Semantic Analysis of Dynamic Connector Based Architecture Styles

Guoxin Su  Mingsheng Ying  Chengqi Zhang

Faculty of Engineering and Information Technology
University of Technology, Sydney

WICSA/ECSA 2012
Outline

Background: where our problem locates

Problem: a motivating example and behavioural properties

Model: formalisation of architectural concepts and properties

Analysis: algorithms for checking desired properties
Background

- Dynamism of connector-based architectural styles: insertion and removal of components
- Type- vs instance-level descriptions and component instantiation: parameterisation or semantic conformance
- Behavioural modeling
Background

- Dynamism of connector-based architectural styles: insertion and removal of components
- Type- vs instance-level descriptions and component instantiation: parameterisation or semantic conformance
- Behavioural modeling

Diagram:

```
C1  C2
  |   |
  C3---C4
  |   |
  C5  C6

C1  C2  C7
  |   |
  C3---C4
  |   |
  C6  C8
```
Background

- Dynamism of connector-based architectural styles: insertion and removal of components
- Type- vs instance-level descriptions and component instantiation: parameterisation or semantic conformance
- Behavioural modeling

Syntax of textual specification:

```
if state = x then
  if pre-conditions then
    input/output and effects [and state := y]
```
A Client-Server System

- The system is connector-based
- A structural view and a scenario:

Components can join in and disconnect to the system dynamically
A Client-Server System

Type-level specification

Component type $CLIENT(c : clt, s : sev)$:

```
if state = 0 then
  $\langle request, c, s \rangle!$ and state := 1
if state = 1 then
  if true then
    $\langle result, c, s \rangle?$ and state := 2
  if true then
    $\langle error, c, s \rangle?$ and state := 2
```
A Client-Server System

Component type \textit{SERVER}(s : sev):

\begin{align*}
\text{if } state = 0 \text{ then} & \quad \langle \text{register}, s \rangle! \text{ and } state := 1 \\
\text{if } state = 1 \text{ then} & \quad \\
\quad \text{if true then} & \quad \langle \text{involve}, x : \textit{clt} \rangle? \text{ and } \\
\qquad \quad \text{enqueue}(x, \textit{Que}) \\
\quad \text{if empty}(\textit{Que}) = \textit{‘n’} \text{ then} & \quad \langle \text{return}, y \rangle! \text{ and } \text{dequeue}(\textit{Que}) \\
\quad \text{if empty}(\textit{Que}) = \textit{‘y’} \text{ then} & \quad \langle \text{unregister}, s \rangle! \text{ and } state := 2
\end{align*}
A Client-Server System

Connector \textit{CSCON}:

\begin{align*}
\text{if } state_1 &= 0 \text{ then } \\
&\langle \text{request}, x : \text{clt}, y : \text{sev} \rangle? \\
\text{if } state_1 &= 1 \text{ then } \\
&\text{if } y \in \text{RegSev} \text{ then } \\
&\langle \text{involve}, x, y \rangle! \text{ and } state_1 := 0 \\
&\text{else } \langle \text{error}, x, y \rangle! \text{ and } state_1 := 0 \\
\text{if } state_2 &= 0 \text{ then } \\
&\langle \text{return}, z : \text{clt}, w : \text{sev} \rangle? \text{ and } state_2 := 1 \\
\text{if } state_2 &= 1 \text{ then } \\
&\langle \text{result}, z, w \rangle! \text{ and } state_2 := 0 \\
\text{if } state_3 &= 0 \text{ then } \\
&\text{if true then } \\
&\langle \text{register}, v : \text{sev} \rangle? \text{ and } \\
&\text{RegSev} := \text{RegSev} \cup \{v\} \\
&\text{if true then } \\
&\langle \text{unregister}, u : \text{sev} \rangle? \text{ and } \\
&\text{RegSev} := \text{RegSev} \setminus \{u\}
\end{align*}
A Client-Server System

Instance-level specification

Component instance $Client_1$ of $CLIENT$:

\[
\text{if } state = 1 \text{ then } \\
\langle \text{request, } c_1, s_1 \rangle \text{! and } state := 2
\]

\[
\text{if } state = 2 \text{ then } \\
\text{if true then } \\
\langle \text{result, } c_1, s_1 \rangle \text{? and } state := 1
\]

\[
\text{if true then } \\
\langle \text{error, } c_1, s_1 \rangle \text{? and } state := 1
\]
A Client-Server System

Component instance $Client_2$ of $CLIENT$:

if $state = 0$ then
    \langle request, c_2, s_1 \rangle \? and state := 1
if $state = 1$ then
    if true then
        \langle result, c_2, s_1 \rangle \? and state := 4
    if true then
        \langle error, c_2, s_1 \rangle \? and state := 2
if $state = 2$ then
    choose any $\in$ sev and
    \langle request, c_2, any\rangle ! and state := 3
if $state = 3$ then
    if true then
        \langle result, clt_2, any\rangle ? and state := 4
    if true then
        \langle error, clt_2, any\rangle ? and state := 4
A Client-Server System

Component instance Server₁ of SERVER:

if state = 0 then
    if upgrade = 'done' then
        ⟨register, s₁⟩! and state := 1
if state = 1 then
    if empty(Que) = 'y' and
        upgrade = 'ready' then
        ⟨unregister, s₁⟩! and
        state := 0
if true then
    ⟨involve, x : clt⟩? and
    enqueue(x, Que)
if empty(Que) = 'n' then
    let y = head(Que) and
    ⟨return, y⟩! and dequeue(Que)
Problems

Basic properties for the client-server system:

- Whether the system is deadlock-free?
- Whether each component, if not terminated, will be deprived of the right to interact with the connector?
- Whether CSCON restricts the system’s behaviours?
- Whether behaviours of each component, if given a suitable configuration, are realisable?

- Can we know the answers to the above questions without exhausting all possibility of runtime system configurations?
- A pitfall: the semantic variances between component types and instances
Basic properties for the client-server system:

- Whether the system is deadlock-free?
- Whether each component, if not terminated, will be deprived of the right to interact with the connector?
- Whether $CSCON$ restricts the system’s behaviours?
- Whether behaviours of each component, if given a suitable configuration, are realisable?

- Can we know the answers to the above questions without exhausting all possibility of runtime system configurations?
- A pitfall: the semantic variances between component types and instances
Process Algebra

Syntax: \( \mathcal{N} \) a set of names, \( a_i \in \mathcal{N} \).

\[
\lambda ::= \langle a_1, \ldots, a_k \rangle \\
\alpha, \beta, \gamma ::= \lambda? \mid \lambda! \mid \tau
\]

\[
P, Q ::= X \mid \text{nil} \mid P \times Q \mid P \parallel Q
\]

\[
M, M' ::= M + M' \mid \lambda?.P \mid \lambda!.P
\]

Operation semantics:

\[
\begin{align*}
\alpha.P &\xrightarrow{\alpha} P \\
P &\xrightarrow{\alpha} P' \\
P + Q &\xrightarrow{\alpha} P'
\end{align*}
\]

\[
\begin{align*}
P &\xrightarrow{\alpha} P' \\
X &\xrightarrow{\alpha} P' \\
P &\xrightarrow{\alpha} P' \\
P &\xrightarrow{\lambda!} P' \times Q \\
P &\xrightarrow{\lambda?} Q'
\end{align*}
\]

where \( M \) is ‘\( \parallel \)’- and ‘\( \times \)’-free.
From behavioural specification to PA processes

*CSCON, CLIENT, SERVER, client_1*, etc. as PA processes

*Recursive equations for Server_1*

(1) if $\text{Que} = \epsilon$ and $\text{update} = \text{‘ready’}$, then

$$X[2, \epsilon] \triangleq \sum_{a \in clt} \langle \text{involve}, a \rangle?. X[3, a] + \langle \text{unregister}, s \rangle!. X[3, \epsilon]$$

(2) if $\text{Que} \neq \epsilon$, then

$$X[2, \text{Que}] \triangleq \sum_{a \in clt} \langle \text{involve}, a \rangle?. X[2, \text{Que}_a] + \langle \text{return}, c \rangle!. X[2, \text{Que}']$$

where $\text{Que}_a = \text{enqueue}(a, \text{Que})$ such that $a \in clt$, $c = \text{head}(\text{Que})$, and $\text{Que}' = \text{dequeue}(\text{Que})$. 
Main Concepts

An informal glimpse

- an architecture type = component types + a connector
- an architecture (instance) = components + a connector

Definition (Components, connectors, component types)

*Components* are ‘∥’-free processes and *connectors* are ‘∥’- and ‘×’-free processes. *Component types* are ‘∥’-free abstract processes of the form

$$\mathcal{I} = Q(x_1 : A_1, \ldots, x_m : A_m)$$

where (1) $A_i \subseteq \mathcal{N}$ ($1 \leq i \leq m$) are name spaces, and (2) $x_i$ ($1 \leq i \leq m$) are formal parameters of $\mathcal{I}$ with $x_1$ being a distinguished one (which, informally speaking, is reserved for the name of an instance of $\mathcal{I}$).
Main Concepts

An informal glimpse

- an architecture type = component types + a connector
- an architecture (instance) = components + a connector

Definition (Components, connectors, component types)

*Components* are ‘||’-free processes and *connectors* are ‘||’- and ‘×’-free processes. *Component types* are ‘||’-free abstract processes of the form

\[ \mathcal{I} = Q(x_1 : A_1, \ldots, x_m : A_m) \]

where (1) \( A_i \subseteq \mathcal{N} \ (1 \leq i \leq m) \) are name spaces, and (2) \( x_i \) (\( 1 \leq i \leq m \)) are formal parameters of \( \mathcal{I} \) with \( x_1 \) being a distinguished one (which, informally specking, is reserved for the name of an instance of \( \mathcal{I} \)).
Main Concepts

Definition (Architecture types and instances)

A (dynamic, connector-based) architecture type is represented as the tuple

\[ \mathcal{A}^t = \langle I_1, \ldots, I_n, C \rangle \]

An architecture instance of \( \mathcal{A}^t \) is the tuple

\[ \mathcal{A} = \langle P^1_1, \ldots P^m_1, \ldots, P^m_n, C \rangle \]

where \( P^i_j \) conforms to \( I_i \).

Example

\[ \text{CStype} = \langle \text{CLIENT, SERVER, CSCON} \rangle \]
\[ \text{CSsystem} = \langle \text{Client}_1, \text{Client}_2, \ldots, \text{Server}_1, \ldots, \text{CSCON} \rangle \]
Main Concepts

Definition (Canonical components)
If \( a \in A_1 \), we call

\[
\mathcal{I}\langle a \rangle = \sum_{a_2 \in A_2, \ldots, a_m \in A_m} Q\langle a, a_2, \ldots, a_m \rangle
\]
a canonical component of \( \mathcal{I} \).

Definition (Component conformance)
\( P \) conforms to \( \mathcal{I}\langle a \rangle \), denoted \( \mathcal{I}\langle a \rangle \vdash P \), if there is \( R \subseteq \text{Proc} \times \text{Proc} \), such that \( \langle \mathcal{I}\langle a \rangle, P \rangle \in R \) and for each \( \langle P_1, P_2 \rangle \in R \):

- if \( P_1 = \text{nil} \) then \( \langle \mathcal{I}\langle a \rangle, P_2 \rangle \in R \) or \( P_2 = \text{nil} \);
- if \( P_1 \xrightarrow{\alpha} P'_1 \) and \( P_1 \neq \mathcal{I}\langle a \rangle \) and \( P_2 \xrightarrow{\alpha} P'_2 \) and \( \langle P'_1, P'_2 \rangle \in R \) for some \( P'_2 \);
- if \( P_2 \xrightarrow{\alpha} P'_2 \) then \( P_1 \xrightarrow{\alpha} P'_1 \) and \( \langle P'_1, P'_2 \rangle \in R \) for some \( P'_1 \).
Main Concepts

Theorem (Properties of ⊢)

1. $\mathcal{I}\langle a \rangle \vdash \mathcal{I}\langle a \rangle$,
2. $\mathcal{I}\langle a \rangle \vdash P_1 \land \mathcal{I}\langle a \rangle \vdash P_2 \Rightarrow \mathcal{I}\langle a \rangle \vdash P_1 + P_2$,
3. $\mathcal{I}\langle a \rangle \vdash P_1 \land P_1 \simeq P_2 \Rightarrow \mathcal{I}\langle a \rangle \vdash P_2$,
4. $\mathcal{I}\langle a \rangle \vdash P \Rightarrow \mathcal{I}\langle a \rangle \vdash P^*$,
5. $\vdash$ is decidable.

Example

$\text{CLIENT}\langle c_1 : \text{clt} \rangle \vdash \text{Client}_1$
$\text{CLIENT}\langle c_2 : \text{clt} \rangle \vdash \text{Client}_2$
$\text{SERVER}\langle s_1 : \text{sev} \rangle \vdash \text{Server}_1$
Main Concepts

Definition (Component configurations)
A *component configuration* of $A^t$ is a process of the form

$$F = P_1\langle a_1 \rangle \times \ldots \times P_n\langle a_n \rangle$$

such that, for each $1 \leq i \neq j \leq n$, $a_i \neq a_j$ and $\mathcal{I}_i\langle a_i \rangle \vdash P_i$ for some interface $\mathcal{I}_i$ of $A^t$.

Definition (Architectures revisited)
The semantics of an architecture $A$ can be considered as the process $F \parallel C$.

Example

$$CS\text{system} = (client_1 \times client_2 \times \ldots \times server_1 \times \ldots) \parallel CSCON$$
Properties

Definition (Deadlock-freedom)

(1) An architecture instance $\mathcal{A} = F \parallel C$ is deadlock-free, if the following proposition holds: if $\mathcal{A} \xrightarrow{*} F' \parallel C'$ and $F' \xrightarrow{}$, then $F' \parallel C' \xrightarrow{}$. (2) An architecture type $\mathcal{A}^t$ is deadlock-free if each instance of $\mathcal{A}^t$ is deadlock-free.

Definition (Non-starvation)

$\mathcal{A} = (F \times P) \parallel C$ is non-starving, if the following holds: if $\mathcal{A} \xrightarrow{*} (F' \times P') \parallel C'$ and $P' \xrightarrow{}$, then there are $F''$ and $C''$ such that $F' \parallel C' \xrightarrow{*} F'' \parallel C''$ and $P' \parallel C'' \xrightarrow{}$. (2) An architecture type $\mathcal{A}^t$ is non-starving if each instance of $\mathcal{A}^t$ is non-starving.

Lemma

Non-starvation implies deadlock-freedom.
Properties

Conservation: behaviours of architecture instances are refined by the connector.

**Definition (Conservation)**
An architecture type $\mathcal{A}^t$ is *conservative*, if, for each $\tilde{\alpha}$ such that $C \xrightarrow{\tilde{\alpha}}$, there is a configuration $F$ such that $F \xrightarrow{\#\tilde{\alpha}}$ where $\#\tilde{\alpha}$ is the dual sequence of $\tilde{\alpha}$ w.r.t. $\{?, !\}$.

Completeness: the connector does not exclude behaviours of components.

**Definition (Completeness)**
$\mathcal{A}^t$ is complete if the following proposition holds: for each component $P$ and $P' \in \text{Proc}(P)$, if $P' \xrightarrow{\alpha}$, then $(F \times P) \parallel C \xrightarrow{*} (F' \times P') \parallel C'$ for some $F, F', C'$ such that $C' \xrightarrow{\#\alpha}$.
The Method

The method is to construct a specific architecture instance which can mimic just all behaviours of possible components.

Definition (Construction procedure)
For any given component $P$, we choose a new process identifier $X_P$. The iteration of $P$, denoted by $P^*$, is obtained by substituting $nil$ in $P$ by $X_P$, and the recursive equation for $X_P$ is $X_P \equiv P^*$. A canonical configuration and a canonical architecture instance are respectively defined as

$$F^*_c = X_{P_{c,1}} \times \ldots \times X_{P_{c,k}} \quad \mathcal{A}^*_c = F^*_c \parallel C$$

where $P_{c,1}, \ldots, P_{c,k}$ enumerate all canonical components of $\mathcal{A}^t$. 
The Method

**N.B.** The number of canonical components of an architecture type is the number of *possible* components. For example, the number of canonical components of \( CStype \) is \(|clt| + |sev|\). But the number of possible configurations for \( CStype \) is \( 2|clt|+|sev| \).

The following lemma says that the iteration of a canonical component’s behaviours are just enough to mimic all of its components’ behaviours in some sense.

**Lemma**

- If \( P \xrightarrow{\tilde{\alpha}} \) and \( \mathcal{I} \langle a \rangle \vdash P \), then \( \mathcal{I} \langle a \rangle^* \xrightarrow{\tilde{\alpha}} \);
- If \( \mathcal{I} \langle a \rangle^* \xrightarrow{\tilde{\alpha}} \), then there is \( P \) such that \( P \xrightarrow{\tilde{\alpha}} \) and \( \mathcal{I} \langle a \rangle \vdash P \).
Deadlock-Freedom

Theorem
\( A^t \) is deadlock-free if and only if the depth-first search algorithm on the right returns ‘yes’.

Significance
It searches in \( A_c^* \)’s state space but verifies \( A^t \)’s property.

---

Data: \( A_c^* \)
Output: ‘yes’ or ‘no’

Let \( bool = 1 \)

\[
\text{foreach } (P_1 \times \ldots \times P_k) \parallel C' \in \text{Proc}(A_c^*) \text{ do}
\]

\[
\text{if } P_1 \times \ldots \times P_k = F_c^* \text{ then}
\]

\[
\quad \text{if } X_{P_{c,i}} \parallel C' \not\rightarrow, \exists 1 \leq i \leq k \text{ then}
\]

\[
\qquad \text{bool} := 0
\]

\[
\quad \text{break}
\]

\[
\text{else}
\]

\[
\quad \text{Let } P'_i = P_i\{\text{nil}/X_{I_i(a)}\}, \forall 1 \leq i \leq k
\]

\[
\quad \text{if } (P'_1 \times \ldots \times P'_k) \parallel C' \not\rightarrow \text{ then}
\]

\[
\qquad \text{bool} := 0
\]

\[
\quad \text{break}
\]

\[
\text{if } bool = 1 \text{ then return ‘yes’}
\]

\[
\text{else return ‘no’}
\]
Non-Starvation

Data: $A_c^*$
Output: ‘yes’ or ‘no’

Let $bool = 1$

foreach $(P_1 \times \ldots \times P_k) \parallel C' \in \text{Proc}(A_c^*)$ do
  Let $P'_i = P_i\{\text{nil}/X_i(a)\}, \forall 1 \leq i \leq k$
  foreach $1 \leq i \leq k$ do
    Let $F_i = P'_1 \times \ldots P'_{i-1} \times P'_{i+1} \times \ldots P'_k$
    if there are $F'_i$ and $C''$ such that
      \[ F_i \parallel C' \xrightarrow{*} F'_i \parallel C'' \]
      \[ P_i \parallel C'' \rightarrow \]
    then
      skip
    else
      $bool := 0$
      break
  
if $bool = 1$ then return ‘yes’
else return ‘no’

Theorem

$A^t$ is non-starving if and only if the depth-first search algorithm on the left returns ‘yes’.

Significance

$A_c^* \leadsto A^t$. 
Conservation (I)

Definition (Determinism)
An architecture type $A^t$ is deterministic if its connector and all of its canonical components are deterministic.

Theorem
Suppose $A^t$ is deterministic. $A^t$ is conservative if and only if the depth-first search algorithm on the right returns ‘yes’.

Significance
$A^*_c \sim A^t$. 

<table>
<thead>
<tr>
<th>Data: $A^*_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: ‘yes’ or ‘no’</td>
</tr>
<tr>
<td>Let $bool = 1$</td>
</tr>
<tr>
<td>foreach $(P_1 \times \ldots \times P_k) \parallel C' \in \text{Proc}(A^*_c)$ do</td>
</tr>
<tr>
<td>foreach $\alpha$ s.t. $C' \xrightarrow{\alpha}$ do</td>
</tr>
<tr>
<td>if $(P_1 \times \ldots \times P_k) \xrightarrow{\not\alpha}$ then</td>
</tr>
<tr>
<td>$bool := 0$</td>
</tr>
<tr>
<td>break</td>
</tr>
<tr>
<td>if $bool = 1$ then return ‘yes’</td>
</tr>
<tr>
<td>else return ‘no’</td>
</tr>
</tbody>
</table>
Completeness (I)

Theorem
Suppose $A^t$ is deterministic. $A^t$ is complete if and only if the algorithm on the right returns ‘yes’.

Data: $A^*_c$
Output: ‘yes’ or ‘no’

Let $bool = 1$
foreach $P \in \text{Proc}(P_{c,i}), 1 \leq i \leq k$ do
    let $S = \{ C' \mid F^*_c \parallel C \xrightarrow{\star} F \parallel C', F[i] = P \}$
    if there exists $\alpha$ s.t. $P \xrightarrow{\alpha}$ and $C' \xrightarrow{\parallel \alpha}$ for all $C' \in S$ then
        $bool := 0$
        break
if $bool = 1$ then return ‘yes’
else return ‘no’

Significance

- $A^*_c \xrightarrow{\sim} A^t$,
- Compositionality: it checks the canonical components on the one-by-one basis.
Two further assumptions:

- Different canonical components share no actions
  \( \text{Act}(P_{c,i}) \cap \text{Act}(P_{c,j}) = \emptyset \text{ if } i \neq j \); 

- Actions of components are dual to actions of the connector, and vice versa
  \( \bigcup_{i=1}^{k} \text{Act}(P_{c,i}) = \{ \#\alpha | \alpha \in \text{Act}(C) \} \).

Let \( \tilde{\alpha}|A \) be the projection of \( \tilde{\alpha} \) on \( A \subseteq \text{Act} \). \( Q \xrightarrow{\tilde{\gamma}}_A Q' \) if and only if there is \( \tilde{\alpha} \) such that \( Q \xrightarrow{\tilde{\alpha}} Q' \) and \( \tilde{\gamma} = \tilde{\alpha}|A \).

**Definition**

1. \( C \preceq P_{c,i} \) if \( C \xrightarrow{\tilde{\alpha}} \text{Act}(P_{c,i}) \) implies \( P^*_{c,i} \xrightarrow{\#\tilde{\alpha}} \); 
2. \( C \succeq P_{c,i} \) if \( P^*_{c,i} \xrightarrow{\tilde{\alpha}} \) implies \( C \xrightarrow{\#\tilde{\alpha}} \text{Act}(P_{c,i}) \).

**Theorem**

\( A^t \) is conservative (resp. complete) if and only if \( C \preceq P_{c,i} \) (resp. \( C \succeq P_{c,i} \)) for each canonical component \( P_{c,i} \) of \( A^t \).
Conservation (II)

**Theorem**

*With the previous three assumptions, $C \preceq P_{c,i}$ if and only if the algorithm on the right returns ‘yes’.*

**Significance**

- $A^*_c \leadsto A^t$,
- Compositional checking.

<table>
<thead>
<tr>
<th>Data: $C, P_{c,i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Output:</strong> ‘yes’ or ‘no’</td>
</tr>
</tbody>
</table>

Construct a graph $G = \langle V, E \rangle$ such that

- $V = \text{Proc}(P^*_c) \times \text{Proc}(C)$
- $E = \{\langle\langle P_1, C_1\rangle, \langle P_2, C_2\rangle\rangle \mid P_1 \xrightarrow{\alpha} P_2, C_1 \xrightarrow{\#\alpha} \text{Act}(P_{c,i}) C_2, \alpha \in \text{Act}(P_{c,i})\}$

Let $bool = 1$

```latex
definizione
\[
\text{foreach } \langle P, C' \rangle \text{ reachable from } \langle P^*_c, i, C \rangle \text{ in } G \text{ do}
\]

```

```latex
\text{if } \text{there is } \gamma \text{ such that}
```

```latex
\[
\begin{align*}
\langle P, C' \rangle & \xrightarrow{\gamma} \\
\end{align*}
```

```latex
\text{then}
```

```latex
\begin{align*}
bool & := 0 \\
\text{break}
\end{align*}
```

```latex
\text{if } bool = 1 \text{ then return ‘yes’}
\text{else return ‘no’}
```
Completeness (II)

Data: C, P_{c,i}
Output: ‘yes’ or ‘no’

Construct a graph G = ⟨V, E⟩ as in the previous algorithm
Let bool = 1
foreach ⟨P, C’⟩ reachable from ⟨P^*, i, C⟩ in G do
  if there is γ such that
    ▶ P ↠ γ
    ▶ C’ ↦ γ Act(P_{c,i})
  then
    bool := 0
    break
if bool = 1 then return ‘yes’
else return ‘no’

Theorem
With the previous three assumptions plus the conservation of A_t, C ⊳ P_{c,i} if and only if the algorithm on the left returns ‘yes’.

Significance
▶ A^*_c ⊳ A_t,
▶ Finer-grained compositional checking.
Summary

- We propose a semantic model for the dynamic connector-based architecture styles;

- We show that the analysis of several basic properties of these architecture styles depends on architecture instances with fixed configurations, reducing the verification state space.

Outlook: the bridge between our semantic model and a mature ADL?