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

Service Template

- **Topology Template**
  - Relationship Template
  - Node Template

- **Node Types**
  - Node Type

- **Relationship Types**
  - Relat. Type

- **Plans**

**Legenda**
- Property
- Interface
- Capability
- Requirement
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 Orchestratio

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

Example: a TOSCA topology

Translator (WebService)

Convertor (WebService)

Apache (Server)

Debian (OperatingSystem)

VMWare (VirtualMachine)
Example: corresponding TOSCA deployment plans
Example: corresponding TOSCA deployment plans

ERROR: Apache must be run before being configurated
Example: corresponding TOSCA deployment plans

(a)

(b)

ERROR: Debian is not started
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**
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**

---

Models and Languages for Service-Oriented and Cloud Computing

Bertinoro 7-11/3/2016
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

1. **Unavailable**  
   - R: {}  
   - C: {}  

2. **Stopped**  
   - R: {}  
   - C: {}  

3. **Working**  
   - R: {ServerCont}  
   - C: {WebAppR}

{ServerCont} Setup → {ServerCont} Run

{ServerCont} Uninstall → {} Stop

{} Configure
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/](http://ranma42.github.io/MProt/)
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
  - ...

Bertinoro 7-11/3/2016

Models and Languages for Service-Oriented and Cloud Computing
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
  - ...

Models and Languages for Service-Oriented and Cloud Computing
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)

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

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

Legend

- [ ] Component
- [ ] State
- [ ] Initial State
- [ ] Require Port
- [ ] Provide Port
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 $\Gamma$ of *component types* of the Aeolus model, ranged over by $T_1, T_2, \ldots$ 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*;
- $P = \langle P, R \rangle$, with $P, R \subseteq I$, 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 $(P \rightarrow \mathbb{N}_\infty^+) \times (R \rightarrow \mathbb{N})$. 
Formalizing the “deployment” problem

**Definition 1** (Component type). The set $\Gamma$ 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*;
- $P = \langle P, R \rangle$, with $P, R \subseteq \mathcal{I}$, 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 $(P \rightarrow \mathbb{N}^+_\infty) \times (R \rightarrow \mathbb{N})$.

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

- $U \subseteq \Gamma$ is the finite *universe* of all available component types;
- $Z \subseteq Z$ 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:

- $\text{stateChange}(z, q_1, q_2)$ where $z \in \mathcal{Z}$: change the state of the component $z$ from $q_1$ to $q_2$;
- $\text{bind}(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$: add a binding between $z_1$ and $z_2$ on port $r$;
- $\text{unbind}(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$: remove the specified binding;
- $\text{new}(z : \mathcal{T})$ where $z \in \mathcal{Z}$ and $\mathcal{T}$ is a component type: add a new component $z$ of type $\mathcal{T}$;
- $\text{del}(z)$ where $z \in \mathcal{Z}$: remove the component $z$ from the configuration.
Formalizing the “deployment” problem

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

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

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

\[
C \xrightarrow{\text{bind}(r,z_1,z_2)} \langle U, Z, S, B \cup \langle r, z_1, z_2 \rangle \rangle \\
\text{if } \langle r, z_1, z_2 \rangle \not\in B \\
\text{and } r \in C[z_1].\text{req} \cap C[z_2].\text{prov}
\]

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

\[
C \xrightarrow{\text{del}(z)} \langle U, Z \setminus \{z\}, S', B' \rangle \\
\text{if } S'(z') = \begin{cases} \\
\bot & \text{if } z' = z \\
C[z'] & \text{otherwise}
\end{cases} \\
\text{and } B' = \{ \langle r, z_1, z_2 \rangle \in B \mid z \not\in \{z_1, z_2\} \}
\]
“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

![Diagram](image-url)
Deployment problem: example

Deployment plan:

\[
\begin{align*}
&\text{new}(k: \text{kerb5}), \text{new}(o: \text{openldap}), \\
&\text{stateChange}(k, \text{uninst}, \text{stage1}), \\
&\text{bind}(\text{libkerb5-dev}, o, k), \text{stateChange}(o, \text{uninst}, \text{normal}), \\
&\text{bind}(\text{libldap2-dev}, k, o), \\
&\text{stateChange}(k, \text{stage1}, \text{normal})
\end{align*}
\]
Deployment problem: example

Deployment plan:

\[
\begin{align*}
\text{new}(k: \text{krb5}), & \text{new}(o: \text{openldap}), \\
\text{stateChange}(k, \text{uninst}, \text{stage1}), & \\
\text{bind}(\text{libkrb5-dev}, o, k), & \text{stateChange}(o, \text{uninst}, \text{normal}), \\
\text{bind}(\text{libldap2-dev}, k, o), & \\
\text{stateChange}(k, \text{stage1}, \text{normal})
\end{align*}
\]
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 plan diagram](image)
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:

  \[
  \text{new}(k:\text{krb5}),\text{new}(o:\text{openldap}), \\
  \text{stateChange}(k,\text{uninst},\text{stage1}), \\
  \text{bind}(\text{libkrb5-dev},o,k),\text{stateChange}(o,\text{uninst},\text{normal}), \\
  \text{bind}(\text{libldap2-dev},k,o), \\
  \text{stateChange}(k,\text{stage1},\text{normal})
  \]

Models and Languages for Service-Oriented and Cloud Computing

Bertinoro 7-11/3/2016
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

<table>
<thead>
<tr>
<th>Component model</th>
<th>Deployment is</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full component model</td>
<td>Undecidable</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Component model</th>
<th>Deployment is</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full component model</td>
<td>Undecidable</td>
</tr>
<tr>
<td>No capacity constraints</td>
<td>Ackermann-hard</td>
</tr>
</tbody>
</table>

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 w.q.o.: finite basis and fixpoint guaranteed.

![Initial conf.](image1) ![Target conf.](image2)
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

Initial conf. .... Target conf.
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
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 \( p \)

∀ \( i \). \( \text{reset}_i \)

∀ \( i \). \( \text{reset'}_i \)

∀ \( i \). \( \text{counter}_i(\neg h_i) \)

∀ \( i \). \( \text{counter}_k(\neg h_k) \)

∀ \( i \). \( \text{counter}_1(\neg h_1) \)
<table>
<thead>
<tr>
<th>Component model</th>
<th>Deployment is</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full component model</td>
<td>Undecidable</td>
</tr>
<tr>
<td>No capacity constraints</td>
<td>Ackermann-hard</td>
</tr>
<tr>
<td>No capacity constraints, No conflicts</td>
<td>Quadratic</td>
</tr>
</tbody>
</table>
Quadratic algorithm without constraints and conflicts

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

---

**Algorithm 1** Checking achievability in the Aeolus\textsuperscript{−} model

```plaintext
function \textsc{Achievability}(U, T, q)

absConf := \{\langle T', \text{init} \rangle | T' \in U\}
provPort := \bigcup_{\langle T', q' \rangle \in absConf} \{\text{dom}(T'.P(q'))\}

repeat
    new := \{\langle T', q' \rangle | \langle T', q'' \rangle \in absConf, (q'', q') \in T'.\text{trans}\} \setminus absConf
    newPort := \bigcup_{\langle T', q' \rangle \in new} \{\text{dom}(T'.P(q'))\}
    while \exists \langle T', q' \rangle \in new . \text{dom}(T'.R(q')) \not\subseteq provPort \cup newPort do
        new := new \setminus \{\langle T', q' \rangle\}
        newPort := \bigcup_{\langle T', q' \rangle \in new} \{\text{dom}(T'.P(q'))\}
    end while
    absConf := absConf \cup new
    provPort := provPort \cup newPort
end while

until new = \emptyset
if \langle T, q \rangle \in absConf then return true
else return false
end if
```

---

Models and Languages for Service-Oriented and Cloud Computing
Example: the kerberos case-study

![Diagram of the kerberos case-study](image.png)
Example: the kerberos case-study

Initial states

Models and Languages for Service-Oriented and Cloud Computing
Example: the kerberos case-study

Models and Languages for Service-Oriented and Cloud Computing

Bertinoro 7-11/3/2016
Example: the kerberos case-study

Models and Languages for Service-Oriented and Cloud Computing
Bertinoro 7-11/3/2016
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)

![Diagram of the reachability graph for the kerberos running example](image-url)
Fully automated deployment (no capacity, no conflicts)

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

```
Time

z,ε,uninst
  ↓
z,uninst,stage1
    ↓
z,stage1,normal
      ↓
z,normal,ε

w,ε,uninst
  ↓
w,uninst,normal
    ↓
w,normal,ε
```

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

In general, we have to consider an abstract plan where we add a new instance duplication to the abstract plan in Fig. 8. This is obtained by moving the pairs of arcs labeled with normal, w, and outgoing arcs of the instance y representing its requirements on the interface that the new instance must provide to the other components. See, for example, the kerberos example in which the component type krb5 requires not only one instance of the configuration of other objects in the configuration. For this reason, the abstract plan of the instance y will have impediments due to incoming arcs and it can be visited. Consider now the case in which there are incoming arcs and there is an incoming arc from their initial position in Fig. 8.

In the context of knowledge representation and reasoning, a very important application of artificial intelligence is that of planning. Since a declarative language for planning has been defined for establishing a common syntax for different tools and limits the number of objects that could be solved as described in the previous case.

Our tool solves a planning problem and therefore we tried to validate our ad-hoc planner against standard planners. To do so we have defined an encoding of our specific planning problem into PDDL. Each component instance is translated automatically generated following the pattern of component type bindings. The encoding abstracts from the bind and unbind actions required interfaces. The encoding can be solved by means of polynomial time in a post processing phase. The planning problem of a single component instance can be transformed when the visit is blocked by a cycle. When a node cannot be visited, we proceed as follows: There is at least an adaptive visit until the target node. This is obtained by moving the pairs of arcs labeled with normal, w, and outgoing arcs of the instance y representing its requirements on the interface that the new instance must provide to the other components.

It is interesting to note that this adaptive visit will eventually terminate because new instances do not introduce new cycles. This limitation was necessary because all the solvers assume a finite number of objects – without this limitation the planning problem is undecidable. This language is known as PDDL, which is a necessary step to provide an interface during a specific phase of the plan, but also to validate our ad-hoc planner against standard planners. To do so we have defined an encoding of our specific planning problem into PDDL.
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

---

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 is NP-complete but we can use Constraint Solving technology.
Bindings establishment

- **Bindings** decided as solution of a max-flow problem
We have realised 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)

---

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

```json
{
  "states": [
    {
      "provide": {},
      "require": {},
      "initial": true,
      "name": "Installed",
      "successors": [
        "Template"
      ]
    },
    {
      "provide": {},
      "require": {},
      "successors": [
        "Configured"
      ],
      "name": "Template"
    },
    {
      "provide": {},
      "require": {
        "@Haproxy/Active/add_database": 1,
        "@Httpd/Active/start": 1,
        "@Httpd/Configured/get_document_root": 1
      },
      "successors": [
        "Active"
      ],
      "name": "Configured"
    },
    {
      "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
      },
      "successors": [
        "ActiveWithNfs"
      ],
      "name": "Configured"
    }
  ],
  "name": "Wordpress"
}
```
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