

# Modeling Complex Systems: A Coalgebraic Perspective

Sun Meng

LMAM & Department of Informatics, School of Mathematical Sciences,  
Peking University,  
Beijing, China

<http://www.math.pku.edu.cn/teachers/sunm>

November 29, 2016

# Roadmap

1. Introduction.
2. Components as Coalgebras.
3. A Coalgebraic Perspective on UML.
4. Coalgebras and Logic (almost  $\emptyset$ ).
5. Future Opportunities.

## Motivation: Question and Approach

- The development of science proceeds in a cycle of activities:
  - **Analysis**: Understand the problem;
  - **Abstraction**: creating a mathematical model by eliminating irrelevant details in order to identify what is essential;
  - **Reasoning**: reasoning within the model, getting a collection of general laws;
  - **Specialization**: the general laws are instantiated to the specific problem and a solution (= an implementation) is calculated, which leads to further understanding, and input for another round of activities.
- Is this process relevant to the development of software systems?

## Motivation: Question and Approach

- The development of science proceeds in a cycle of activities:
  - **Analysis**: Understand the problem;
  - **Abstraction**: creating a mathematical model by eliminating irrelevant details in order to identify what is essential;
  - **Reasoning**: reasoning within the model, getting a collection of general laws;
  - **Specialization**: the general laws are instantiated to the specific problem and a solution (= an implementation) is calculated, which leads to further understanding, and input for another round of activities.
- Is this process relevant to the development of software systems?

## Motivation: Question and Approach

- The development of science proceeds in a cycle of activities:
  - **Analysis**: Understand the problem;
  - **Abstraction**: creating a mathematical model by eliminating irrelevant details in order to identify what is essential;
  - **Reasoning**: reasoning within the model, getting a collection of general laws;
  - **Specialization**: the general laws are instantiated to the specific problem and a solution (= an implementation) is calculated, which leads to further understanding, and input for another round of activities.
- Is this process relevant to the development of software systems?

## Motivation: Question and Approach

- The development of science proceeds in a cycle of activities:
  - **Analysis**: Understand the problem;
  - **Abstraction**: creating a mathematical model by eliminating irrelevant details in order to identify what is essential;
  - **Reasoning**: reasoning within the model, getting a collection of general laws;
  - **Specialization**: the general laws are instantiated to the specific problem and a solution (= an implementation) is calculated, which leads to further understanding, and input for another round of activities.
- Is this process relevant to the development of software systems?

## Motivation: Question and Approach

- The development of science proceeds in a cycle of activities:
  - **Analysis**: Understand the problem;
  - **Abstraction**: creating a mathematical model by eliminating irrelevant details in order to identify what is essential;
  - **Reasoning**: reasoning within the model, getting a collection of general laws;
  - **Specialization**: the general laws are instantiated to the specific problem and a solution (= an implementation) is calculated, which leads to further understanding, and input for another round of activities.
- Is this process relevant to the development of software systems?

## Motivation: Question and Approach

- The development of software systems proceeds in a cycle of activities:
  - **Requirement Analysis**: Understanding the domain problem;
  - **System Modeling**: Creating a model for the system by eliminating irrelevant details in order to identify essential properties;
  - **Detailed Design**: Providing detailed specification of the system;
  - **Implementation**: Final System;
  - **Verification and Testing**: To guarantee the correctness of the final implementation w.r.t. the specification.



## Motivation: Question and Approach

- The development of software systems proceeds in a cycle of activities:
  - **Requirement Analysis**: Understanding the domain problem;
  - **System Modeling**: Creating a model for the system by eliminating irrelevant details in order to identify essential properties;
  - **Detailed Design**: Providing detailed specification of the system;
  - **Implementation**: Final System;
  - **Verification and Testing**: To guarantee the correctness of the final implementation w.r.t. the specification.

## Motivation: Question and Approach

- The development of software systems proceeds in a cycle of activities:
  - **Requirement Analysis**: Understanding the domain problem;
  - **System Modeling**: Creating a model for the system by eliminating irrelevant details in order to identify essential properties;
  - **Detailed Design**: Providing detailed specification of the system;
  - **Implementation**: Final System;
  - **Verification and Testing**: To guarantee the correctness of the final implementation w.r.t. the specification.

## Motivation: Question and Approach

- The development of software systems proceeds in a cycle of activities:
  - **Requirement Analysis**: Understanding the domain problem;
  - **System Modeling**: Creating a model for the system by eliminating irrelevant details in order to identify essential properties;
  - **Detailed Design**: Providing detailed specification of the system;
  - **Implementation**: Final System;
  - **Verification and Testing**: To guarantee the correctness of the final implementation w.r.t. the specification.

## Motivation: Question and Approach

- The development of software systems proceeds in a cycle of activities:
  - **Requirement Analysis**: Understanding the domain problem;
  - **System Modeling**: Creating a model for the system by eliminating irrelevant details in order to identify essential properties;
  - **Detailed Design**: Providing detailed specification of the system;
  - **Implementation**: Final System;
  - **Verification and Testing**: To guarantee the correctness of the final implementation w.r.t. the specification.

## Motivation: Question and Approach

- **Component based systems:** The notion of **components** and the **compositional design** principle are well established in all other engineering disciplines, but until 1990s, were unsuccessful in the world of software systems.
- Component Software Technologies: CORBA (OMG), COM+ (MicroSoft), JavaBeans (Sun), ...
- **What is a software component?**
  - Software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system. (C. Szyperski, D. Gruntz and S.Murer 2003)
- The characteristic properties of a component are that it:
  - is a unit of independent deployment;
  - is a unit of third-party composition;
  - has no external observable state

## Motivation: Question and Approach

- **Component based systems:** The notion of **components** and the **compositional design** principle are well established in all other engineering disciplines, but until 1990s, were unsuccessful in the world of software systems.
- Component Software Technologies: CORBA (OMG), COM+ (MicroSoft), JavaBeans (Sun), ...
- **What is a software component?**
  - Software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system. (C. Szyperski, D. Gruntz and S.Murer 2003)
- The characteristic properties of a component are that it:
  - is a unit of independent deployment;
  - is a unit of third-party composition;
  - has no external observable state

## Motivation: Question and Approach

- **Component based systems:** The notion of **components** and the **compositional design** principle are well established in all other engineering disciplines, but until 1990s, were unsuccessful in the world of software systems.
- Component Software Technologies: CORBA (OMG), COM+ (MicroSoft), JavaBeans (Sun), ...
- **What is a software component?**
  - Software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system. (C. Szyperski, D. Gruntz and S.Murer 2003)
- The characteristic properties of a component are that it:
  - is a unit of independent deployment;
  - is a unit of third-party composition;
  - has no external observable state

## Motivation: Question and Approach

- **Component based systems:** The notion of **components** and the **compositional design** principle are well established in all other engineering disciplines, but until 1990s, were unsuccessful in the world of software systems.
- Component Software Technologies: CORBA (OMG), COM+ (MicroSoft), JavaBeans (Sun), ...
- **What is a software component?**
  - Software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system. (C. Szyperski, D. Gruntz and S.Murer 2003)
- The characteristic properties of a component are that it:
  - is a unit of independent deployment;
  - is a unit of third-party composition;
  - has no external observable state



## Motivation: Question and Approach

- **Component based systems:** The notion of **components** and the **compositional design** principle are well established in all other engineering disciplines, but until 1990s, were unsuccessful in the world of software systems.
- Component Software Technologies: CORBA (OMG), COM+ (MicroSoft), JavaBeans (Sun), ...
- **What is a software component?**
  - Software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system. (C. Szyperski, D. Gruntz and S.Murer 2003)
- The characteristic properties of a component are that it:
  - is a unit of independent deployment;
  - is a unit of third-party composition;
  - has no external observable state

## Motivation: Question and Approach

- Could we apply the general approach to the development of component-based systems?
- **Key:** resort to the *algebra* vs. *coalgebra duality* as a mathematical explanation of the intuitive symmetry between data and behavioral structures.
  - **Algebra:** abstract description of data structures. The emphasis is on **construction**.
  - **Coalgebra:** abstract description of systems' behaviors. The emphasis is on **observation**.
- A mathematical model for components and their composition.
- Applying the model to component-based complex system modeling and design.

## Motivation: Question and Approach

- Could we apply the general approach to the development of component-based systems?
- **Key:** resort to the *algebra* vs. *coalgebra duality* as a mathematical explanation of the intuitive symmetry between data and behavioral structures.
  - **Algebra:** abstract description of data structures. The emphasis is on **construction**.
  - **Coalgebra:** abstract description of systems' behaviors. The emphasis is on **observation**.
- A mathematical model for components and their composition.
- Applying the model to component-based complex system modeling and design.

## Motivation: Question and Approach

- Could we apply the general approach to the development of component-based systems?
- **Key:** resort to the *algebra* vs. *coalgebra duality* as a mathematical explanation of the intuitive symmetry between data and behavioral structures.
  - **Algebra:** abstract description of data structures. The emphasis is on **construction**.
  - **Coalgebra:** abstract description of systems' behaviors. The emphasis is on **observation**.
- A mathematical model for components and their composition.
- Applying the model to component-based complex system modeling and design.

## Motivation: Question and Approach

- Could we apply the general approach to the development of component-based systems?
- **Key:** resort to the *algebra* vs. *coalgebra duality* as a mathematical explanation of the intuitive symmetry between data and behavioral structures.
  - **Algebra:** abstract description of data structures. The emphasis is on **construction**.
  - **Coalgebra:** abstract description of systems' behaviors. The emphasis is on **observation**.
- A mathematical model for components and their composition.
- Applying the model to component-based complex system modeling and design.

## The Basic Duality

- Algebra: abstract description of data structures.

$$[\text{nil}, \text{cons}] : \mathbf{1} + A \times A^* \rightarrow A^*$$

- The emphasis is on **construction**;
- In general:

a tool box:



an assembly process:



artifact  $\xrightarrow{p}$  artifact

## The Basic Duality

- Coalgebra: abstract description of systems' behaviors.

$$\langle \text{head}, \text{tail} \rangle : A^\infty \longrightarrow \mathbf{1} + A \times A^\infty$$

- The emphasis is on **observation**;
- In general:

lens:

an observational structure:



universe



universe

## Functional Components

- Question: What is the appropriate model for a component?

$$f : I \longrightarrow O$$

- The behavior of a function is captured by the output it produces, which is completely determined by the supplied input.
- **Reality is not so simple!**



## Functional Components

- Question: What is the appropriate model for a component?

$$f : I \longrightarrow O$$

- The behavior of a function is captured by the output it produces, which is completely determined by the supplied input.
- **Reality is not so simple!**

## Functional Components

- Question: What is the appropriate model for a component?

$$f : I \longrightarrow O$$

- The behavior of a function is captured by the output it produces, which is completely determined by the supplied input.
- **Reality is not so simple!**

## Components as Coalgebras

- Is there any other possibilities?
- One may know how to produce output from input but not in all cases:

$$f : I \longrightarrow O + 1$$

- One may be uncertain of the outcome of , in the sense that the evolution of the system being observed may be non-deterministic:

$$f : I \longrightarrow \mathcal{P}O$$

- One may recognize that there is some environmental (or context) information. (For example, it might be the case that the computation will modify the environment, thus influencing latter computation:

$$f : I \longrightarrow (O \times U)^U$$

## Components as Coalgebras

- Is there any other possibilities?
- One may know how to produce output from input but not in all cases:

$$f : I \longrightarrow O + 1$$

- One may be uncertain of the outcome of , in the sense that the evolution of the system being observed may be non-deterministic:

$$f : I \longrightarrow \mathcal{P}O$$

- One may recognize that there is some environmental (or context) information. (For example, it might be the case that the computation will modify the environment, thus influencing latter computation:

$$f : I \longrightarrow (O \times U)^U$$

## Components as Coalgebras

- Is there any other possibilities?
- One may know how to produce output from input but not in all cases:

$$f : I \longrightarrow O + 1$$

- One may be uncertain of the outcome of , in the sense that the evolution of the system being observed may be non-deterministic:

$$f : I \longrightarrow \mathcal{P}O$$

- One may recognize that there is some environmental (or context) information. (For example, it might be the case that the computation will modify the environment, thus influencing latter computation:

$$f : I \longrightarrow (O \times U)^U$$

## Components as Coalgebras

- Is there any other possibilities?
- One may know how to produce output from input but not in all cases:

$$f : I \longrightarrow O + 1$$

- One may be uncertain of the outcome of , in the sense that the evolution of the system being observed may be non-deterministic:

$$f : I \longrightarrow \mathcal{P}O$$

- One may recognize that there is some environmental (or context) information. (For example, it might be the case that the computation will modify the environment, thus influencing latter computation:

$$f : I \longrightarrow (O \times U)^U$$

## Components as Coalgebras

- A function computed within a context is often referred to as state-based, in the sense the word ‘state’ has in automata theory - the internal memory of the automata which both constraints and is constrained by the execution of actions.
- The ‘nature’ of  $f : I \longrightarrow (O \times U)^U$  as a ‘state-based function’ is made more explicit by rewriting it as

$$f : U \longrightarrow (O \times U)^I$$

- One’s focus becomes the ‘universe’ or, more pragmatically, the state space. Input and output parameters may or may not be relevant, depending on the particular kind of observation one may want to perform.

## Components as Coalgebras

- A function computed within a context is often referred to as state-based, in the sense the word ‘state’ has in automata theory - the internal memory of the automata which both constraints and is constrained by the execution of actions.
- The ‘nature’ of  $f : I \longrightarrow (O \times U)^U$  as a ‘state-based function’ is made more explicit by rewriting it as

$$f : U \longrightarrow (O \times U)^I$$

- One’s focus becomes the ‘universe’ or, more pragmatically, the state space. Input and output parameters may or may not be relevant, depending on the particular kind of observation one may want to perform.




## Components as Coalgebras

- A function computed within a context is often referred to as state-based, in the sense the word ‘state’ has in automata theory - the internal memory of the automata which both constraints and is constrained by the execution of actions.
- The ‘nature’ of  $f : I \longrightarrow (O \times U)^U$  as a ‘state-based function’ is made more explicit by rewriting it as


$$f : U \longrightarrow (O \times U)^I$$

- One’s focus becomes the ‘universe’ or, more pragmatically, the state space. Input and output parameters may or may not be relevant, depending on the particular kind of observation one may want to perform.


## Components as Coalgebras

- Informal understanding of components shows that it has:
  - internal state space that persists in time;
  - the possibility of interaction with other components during the overall computation;
  - observable through well-defined interfaces to ensure flow of data.
- Such components can be found “everywhere”, from sophisticated plant control systems, to formal automata or domestic appliances.
- To investigate components one should equip himself with an appropriate ‘lens’  which necessarily entails a particular shape for observation.

## Components as Coalgebras

- Informal understanding of components shows that it has:
  - internal state space that persists in time;
  - the possibility of interaction with other components during the overall computation;
  - observable through well-defined interfaces to ensure flow of data.
- Such components can be found “everywhere”, from sophisticated plant control systems, to formal automata or domestic appliances.
- To investigate components one should equip himself with an appropriate ‘lens’  which necessarily entails a particular shape for observation.

## Components as Coalgebras

- Informal understanding of components shows that it has:
  - internal state space that persists in time;
  - the possibility of interaction with other components during the overall computation;
  - observable through well-defined interfaces to ensure flow of data.
- Such components can be found “everywhere”, from sophisticated plant control systems, to formal automata or domestic appliances.
- To investigate components one should equip himself with an appropriate ‘lens’  which necessarily entails a particular shape for observation.

## Components as Coalgebras

- A lens  $\circ \rightsquigarrow \circ$  for components is an operation mapping a set (of states)  $U$  to a set  $\circ \rightsquigarrow \circ U$  containing the possible effects of an observable transition.

$\circ \rightsquigarrow \circ$	$\circ \rightsquigarrow \circ U$	Component Behavior
1	1	stop
$A$	$A$	outputs $a \in A$ once
$Id$	$U$	running forever
$A \times -$	$A \times U$	stream over $A$
$A \times - + 1$	$A \times U + 1$	finite or infinite list over $A$

- $\circ \rightsquigarrow \circ$  provides an appropriate notion of interface for components.

## Components as Coalgebras

- A lens  $\circ \rightsquigarrow \circ$  for components is an operation mapping a set (of states)  $U$  to a set  $\circ \rightsquigarrow \circ U$  containing the possible effects of an observable transition.

$\circ \rightsquigarrow \circ$	$\circ \rightsquigarrow \circ U$	Component Behavior
1	1	stop
$A$	$A$	outputs $a \in A$ once
$Id$	$U$	running forever
$A \times -$	$A \times U$	stream over $A$
$A \times - + 1$	$A \times U + 1$	finite or infinite list over $A$

- $\circ \rightsquigarrow \circ$  provides an appropriate notion of interface for components.

## Components as Coalgebras

- The behavior of the sort of computational structures is determined not only by an external input stimuli, but also by some internal ‘memory’ to which there is, in general, no direct access. Such systems can always be represented by functions typed as

$$p : U \longrightarrow \text{○} \text{---} \text{○} U$$

- Interpretation:
  - $U$ : the set of states;
  - $p$ : the component’s dynamics, describing the observable effects of an elementary step (transition) in the evolution of the component;
  - $\text{○} \text{---} \text{○}$ : the shape for the observation structure (*interface*);
  - $\text{○} \text{---} \text{○} U$ : the set of all possible outcomes of taking one step transition.

## Components as Coalgebras

- Mealy machines (on and ) were introduced as components observed through the interface (lens)  $\bigcirc \curvearrowright \bigcirc = (\mathcal{O} \times -)^!$ .
- $\bigcirc \curvearrowright \bigcirc$  can be regarded as the type of a mapping which decomposes the 'observable universe'  $U$  into an 'observation context'  $(\mathcal{O} \times U)^!$ .
- Different interfaces give rise to different classes of components.












## Components as Coalgebras

- Mealy machines (on and ) were introduced as components observed through the interface (lens)  $\text{O} \curvearrowright \text{O} = (\text{O} \times -)^!$ .
- $\text{O} \curvearrowright \text{O}$  can be regarded as the type of a mapping which decomposes the 'observable universe'  $U$  into an 'observation context'  $(\text{O} \times U)^!$ .
- Different interfaces give rise to different classes of components.

## Components as Coalgebras

- Mealy machines (on and ) were introduced as components observed through the interface (lens)  $\text{O} \curvearrowright \text{O} = (\text{O} \times -)^!$ .
- $\text{O} \curvearrowright \text{O}$  can be regarded as the type of a mapping which decomposes the 'observable universe'  $U$  into an 'observation context'  $(\text{O} \times U)^!$ .
- Different interfaces give rise to different classes of components.

## Components as Coalgebras

- Interface shapes for observations:
  - 'opaque':   $U = \mathbf{1}$
  - black and white:   $U = \mathbf{2}$
  - colouring:   $U = \mathcal{O}$
  - multi-attribute:   $U = \prod_{k \in K} \mathcal{O}_k$
- Interface shapes for actions:
  - partiality:   $U = U + \mathbf{1}$
  - visible attributes (outputs):   $U = \mathcal{O} \times U$
  - external stimulus triggered evolution (inputs):   $U = U'$
  - non-determinism:   $U = \mathcal{P}U$
  - probability:   $U = \mathcal{D}U$ , where for a set  $X$ ,  $\mathcal{D}X = \{\xi : X \rightarrow [0, 1] \mid \sum_{x \in X} \xi(x) \leq 1\}$ , and  $\xi$  is called a probability sub-distribution over  $X$ .
- The elementary 'lens' can be glued with set-theoretic constructions.

## Components as Coalgebras

- Interface shapes for observations:
  - 'opaque':  $\bigcirc \rightsquigarrow \bigcirc \ U = \mathbf{1}$
  - black and white:  $\bigcirc \rightsquigarrow \bigcirc \ U = \mathbf{2}$
  - colouring:  $\bigcirc \rightsquigarrow \bigcirc \ U = \mathcal{O}$
  - multi-attribute:  $\bigcirc \rightsquigarrow \bigcirc \ U = \prod_{k \in K} \mathcal{O}_k$
- Interface shapes for actions:
  - partiality:  $\bigcirc \rightsquigarrow \bigcirc \ U = U + \mathbf{1}$
  - visible attributes (outputs):  $\bigcirc \rightsquigarrow \bigcirc \ U = \mathcal{O} \times U$
  - external stimulus triggered evolution (inputs):  $\bigcirc \rightsquigarrow \bigcirc \ U = U'$
  - non-determinism:  $\bigcirc \rightsquigarrow \bigcirc \ U = \mathcal{P}U$
  - probability:  $\bigcirc \rightsquigarrow \bigcirc \ U = \mathcal{D}U$ , where for a set  $X$ ,  $\mathcal{D}X = \{\xi : X \rightarrow [0, 1] \mid \sum_{x \in X} \xi(x) \leq 1\}$ , and  $\xi$  is called a probability sub-distribution over  $X$ .
- The elementary 'lens' can be glued with set-theoretic constructions.

## Components as Coalgebras


- Interface shapes for observations:
  - 'opaque':  $\bigcirc \sim \bigcirc \quad U = \mathbf{1}$
  - black and white:  $\bigcirc \sim \bigcirc \quad U = \mathbf{2}$
  - colouring:  $\bigcirc \sim \bigcirc \quad U = \mathcal{O}$
  - multi-attribute:  $\bigcirc \sim \bigcirc \quad U = \prod_{k \in K} \mathcal{O}_k$
- Interface shapes for actions:
  - partiality:  $\bigcirc \sim \bigcirc \quad U = U + \mathbf{1}$
  - visible attributes (outputs):  $\bigcirc \sim \bigcirc \quad U = \mathcal{O} \times U$
  - external stimulus triggered evolution (inputs):  $\bigcirc \sim \bigcirc \quad U = U'$
  - non-determinism:  $\bigcirc \sim \bigcirc \quad U = \mathcal{P}U$
  - probability:  $\bigcirc \sim \bigcirc \quad U = \mathcal{D}U$ , where for a set  $X$ ,  $\mathcal{D}X = \{\xi : X \rightarrow [0, 1] \mid \sum_{x \in X} \xi(x) \leq 1\}$ , and  $\xi$  is called a probability sub-distribution over  $X$ .
- The elementary 'lens' can be glued with set-theoretic constructions.

## Components as Coalgebras


- For a universe  $U$  and observation structure  $p : U \rightarrow \text{O} \sim \text{O} U$ , the pair  $\langle U, p \rangle$  is called a  $\text{O} \sim \text{O}$ -coalgebra.
- A morphism between coalgebras is a function between their carriers which preserves the dynamics, i.e.

$$\begin{array}{ccc}
 U & \xrightarrow{p} & \text{O} \sim \text{O} U \\
 h \downarrow & & \downarrow \text{O} \sim \text{O} h \\
 V & \xrightarrow{q} & \text{O} \sim \text{O} V
 \end{array}$$

## Functors


-  should be applicable not only to *sets*, but also to *functions*.
- The idea of an uniform transformation of both sets and functions is captured by the notion of a **functor**.
- A functor is a function over our working universe which preserves *identities* and *composition*, i.e., the graph and monoidal structure:
  - For each function  $f : A \rightarrow B$ ,  $Tf : TA \rightarrow TB$ ;
  - $Tid_X = id_{TX}$ ;
  - $T(f \circ g) = Tf \circ Tg$ .
- The functor should capture both a signature of actions and observers, as well as a particular behavior model.

## Functors


-  should be applicable not only to *sets*, but also to *functions*.
- The idea of a uniform transformation of both sets and functions is captured by the notion of a **functor**.
- A functor is a function over our working universe which preserves *identities* and *composition*, i.e., the graph and monoidal structure:
  - For each function  $f : A \rightarrow B$ ,  $Tf : TA \rightarrow TB$ ;
  - $Tid_X = id_{TX}$ ;
  - $T(f \circ g) = Tf \circ Tg$ .
- The functor should capture both a signature of actions and observers, as well as a particular behavior model.



## Functors

-  should be applicable not only to *sets*, but also to *functions*.
- The idea of a uniform transformation of both sets and functions is captured by the notion of a **functor**.
- A functor is a function over our working universe which preserves *identities* and *composition*, i.e., the graph and monoidal structure:
  - For each function  $f : A \rightarrow B$ ,  $Tf : TA \rightarrow TB$ ;
  - $Tid_X = id_{TX}$ ;
  - $T(f \circ g) = Tf \circ Tg$ .
- The functor should capture both a signature of actions and observers, as well as a particular behavior model.

## Functors

-  should be applicable not only to *sets*, but also to *functions*.
- The idea of an uniform transformation of both sets and functions is captured by the notion of a **functor**.
- A functor is a function over our working universe which preserves *identities* and *composition*, i.e., the graph and monoidal structure:
  - For each function  $f : A \rightarrow B$ ,  $Tf : TA \rightarrow TB$ ;
  - $Tid_X = id_{TX}$ ;
  - $T(f \circ g) = Tf \circ Tg$ .
- The functor should capture both a signature of actions and observers, as well as a particular behavior model.

## Components as Coalgebras

- These aspects are orthogonal and should be dealt separately.
- Therefore we consider the interface functor

$$T_{I,O}^B = B(\text{Id} \times O)^I$$

where  $I$  and  $O$  are sets acting as component input and output interfaces.

- For a component  $p$  with such an interface, the transition structure of the corresponding coalgebra can be represented as

$$\overline{\alpha}_p : U_p \longrightarrow B(U_p \times O)^I$$

where  $\alpha_p : U_p \times I \rightarrow B(U_p \times O)$  represents the state transitions.

- A component  $p$  for interface  $T_{I,O}^B$  can be represented as a seeded coalgebra  $p = \langle U_p, \overline{\alpha}_p : U_p \longrightarrow T_{I,O}^B U_p, u_0 \rangle$ .

## Components as Coalgebras

- These aspects are orthogonal and should be dealt separately.
- Therefore we consider the interface functor

$$T_{I,O}^B = B(\text{Id} \times O)'$$

where  $I$  and  $O$  are sets acting as component input and output interfaces.

- For a component  $p$  with such an interface, the transition structure of the corresponding coalgebra can be represented as

$$\bar{\alpha}_p : U_p \longrightarrow B(U_p \times O)'$$

where  $\alpha_p : U_p \times I \rightarrow B(U_p \times O)$  represents the state transitions.

- A component  $p$  for interface  $T_{I,O}^B$  can be represented as a seeded coalgebra  $p = \langle U_p, \bar{\alpha}_p : U_p \longrightarrow T_{I,O}^B U_p, u_0 \rangle$ .

## Components as Coalgebras

- These aspects are orthogonal and should be dealt separately.
- Therefore we consider the interface functor

$$T_{I,O}^B = B(\text{Id} \times O)^I$$

where  $I$  and  $O$  are sets acting as component input and output interfaces.

- For a component  $p$  with such an interface, the transition structure of the corresponding coalgebra can be represented as

$$\overline{\alpha}_p : U_p \longrightarrow B(U_p \times O)^I$$

where  $\alpha_p : U_p \times I \rightarrow B(U_p \times O)$  represents the state transitions.

- A component  $p$  for interface  $T_{I,O}^B$  can be represented as a seeded coalgebra  $p = \langle U_p, \overline{\alpha}_p : U_p \longrightarrow T_{I,O}^B U_p, u_0 \rangle$ .

## Components as Coalgebras

- These aspects are orthogonal and should be dealt separately.
- Therefore we consider the interface functor

$$T_{I,O}^B = B(\text{Id} \times O)^I$$

where  $I$  and  $O$  are sets acting as component input and output interfaces.

- For a component  $p$  with such an interface, the transition structure of the corresponding coalgebra can be represented as

$$\overline{\alpha}_p : U_p \longrightarrow B(U_p \times O)^I$$

where  $\alpha_p : U_p \times I \rightarrow B(U_p \times O)$  represents the state transitions.

- A component  $p$  for interface  $T_{I,O}^B$  can be represented as a seeded coalgebra  $p = \langle U_p, \overline{\alpha}_p : U_p \longrightarrow T_{I,O}^B U_p, u_0 \rangle$ .

## Component Behavior

- Successive observations of a component  $p$  reveal its allowed behavioral patterns.
- For each state value  $u \in U_p$ , the behavior of  $p$  at  $u$  (more precisely, from  $u$  onwards) organize itself into a tree-like structure, because it depends on the sequences of input items processed.
- Such trees, whose arcs are labelled with  $I$  values and nodes with  $O$  values, can be represented by functions from non empty sequence of  $I$  to B-structures of output items.
- In other words, the space of behaviors of a component with input  $I$  and output  $O$  is the set  $(BO)^{I^+}$ , which is in fact the carrier  $\nu_T$  of the final T-coalgebra  $(\nu_T, \omega_T : \nu_T \rightarrow T\nu_T)$ .

## Component Behavior

- Successive observations of a component  $p$  reveal its allowed behavioral patterns.
- For each state value  $u \in U_p$ , the behavior of  $p$  at  $u$  (more precisely, from  $u$  onwards) organize itself into a tree-like structure, because it depends on the sequences of input items processed.
- Such trees, whose arcs are labelled with  $I$  values and nodes with  $O$  values, can be represented by functions from non empty sequence of  $I$  to  $B$ -structures of output items.
- In other words, the space of behaviors of a component with input  $I$  and output  $O$  is the set  $(BO)^{I^+}$ , which is in fact the carrier  $\nu_T$  of the final T-coalgebra  $(\nu_T, \omega_T : \nu_T \rightarrow T\nu_T)$ .



## Component Behavior

- Successive observations of a component  $p$  reveal its allowed behavioral patterns.
- For each state value  $u \in U_p$ , the behavior of  $p$  at  $u$  (more precisely, from  $u$  onwards) organize itself into a tree-like structure, because it depends on the sequences of input items processed.
- Such trees, whose arcs are labelled with  $I$  values and nodes with  $O$  values, can be represented by functions from non empty sequence of  $I$  to  $B$ -structures of output items.
- In other words, the space of behaviors of a component with input  $I$  and output  $O$  is the set  $(BO)^{I^+}$ , which is in fact the carrier  $\nu_T$  of the final T-coalgebra  $(\nu_T, \omega_T : \nu_T \rightarrow T\nu_T)$ .

## Component Behavior

- Successive observations of a component  $p$  reveal its allowed behavioral patterns.
- For each state value  $u \in U_p$ , the behavior of  $p$  at  $u$  (more precisely, from  $u$  onwards) organize itself into a tree-like structure, because it depends on the sequences of input items processed.
- Such trees, whose arcs are labelled with  $I$  values and nodes with  $O$  values, can be represented by functions from non empty sequence of  $I$  to B-structures of output items.
- In other words, the space of behaviors of a component with input  $I$  and output  $O$  is the set  $(BO)^{I^+}$ , which is in fact the carrier  $\nu_T$  of the final T-coalgebra  $(\nu_T, \omega_T : \nu_T \rightarrow T\nu_T)$ .

## Component Behavior

- By finality, from any other T-coalgebra  $p$ , there is a unique morphism  $\llbracket p \rrbracket$  making the following diagram to commute:

$$\begin{array}{ccc}
 \nu_T & \xrightarrow{\omega_T} & B(\nu_T \times O)^I \\
 \llbracket p \rrbracket \uparrow & & \uparrow B(\llbracket p \rrbracket \times O)^I \\
 U_p & \xrightarrow{\bar{\alpha}_p} & B(U_p \times O)^I
 \end{array}$$

- Applying morphism  $\llbracket p \rrbracket$  to a state value  $u \in U_p$  gives the observable behavior of a sequence of  $p$  transitions starting at  $u$ .

## Bisimulation

Bisimulation for two T-coalgebras  $(U, \alpha)$  and  $(V, \beta)$  is a relation  $R \subseteq U \times V$  such that there is a T-coalgebra  $(R, \gamma)$  satisfying

$$T(\pi_1) \circ \gamma = \alpha \circ \pi_1$$

$$T(\pi_2) \circ \gamma = \beta \circ \pi_2$$

The following diagram is the corresponding instantiation for the functor T underlying our model of components.

$$\begin{array}{ccccc}
 U & \xleftarrow{\pi_1} & R & \xrightarrow{\pi_2} & V \\
 \alpha \downarrow & & \downarrow \gamma & & \downarrow \beta \\
 B(U \times O)^I & \xleftarrow{B(\pi_1 \times O)^I} & B(R \times O)^I & \xrightarrow{B(\pi_2 \times O)^I} & B(V \times O)^I
 \end{array}$$

## Bisimulation

- Provides a ‘relational’ view of coalgebra morphisms, as the graph of a T-coalgebra morphism is a T-bisimulation.
- Has a large theory (e.g. closed under converse and composition).
- Entails a *local* proof theory for observational equivalence: the *coinductive* proof principle being widely used in the coalgebra literature is the explicit construction of a bisimulation containing the pair of states one want to prove equivalent.
- Parametric on system’s interface T.

## Components as Coalgebras

- For every functor  $T$ , the  $T$ -coalgebras together with the homomorphisms between them form a category.
- A new category **Co** is defined as a “total category” which contains the seeded coalgebras for different functors.
- For two components  $p = \langle U_p, \overline{\alpha}_p : U_p \longrightarrow T_{I_p, O_p}^{B_p} U_p, u_p \rangle$  and  $q = \langle U_q, \overline{\alpha}_q : U_q \longrightarrow T_{I_q, O_q}^{B_q} U_q, u_q \rangle$  in **Co**, the morphism between them is  $\langle k, \tau_{f,g}^\psi \rangle$  where  $f : I_p \longrightarrow I_q$ ,  $g : O_p \longrightarrow O_q$ ,  $\psi : B_p \Rightarrow B_q$ ,  $\tau_{f,g}^\psi = \psi(\text{id} \times g)^f : T_{I_p, O_p}^{B_p} \Rightarrow T_{I_q, O_q}^{B_q}$  is a natural transformation, and  $k : U_p \longrightarrow U_q$  is the coalgebra morphism under natural transformation  $\tau_{f,g}^\psi$ , such that  $k u_p = u_q$  and the following diagram commutes:

## Components as Coalgebras

$$\begin{array}{ccc}
 U_p & \xrightarrow{k} & U_q \\
 \rho \downarrow & & \downarrow q \\
 T_{I_p, O_p}^{B_p} U_p & \xrightarrow{(\tau_{f,g}^\psi) U_p} & T_{I_q, O_q}^{B_q} U_p \xrightarrow{T_{I_q, O_q}^{B_q} k} T_{I_q, O_q}^{B_q} U_q
 \end{array}$$

## Bisimulation of Components

### Definition

Two T-components  $p = \langle U_p, \bar{\alpha}_p : U_p \longrightarrow T U_p, u_p \rangle$  and  $q = \langle U_q, \bar{\alpha}_q : U_q \longrightarrow T U_q, u_q \rangle$  are bisimilar iff there is a T-bisimulation containing the pair  $\langle u_p, u_q \rangle$ .

### Definition

Let  $p = \langle U_p, \bar{\alpha}_p : U_p \longrightarrow T_{I_p, O_p}^{B_p} U_p, u_p \rangle$  and  $q = \langle U_q, \bar{\alpha}_q : U_q \longrightarrow T_{I_q, O_q}^{B_q} U_q, u_q \rangle$  be two components, and  $\tau_{f,g}^\psi = \psi(\text{id} \times g)^f$  is a natural transformation, where  $f : I_p \longrightarrow I_q$ ,  $g : O_p \longrightarrow O_q$ ,  $\psi : B_p \Rightarrow B_q$ , if  $p' = \langle U_p, (\tau_{f,g}^\psi)_{U_p} \cdot \bar{\alpha}_p, u_p \rangle$  and  $q$  are bisimilar, then  $p$  and  $q$  are bisimilar under the natural transformation  $\tau_{f,g}^\psi$ , denoted by  $p \approx_{\tau_{f,g}^\psi} q$ .



## Bisimulation of Components

### Definition

Two T-components  $p = \langle U_p, \bar{\alpha}_p : U_p \longrightarrow T U_p, u_p \rangle$  and  $q = \langle U_q, \bar{\alpha}_q : U_q \longrightarrow T U_q, u_q \rangle$  are bisimilar iff there is a T-bisimulation containing the pair  $\langle u_p, u_q \rangle$ .

### Definition

Let  $p = \langle U_p, \bar{\alpha}_p : U_p \longrightarrow T_{I_p, O_p}^{B_p} U_p, u_p \rangle$  and  $q = \langle U_q, \bar{\alpha}_q : U_q \longrightarrow T_{I_q, O_q}^{B_q} U_q, u_q \rangle$  be two components, and  $\tau_{f,g}^\psi = \psi(\text{id} \times g)^f$  is a natural transformation, where  $f : I_p \longrightarrow I_q$ ,  $g : O_p \longrightarrow O_q$ ,  $\psi : B_p \Rightarrow B_q$ , if  $p' = \langle U_p, (\tau_{f,g}^\psi)_{U_p} \cdot \bar{\alpha}_p, u_p \rangle$  and  $q$  are bisimilar, then  $p$  and  $q$  are bisimilar under the natural transformation  $\tau_{f,g}^\psi$ , denoted by  $p \approx_{\tau_{f,g}^\psi} q$ .

## The Calculus of Components

### Definition

For two components  $p = \langle U_p, \bar{\alpha}_p : U_p \longrightarrow T_{I,K}^{B_p} U_p, u_p \rangle$  and  $q = \langle U_q, \bar{\alpha}_q : U_q \longrightarrow T_{K,O}^{B_q} U_q, u_q \rangle$ , the sequential composition of  $p$  and  $q$  is defined as

$$p ; q = \langle U, \bar{\alpha}_{p;q} : U \longrightarrow T_{I,O}^B U, u_{p;q} \rangle$$

where  $U = U_p \times U_q$ ,  $B = B_p B_q$ ,  $u_{p;q} = \langle u_p, u_q \rangle$  and

$$\begin{aligned} \alpha_{p;q} &= U_p \times U_q \times I \xrightarrow{xr} U_p \times I \times U_q \xrightarrow{\alpha_p \times \text{id}} B_p(U_p \times K) \times U_q \\ &\xrightarrow{\tau_r} B_p(U_p \times K \times U_q) \xrightarrow{B_p(a \cdot xr)} B_p(U_p \times (U_q \times K)) \\ &\xrightarrow{B_p(\text{id} \times \alpha_q)} B_p(U_p \times B_q(U_q \times O)) \xrightarrow{B_p \tau_l} B_p B_q(U_p \times (U_q \times O)) \\ &\xrightarrow{B_p B_q a^\circ} B_p B_q(U_p \times U_q \times O) = B(U \times O) \end{aligned}$$

## The Calculus of Components

### Definition

For two components  $p = \langle U_p, \bar{\alpha}_p : U_p \longrightarrow T_{l_p, O_p}^{B_p} U_p, u_p \rangle$  and  $q = \langle U_q, \bar{\alpha}_q : U_q \longrightarrow T_{l_q, O_q}^{B_q} U_q, u_q \rangle$ , their external choice is defined as  $p \boxplus q = \langle U, \bar{\alpha}_{p \boxplus q} : U \longrightarrow T_{l, O}^{B_{p \boxplus q}} U, u_{p \boxplus q} \rangle$ , where  $U = U_p \times U_q$ ,  $l = l_p + l_q$ ,  $O = O_p + O_q$ ,  $B_{p \boxplus q} = B_p + B_q$ ,  $u_{p \boxplus q} = \langle u_p, u_q \rangle$ , and

$$\begin{aligned}
 \alpha_{p \boxplus q} &= U \times (l_p + l_q) \xrightarrow{dr} (U \times l_p) + (U \times l_q) \\
 &\xrightarrow{xr+a} (U_p \times l_p) \times U_q + U_p \times (U_q \times l_q) \\
 &\xrightarrow{\alpha_p \times \text{id} + \text{id} \times \alpha_q} B_p (U_p \times O_p) \times U_q + U_p \times B_q (U_q \times O_q) \\
 &\xrightarrow{\tau_r + \tau_l} B_p ((U_p \times O_p) \times U_q) + B_q (U_p \times (U_q \times O_q)) \\
 &\xrightarrow{B_p \text{ xr} + B_q \text{ a}^\circ} B_p (U \times O_p) + B_q (U \times O_q) \\
 &\xrightarrow{[B_p \iota_1, B_q \iota_2]} B_{p \boxplus q} (U \times O)
 \end{aligned}$$

## The Calculus of Components

### Definition

For two components  $p = \langle U_p, \bar{\alpha}_p : U_p \longrightarrow T_{I_p, O_p}^{B_p} U_p, u_p \rangle$  and  $q = \langle U_q, \bar{\alpha}_q : U_q \longrightarrow T_{I_q, O_q}^{B_q} U_q, u_q \rangle$ , their parallel composition  $p \boxtimes q = \langle U, \bar{\alpha}_{p \boxtimes q} : U \longrightarrow T_{I, O}^{B_{p \boxtimes q}} U, u_{p \boxtimes q} \rangle$  where  $U = U_p \times U_q$ ,  $I = I_p \times I_q$ ,  $O = O_p \times O_q$ ,  $B_{p \boxtimes q} = B_p B_q$ ,  $u_{p \boxtimes q} = \langle u_p, u_q \rangle$  and

$$\begin{aligned}
 \alpha_{p \boxtimes q} &= U_p \times U_q \times (I_p \times I_q) \xrightarrow{m} (U_p \times I_p) \times (U_q \times I_q) \\
 &\xrightarrow{\alpha_p \times \alpha_q} B_p (U_p \times O_p) \times B_q (U_q \times O_q) \\
 &\xrightarrow{\tau_r} B_p (U_p \times O_p \times B_q (U_q \times O_q)) \\
 &\xrightarrow{B_p \tau_l} B_p B_q (U_p \times O_p \times (U_q \times O_q)) \\
 &\xrightarrow{B_p B_q m} B_p B_q (U_p \times U_q \times (O_p \times O_q)) = B_{p \boxtimes q} (U \times O)
 \end{aligned}$$

## The Calculus of Components

For a component  $p = \langle U, \bar{\alpha}_p : U \longrightarrow T_{I,O}^B U, u_0 \rangle$  and functions  $f : I' \longrightarrow I, g : O \longrightarrow O'$ , the wrapping

$$p[f, g] = \langle U, \bar{\alpha}_{p[f,g]} : U \longrightarrow T_{I',O'}^B U, u_0 \rangle$$

where

$$\begin{array}{c} \alpha_{p[f,g]} = U \times I' \xrightarrow{\text{id} \times f} U \times I \xrightarrow{\alpha_p} \\ B(U \times O) \xrightarrow{B(\text{id} \times g)} B(U \times O') \end{array}$$

## The Calculus of Components

$$(p; q); r \approx p; (q; r) \quad (1)$$

$$(p \boxtimes q) \boxtimes r \approx (p \boxtimes (q \boxtimes r))[a, a^\circ] \quad (2)$$

$$q \boxtimes p \approx_{\tau_{s,s}^\gamma} p \boxtimes q \quad (3)$$

$$\text{idle} \boxtimes p \approx_{\tau_{\text{id},\text{id}}^\gamma} p[r, r^\circ] \quad (4)$$

$$\text{nil} \boxtimes p \approx_{\tau_{\text{id},\text{id}}^\gamma} \text{nil}[z!, z!^\circ] \quad (5)$$

$$(p \boxplus q) \boxplus r \approx (p \boxplus (q \boxplus r))[a_+, a^\circ] \quad (6)$$

$$p \boxplus q \approx_{\tau_{s_+,s}^{s_+}} q \boxplus p \quad (7)$$

$$\text{nil} \boxplus p \approx_{\tau_{\text{id},\pi_2}^\gamma} p[r_+, r^\circ] \quad (8)$$

$$p \boxplus \text{nil} \approx_{\tau_{\text{id},\pi_1}^\gamma} p[l_+, l^\circ] \quad (9)$$

## The Calculus of Components

$$\begin{array}{ccc}
 ((U_p \times U_q) \times U_r) \times I & \xrightarrow{a \times \text{id}} & (U_p \times (U_q \times U_r)) \times I \\
 \phi_p \downarrow & & \downarrow \psi_p \\
 B_p (U_p \times (U_q \times K)) \times U_r & \xrightarrow{B_p a \cdot \tau_r} & B_p (U_p \times ((U_q \times U_r) \times K)) \\
 \phi_q \downarrow & & \downarrow \psi_q \\
 B_p B_q ((U_p \times U_q) \times (U_r \times L)) & \xleftarrow{B_p B_q a^\circ \cdot B_p \tau_l} & B_p (U_p \times B_q (U_q \times (U_r \times L))) \\
 \phi_r \downarrow & & \downarrow \psi_r \\
 B_p B_q B_r (((U_p \times U_q) \times U_r) \times O) & \xrightarrow{B_p B_q B_r (a \times \text{id})} & B_p B_q B_r ((U_p \times (U_q \times U_r)) \times O)
 \end{array}$$

## What does refinement mean?

- Refinement: A *transformation* of an “abstract” into a more “concrete” design, entailing a notion of *substitution*.
- Data refinement, being traced back to Hoare’s work, retrieve function from the concrete into the abstract model is defined.
- Object-orientation, substitution is expressed in terms of behavior typing.
- Process algebra, reduction of nondeterminism.
- **Semantic characterization of refinement for state-based components.**



## What does refinement mean?

- Refinement: A *transformation* of an “abstract” into a more “concrete” design, entailing a notion of *substitution*.
- Data refinement, being traced back to Hoare’s work, retrieve function from the concrete into the abstract model is defined.
- Object-orientation, substitution is expressed in terms of behavior typing.
- Process algebra, reduction of nondeterminism.
- **Semantic characterization of refinement for state-based components.**

## What does refinement mean?

- Refinement: A *transformation* of an “abstract” into a more “concrete” design, entailing a notion of *substitution*.
- Data refinement, being traced back to Hoare’s work, retrieve function from the concrete into the abstract model is defined.
- Object-orientation, substitution is expressed in terms of behavior typing.
- Process algebra, reduction of nondeterminism.
- **Semantic characterization of refinement for state-based components.**

## What does refinement mean?

- Refinement: A *transformation* of an “abstract” into a more “concrete” design, entailing a notion of *substitution*.
- Data refinement, being traced back to Hoare’s work, retrieve function from the concrete into the abstract model is defined.
- Object-orientation, substitution is expressed in terms of behavior typing.
- Process algebra, reduction of nondeterminism.
- **Semantic characterization of refinement for state-based components.**

## What does refinement mean?

- Refinement: A *transformation* of an “abstract” into a more “concrete” design, entailing a notion of *substitution*.
- Data refinement, being traced back to Hoare’s work, retrieve function from the concrete into the abstract model is defined.
- Object-orientation, substitution is expressed in terms of behavior typing.
- Process algebra, reduction of nondeterminism.
- **Semantic characterization of refinement for state-based components.**

## Refinement

- Based on the coalgebraic framework, three kinds of refinement relations can be defined for state-based systems:
- *Behavioral Refinement*: typically relates systems of the same interface, where the refinement is based on a simulation preorder between the two systems.
- *Interface Refinement*: relates systems of different interfaces, and the question is whether a system can be transformed, by suitable wiring, to replace another system with a different interface.
- *Architectural Refinement*: being used for decomposing a system with a specified behavior into a distributed system architecture, i.e., a family of systems (components) combined in parallel.

## Refinement

- Based on the coalgebraic framework, three kinds of refinement relations can be defined for state-based systems:
- *Behavioral Refinement*: typically relates systems of the same interface, where the refinement is based on a simulation preorder between the two systems.
- *Interface Refinement*: relates systems of different interfaces, and the question is whether a system can be transformed, by suitable wiring, to replace another system with a different interface.
- *Architectural Refinement*: being used for decomposing a system with a specified behavior into a distributed system architecture, i.e., a family of systems (components) combined in parallel.

## Refinement

- Based on the coalgebraic framework, three kinds of refinement relations can be defined for state-based systems:
- *Behavioral Refinement*: typically relates systems of the same interface, where the refinement is based on a simulation preorder between the two systems.
- *Interface Refinement*: relates systems of different interfaces, and the question is whether a system can be transformed, by suitable wiring, to replace another system with a different interface.
- *Architectural Refinement*: being used for decomposing a system with a specified behavior into a distributed system architecture, i.e., a family of systems (components) combined in parallel.

## Refinement

- Based on the coalgebraic framework, three kinds of refinement relations can be defined for state-based systems:
- *Behavioral Refinement*: typically relates systems of the same interface, where the refinement is based on a simulation preorder between the two systems.
- *Interface Refinement*: relates systems of different interfaces, and the question is whether a system can be transformed, by suitable wiring, to replace another system with a different interface.
- *Architectural Refinement*: being used for decomposing a system with a specified behavior into a distributed system architecture, i.e., a family of systems (components) combined in parallel.



## Behavioral Refinement

- Behavior refinement affects the internal dynamics of a system while leaving unchanged its external interface.
- $p$  *behaviorally* refines  $q$  if the behavior patterns observed for  $p$  are a structural restriction, with respect to the *behavioral model* captured by monad  $B$ , of those of  $q$ .
- Any coalgebra  $\langle U, p : U \rightarrow TU \rangle$  specifies a transition structure over  $U$ .
- For extended polynomial functors such a structure may be expressed as a relation  $\longrightarrow_p \subseteq U \times U$ , defined in terms of the *structural membership* relation  $\in_T \subseteq U \times TU$ , i.e.,

$$u \longrightarrow_p u' \quad \text{iff} \quad u' \in_T p u$$

## Behavioral Refinement

- Behavior refinement affects the internal dynamics of a system while leaving unchanged its external interface.
- $p$  *behaviorally* refines  $q$  if the behavior patterns observed for  $p$  are a structural restriction, with respect to the *behavioral model* captured by monad  $B$ , of those of  $q$ .
- Any coalgebra  $\langle U, p : U \rightarrow TU \rangle$  specifies a transition structure over  $U$ .
- For extended polynomial functors such a structure may be expressed as a relation  $\longrightarrow_p \subseteq U \times U$ , defined in terms of the *structural membership* relation  $\in_T \subseteq U \times TU$ , i.e.,

$$u \longrightarrow_p u' \quad \text{iff} \quad u' \in_T p u$$

## Behavioral Refinement

- Behavior refinement affects the internal dynamics of a system while leaving unchanged its external interface.
- $p$  *behaviorally* refines  $q$  if the behavior patterns observed for  $p$  are a structural restriction, with respect to the *behavioral model* captured by monad  $B$ , of those of  $q$ .
- Any coalgebra  $\langle U, p : U \rightarrow TU \rangle$  specifies a transition structure over  $U$ .
- For extended polynomial functors such a structure may be expressed as a relation  $\longrightarrow_p \subseteq U \times U$ , defined in terms of the *structural membership* relation  $\in_T \subseteq U \times TU$ , i.e.,

$$u \longrightarrow_p u' \quad \text{iff} \quad u' \in_T p u$$

## Behavioral Refinement

- Structural Membership Relation for extended polynomial functors:

$$x \in_{\text{Id}} y \quad \text{iff} \quad x = y$$

$$x \in_{\underline{K}} y \quad \text{iff} \quad \text{false}$$

$$x \in_{T_1 \times T_2} y \quad \text{iff} \quad x \in_{T_1} \pi_1 y \vee x \in_{T_2} \pi_2 y$$

$$x \in_{T_1 + T_2} y \quad \text{iff} \quad \begin{cases} y = \iota_1 y' \Rightarrow x \in_{T_1} y' \\ y = \iota_2 y' \Rightarrow x \in_{T_2} y' \end{cases}$$

$$x \in_{T \underline{K}} y \quad \text{iff} \quad \exists k \in K. x \in_T y k$$

$$x \in_{\mathcal{P}T} y \quad \text{iff} \quad \exists y' \in y. x \in_T y'$$

## Behavioral Refinement

- In data refinement, there is a ‘recipe’ to identify a refinement situation: look for a *retrieve function* to witness it. I.e., a morphism in the relevant category, from the ‘concrete’ to the ‘abstract’ model such that the latter can be *recovered* from the former up to a suitable notion of equivalence, though, typically, not in a unique way.
- In the coalgebraic framework, however, things do not work this way. The reason is obvious: initial states preserving coalgebra morphisms are known to entail bisimilarity. Therefore we have to look for some *weaker* notion of morphism between coalgebras.

## Behavioral Refinement

- In data refinement, there is a ‘recipe’ to identify a refinement situation: look for a *retrieve function* to witness it. I.e., a morphism in the relevant category, from the ‘concrete’ to the ‘abstract’ model such that the latter can be *recovered* from the former up to a suitable notion of equivalence, though, typically, not in a unique way.
- In the coalgebraic framework, however, things do not work this way. The reason is obvious: initial states preserving coalgebra morphisms are known to entail bisimilarity. Therefore we have to look for some *weaker* notion of morphism between coalgebras.

## Behavioral Refinement

- Coalgebra morphism from  $p$  to  $q$ :

$$\mathbf{B}(h \times \text{id}) \circ \alpha_p = \alpha_q \circ (h \times \text{id})$$

- In terms of transitions, the equation can be translated into the requirements:

$$u \xrightarrow{\langle i, o \rangle}_p u' \Rightarrow h u \xrightarrow{\langle i, o \rangle}_q h u'$$

$$h u \xrightarrow{\langle i, o \rangle}_q v' \Rightarrow \exists u' \in U. u \xrightarrow{\langle i, o \rangle}_p u' \wedge u' = h v'$$

which jointly state that, not only  $p$  dynamics is *preserved* by  $h$ , but also  $q$  dynamics is *reflected* back over the same  $h$ .

## Behavioral Refinement

- Coalgebra morphism from  $p$  to  $q$ :

$$B(h \times \text{id}) \circ \alpha_p = \alpha_q \circ (h \times \text{id})$$

- In terms of transitions, the equation can be translated into the requirements:

$$u \xrightarrow{\langle i, o \rangle}_p u' \Rightarrow h u \xrightarrow{\langle i, o \rangle}_q h u'$$

$$h u \xrightarrow{\langle i, o \rangle}_q v' \Rightarrow \exists u' \in U. u \xrightarrow{\langle i, o \rangle}_p u' \wedge u' = h v'$$

which jointly state that, not only  $p$  dynamics is *preserved* by  $h$ , but also  $q$  dynamics is *reflected* back over the same  $h$ .



## Behavioral Refinement

Given a **Set** endofunctor  $T$ , an order  $\leq$  on  $T$  is defined as a functor  $\leq$  from **Set** to **PreOrder** (concretely, mapping every set  $U$  into a collection of preorders  $\leq_{T(U)}$ ) making the following diagram to commute:

$$\begin{array}{ccc}
 & (T(U), \leq_{T(U)}) & \\
 \swarrow \leq & & \downarrow G \\
 U & \xrightarrow{T} & T(U)
 \end{array}$$

where  $G$  is the forgetful functor which forgets the preorder structure for every preordered set and gives its underlying set.

## Behavioral Refinement

- Let  $T$  be an extended polynomial functor on **Set** and consider two  $T$ -coalgebras  $c = (U, \alpha : U \rightarrow TU)$  and  $a = (V, \beta : V \rightarrow TV)$ . A *forward* morphism  $h : c \rightarrow a$  with respect to a refinement preorder  $\leq$ , is a function from  $U$  to  $V$  such that

$$Th \circ \alpha \leq \beta \circ h$$

- Dually,  $h$  is called a *backward* morphism if  $\beta \circ h \leq Th \circ \alpha$ .
- For coalgebras  $c$  and  $a$ ,  $c$  is a *behavior refinement* of  $a$ , written  $c \sqsubseteq_B a$ , if there exist coalgebras  $p$  and  $q$  such that  $c \sim p$ ,  $a \sim q$  and there exists a (initial state preserving) forward morphism from  $p$  to  $q$ .

## Behavioral Refinement

- Let  $T$  be an extended polynomial functor on **Set** and consider two  $T$ -coalgebras  $c = (U, \alpha : U \rightarrow TU)$  and  $a = (V, \beta : V \rightarrow TV)$ . A *forward* morphism  $h : c \rightarrow a$  with respect to a refinement preorder  $\leq$ , is a function from  $U$  to  $V$  such that

$$Th \circ \alpha \leq \beta \circ h$$

- Dually,  $h$  is called a *backward* morphism if  $\beta \circ h \leq Th \circ \alpha$ .
- For coalgebras  $c$  and  $a$ ,  $c$  is a *behavior refinement* of  $a$ , written  $c \sqsubseteq_B a$ , if there exist coalgebras  $p$  and  $q$  such that  $c \sim p$ ,  $a \sim q$  and there exists a (initial state preserving) forward morphism from  $p$  to  $q$ .

## Behavioral Refinement

The exact meaning of a refinement assertion  $c \sqsubseteq_B a$  depends on the refinement preorder  $\leq$  adopted. For example, we can define the preorder for extended polynomial functor  $T$  by induction as follows:

$$x \sqsubseteq_{Id} y \quad \text{iff} \quad x = y$$

$$x \sqsubseteq_K y \quad \text{iff} \quad x =_K y$$

$$x \sqsubseteq_{T_1 \times T_2} y \quad \text{iff} \quad \pi_1 x \sqsubseteq_{T_1} \pi_1 y \wedge \pi_2 x \sqsubseteq_{T_2} \pi_2 y$$

$$x \sqsubseteq_{T_1 + T_2} y \quad \text{iff} \quad \begin{cases} x = \iota_1 x' \wedge y = \iota_1 y' \Rightarrow x' \sqsubseteq_{T_1} y' \\ x = \iota_2 x' \wedge y = \iota_2 y' \Rightarrow x' \sqsubseteq_{T_2} y' \end{cases}$$

$$x \sqsubseteq_{TK} y \quad \text{iff} \quad \forall k \in K. x(k) \sqsubseteq_T y(k)$$

$$x \sqsubseteq_{\emptyset T} y \quad \text{iff} \quad \forall e \in x. \exists e' \in y. e \sqsubseteq_T e'$$

## A Coalgebraic Semantics of UML

- UML is “a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system”.
- In practice, it stands for a collection of inter-related, semi-formal design notations for software development, providing a unified notation, expressive and widely adopted.
- It lacks a rigorous and consensual semantic definition leading, therefore, to weak effective support to the design of complex systems and, often, to conflicting support tools.

## A Coalgebraic Semantics of UML

- UML is “a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system”.
- In practice, it stands for a collection of inter-related, semi-formal design notations for software development, providing a unified notation, expressive and widely adopted.
- It lacks a rigorous and consensual semantic definition leading, therefore, to weak effective support to the design of complex systems and, often, to conflicting support tools.

## A Coalgebraic Semantics of UML

- UML is “a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system”.
- In practice, it stands for a collection of inter-related, semi-formal design notations for software development, providing a unified notation, expressive and widely adopted.
- It lacks a rigorous and consensual semantic definition leading, therefore, to weak effective support to the design of complex systems and, often, to conflicting support tools.

## A Coalgebraic Semantics of UML

- We introduced a generic coalgebraic semantic framework for different models in UML, including class diagrams, use cases, statecharts and sequence diagrams, where the semantics of different kinds of models are given as coalgebras.
- Notions of bisimulation and refinement capture observational equivalence and simulation preorders, respectively, and form the basis of a whole discipline of reasoning and transforming UML designs.



## A Coalgebraic Semantics of UML

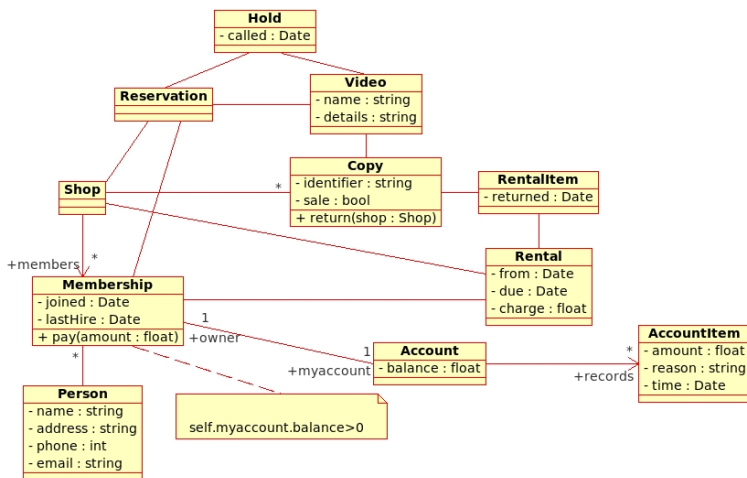
- We introduced a generic coalgebraic semantic framework for different models in UML, including class diagrams, use cases, statecharts and sequence diagrams, where the semantics of different kinds of models are given as coalgebras.
- Notions of bisimulation and refinement capture observational equivalence and simulation preorders, respectively, and form the basis of a whole discipline of reasoning and transforming UML designs.

## Class Diagrams

- In UML a *class diagram* captures the static structure of a system, as a set of classes and relationships, called *associations*, between them.
- Classes may be further annotated with *constraints*, i.e., properties that must hold for every object in the class along its lifetime.
- We concentrate here in class declarations. The aim of a class declaration is introduce a signature of attributes and methods.

## Class Diagrams

- Consider the simplified model of a video renting e-business:



## Class Diagrams

- Consider class **Membership** in the previous example. It introduces two attributes and a method over a state space, identified by variable  $U$  below, which is made observable exactly (and uniquely) by the attributes and methods it declares:

$$\text{joined} : U \longrightarrow \text{Date}$$

$$\text{lastHire} : U \longrightarrow \text{Date}$$

$$\text{pay} : U \times \mathbb{R} \longrightarrow U$$

These three declarations can be grouped in one through a *split* construction

$$\langle \text{joined}, \text{lastHire}, \overline{\text{pay}} \rangle : U \longrightarrow \text{Date} \times \text{Date} \times U^{\mathbb{R}}$$

which is a *coalgebra* for functor  $\text{T}X = \text{Date} \times \text{Date} \times X^{\mathbb{R}}$ .

## Class Diagrams

- In general, the semantics  $\llbracket c \rrbracket$  of a class  $c$  is given by a specification of a coalgebra

$$\langle \text{at}, \overline{\text{md}} \rangle : U \longrightarrow A \times (O \times U)^I$$

where  $A$  is the attribute domain, and each method accepts a parameter, of type  $I$ , and delivers both a state change and an output value, of type  $O$ . I.e., a coalgebra for functor

$$T : X \longrightarrow A \times (O \times X)^I$$

Typically,  $I$  and  $O$  are *sum* types, aggregating the input-output parameters of each declared method. On its turn,  $A$  is usually a *product* type joining all attribute outputs in a way which emphasises that each of them is available independent of the others, and therefore always able to be accessed in parallel.

## Class Diagrams

- More generally, as methods are typically implemented by *partial functions* or even by arbitrary *relations*, this definition should be generalized to

$$\langle \text{at}, \overline{\text{md}} \rangle : U \longrightarrow A \times B(O \times U)^I$$

where  $B$  is a strong monad capturing some sort of behavioral effect.

## Class Diagrams

- A UML class diagram introduces a number of class specifications which types the object population of any corresponding model implementation. Typically, different ways of putting classes together in a class diagram correspond to different operators between the T-coalgebras.
- In particular, one may consider a form of *parallel* aggregation, denoted by  $\boxtimes$ , in which methods in both classes can be called simultaneously (as they always act upon disjoint state spaces), and a form of *interleaving*, denoted by  $\boxplus$ , which offers a choice of which class to call.
- Note that in both cases, attributes are always available to be observed, and therefore are composed in a multiplicative context. Initial conditions are joined by logical conjunction.

## Class Diagrams

- Constraints are typically attached to class specifications and their semantic effect is to constraint what coalgebras count as valid implementation for the class. Such is the case, for example, of constraint

$\text{balance} > 0$

attached to class **Membership** in our example.



## Class Diagrams

- Associations can also be interpreted as constraints, with respect to a fragment of the diagram containing the two associated classes.
- For this, one has to assume that the state space of each class has a component recording the collection of live instances.
- An *association* becomes a constraint over such components of the (joint) state space.
- For example a 'one-to-one' association corresponds to a predicate asserting the existence of an injective function relating the collection of instances of each class.
- Similarly, a 'one-to-many' association corresponds to a relation whose kernel is the identity, i.e., a total relation whose converse is simple.

## Class Diagrams

- In general, constraints and associations are predicates which are supposed to be preserved along the system life-time.
- Formally, they are incorporated in the semantics as *invariants*. Such predicates, once encoded as coreflexives, i.e., fragments of the identity, according to

$$y \Phi_P x \equiv y = x \wedge P x$$

can be specified as  $c \cdot \Phi_P \subseteq T \Phi_P \cdot c$ .

- When reasoning about diagram transformations, such as refactoring, constraints entail for *proof obligations*. For example,

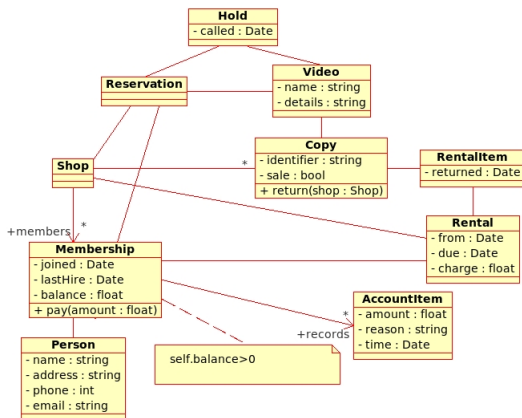
$$\llbracket \text{balance} > 0 \rrbracket =$$

$$\llbracket \mathbf{Membership} \rrbracket \cdot \Phi_{\text{balance} > 0} \subseteq T \Phi_{\text{balance} > 0} \cdot \llbracket \mathbf{Membership} \rrbracket$$

needs to be discarded whenever justifying a refactoring involving class **Membership**.

## Class Diagrams

- Consider the *Inline Class Refactoring* pattern: Inline class refactoring allows two classes to be merged together provided one of them has no methods available.
- The previous example can be transformed into the following diagram:



## Class Diagrams

- classes **Membership** and **Account** are replaced by a new class **Membership'** whose semantics is a new coalgebra over the state space of  $\llbracket \mathbf{Membership} \rrbracket$  to which a new attribute balance is added.

$$\llbracket \mathbf{Membership}' \rrbracket = \langle \langle \text{at}_{\mathbf{Membership}}, \text{at}_{\mathbf{Account}} \rangle, \overline{\text{md}}_{\mathbf{Membership}} \rangle$$

- Assuming the remaining part of the diagram remains unchanged, clearly the new class  $\llbracket \mathbf{Membership}' \rrbracket$  and the relevant fragment of the original class diagram, i.e.,

$$\llbracket \mathbf{Membership} \rrbracket \boxtimes \llbracket \mathbf{Account} \rrbracket$$

are not bisimilar.

- But we can prove that the inline refactoring is actually a refinement between the two diagrams.

## Beyond Class Diagrams ...

- Sun Meng, Zhang Naixiao and Luis Barbosa. On Semantics and Refinement of UML Statecharts: A Coalgebraic View. In Proceedings of SEFM'04, pages 164-173, IEEE Computer Society, 2004.
- Sun Meng and Luis Barbosa. A Coalgebraic Semantic Framework for Reasoning about UML Sequence Diagrams. In Proceedings of QSIC'08. IEEE Computer Society, 2008.
- Sun Meng and Luis Barbosa. A Coalgebraic Semantic Framework for Reasoning about Interaction Designs. In Kevin Lano eds. UML Semantics and its Applications. pages 249-280, Wiley, 2009.

## Coalgebra and Logic ...

- Coalgebra offers tools that apply uniformly to a large class of systems.
- An obvious question from this perspective is whether we can deal with logics for coalgebras in a uniform way.
- This question is of interest from a computer science point of view because coalgebras are systems and logics are specification languages.

## Coalgebra and Logic ...

- As an example, consider the coalgebraic logic invented by Lawrence Moss, for which the syntax and semantics working in a uniform way for all signatures  $\Sigma : \mathbf{Set} \rightarrow \mathbf{Set}$ .
- Formulas of the logic are invariants under behavioral equivalence and the logic is reasonably expressive.
- Here “reasonably expressive” means by requiring that admitting infinite conjunctions, the logic should be able to characterize processes (elements of coalgebras) up to behavioral equivalence.
- The aim is to find a language  $\mathcal{L}_\Sigma$  and for each  $\Sigma$ -coalgebra  $(X, \psi)$  a relation  $\models_\Sigma \subset X \times \mathcal{L}_\Sigma$  satisfying the above requirements.

## Coalgebra and Logic ...

- The starting point is that signatures are functors on **Set** and may hence also be applied to sets of formulas  $\mathcal{L}_\Sigma$  and relations  $\models_\Sigma$ .
- Functors  $\Sigma$  on **Set** are extended to functors on the category of classes **SET** via  $\Sigma K = \bigcup \{ \Sigma X : X \subset K, X \text{ a set} \}$  for classes  $K$ . Moreover,  $\Sigma$  is assumed to weakly preserve pullbacks.
- The syntax of coalgebraic logic:

### Definition

$\mathcal{L}_\Sigma$  is defined to be the least class satisfying:

$$\Phi \subset \mathcal{L}_\Sigma, \Phi \text{ a set} \Rightarrow \bigwedge \Phi \in \mathcal{L}_\Sigma$$

$$\phi \in \Sigma(\mathcal{L}_\Sigma) \Rightarrow \phi \in \mathcal{L}_\Sigma$$

That is,  $\mathcal{L}_\Sigma$  is the initial algebra wrt the functor  $\mathcal{P} + \Sigma$ .



## Coalgebra and Logic ...

- The starting point is that signatures are functors on **Set** and may hence also be applied to sets of formulas  $\mathcal{L}_\Sigma$  and relations  $\models_\Sigma$ .
- Functors  $\Sigma$  on **Set** are extended to functors on the category of classes **SET** via  $\Sigma K = \bigcup \{ \Sigma X : X \subset K, X \text{ a set} \}$  for classes  $K$ . Moreover,  $\Sigma$  is assumed to weakly preserve pullbacks.
- The syntax of coalgebraic logic:

### Definition

$\mathcal{L}_\Sigma$  is defined to be the least class satisfying:

$$\Phi \subset \mathcal{L}_\Sigma, \Phi \text{ a set} \Rightarrow \bigwedge \Phi \in \mathcal{L}_\Sigma$$

$$\phi \in \Sigma(\mathcal{L}_\Sigma) \Rightarrow \phi \in \mathcal{L}_\Sigma$$

That is,  $\mathcal{L}_\Sigma$  is the initial algebra wrt the functor  $\mathcal{P} + \Sigma$ .

## Coalgebra and Logic ...

- The semantics of coalgebraic logic goes as follows:

### Definition

Given a coalgebra  $(X, \psi)$  define  $\models_{\Sigma} \subset X \times \mathcal{L}_{\Sigma}$  as the least relation such that (let  $x \in X$ ):

$$x \models_{\Sigma} \phi \text{ for all } \phi \in \Phi, \Phi \subset \mathcal{L}_{\Sigma}, \Phi \text{ a set} \Rightarrow x \models_{\Sigma} \bigwedge \Phi$$

$$\exists w \in \Sigma(\models_{\Sigma}) \text{ s.t. } \Sigma\pi_1(w) = \psi(x), \Sigma\pi_2(w) = \phi \Rightarrow x \models_{\Sigma} \phi$$

where  $\pi_1, \pi_2$  denotes the projections from the product  $X \times \mathcal{L}_{\Sigma}$  to its components.

## Coalgebra and Logic ...

- The following theorem shows that coalgebraic logic reflects precisely the notion of behavioral equivalence:

### Theorem

Let  $\Sigma$  be a functor on **Set** which weakly preserving pullbacks.  
Then

1. Formulas of  $\mathcal{L}_\Sigma$  are invariant under behavioral equivalence and
2. For each coalgebra  $(X, \psi)$  and each  $x \in X$  there is a formula  $\phi_x \in \mathcal{L}_\Sigma$  such that for all coalgebras  $(X', \psi')$  and all  $x' \in X'$ ,

$$x' \models_\Sigma \phi_x \text{ iff } x, x' \text{ behaviorally equivalent}$$

For more details about the coalgebraic logic:

Lawrence Moss. Coalgebraic logic. *Annals of Pure and Applied Logic*. 96: 277-317, 1999.

## Coalgebra and Logic ...

In fact, I am not an expert in this area ...

But there are many references:

- Yde Venema. Algebras and coalgebras. In *Handbook of Modal Logic*, pages 331-426, Elsevier, 2006.
- Alexander Kurz. Coalgebras and their Logics. *SIGACT News*, vol. 37, pages 57-77, 2006.
- Bart Jacobs. The temporal logic of coalgebras via Galois algebras. *Mathematical Structures in Computer Science*, vol. 12(6), pages 875-903, 2002.
- ...

## Coalgebra and Logic ...

In fact, I am not an expert in this area ...

But there are many references:

- Yde Venema. Algebras and coalgebras. In *Handbook of Modal Logic*, pages 331-426, Elsevier, 2006.
- Alexander Kurz. Coalgebras and their Logics. *SIGACT News*, vol. 37, pages 57-77, 2006.
- Bart Jacobs. The temporal logic of coalgebras via Galois algebras. *Mathematical Structures in Computer Science*, vol. 12(6), pages 875-903, 2002.
- ...

## Why Coalgebra for Logic?

- A coalgebraic perspective on evolving state-based systems is of interest to logicians, because ...
  - Uniform treatment of different types of systems. For example, one can establish that satisfiability of coalgebraic logic is in PSPACE and that complete coalgebraic logics have the finite model property. The uniformity of the metatheory might well translate into software tools that are easier to design, maintain, and to implement.
  - Modularity. Different functors can be combined using composition of functors, product, coproduct, etc. Theorems and algorithms for basic types can then be lifted to arbitrarily complex combinations.
  - One-step analysis. Coalgebraic analysis of dynamic systems is particularly successful where the class of all complete behaviors is determined by the possible one-step behaviors. This is the basis of **coinduction**. It also plays an important role in applications to automata theory.
  - ...

## Why Coalgebra for Logic?

- A coalgebraic perspective on evolving state-based systems is of interest to logicians, because ...
  - Uniform treatment of different types of systems. For example, one can establish that satisfiability of coalgebraic logic is in PSPACE and that complete coalgebraic logics have the finite model property. The uniformity of the metatheory might well translate into software tools that are easier to design, maintain, and to implement.
  - Modularity. Different functors can be combined using composition of functors, product, coproduct, etc. Theorems and algorithms for basic types can then be lifted to arbitrarily complex combinations.
  - One-step analysis. Coalgebraic analysis of dynamic systems is particularly successful where the class of all complete behaviors is determined by the possible one-step behaviors. This is the basis of **coinduction**. It also plays an important role in applications to automata theory.
  - ...

## Why Coalgebra for Logic?

- A coalgebraic perspective on evolving state-based systems is of interest to logicians, because ...
  - Uniform treatment of different types of systems. For example, one can establish that satisfiability of coalgebraic logic is in PSPACE and that complete coalgebraic logics have the finite model property. The uniformity of the metatheory might well translate into software tools that are easier to design, maintain, and to implement.
  - Modularity. Different functors can be combined using composition of functors, product, coproduct, etc. Theorems and algorithms for basic types can then be lifted to arbitrarily complex combinations.
  - One-step analysis. Coalgebraic analysis of dynamic systems is particularly successful where the class of all complete behaviors is determined by the possible one-step behaviors. This is the basis of **coinduction**. It also plays an important role in applications to automata theory.
  - ...



## Why Coalgebra for Logic?

- A coalgebraic perspective on evolving state-based systems is of interest to logicians, because ...
  - Uniform treatment of different types of systems. For example, one can establish that satisfiability of coalgebraic logic is in PSPACE and that complete coalgebraic logics have the finite model property. The uniformity of the metatheory might well translate into software tools that are easier to design, maintain, and to implement.
  - Modularity. Different functors can be combined using composition of functors, product, coproduct, etc. Theorems and algorithms for basic types can then be lifted to arbitrarily complex combinations.
  - One-step analysis. Coalgebraic analysis of dynamic systems is particularly successful where the class of all complete behaviors is determined by the possible one-step behaviors. This is the basis of **coinduction**. It also plays an important role in applications to automata theory.
  - ...

## Future Directions

- Coalgebras have been successfully applied to many areas in computer science.
- Applications in mathematics and logic have been developed much less than applications to computer science, but coalgebra as an area naturally overlaps with Universal Algebra, Modal Logic, Domain Theory and Category Theory.
- We believe that this presents many opportunities for exciting future research in different areas.

Thank you!