)



Deployment problem: example

- Deployment plan:

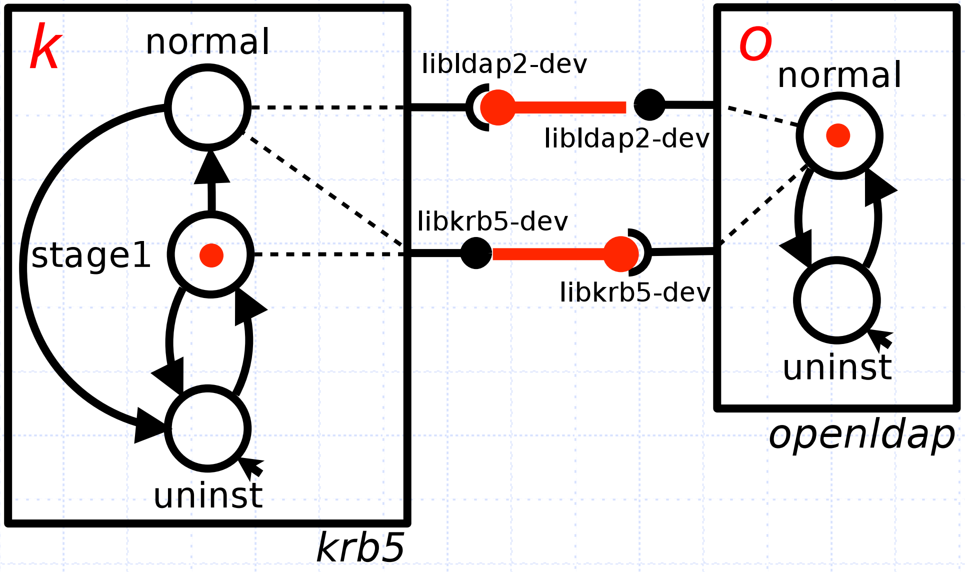
```
new(k:krb5),new(o:openldap),
stateChange(k,uninst,stage1),
bind(libkrb5-dev,o,k),stateChange(o,uninst,normal),
bind(libldap2-dev,k,o),
stateChange(k,stage1,normal)
```



Deployment problem: example

- Deployment plan:

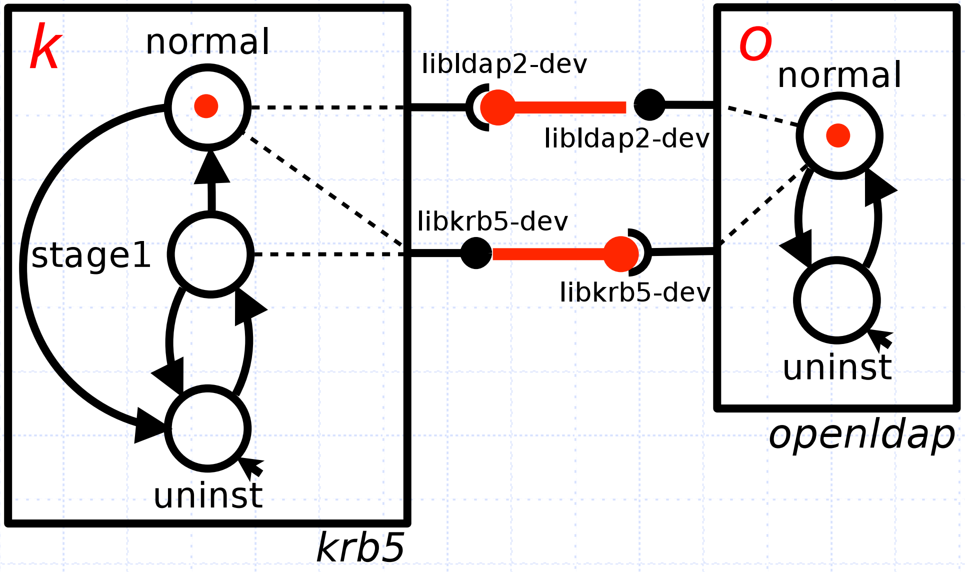
```
new(k:krb5),new(o:openldap),
stateChange(k,uninst,stage1),
bind(libkrb5-dev,o,k),stateChange(o,uninst,normal),
bind(libldap2-dev,k,o),
stateChange(k,stage1,normal)
```



Deployment problem: example

- Deployment plan:

```
new(k:krb5),new(o:openldap),
stateChange(k,uninst,stage1),
bind(libkrb5-dev,o,k),stateChange(o,uninst,normal),
bind(libldap2-dev,k,o),
stateChange(k,stage1,normal)
```



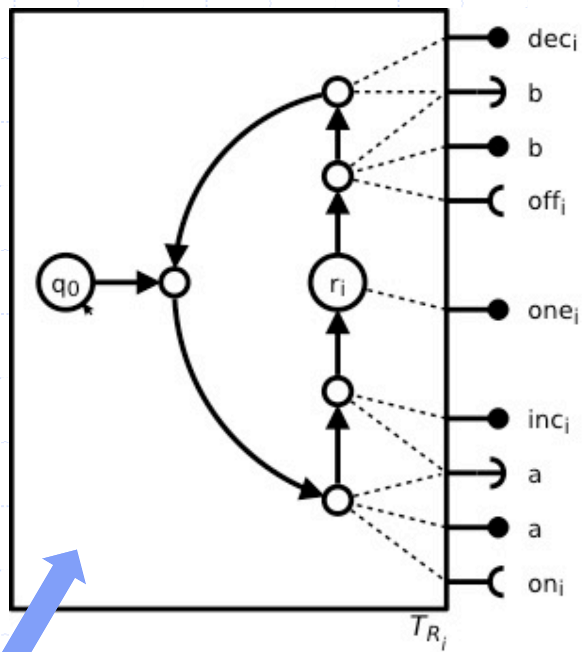
Summary of decidability/complexity results

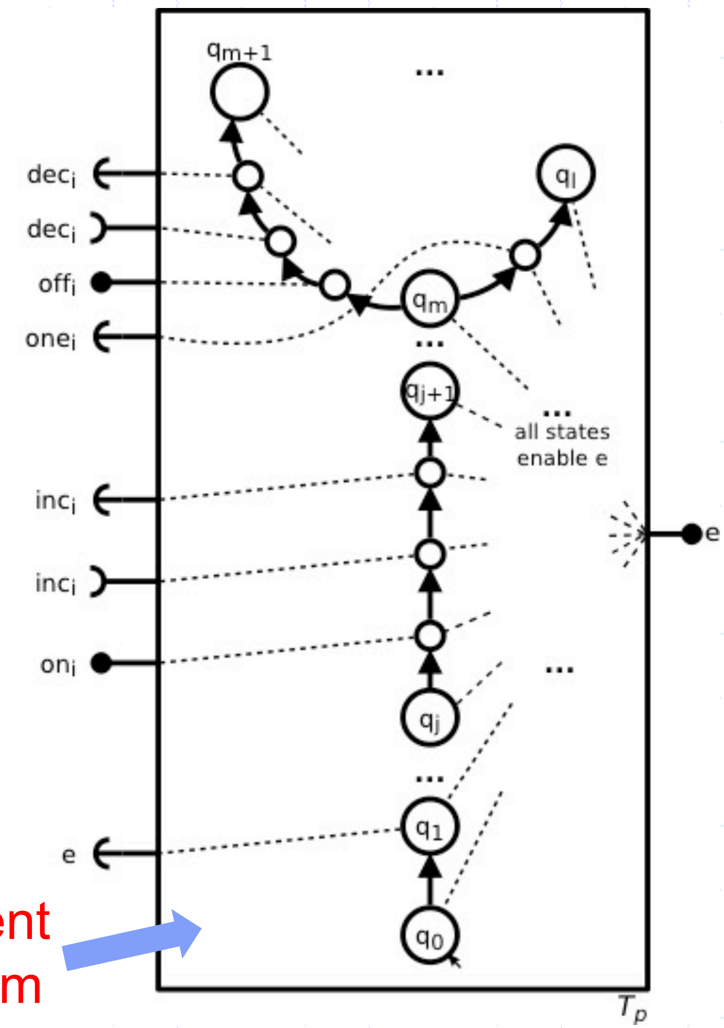
Component model	**Deployment is**
Full component model	Undecidable

Deployment undecidable

- We prove that the deployment problem is **undecidable**
- The proof is by reduction from 2 counter machines (2CMs)
  - A program composed of **increment**, **decrement** or **jump-if-zero**, or **halt** instructions…
  - …on two **counters** holding natural numbers

Encoding 2CMs



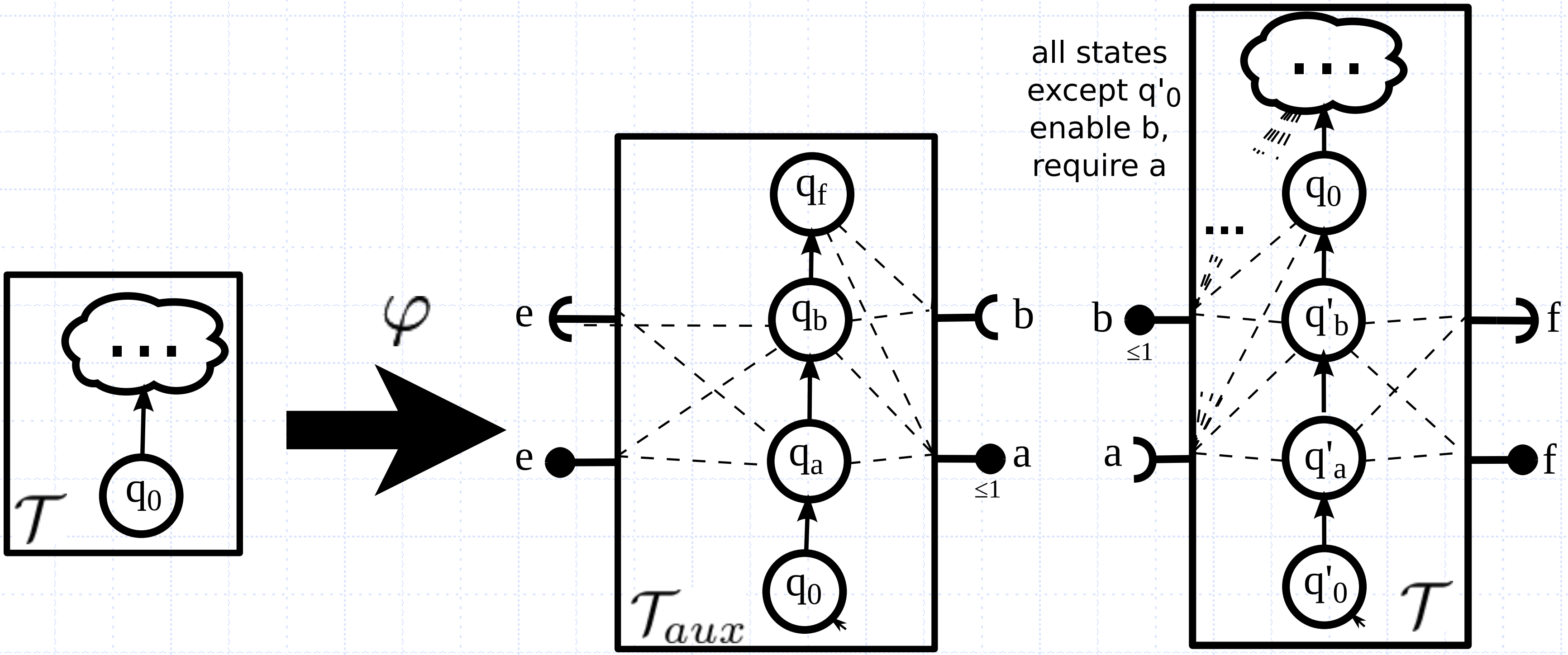


One component for
each unit in a counter

One component
for the program

..with persistent unit components

- To avoid unit component **deletion**, we realise a "lively" embrace

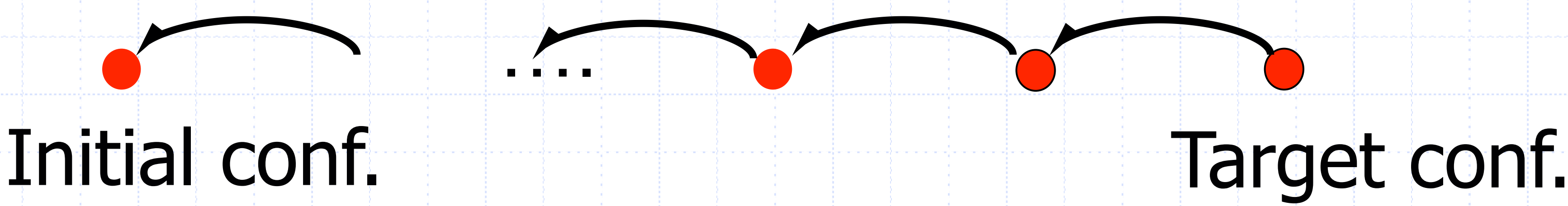


Summary of decidability/complexity results

Component model	Deployment is
Full component model	Undecidable
No capacity constraints	Ackermann-hard

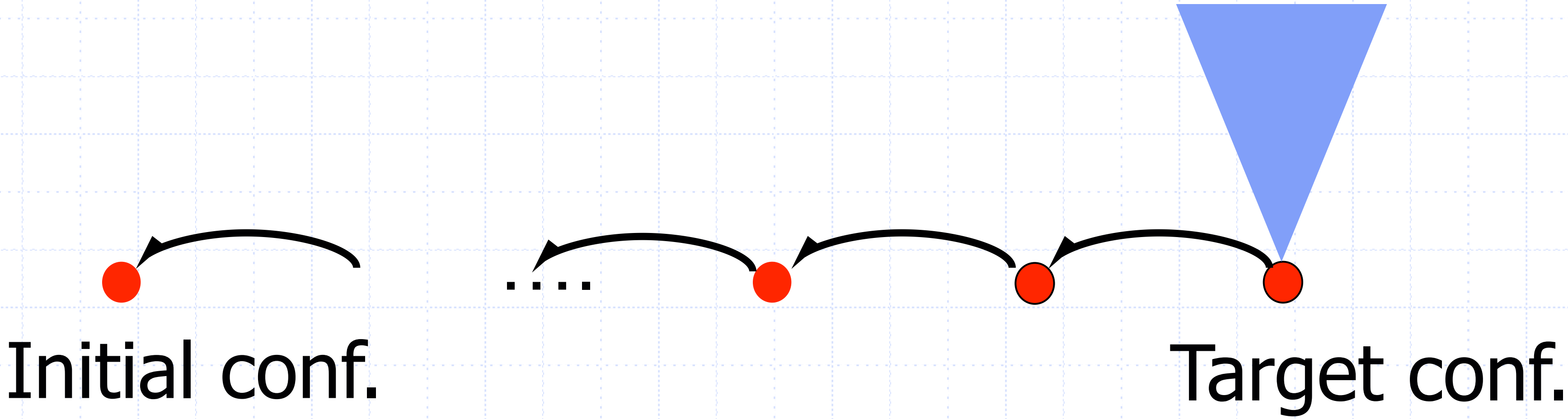
Decidability result without capacity constraints

- **Backward** search algorithm based on the theory of WSTS (Well-Structured Transition Systems)
  - WSTS are popular in the context of infinite state systems verification



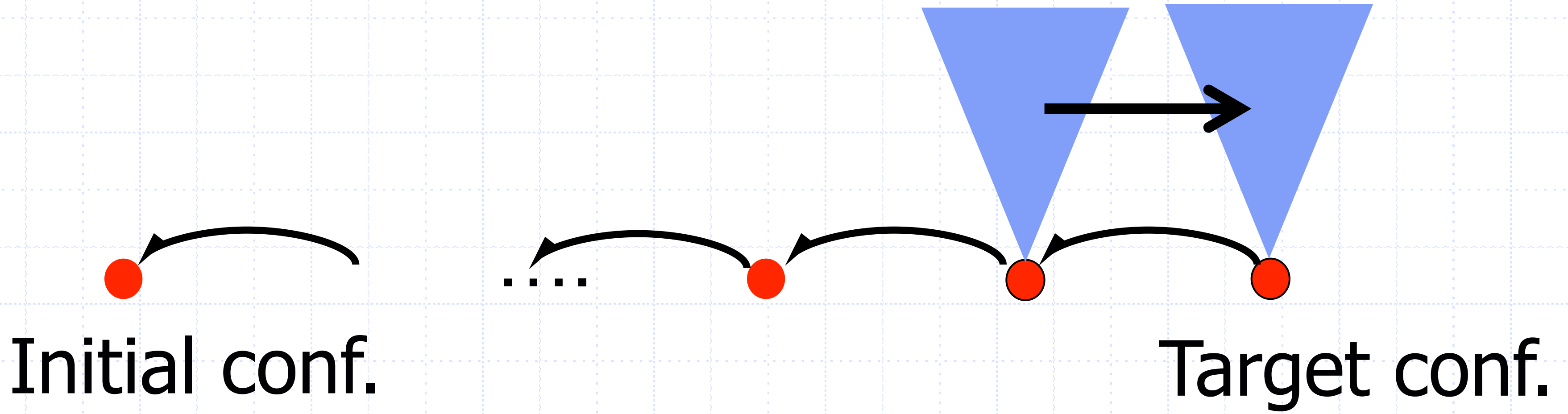
Decidability result without capacity constraints

- Key point:
  ordering $C_1 \leq C_2$ on configurations s.t.
  - if C_1 has a given component, also C_2 has it



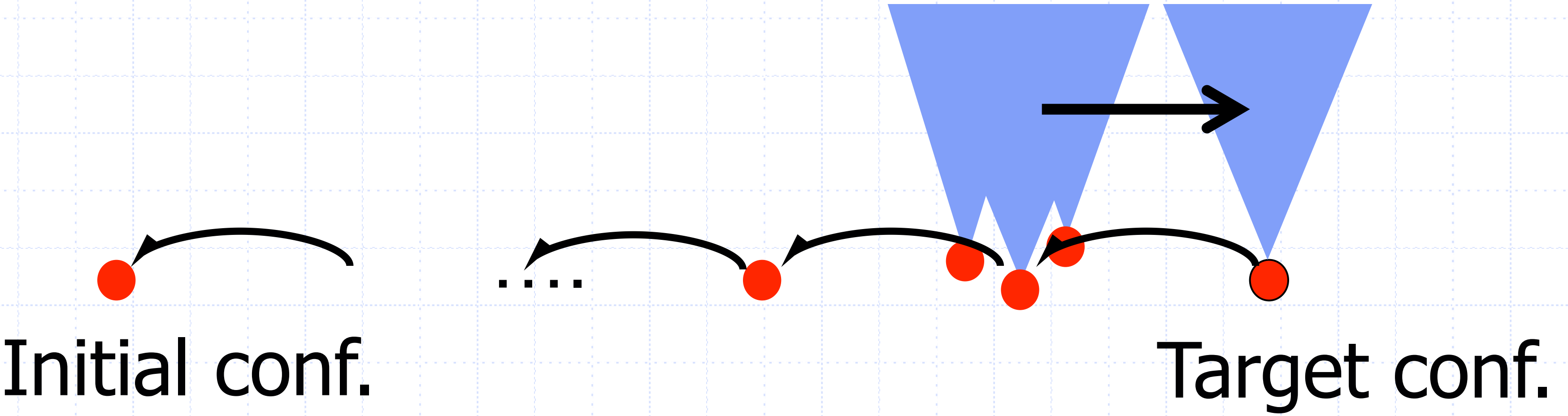
Decidability result without capacity constraints

- Key point:
  ordering $C_1 \leq C_2$ on configurations s.t.
  - if C_1 has a given component, also C_2 has it
  - if $C_1 \leq C_2$ and $C_1 \rightarrow C_1'$ then $C_2 \rightarrow C_2'$ with $C_1' \leq C_2'$



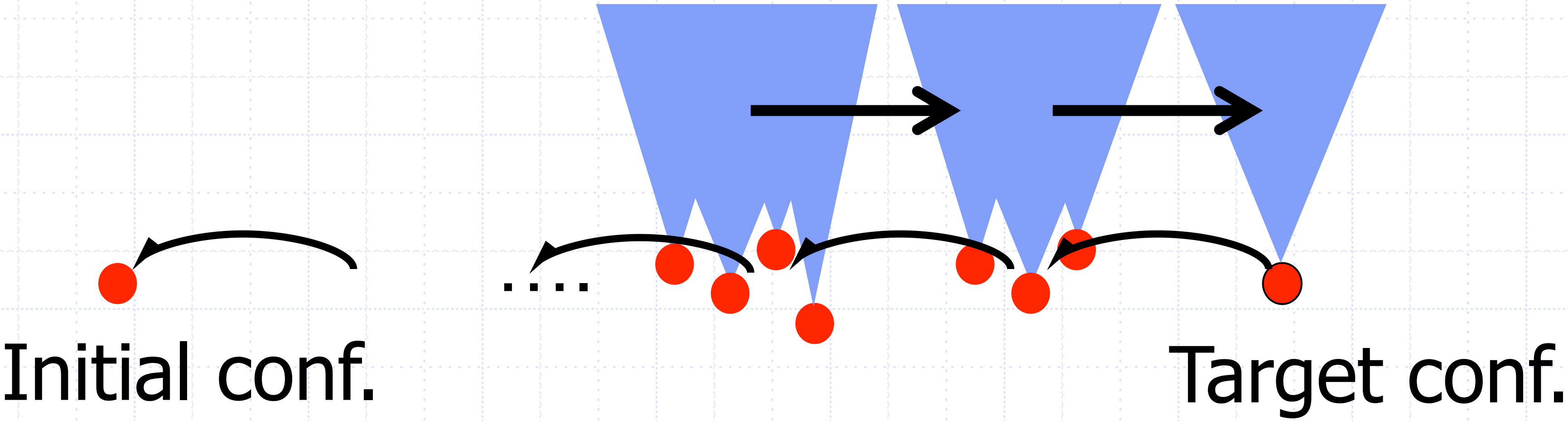
Decidability result without capacity constraints

- Key point:
  ordering $C_1 \leq C_2$ on configurations s.t.
  - if C_1 has a given component, also C_2 has it
  - if $C_1 \leq C_2$ and $C_1 \rightarrow C_1'$ then $C_2 \rightarrow C_2'$ with $C_1' \leq C_2'$
  - $\leq$ is a wqo: finite basis and fixpoint guaranteed



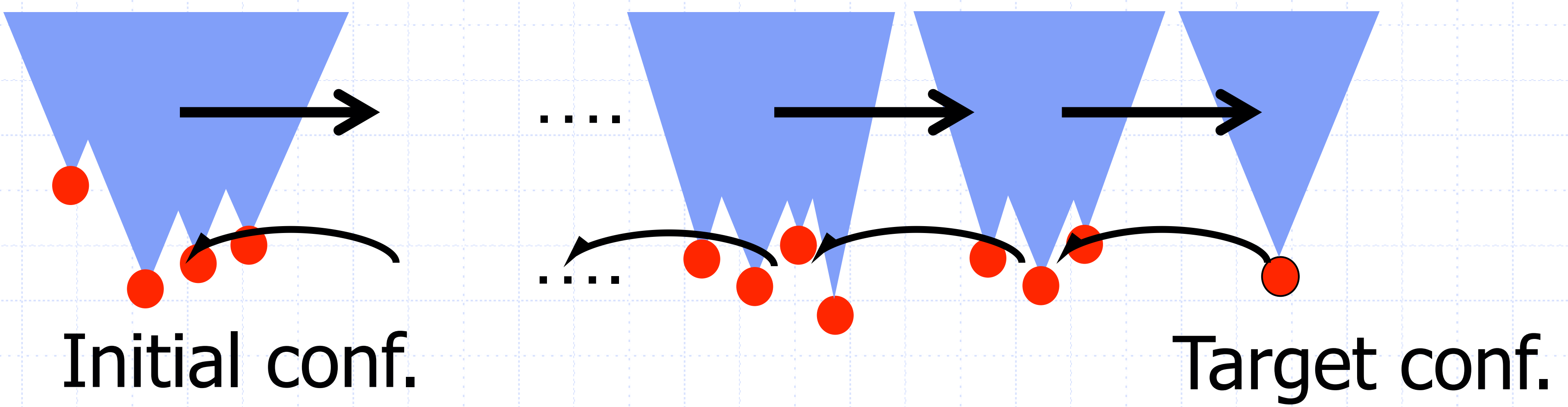
Decidability result without capacity constraints

- Key point:
  ordering $C_1 \leq C_2$ on configurations s.t.
  - if C_1 has a given component, also C_2 has it
  - if $C_1 \leq C_2$ and $C_1 \rightarrow C_1'$ then $C_2 \rightarrow C_2'$ with $C_1' \leq C_2'$
  - $\leq$ is a wqo: finite basis and fixpoint guaranteed



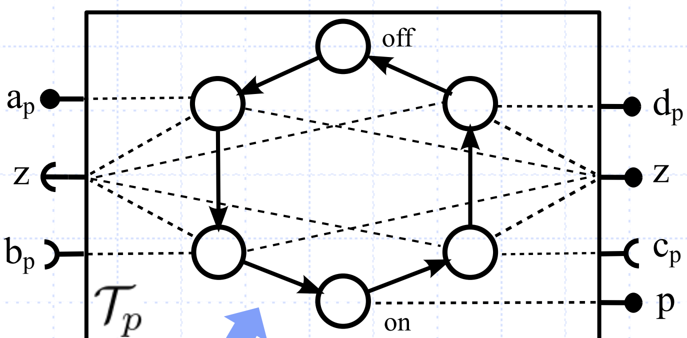
Decidability result without capacity constraints

- Key point:
  ordering $C_1 \leq C_2$ on configurations s.t.
  - if C_1 has a given component, also C_2 has it
  - if $C_1 \leq C_2$ and $C_1 \rightarrow C_1'$ then $C_2 \rightarrow C_2'$ with $C_1' \leq C_2'$
  - $\leq$ is a wqo: finite basis and fixpoint guaranteed

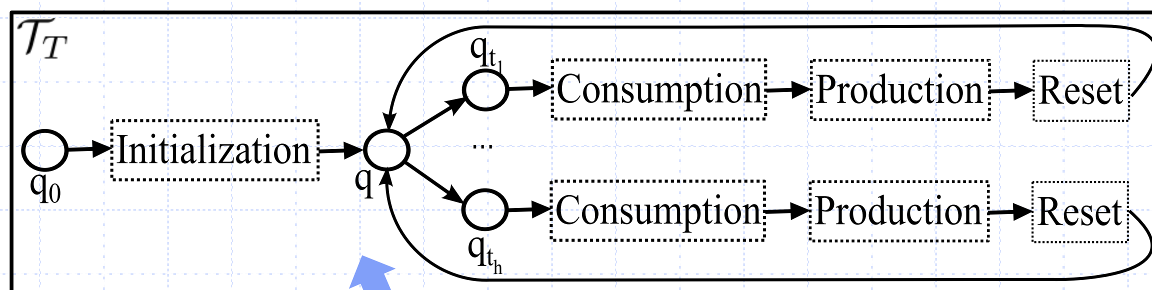


Complexity

- The complexity of the problem is Ackermann-hard (reduction from **coverability** in reset Petri nets)

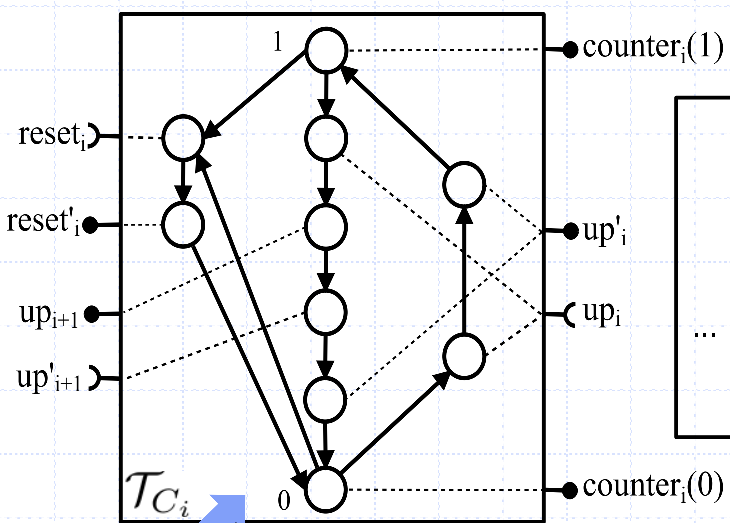


One component for each token in a place *p*

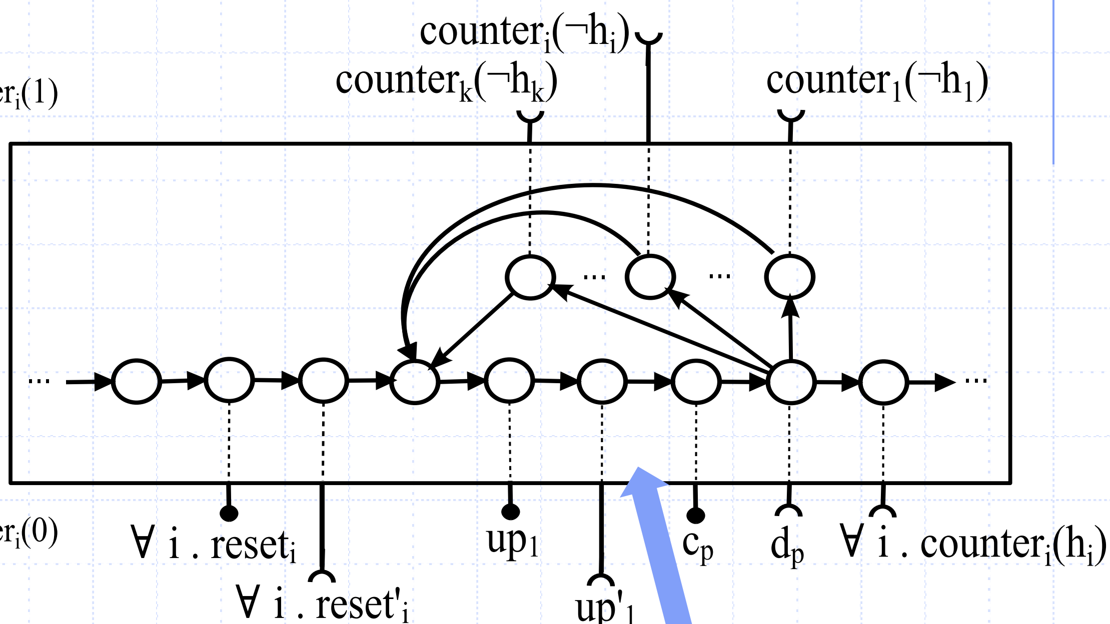


One component for all transitions

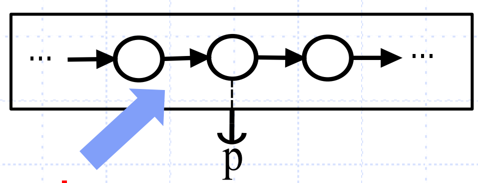
Complexity



One bit in a counter to count the tokens to consume/produce



Counting the tokens to be consumed



Resetting place *p*

Summary of decidability/complexity results

Component model	Deployment is
Full component model	Undecidable
No capacity constraints	Ackermann-hard
No capacity constraints, No conflicts	Quadratic

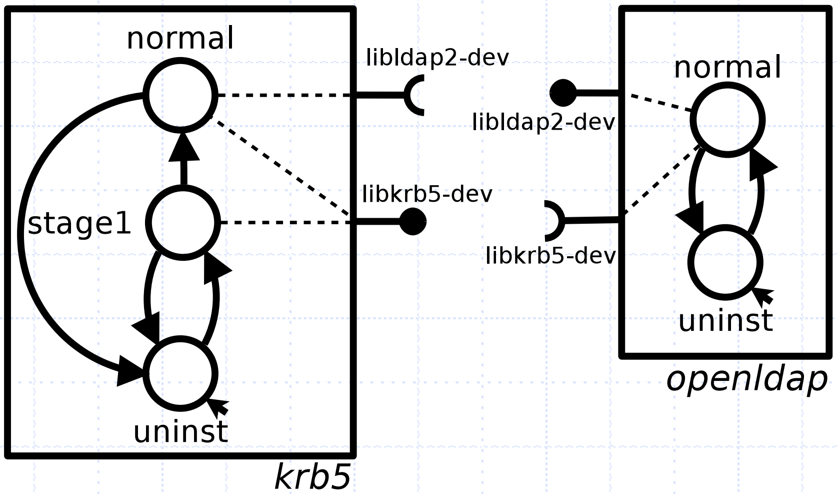
Quadratic algorithm without constraints and conflicts

- **Forward** reachability algorithm
  - all reachable states computed by saturation

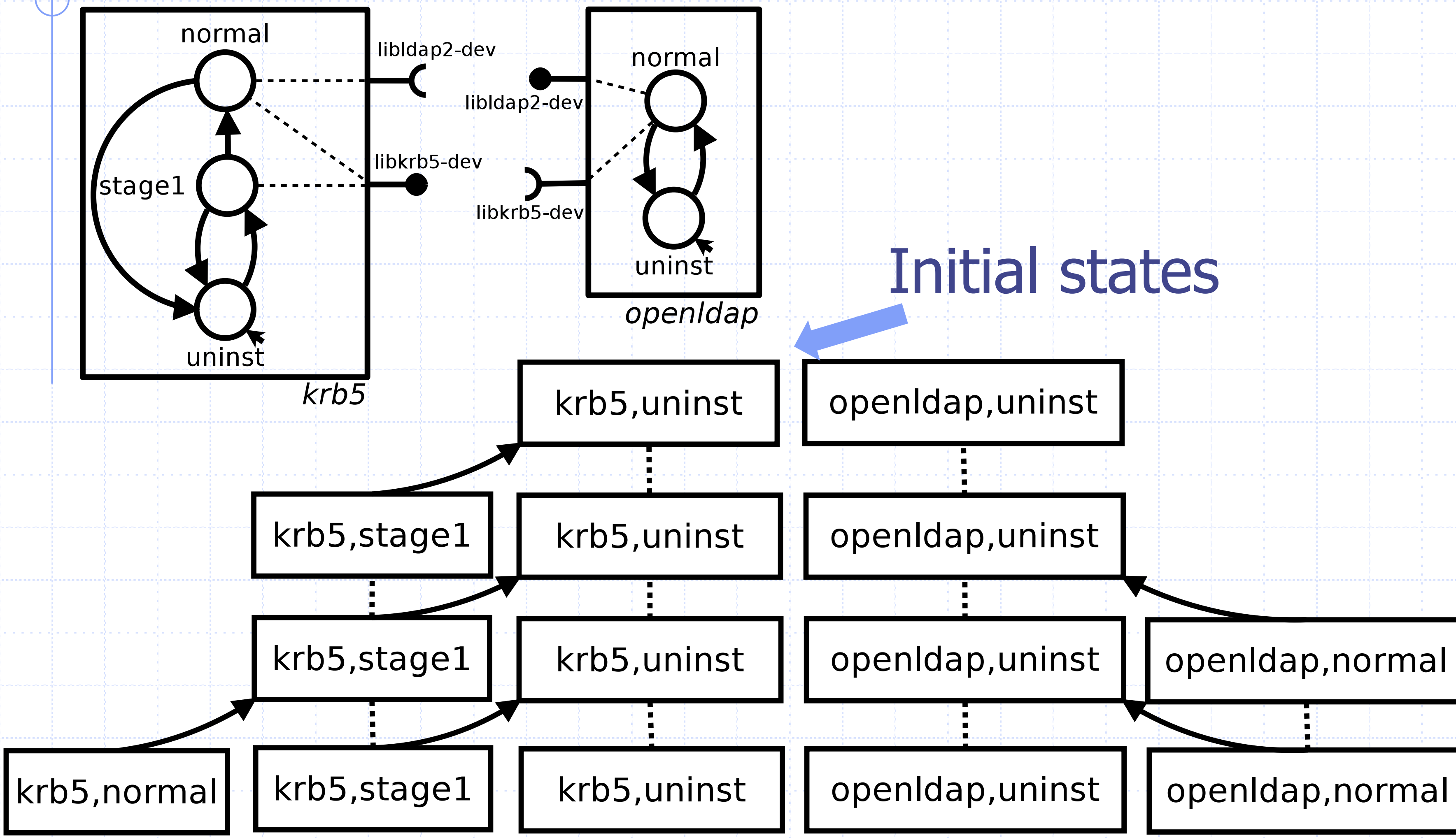
Algorithm 1 Checking achievability in the Aeolus$^-$ model

function ACHIEVABILITY(U, $\mathcal{T}$, q)
  $absConf := \{\langle \mathcal{T}', \mathcal{T}'.\texttt{init}\rangle \mid \mathcal{T}' \in U\}$
  $provPort := \bigcup_{\langle \mathcal{T}', q'\rangle \in absConf} \{dom(\mathcal{T}'.\mathbf{P}(q'))\}$
  repeat
    $new := \{\langle \mathcal{T}', q'\rangle \mid \langle \mathcal{T}', q''\rangle \in absConf, (q'', q') \in \mathcal{T}'.\texttt{trans}\} \setminus absConf$
    $newPort := \bigcup_{\langle \mathcal{T}', q'\rangle \in new} \{dom(\mathcal{T}'.\mathbf{P}(q'))\}$
    while $\exists \langle \mathcal{T}', q'\rangle \in new \;.\; dom(\mathcal{T}'.\mathbf{R}(q')) \not\subseteq provPort \cup newPort$ **do**
      $new := new \setminus \{\langle \mathcal{T}', q'\rangle\}$
      $newPort := \bigcup_{\langle \mathcal{T}', q'\rangle \in new} \{dom(\mathcal{T}'.\mathbf{P}(q'))\}$
    end while
    $absConf := absConf \cup new$
    $provPort := provPort \cup newPort$
  until $new = \emptyset$
  if $\langle \mathcal{T}, q\rangle \in absConf$ **then return true**
  else return false
  end if
end function

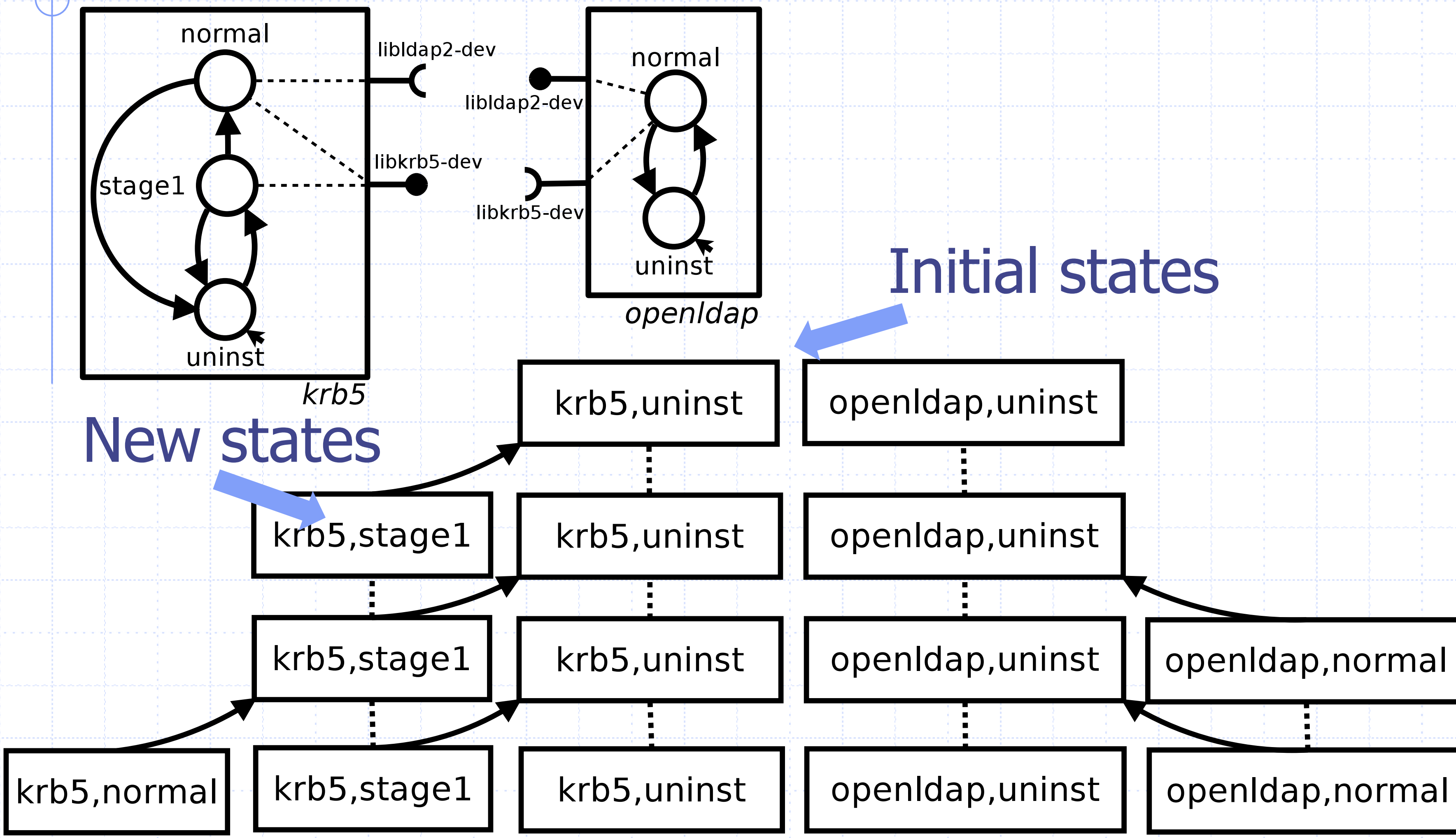
Example: the kerberos case-study



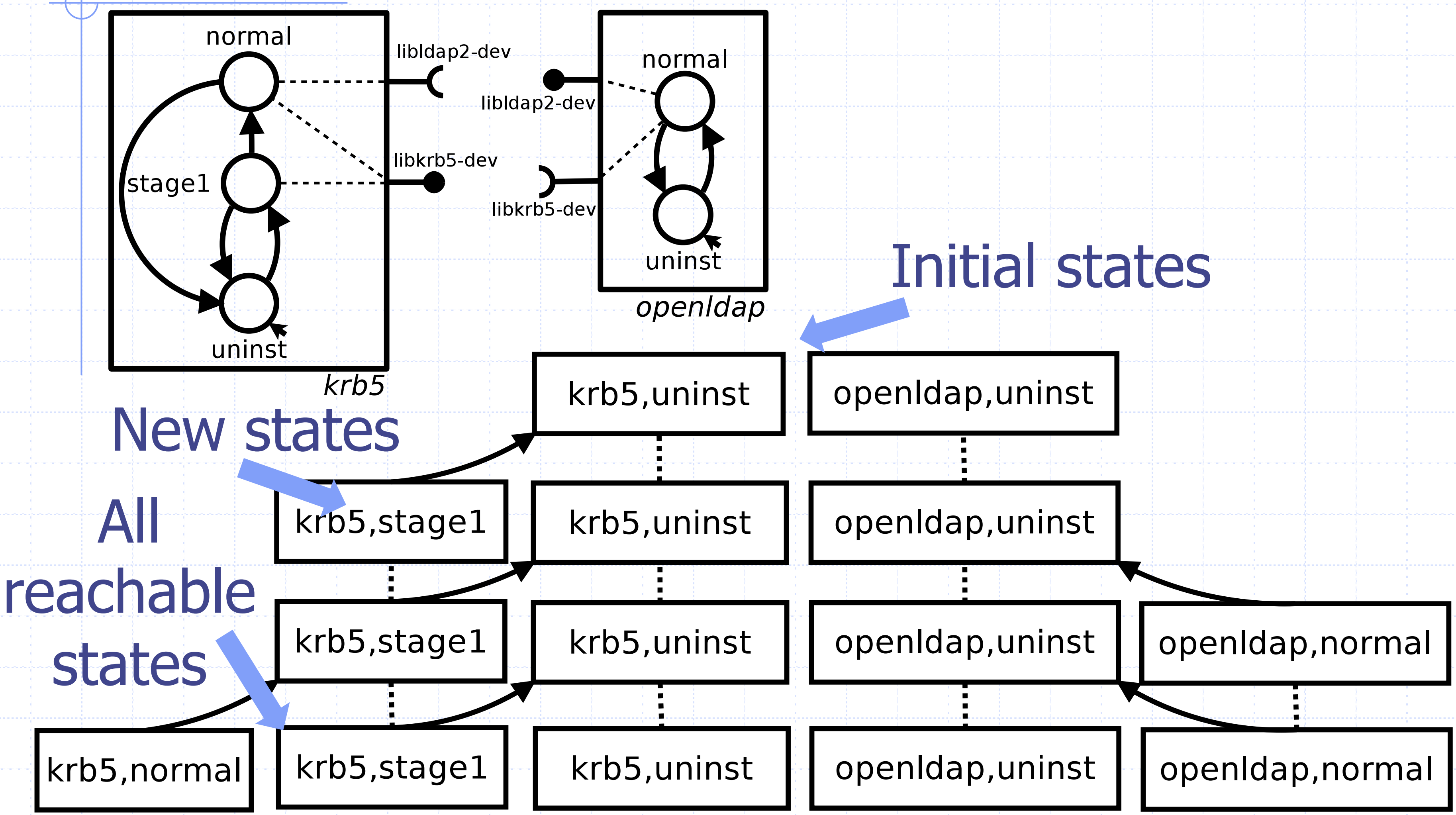
Example:
the kerberos case-study



Example: the kerberos case-study



Example: the kerberos case-study

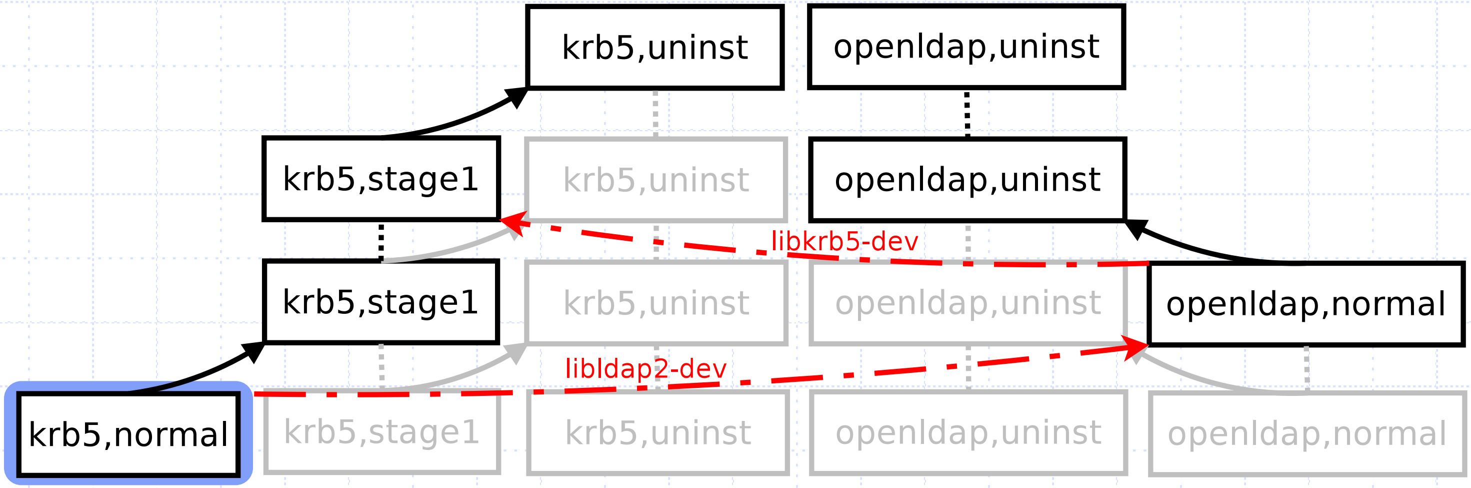


Lesson learned from the foundational study

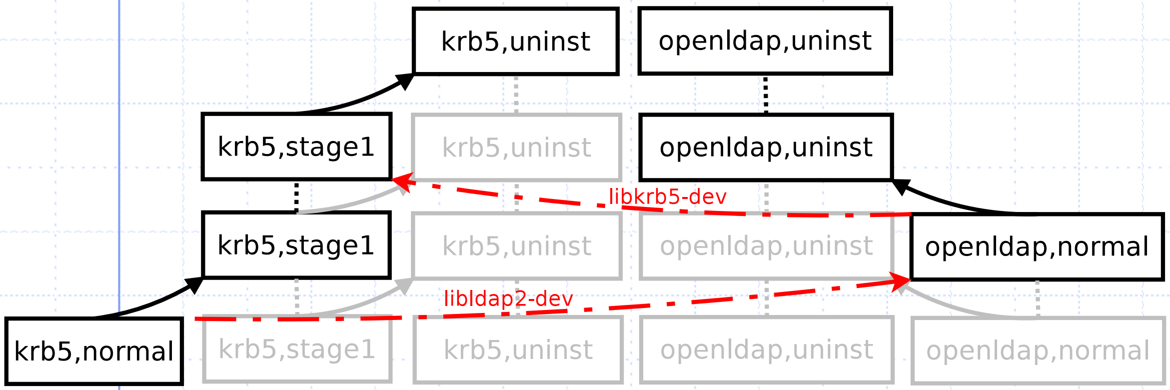
Deployment can be reasonably **fully automatised** if we do not consider capacity constraints and conflicts

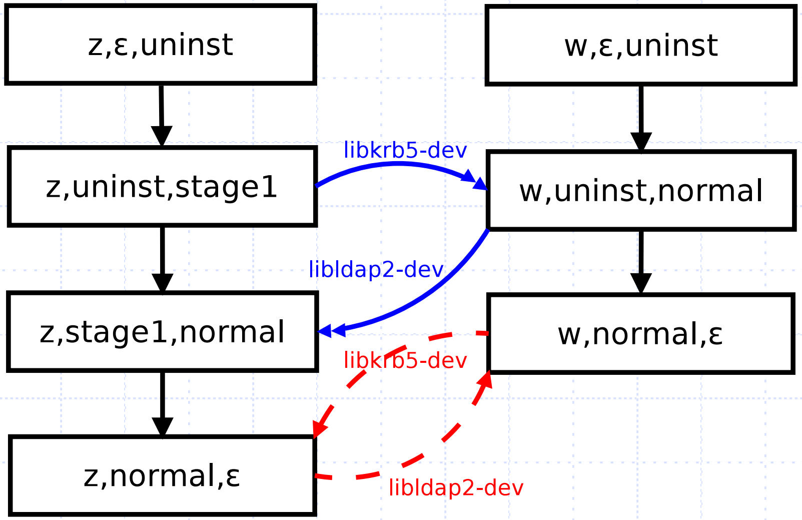
Fully automated deployment (no capacity, no conflicts)

- Use the graph of the reachability algorithm **bottom-up** from the target state
  - select the **bindings** (red arrows)
  - select the **predecessors** (black arrows)



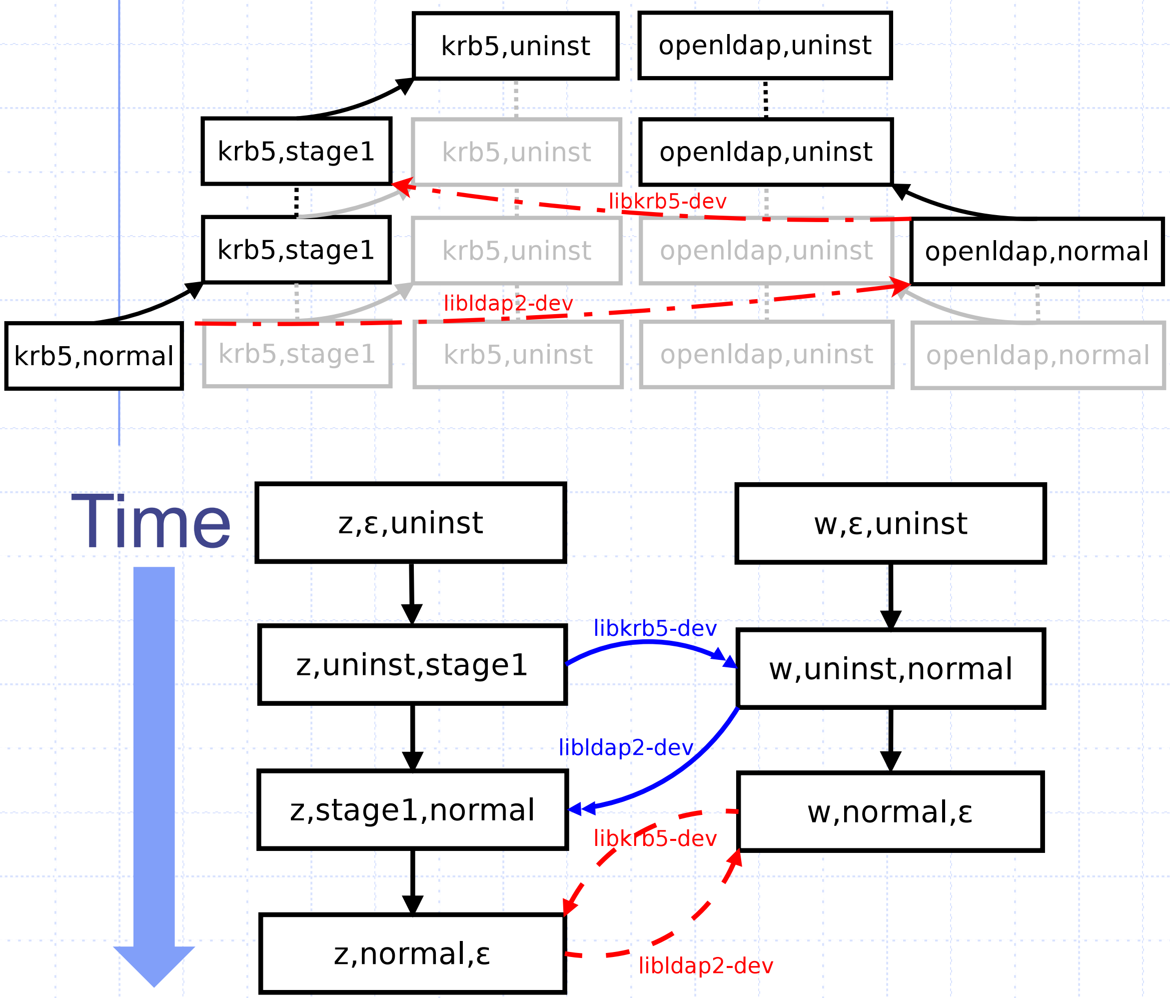
Fully automated deployment (no capacity, no conflicts)





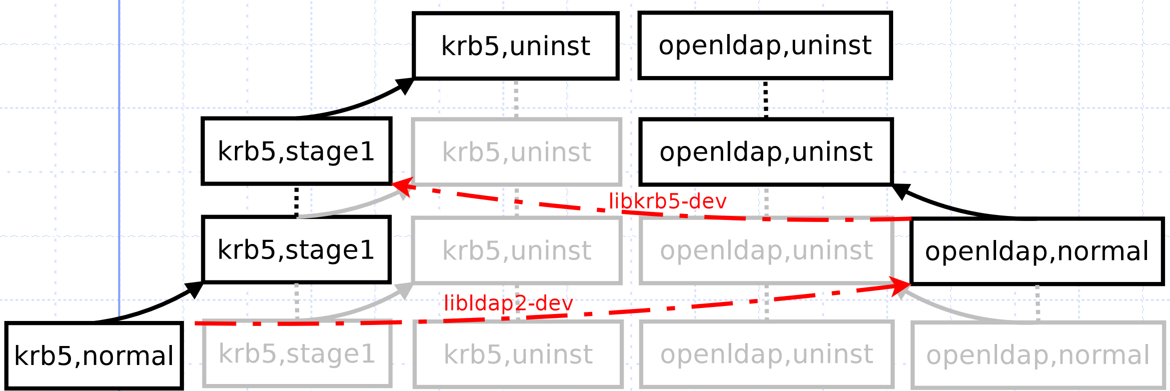
- Generate an **abstract plan** (one component for each maximal path)

Fully automated deployment (no capacity, no conflicts)

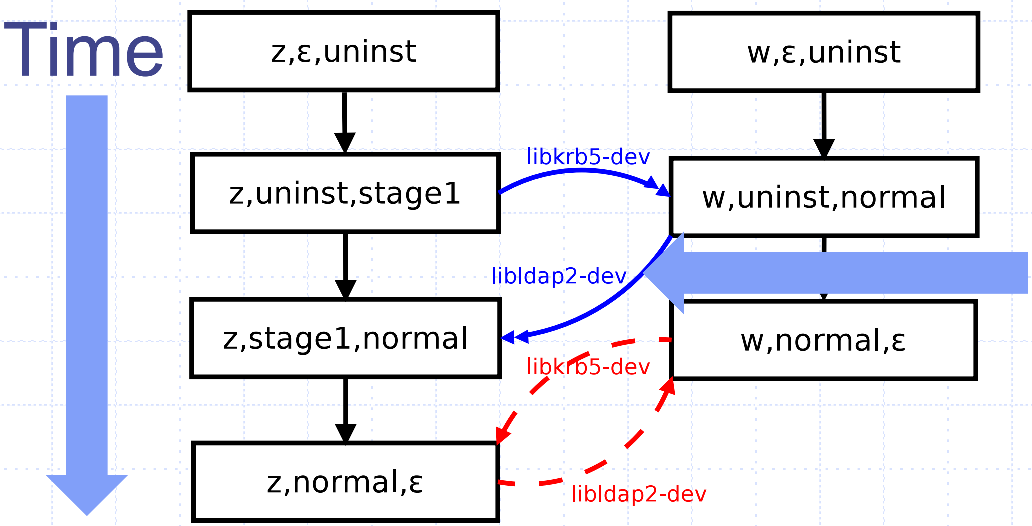


- Generate an **abstract plan** (one component for each maximal path)

Fully automated deployment (no capacity, no conflicts)



- Generate an **abstract plan** (one component for each maximal path)



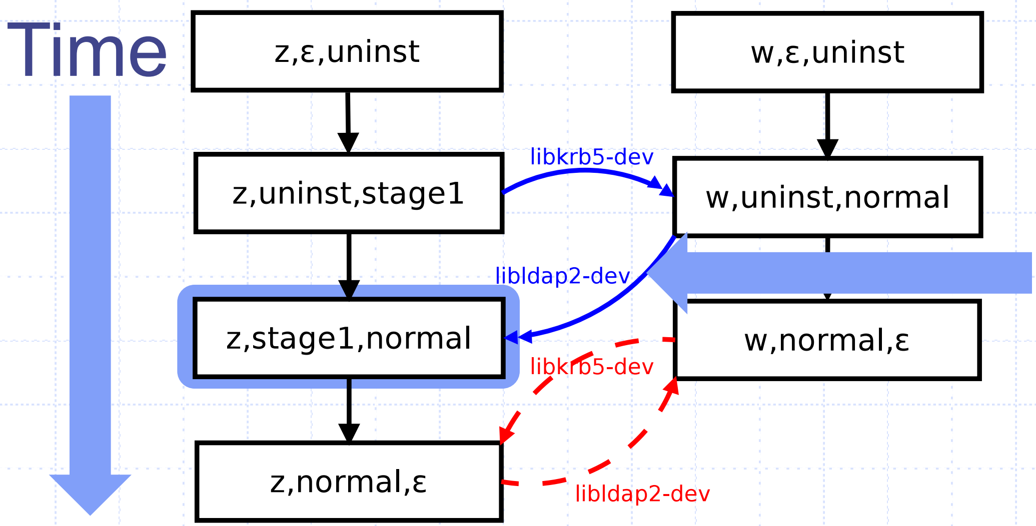
Arrows represent a **precedence** relation:

- **blue**: start requirement
- **red**: end requirement

Fully automated deployment (no capacity, no conflicts)

- Plan as a **topological** visit until target:

```
new(k:krb5),new(o:openldap),
stateChange(k,uninst,stage1),
bind(libkrb5-dev,o,k),stateChange(o,uninst,normal),
bind(libldap2-dev,k,o),
stateChange(k,stage1,normal)
```

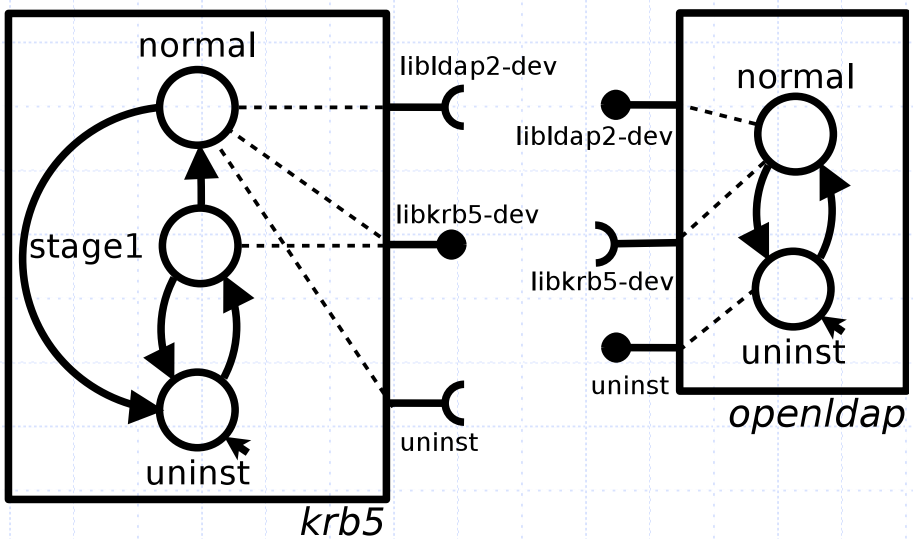


Arrows represent a **precedence** relation:

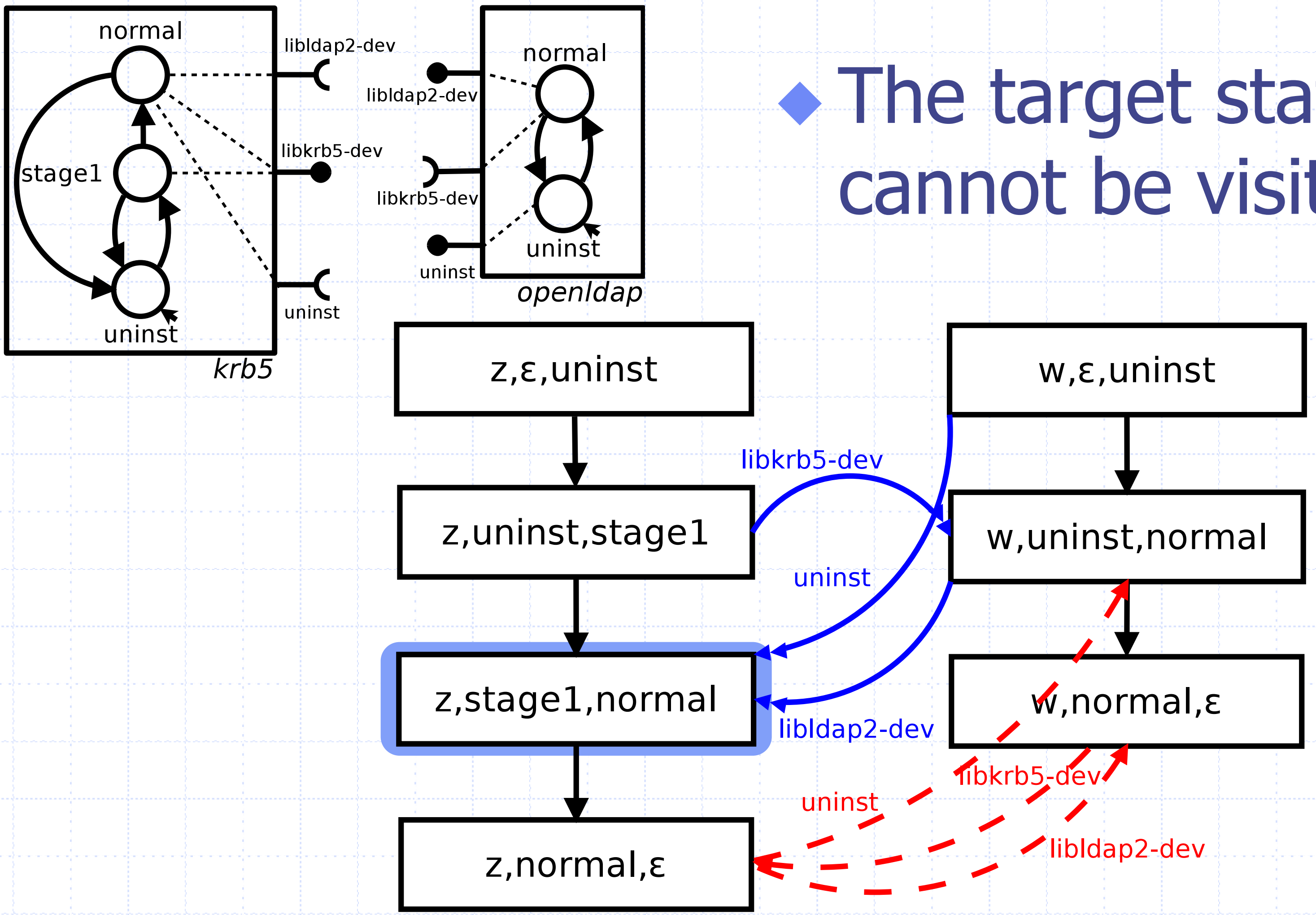
- **blue**: start requirement
- **red**: end requirement

Fully automated deployment (no capacity, no conflicts)

- Problem:
  cycles could forbid the **topological** visit
- Example: krb5 in normal requires an openldap in uninst state

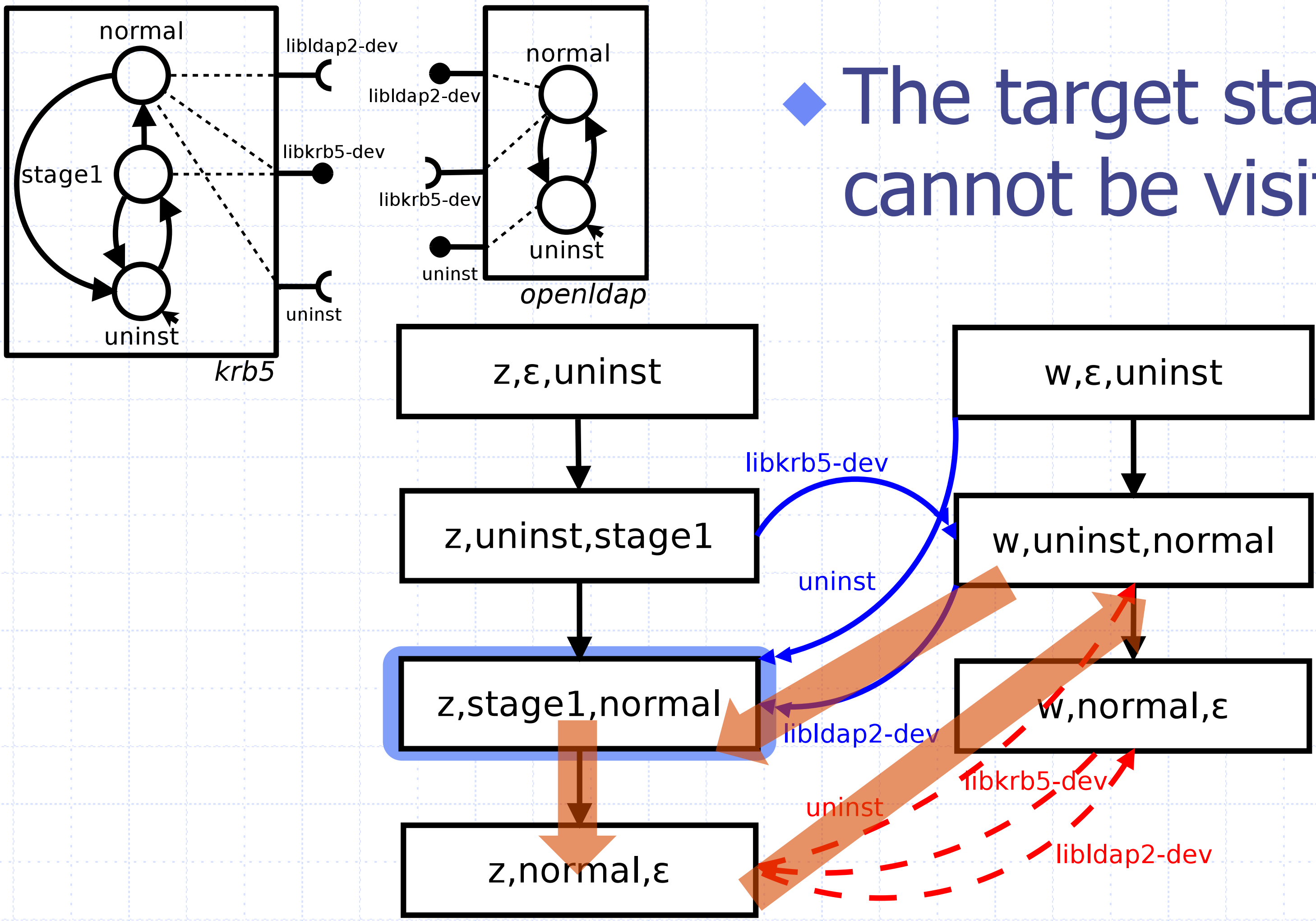


Fully automated deployment (no capacity, no conflicts)



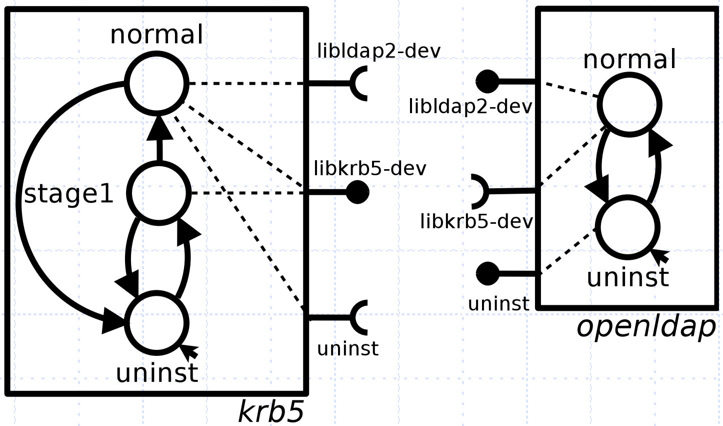
- The target state cannot be visited!

Fully automated deployment (no capacity, no conflicts)

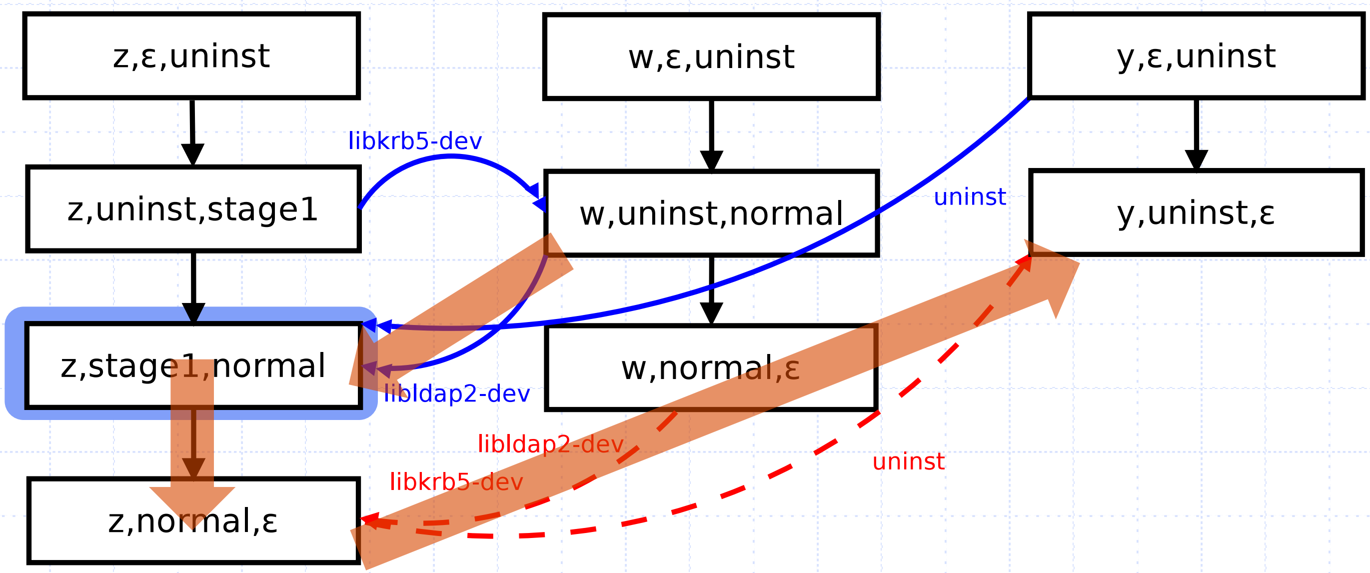


- The target state cannot be visited!

Fully automated deployment (no capacity, no conflicts)



- Solution: component **duplication**



Capacity constraints and conflicts strike back

- We have investigated the problem of synthesising the **final** configuration
  - considering **capacity constraints** and **conflicts** but …
  - … abstracting away from the internal **configuration** automata

R. Di Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, A. Eiche, A. Agahi: *Automated synthesis and deployment of cloud applications.* In Proc. ASE 2014: 211-222

Basic idea

- **Idea** for computing the final configuration:
  - first perform component **selection**
    - **abstract away** from the specific bindings among the selected components …
    - … considering only the overall **requirements** / **capacity constraints** / **conflicts** to be satisfied
  - Subsequently establish the **bindings** among the selected components
    - thus forming the expected **configuration**

Component selection

- Component **selection** is NP-complete but we can use Contraint Solving technology

$$\bigwedge_{p \in \mathcal{I}} \bigwedge_{\langle \mathcal{T}, q \rangle} \mathcal{T}.\mathbf{R}(q)(p) \times \mathtt{comp}(\langle \mathcal{T}, q \rangle) \leq \sum_{\langle \mathcal{T}', q' \rangle} \mathtt{bind}(p, \langle \mathcal{T}', q' \rangle, \langle \mathcal{T}, q \rangle)$$

$$\bigwedge_{p \in \mathcal{I}} \bigwedge_{\langle \mathcal{T}, q \rangle \,.\, \mathcal{T}.\mathbf{P}(q)(p) < \infty} \mathcal{T}.\mathbf{P}(q)(p) \times \mathtt{comp}(\langle \mathcal{T}, q \rangle) \geq \sum_{\langle \mathcal{T}', q' \rangle} \mathtt{bind}(p, \langle \mathcal{T}, q \rangle, \langle \mathcal{T}', q' \rangle)$$

$$\bigwedge_{p \in \mathcal{I}} \bigwedge_{\langle \mathcal{T}, q \rangle \,.\, \mathcal{T}.\mathbf{P}(q)(p) = \infty} \mathtt{comp}(\langle \mathcal{T}, q \rangle) = 0 \;\Rightarrow\; \sum_{\langle \mathcal{T}', q' \rangle} \mathtt{bind}(p, \langle \mathcal{T}, q \rangle, \langle \mathcal{T}', q' \rangle) = 0$$

$$\bigwedge_{p \in \mathcal{I}} \bigwedge_{\substack{\langle \mathcal{T}, q \rangle \,.\, \mathcal{T}.\mathbf{R}(q)(p) = 0 \,\wedge \\ \mathcal{T}.\mathbf{P}(q)(p) > 0)}} \mathtt{comp}(\langle \mathcal{T}, q \rangle) \leq 1$$

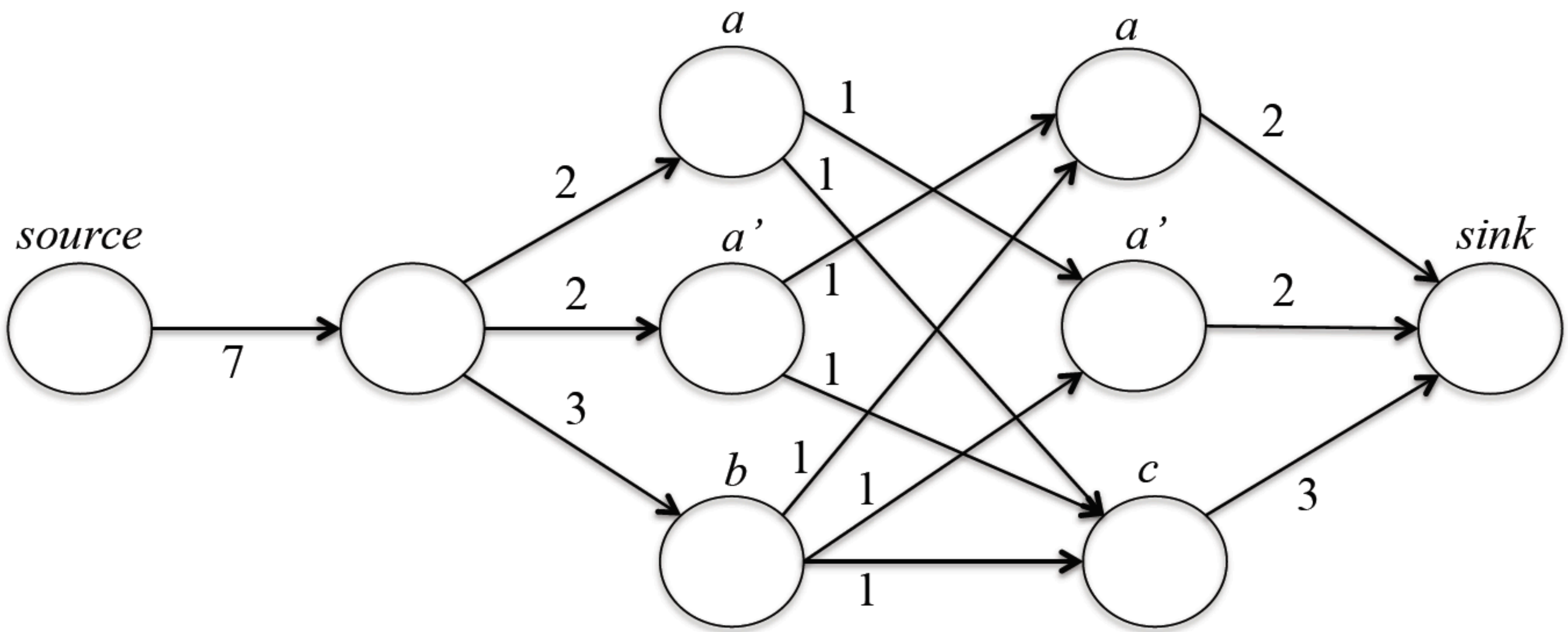
$$\bigwedge_{p \in \mathcal{I}} \bigwedge_{\substack{\langle \mathcal{T}, q \rangle . \\ \mathcal{T}.\mathbf{R}(q)(p) = 0}} \bigwedge_{\substack{\langle \mathcal{T}', q' \rangle \neq \langle \mathcal{T}, q \rangle \,. \\ \mathcal{T}'.\mathbf{P}(q')(p) > 0}} \mathtt{comp}(\langle \mathcal{T}, q \rangle) > 0 \;\Rightarrow\; \mathtt{comp}(\langle \mathcal{T}', q' \rangle) = 0$$

$$\bigwedge_{p \in \mathcal{I}} \bigwedge_{\langle \mathcal{T}, q \rangle} \bigwedge_{\langle \mathcal{T}', q' \rangle \neq \langle \mathcal{T}, q \rangle} \mathtt{bind}(p, \langle \mathcal{T}, q \rangle, \langle \mathcal{T}', q' \rangle) \leq \mathtt{comp}(\langle \mathcal{T}, q \rangle) \times \mathtt{comp}(\langle \mathcal{T}', q' \rangle)$$

$$\bigwedge_{p \in \mathcal{I}} \bigwedge_{\langle \mathcal{T}, q \rangle} \mathtt{bind}(p, \langle \mathcal{T}, q \rangle, \langle \mathcal{T}, q \rangle) \leq \mathtt{comp}(\langle \mathcal{T}, q \rangle) \times (\mathtt{comp}(\langle \mathcal{T}, q \rangle) - 1)$$

Bindings establishment

- **Bindings** decided as solution of a max-flow problem



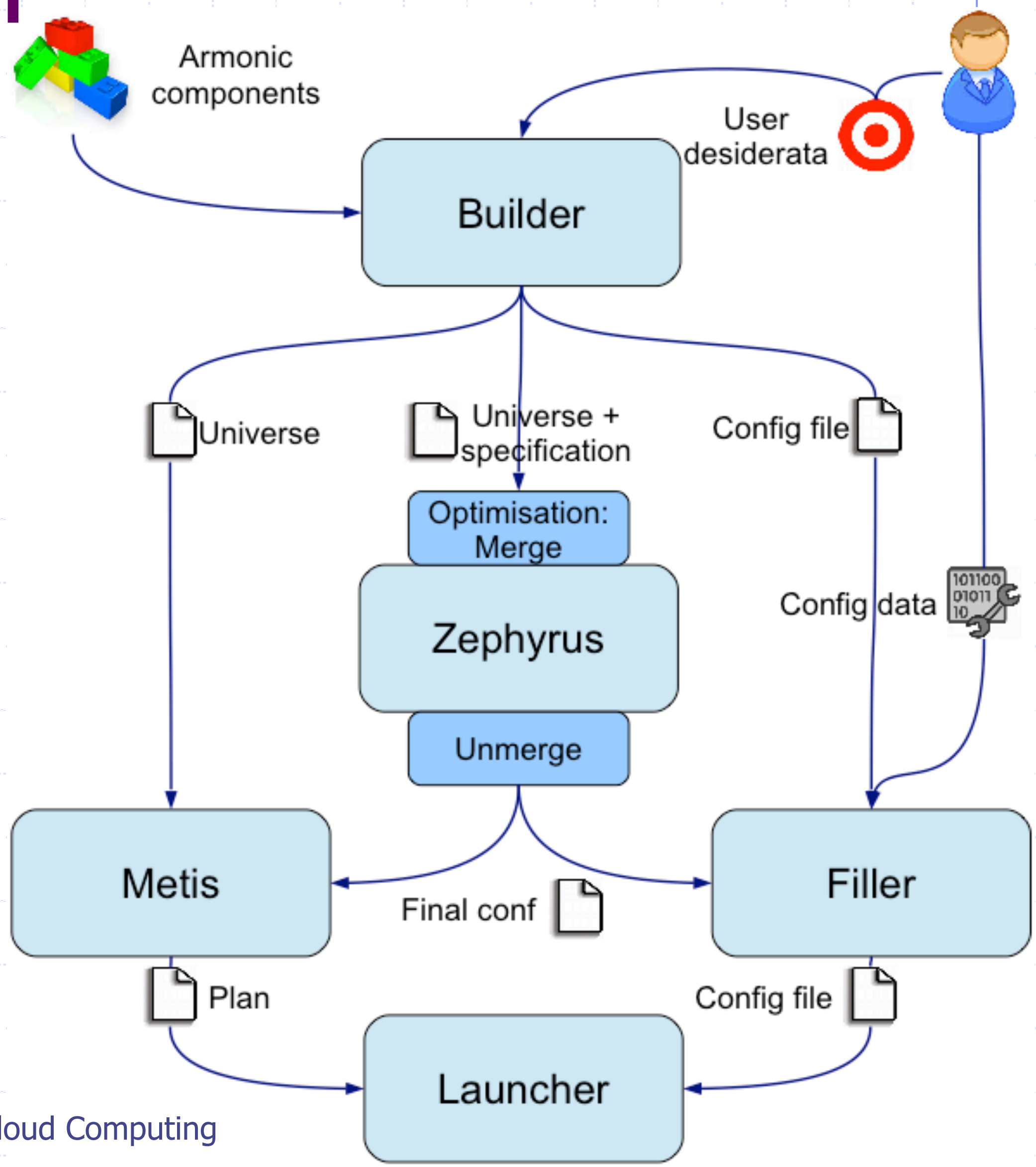
Putting everything together: Aeolus Blender

- We have relised a **tool-chain** that:
  - Starting from a **library** of components and the specification of the **desired** configuration
  - First computes the **final configuration** (considering capacity and conflicts) …
  - … then computes a deployment **plan** to reach it (capacity and conflicts not guaranteed)

R. Di Cosmo, A. Eiche, J. Mauro, S. Zacchiroli, G. Zavattaro, J. Zwolakowski: *Automatic Deployment of Services in the Cloud with Aeolus Blender.* In Proc ICSOC 2015: 397-411

Putting everything together: Aeolus Blender

- **Armonic**: library of components
- **Zephyrus**: synthesis of the final architecture
- **Metis**: plan the configuration actions



Armonic: component description

- Definition of a language for describing component's **repositories**

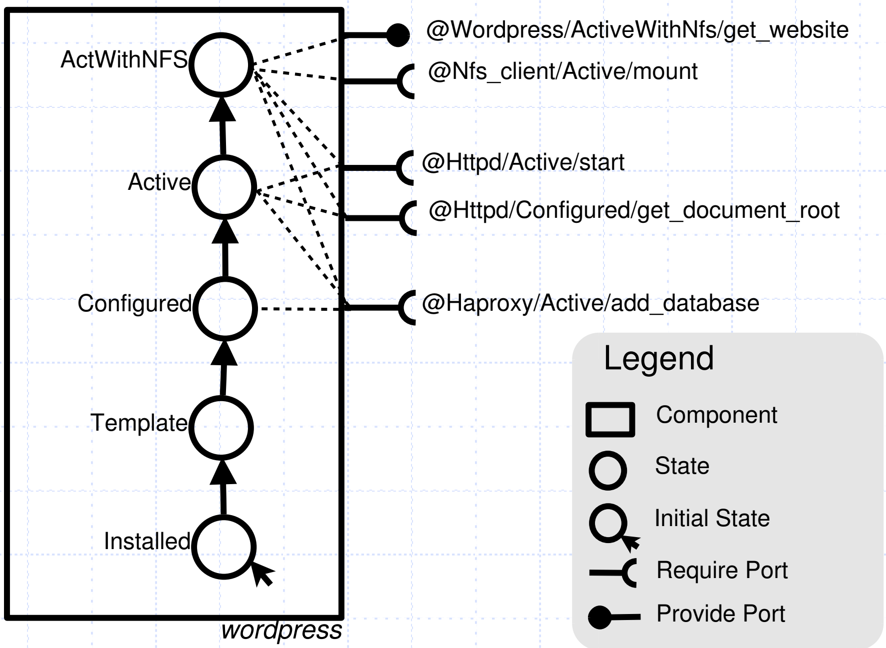
```
{
  "states": [
   {
    "provide": {},
    "require": {},
    "initial": true,
    "name": "Installed",
    "successors": [
     "Template"
    ]
   },
   {
    "provide": {},
    "require": {},
    "successors": [
     "Configured"
    ],
    "name": "Template"
   },
```

```
{
 "provide": {},
 "require": {
  "@Haproxy/Active/add_database": 1
 },
 "successors": [
  "Active"
 ],
 "name": "Configured"
},
{
 "provide": {},
 "require": {
  "@Haproxy/Active/add_database": 1,
  "@Httpd/Active/start": 1,
  "@Httpd/Configured/get_document_root": 1
 },
 "successors": [
  "ActiveWithNfs"
 ],
 "name": "Active"
},
```

```
  {
   "provide": {
    "@Wordpress/ActiveWithNfs/get_website": 1000
   },
   "require": {
    "@Haproxy/Active/add_database": 1,
    "@Httpd/Active/start": 1,
    "@Httpd/Configured/get_document_root": 1,
    "@Nfs_client/Active/mount": 1
   },
   "name": "ActiveWithNfs"
  }
 ],
 "name": "Wordpress"
}
```

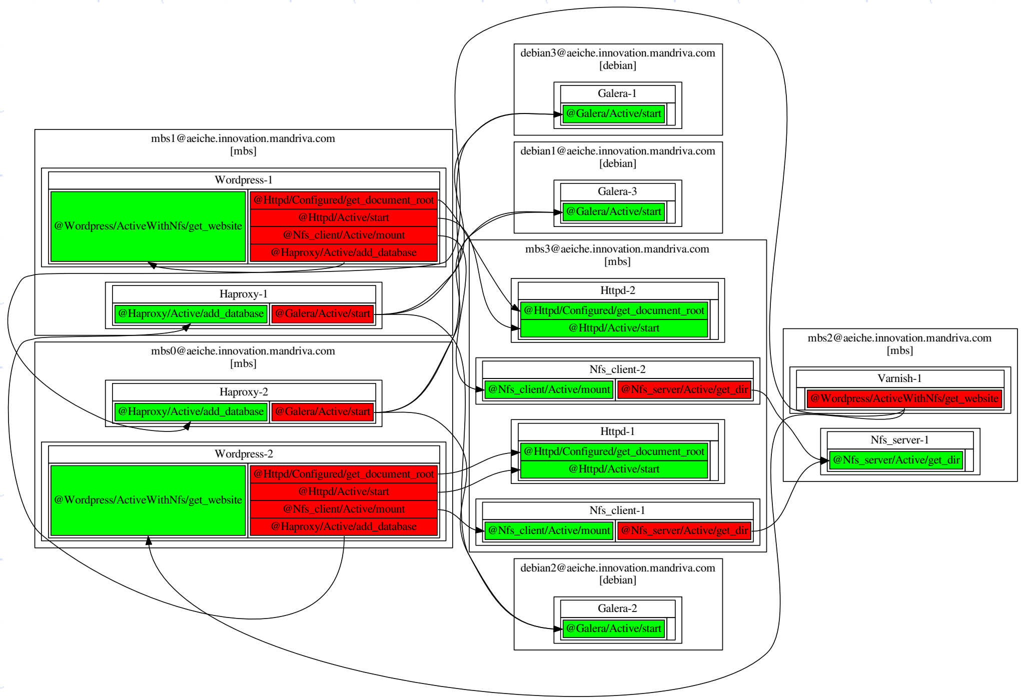
Armonic: component description

- Definition of a language for describing component's **repositories**



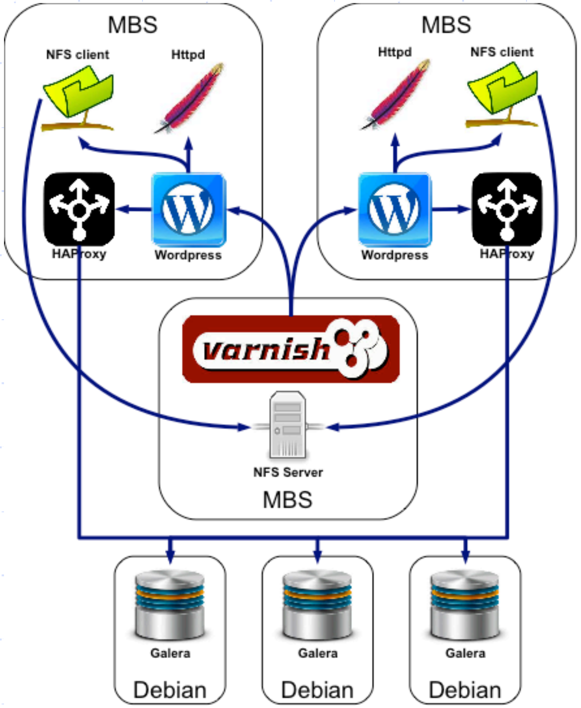
Zephyrus: final configuration computation

- Realization of a tool for component's selection and architecture **synthesis**



Zephyrus: final configuration computation

- Realization of a tool for component's selection and architecture **synthesis**



Metis: deployment plan (conflicts/capacity not guaranteed)

- Realization of a tool for **planning** the configuration actions to be executed