CALM
Explaining CADENA’s Type System

Georg Jung
jung@cis.ksu.edu

SAnToS Laboratory, Department of Computing and Information Sciences, Kansas State University

SAnToS Laboratory Internal Presentation
February 14, 2007
CALM features three basic categories of elements:

- Interfaces
- Components
- Connectors

Each category is characterized through their specification elements, which are attributes, ports, and roles.
CALM features three basic categories of elements:

- Interfaces
  - Attributes
- Components
- Connectors

Each category is characterized through their specification elements, which are attributes, ports, and roles.
CALM features three basic categories of elements:

- Interfaces
  - Attributes

- Components
  - Attributes + Ports

- Connectors

Each category is characterized through their specification elements, which are attributes, ports, and roles.
CALM features three basic categories of elements:

- Interfaces
  - Attributes
- Components
  - Attributes + Ports
- Connectors
  - Attributes + Roles

Each category is characterized through their specification elements, which are attributes, ports, and roles.
The specifiers attributes, ports, and roles are recorded in mappings.
Specifier Mappings

The specifiers attributes, ports, and roles are recorded in mappings.

Definition

An attribute mapping $\alpha \in \hat{A}$ maps identifiers $id \in Id$ to pairs formed of the constructor term $a^m$ and an attribute type $\hat{\tau}_a \in \hat{T}_a$

$$\hat{A} = Id \rightarrow_{fin} \{(a^m, \hat{\tau}_a) | \hat{\tau}_a \in \hat{T}_a\}$$
Specifier Mappings

The specifiers **attributes, ports, and roles** are recorded in **mappings**.

**Definition**

An **attribute mapping** $\hat{\alpha} \in \hat{A}$ maps identifiers $id \in Id$ to pairs formed of the constructor term $a^m$ and an attribute type $\hat{\tau}_a \in \hat{T}_a$

$$\hat{A} = Id \rightarrow^{fin} \{(a^m, \hat{\tau}_a) | \hat{\tau}_a \in \hat{T}_a\},$$

**Notations**

- **Shorthand Mapping**: $\hat{A} = Id \rightarrow (a^m \hat{\tau}_a)$
- **Judgement Style**: $\hat{\alpha} \vdash id \leftrightarrow (a^m \hat{\tau}_a)$
Greek...
Specifier Mappings

Attributes
\[ \hat{\alpha} \in \hat{\mathcal{A}}, \quad \hat{\mathcal{A}} = \text{id} \rightarrow (a^m \hat{\tau}_a), \quad \hat{\alpha} \vdash \text{id} \leftrightarrow (a^m \hat{\tau}_a). \]

Port (options)
\[ \hat{\pi} \in \hat{\Pi}, \quad \hat{\Pi} = \text{id} \rightarrow (p^m \hat{\tau}_p), \quad \hat{\pi} \vdash \text{id} \leftrightarrow (p^m \hat{\tau}_p). \]

Role (options)
\[ \hat{\rho} \in \hat{\mathcal{R}}, \quad \hat{\mathcal{R}} = \text{id} \rightarrow (r^m \hat{\tau}_r), \quad \hat{\rho} \vdash \text{id} \leftrightarrow (r^m \hat{\tau}_r). \]
Meta-kinds are tuples of mappings, mapping their identifiers to attributes/ports/roles. The three categories are distinguished by which sorts of mapping they contain.
Meta-kind Mappings

Definition

The meta-kind mapping is some $\gamma \in \Gamma$ with

$$\Gamma = \text{Id} \rightarrow ((i^m \hat{A}) \cup (l^m \hat{A} \times \hat{R}) \cup (c^m \hat{A} \times \hat{N}))$$

Meta-kinds are tuples of mappings, mapping their identifiers to attributes/ports/roles. The three categories are distinguished by which sorts of mapping they contain.
Greek...

Meta-kind Mappings

Definition

The meta-kind mapping is some $\gamma \in \Gamma$ with

$$\Gamma = \text{id} \rightarrow (((i^m \hat{A}) \cup (l^m \hat{A} \times \hat{R}) \cup (c^m \hat{A} \times \hat{\Pi}))$$

i. e.,

- Interface Meta-kinds: $\gamma \vdash id_i \leftrightarrow (i^m \hat{\alpha})$,
- Component Meta-kinds: $\gamma \vdash id_c \leftrightarrow (c^m (\hat{\alpha}, \hat{\pi}))$, and
- Connector Meta-kinds: $\gamma \vdash id_l \leftrightarrow (l^m (\hat{\alpha}, \hat{\rho}))$.

Meta-kinds are tuples of mappings, mapping their identifiers to attributes/ports/roles. The three categories are distinguished by which sorts of mapping they contain.
Greek...

Kind Mappings

The kind mapping is some $\kappa \in K$ with $K = \text{Id} \rightarrow (\bar{i}_A \cup (\bar{l}_A \times \bar{R}) \cup (\bar{c}_A \times \bar{\Pi}))$. i.e., derived from the respective meta-kinds: $\gamma \vdash \text{id} \mapsto \bar{i}\hat{\alpha}$; $\kappa \vdash \text{id} \mapsto \bar{i}_\bar{\alpha}$, $\gamma \vdash \text{id} \mapsto \bar{c}_m(\hat{\alpha}, \hat{\pi})$; $\kappa \vdash \text{id} \mapsto \bar{c}(\bar{\alpha}, \bar{\pi})$, and $\gamma \vdash \text{id} \mapsto \bar{l}_m(\hat{\alpha}, \hat{\rho})$; $\kappa \vdash \text{id} \mapsto \bar{l}(\bar{\alpha}, \bar{\rho})$. 
Greek...
Kind Mappings

Definition

The kind mapping is some \( \kappa \in K \) with

\[
K = Id \rightarrow ((i \tilde{A}) \cup (l \tilde{A} \times \tilde{R}) \cup (c \tilde{A} \times \tilde{\Pi}))
\]
Greek...

Kind Mappings

Definition

The kind mapping is some $\kappa \in K$ with

$$K = \text{Id} \rightarrow ((i \tilde{A}) \cup (l \tilde{A} \times \tilde{R}) \cup (c \tilde{A} \times \tilde{Π}))$$

i. e., derived from the respective meta-kinds:

$$\gamma \vdash id_0 \leftrightarrow (i^m \hat{\alpha}) \leadsto \kappa \vdash id_1 \leftrightarrow (i \tilde{\alpha}),$$

$$\gamma \vdash id_0 \leftrightarrow (c^m (\hat{\alpha}, \hat{\pi})) \leadsto \kappa \vdash id_1 \leftrightarrow (c (\tilde{\alpha}, \tilde{\pi})), \text{ and}$$

$$\gamma \vdash id_0 \leftrightarrow (l^m (\hat{\alpha}, \hat{\rho})) \leadsto \kappa \vdash id_1 \leftrightarrow (l (\tilde{\alpha}, \tilde{\rho})).$$
The General Idea:

\[ \gamma \vdash \text{meta-kind} \mapsto (\text{category} (\hat{\alpha}, \hat{\pi}, \hat{\rho})) \]

\[ \kappa \vdash \text{kind-name} \mapsto (\text{category} (\bar{\alpha}, \bar{\pi}, \bar{\rho})) \]

\[ \psi \vdash \text{type-name} \mapsto (\text{kind-name} (\alpha, \pi, \rho)) \]

\[ \theta \vdash \text{inst-name} \mapsto (\text{type-name} (\hat{\alpha}, \hat{\pi}, \hat{\rho})). \]
The General Idea:

\[ \gamma \vdash \text{meta-kind} \mapsto (\text{category} (\hat{\alpha}, \hat{\pi}, \hat{\rho})) \]

\[ \kappa \vdash \text{kind-name} \mapsto (\text{category} (\bar{\alpha}, \bar{\pi}, \bar{\rho})) \]

\[ \psi \vdash \text{type-name} \mapsto (\text{kind-name} (\alpha, \pi, \rho)) \]

\[ \theta \vdash \text{inst-name} \mapsto (\text{type-name} (\hat{\alpha}, \hat{\pi}, \hat{\rho})). \]
The General Idea:

\[ \gamma \vdash \text{meta-kind} \mapsto (\text{category} (\hat{\alpha}, \hat{\pi}, \hat{\rho})) \]
\[ \kappa \vdash \text{kind-name} \mapsto (\text{category} (\bar{\alpha}, \bar{\pi}, \bar{\rho})) \]
\[ \psi \vdash \text{type-name} \mapsto (\text{kind-name} (\alpha, \pi, \rho)) \]
\[ \theta \vdash \text{inst-name} \mapsto (\text{type-name} (\check{\alpha}, \check{\pi}, \check{\rho})). \]
The General Idea:

\[
\begin{align*}
\gamma \vdash \text{meta-kind} & \mapsto (\text{category} (\hat{\alpha}, \hat{\pi}, \hat{\rho})) \\
\kappa \vdash \text{kind-name} & \mapsto (\text{category} (\bar{\alpha}, \bar{\pi}, \bar{\rho})) \\
\psi \vdash \text{type-name} & \mapsto (\text{kind-name} (\alpha, \pi, \rho)) \\
\theta \vdash \text{inst-name} & \mapsto (\text{type-name} (\dot{\alpha}, \dot{\pi}, \dot{\rho})).
\end{align*}
\]

The mappings allow to reconstruct type and kind of any instance.
Background
Adjusting minimum/maximum fan-out

Definition
Ranges can be *shifted* by a subtrahend $c \in \mathbb{N}_0$ with

\[
[n..m] - c = \begin{cases} 
[n-c..m-c], & \text{if } n \geq c \\
[0..m-c], & \text{if } n < c, m \geq c \\
\text{undefined}, & \text{if } m < c 
\end{cases}
\]

where $* > c$ and $* - c = *$ for all $c \in \mathbb{N}_0$. 
Background
Adjusting minimum/maximum fan-out

Definition
Ranges can be *shifted* by a subtrahend \( c \in \mathbb{N}_0 \) with

\[
[n..m] - c = \begin{cases} 
[n - c..m - c], & \text{if } n \geq c \\
[0..m - c], & \text{if } n < c, m \geq c \\
\text{undefined}, & \text{if } m < c
\end{cases}
\]

where \( \ast > c \) and \( \ast - c = \ast \) for all \( c \in \mathbb{N}_0 \).

Example
- \([0..\ast] - 2 = [0..\ast]\)
Background
Adjusting minimum/maximum fan-out

Definition
Ranges can be *shifted* by a subtrahend \( c \in \mathbb{N}_0 \) with

\[
[n..m] - c = \begin{cases} 
[n - c..m - c], & \text{if } n \geq c \\
[0..m - c], & \text{if } n < c, m \geq c \\
\text{undefined}, & \text{if } m < c 
\end{cases}
\]

where \( * > c \) and \( * - c = * \) for all \( c \in \mathbb{N}_0 \).

Example
- \([0..*] - 2 = [0..*]\)
- \([1..3] - 2 = [0..1]\)
Background
Adjusting minimum/maximum fan-out

Definition
Ranges can be shifted by a subtrahend $c \in \mathbb{N}_0$ with

$$[n..m] - c = \begin{cases} [n - c..m - c], & \text{if } n \geq c \\ [0..m - c], & \text{if } n < c, m \geq c \\ \text{undefined}, & \text{if } m < c \end{cases}$$

where $\ast > c$ and $\ast - c = \ast$ for all $c \in \mathbb{N}_0$.

Example
- $[0..\ast] - 2 = [0..\ast]$
- $[1..3] - 2 = [0..1]$
- $[1..3] - 4 = \$ undefined \$
### Typing an Assembly

<table>
<thead>
<tr>
<th>Open—Closed</th>
<th>Complete—Incomplete</th>
</tr>
</thead>
</table>

**Definition**

A port/role on a component/connector instance is **open**, iff its **multiplexity** (i.e., minimum to maximum fan-out interval) allows further connections.

**Definition**

A port/role on a component/connector instance is **incomplete**, iff its **multiplexity** requires further connections.

**Example**

In nesC, ports have multiplexity \([0..1]\), roles have multiplexity \([1]\).

- all ports can be exposed
- unconnected roles have to be exposed
Typing an Assembly

**Open—Closed**

**Complete—Incomplete**

**Definition**

A port/role on a component/connector instance is **open**, iff its multiplexity (*i.e.*, minimum to maximum fan-out interval) **allows** further connections.

Example

In nesC, ports have multiplexity $[0..1]$, roles have multiplexity $[1]$. All ports can be exposed, unconnected roles have to be exposed.
Typing an Assembly

Definition
A port/role on a component/connector instance is **open**, iff its multiplexity (*i.e.*, minimum to maximum fan-out interval) **allows** further connections.

Definition
A port/role on a component/connector instance is **incomplete**, iff its multiplexity **requires** further connections.
Typing an Assembly

Open—Closed | Complete—Incomplete

Definition
A port/role on a component/connector instance is **open**, iff its multiplexity (i.e., minimum to maximum fan-out interval) allows further connections.

Definition
A port/role on a component/connector instance is **incomplete**, iff its multiplexity requires further connections.

Example
In nesC, ports have multiplexity \([0..1]\), roles have multiplexity \([1]\).
- all ports can be exposed
- unconnected roles have to be exposed
Typing an Assembly

Concept

To type an assembly one has to
Typing an Assembly

Concept

To type an assembly one has to

- **shift** all port/role multiplexities according to existing connections
Typing an Assembly

To type an assembly one has to

- **shift** all port/role multiplexities according to existing **connections**
- **expose** ports/roles according to **openness/completeness** constraints
Typing an Assembly

Concept

To type an assembly one has to

- **shift** all port/role multiplexities according to existing **connections**
- **expose** ports/roles according to **openness/completeness** constraints
- **unify with existing type**
Typing an Assembly

Concept

To type an assembly one has to

- **shift** all port/role multiplexities according to existing **connections**
- **expose** ports/roles according to **openness/completeness** constraints
- **unify with** existing **type**

Assembly Instance

\[ \theta \vdash \text{assembly} \mapsto (\dot{\alpha}, \dot{\pi}, \dot{\rho}) \]

Assembly Type

\[ \psi \vdash \tau_{\text{assembly}} \mapsto (\alpha, \pi, \rho) \]
Typing an Assembly
Component or Connector

Assembly Instance

\[ \theta \vdash \text{assembly} \mapsto (\dot{\alpha}, \dot{\pi}, \dot{\rho}) \]
Typing an Assembly
Component or Connector

Assembly Instance

\[ \theta \vdash \text{assembly} \leftrightarrow (\dot{\alpha}, \dot{\pi}, \dot{\rho}) \]

Typing as Component
All roles must be complete

\[ \psi \vdash \tau_{\text{assembly}} \leftrightarrow (\alpha, \pi) \]
Typing an Assembly

Component or Connector

Assembly Instance

\[ \theta \vdash \text{assembly} \mapsto (\dot{\alpha}, \dot{\pi}, \dot{\rho}) \]

Typing as Component

All roles must be complete

\[ \psi \vdash \tau_{\text{assembly}} \mapsto (\alpha, \pi) \]

Typing as Connector

All ports must be complete

\[ \psi \vdash \tau_{\text{assembly}} \mapsto (\alpha, \rho) \]
Typing an Assembly

Summary

- Open ports/roles can be omitted if they are complete.
Typing an Assembly

Summary

- Open ports/roles can be omitted if they are complete
- If all ports are complete, the category of the assembly can be changed to connector
Typing an Assembly

Summary

- Open ports/roles can be omitted if they are complete.
- If all ports are complete, the category of the assembly can be changed to connector.
- If all roles are complete, the category of the assembly can be changed to component.
Typing an Assembly

Summary

- Open ports/roles can be omitted if they are complete.
- If all **ports** are complete, the **category** of the assembly can be changed to **connector**.
- If all **roles** are complete, the **category** of the assembly can be changed to **component**.
- The unification with a component/connector type is essentially wrapping!
Typing an Assembly

Summary

- Open ports/roles can be omitted if they are complete.
- If all ports are complete, the category of the assembly can be changed to connector.
- If all roles are complete, the category of the assembly can be changed to component.
- The unification with a component/connector type is essentially wrapping!

Note:

- Wraps between different architectural styles need to unify ports or roles.
- Therefore, inter style coercion has to work on interfaces.
- Components and connectors are not essential for inter style coercions.
Style Development

The Root Style

The Root style has no basis in the formal concept. The original idea was that the category is decided through the root of the inheritance tree. The category is fixed by keyword instead. Meta-kinds should allow multiple inheritance. Problems of multiple inheritance come from overloading/overriding. In purely structural inheritance they do not matter. Common functionalities (captured by interfaces in Java) should be grouped in specific meta-kinds.

Note: In absence of method-implementations, inheritance boils down to set-union.
Style Development

The Root Style

- The Root style has no basis in the formal concept
  - The original idea was that the *category* is decided through the root of the inheritance tree
  - The category is fixed by *keyword* instead
Style Development

The Root Style

- The Root style has no basis in the formal concept
  - The original idea was that the *category* is decided through the root of the inheritance tree
  - The category is fixed by *keyword* instead
- Meta-kinds should allow *multiple inheritance*
  - Problems of multiple inheritance come from overloading/overriding. In purely structural inheritance they do not matter.
  - Common functionalities (captured by *interfaces* in Java) should be grouped in specific meta-kinds.
Style Development

The Root Style

- The Root style has no basis in the formal concept
  - The original idea was that the *category* is decided through the root of the inheritance tree
  - The category is fixed by *keyword* instead

- Meta-kinds should allow *multiple inheritance*
  - Problems of multiple inheritance come from overloading/overriding. In purely structural inheritance they do not matter.
  - Common functionalities (captured by *interfaces* in Java) should be grouped in specific meta-kinds.

Note

In absence of method-implementations, inheritance boils down to *set-union*

- Eliminate Root style
- Allow multiple inheritance
A style is characterized through its meta-kinds and kinds
Style Development

Inter Style Inheritance

A style is characterized through its *meta-kinds* and *kinds*

- if kinds are inherited, modules/scenarios which use the respective kinds need to *still work*
  - Inheritance and Elision are meant to support *inter style migration*
Style Development

Inter Style Inheritance

A style is characterized through its meta-kinds and kinds

- if kinds are inherited, modules/scenarios which use the respective kinds need to still work
  - Inheritance and Elision are meant to support inter style migration
- Styles should allow more than union through inheritance
  - styles should be seen as sets of (meta-) kinds
  - general set operations should be possible (union, intersection, etc.)
A style is characterized through its meta-kinds and kinds
- if kinds are inherited, modules/scenarios which use the respective kinds need to still work
  - Inheritance and Elision are meant to support inter style migration
- Styles should allow more than union through inheritance
  - styles should be seen as sets of (meta-) kinds
  - general set operations should be possible (union, intersection, etc.)
- Compliance to styles should be inferred, not hardcoded
  - modules comply to every style which contains all the kinds they use
  - scenarios comply to styles which all their modules comply to
Style Development

Inter Style Inheritance

A style is characterized through its meta-kinds and kinds

- if kinds are inherited, modules/scenarios which use the respective kinds need to still work
  - Inheritance and Elision are meant to support inter style migration
- Styles should allow more than union through inheritance
  - styles should be seen as sets of (meta-) kinds
  - general set operations should be possible (union, intersection, etc.)
- Compliance to styles should be inferred, not hardcoded
  - modules comply to every style which contains all the kinds they use
  - scenarios comply to styles which all their modules comply to

Note

The rigid inheritance structure and the fact that binding of modules/scenarios to styles is “hardwired” makes most refinement capabilities we claim impossible.
Style Development

Interface-typing constraints

The interface-type constraints on the style tier have to be more precise.
Style Development

Interface-typing constraints

The interface-type constraints on the style tier have to be more precise.

- Typing constraints (currently available for connectors) have to be available globally, especially for components
  - Component kinds might require very specific interfaces (e.g., a monitor interface)
Style Development

Interface-typing constraints

The interface-type constraints on the style tier have to be more precise.

- Typing constraints (currently available for connectors) have to be available **globally**, especially for **components**
  - Component kinds might require very specific interfaces (e.g., a monitor interface)
- A style (especially a **refined** style) should be enabled to **require** specific **modules**
  - This way, styles can **explicitly** reference interface types (e.g., interfaces representing a library)
Style Development

Interface-typing constraints

The interface-type constraints on the style tier have to be more precise.

- Typing constraints (currently available for connectors) have to be available globally, especially for components
  - Component kinds might require very specific interfaces (e.g., a monitor interface)

- A style (especially a refined style) should be enabled to require specific modules
  - This way, styles can explicitly reference interface types (e.g., interfaces representing a library)

Note

Referencing library interfaces from a style is an important step from platform-oriented styles towards product-line-oriented styles
We need to introduce Implementation Tables
Style Development

Implementation Tables -1-

We need to introduce **Implementation Tables**

- Implementation Tables associate **types** with **possible implementations**
  - Implementations can be **CADENA assemblies**, or **code**, etc.
  - Each type can have **multiple alternative implementations**, organized by attributes (e.g., QoS, platform particularities, etc.)
We need to introduce **Implementation Tables**

- Implementation Tables associate *types with possible implementations*
  - Implementations can be **CADENA** assemblies, or *code, etc.*
  - Each type can have *multiple alternative implementations*, organized by attributes (*e.g.*, QoS, platform particularities, *etc.*)

- Implementation tables must associate a type with
  - a reference to an implementing object
  - a *wrap* which justifies the association
  - *attributes* which characterize this implementation wrt. alternatives
Style Development
Implementation Tables -2-

We need to introduce Implementation Tables
We need to introduce **Implementation Tables**

- **CADENA** scenarios should not emphasize component *internals* but rather global *scenario properties*
  - Drop distinction between “component” and “sub-assembly”
  - *Every* component/service has to be associated with an implementation
  - sub-assemblies form inner nodes of an *implementation tree*
  - distinguish scenario *completion levels*
    - *assemblies* being networks of components without attached implementations
    - *scenarios* include implementation information, they can be incomplete or deployable
We need to introduce **Implementation Tables**
We need to introduce Implementation Tables

Notes

- The implementation table spans the three tiers (style, module, and scenario). It has to be an independent entity.
- Eventually, implementations for connectors have to be type parametric or type polymorphic.
Wraps combine aspects of all three CALM tiers
Wraps combine aspects of all three CALM tiers

- On the style tier, the possibility to implement elements of an interface kind by sets of interfaces of a different style has to be declared and constrained
Wraps combine aspects of all three CALM tiers

- On the style tier, the possibility to implement elements of an interface kind by sets of interfaces of a different style has to be declared and constrained

- On the module tier, interface type conversions have to be defined in accord with style-level constraints
Wraps combine aspects of all three CALM tiers

- On the style tier, the possibility to implement elements of an interface kind by sets of interfaces of a different style has to be declared and constrained.

- On the module tier, interface type conversions have to be defined in accord with style-level constraints.

- On the scenario tier, there must be a mechanism to summarize exposed interfaces and convert them according to module level declarations.
Questions?