<table>
<thead>
<tr>
<th>1. Introduction</th>
<th>Design complexity, examples of embedded and cyber-physical systems, traditional design flows, Platform-Based Design, design capture and entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. Architecture and performance abstraction</td>
<td>Definition of architecture, examples. Real time operating systems, scheduling of computation and communication.</td>
</tr>
</tbody>
</table>
EE249 Fall’12: where we are going

1. Introduction
   - Design complexity, examples of embedded and cyber-physical systems, traditional design flows, Platform-Based Design, design capture and entry

2. Functional modeling, analysis and simulation

3. Architecture and performance abstraction
   - Definition of architecture, examples. Real time operating systems, scheduling of computation and communication.

4. Mapping

5. Verification

6. Applications
   - Automotive: car architecture, communication standards (OSEK/AUTOSAR), scheduling and timing analysis. Building automation. Aircraft electric power system.

Contract-Based Design: an all-encompassing framework
The key to Platform Based Design

- Components
- Composition rules
- Refinement rules
- Abstraction rules
Outline: contracts and compositional methods for system design

- Where and why using contracts?
- Introduction to contracts
- Mathematical meta-theory of contracts
- Overview of concrete contract theories
- Application examples
Outline: contracts and compositional methods for system design

- Where and why using contracts?
  - Structuring top-level specifications
  - Sub-contracting and reusing
  - Deployment and mapping
- Introduction to contracts
- Mathematical meta-theory of contracts
- Overview of concrete contract theories
- Application examples
Structuring top-level specifications

• A desirable objective at each step of the design

• Requirement documents are structured into viewpoints (sometimes referred to as chapters, aspects, sub-documents, … depending on the company or sector)
  – Behavioral viewpoint: the functions are specified
  – Timing viewpoint: timing budgets are allocated to activities
  – Safety viewpoint: fault propagation, reliability
  – …

• Viewpoints are generally developed by different teams using different skills, frameworks and tools
Example: a monitoring system

Software for monitoring the physical system

- Requirements must specify
  - Decision logic
  - Sampling period
  - When to activate/inhibit monitoring
  - Time lag before fault confirmation
  - Fault identifier
  - Fault compensation
  - Returns to the cockpit
  - *Very diverse in nature*

- Requirements must also specify *at a higher specification level* what is the objective of the detection, which fault to detect and isolate

*decision logic ≈ implementation*

Source: A. Benveniste
The Landscape: sub-contracting

structure

- System architecture
  - Context
    - System
      - Sub-system
      - Sub-system
      - Component

semantics

- Specification and development
  - Viewpoint
  - Viewpoint
  - Requirements document
    - Design
      - Designed System
  - Requirements document
  - Requirements document

Developed by different teams

Developed by different suppliers

Source: A. Benveniste
Need for requirement engineering

**Traceability**
- Requirements attached to “everything” via hyperlinks (tests, V&V, integration)

**Ontology**
- Terms used for entities should be precise and unambiguous (important)
- Terms used for entities should be structured (ontology)

**Identifying responsibilities**
- Some requirements express guarantees; other express assumptions

**Partitioning and sub-contracting**
- Allocating requirements to suppliers, budgeting

**Modular handling of viewpoints & subsystems**
- Separation of concerns: function, QoS, safety/reliability…

**Fundamental properties (certification bodies)**
- Completeness, Consistency, Compatibility, … (from INCOSE)

Source: A. Benveniste
Overall, requirements engineering

- has been considered by the AI community (ontologies)
- has been considered by the Software Engineering community as part of MDE
- has been mostly ignored by other research communities
  - control science
  - formal methods in computer science

Source: A. Benveniste
Requirements on the meta-theory

Source: A. Benveniste
Requirements on the meta-theory
{environment, component}

Contexts are important

- What the system guarantees: must be met by any implementation
- What the system assumes about its context of use: must be met by any legal environment

Source: A. Benveniste
Requirements on the meta-theory

Source: A. Benveniste
Requirements on the meta-theory conjunction and parallel composition

From \( \land \) to \( \not\land \)

- Requirements documents decompose into chapters/viewpoints: conjunction \( \land \)
- System = architecture of sub-systems: composition \( \not\land \)
- Independent development

Source: A. Benveniste
Conjunction of contracts

• Development of each viewpoint is performed under assumptions regarding its context of use, including the other viewpoints

• Conjunction is used to fuse viewpoints and get the full system specification

• Each viewpoint is itself a conjunction of requirements

• Consistency checking for contracts obtained as a conjunction is mandatory (is there an implementation satisfying all the requirements?)
Requirements on the meta-theory implements and refines

Designed component $\models$ local contract
- Meets the guarantees under any legal environment

Decomposed contract $\preceq$ global contract
- Stronger guarantees
- Relaxed context

Source: A. Benveniste
Refinement and composition of contracts

\[ C_{11} \otimes C_{12} \otimes C_{13} \preceq C_1 \]

- Obtaining the three sub-contracts \( C_{11}, C_{12}, C_{13} \) is the art of the designer, based on architectural considerations.

- Subsystems can also be developed by re-using off-the-shelf components.

- Contract theories offer the following services:
  - Firmly assess whether the above relation holds for the decomposition step.
  - Formally check the compatibility of \( C_{11}, C_{12}, C_{13} \).
  - Guarantee that the information provided to suppliers is self-contained.
Facilitating integration of specialized tool and frameworks

• Systems are developed by composing pieces that have been (in part) pre-designed by other groups or companies
  – Routinely done in vertical design chains (avionics, automotive,…)
  – …but in a heuristic and ad hoc way

• Need for standards, methods and tools in the software and hardware domains to allow integration of subsystems and their implementations
  – Across the electronic and mechanical domains (near future), but also across chemical and biology domains (further future) for nano-systems
  – From a static standpoint: data dictionaries, off-line model transformations,…
  – From a dynamic standpoint: co-simulation, HW-in-the-loop simulations and emulation
Deployment and mapping

• The satisfaction of safety or timing viewpoints by a considered deployment depends on
  – the supporting execution platform
  – the mapping of the application to the execution platform.

• How to check deployment compositionally?
Deployment and mapping

\[
C = \bigwedge_k \left( \bigotimes_{i \in I_k} C_{ik} \right)
\]
\[
P = \bigotimes_{j \in J} \left( \bigwedge_{\ell \in L_j} P_{j\ell} \right)
\]

- Virtual model of the execution platform \( \mathcal{P} \)
  - available computing units
  - bus protocol
  - library of RTOS services

- Components enhanced with timing information, fault propagation information
Deployment and mapping as parallel composition

\[
    \mathcal{C} = \bigwedge_k \left( \bigotimes_{i \in I_k} \mathcal{C}_{ik} \right) \\
    \mathcal{P} = \bigotimes_{j \in J} \left( \bigwedge_{\ell \in L_j} \mathcal{P}_{j\ell} \right)
\]

- Contracts attached to sub-systems or components
  - Application contract (top)
  - Platform (bottom)
- Composition defined by synchronization tuples, where both occurrences and values of the different elements are unified.
Deployment and mapping as parallel composition

- Checking $C \otimes P \preceq C$ can be performed compositionally!
- $C \otimes P$ also called vertical contract
  - Relates application and computing platforms
  - Horizontal contracts relate components at the same level
- Mapping in RT-Builder and Metropolis

\[ C \otimes P = \left[ \bigwedge_k (\bigotimes_{i \in I_k} C_{ik}) \right] \otimes \left[ \bigotimes_{j \in J} \left( \bigwedge_{\ell \in L_j} P_{j\ell} \right) \right] \]
Outline: contracts and compositional methods for system design

- Where and why using contracts?
- Introduction to contracts
  - Components and contracts
  - Contract operators and properties
    - Incremental design
    - Independent implementability
- Mathematical meta-theory of contracts
- Overview of concrete contract theories
- Application examples
Compositional Reasoning

Reliably derive global properties of systems based on local properties of components

- Contracts as Assume/Guarantee pairs
- Component properties guaranteed under a set of assumptions on the environment
- Composition valid iff all assumptions are satisfied
Components

Set $P$ of ports, $P = I \cup O$
Set $A$ of assumptions
Set $G$ of guarantees

An implementation $M$ satisfies a contract $(A, G)$, if $M$ refines $G$ in the context of $A$

$$M \cap A \subseteq G$$

Set $P$ of ports, $P = I \cup O$
Set $M$ of behaviors
# Implementations and Contracts

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Contract</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defines the <strong>behavior</strong> of a component</td>
<td>Defines the <strong>boundary of use</strong> of a component</td>
</tr>
<tr>
<td>Does not restrict the environment of use</td>
<td>Declares the acceptable environments A (assumptions) and the limits of operation G (guarantees)</td>
</tr>
<tr>
<td>Usually deterministic</td>
<td>Usually non-deterministic</td>
</tr>
<tr>
<td>Usually tied to some particular architectural solution</td>
<td>Usually abstract, to encompass several possible different implementations</td>
</tr>
</tbody>
</table>
A simple static system

• $M_1$ computes the division between two real inputs and returns the result as a real output

\[
M_1 : \begin{cases}
\text{variables: } & \{ \text{inputs: } x, y \\
\text{outputs: } & z \}
\text{types: } & x, y, z \in \mathbb{R}
\text{behaviors: } & (y \neq 0 \rightarrow z = x/y) \land (y = 0 \rightarrow z = 0)
\end{cases}
\]

• $C_1$ intuitively specifies the intended behavior of components that implement division

\[
C_1 : \begin{cases}
\text{variables: } & \{ \text{inputs: } x, y \\
\text{outputs: } & z \}
\text{types: } & x, y, z \in \mathbb{R}
\text{assumptions: } & y \neq 0
\text{guarantees: } & z = x/y
\end{cases}
\]

Environments for $C_1$? Implementations for $C_1$? Is $C_1$ consistent?
Satisfaction

- \( M \cap A \subseteq G \)
- \( M \models (A, G) \)

- An implementation must refine the guarantees, but only in the context of the acceptable environments
  - Less restrictive than regular refinement (trace containment)
  - Refinement is only up to the acceptable contexts
  - Implementations are free to behave as they like (and even break guarantees) for non acceptable contexts
Composition

- Contract composition enables incremental design
- Component *composability* is a syntactic property (type matching)

\[ M_1: \begin{align*}
\text{variables:} & \quad \{ \text{inputs: } x, y \\
\text{types: } & \quad x, y, z \in \mathbb{R} \\
\text{behaviors:} & \quad (y \neq 0 \rightarrow z = x/y) \land (y = 0 \rightarrow z = 0) \}
\end{align*} \]

\[ M_2: \begin{align*}
\text{variables:} & \quad \{ \text{inputs: } x \\
\text{types: } & \quad x, y \in \mathbb{R} \\
\text{behaviors:} & \quad y = e^x \}
\end{align*} \]

Are \( M_1 \) and \( M_2 \) composable?
Composition

• Contract composition enables incremental design
• Component **composability** is a syntactic property (type matching)

Are $M_1$ and $M'_2$ composable?
Composition

\[ C_1 : \begin{cases} \text{variables: } & \{ \text{inputs: } x, y \\ \text{outputs: } & z \} \\ \text{types: } & x, y, z \in \mathbb{R} \\ \text{assumptions: } & y \neq 0 \\ \text{guarantees: } & z = x/y \end{cases} \]

\[ C_2 : \begin{cases} \text{variables: } & \{ \text{inputs: } u \\ \text{outputs: } & x \} \\ \text{types: } & u, x \in \mathbb{R} \\ \text{assumptions: } & T \\ \text{guarantees: } & x > u \end{cases} \]
Compatibility

- Two contracts are compatible when the guarantees of one do not violate the assumptions of the other
  - And/or when an environment can enforce that condition

- A notion that cannot be expressed on implementations
  - They lack sufficient expressiveness

- A combined syntactic and semantic property
Composition

\[ G_{c_1 \otimes c_2} = G_{c_1} \wedge G_{c_2} \]

\[ A_{c_1 \otimes c_2} = \max \left\{ A \mid \begin{array}{c} A \wedge G_{c_2} \Rightarrow A_{c_1} \\ A \wedge G_{c_1} \Rightarrow A_{c_2} \end{array} \right\} \]
Composition

\[(A_1, G_1)\]

\[(A_1 \cap A_2) \cup \neg G_1 \cup \neg G_2\]

\[G_1 \cap G_2\]

\[(A_2, G_2)\]
Composition

• The composite must guarantee what the components guarantee
  – $G_1 \cap G_2$

• The composite accepts only what is accepted by both components
  – $A_1 \cap A_2$

• However, part of the assumptions of one component can be discharged directly by the other, and vice-versa
  – $(A_1 \cap A_2) \cup \neg G_1 \cup \neg G_2$
  – Look at the weakest assumption assuring that both the initial contract assumptions are met

• Composition must be both associative and commutative
Composition

$C_1 : \begin{cases} 
\text{variables:} & \{ \text{inputs: } x, y \\
& \text{outputs: } z \\
\text{types:} & x, y, z \in \mathbb{R} \\
\text{assumptions:} & y \neq 0 \\
\text{guarantees:} & z = x/y \end{cases}
$

$C_2 : \begin{cases} 
\text{variables:} & \{ \text{inputs: } u \\
& \text{outputs: } x \\
\text{types:} & u, x \in \mathbb{R} \\
\text{assumptions:} & \text{true} \\
\text{guarantees:} & x > u \end{cases}
$

$C_1 \otimes C_2 : \begin{cases} 
\text{variables:} & \{ \text{inputs: } u, y \\
& \text{outputs: } x, z \\
\text{types:} & x, y, u, z \in \mathbb{R} \\
\text{assumptions:} & y \neq 0 \\
\text{guarantees:} & x > u \land z = x/y \end{cases}$
Viewpoints

- Same component
- Different aspects
- Aspects do not combine by parallel composition
- We must instead take their conjunction
Refinement and conjunction

• Conjunction better defined as the greatest lower bound of a refinement order
  – A contract $C = (A, G)$ is stronger than, or refines a contract $C' = (A', G')$ whenever it guarantees more while assuming less
  – $G \subseteq G'$
  – $A \supseteq A'$
  – (If $M$ satisfies $C$, then $M$ also satisfies $C'$)

• Greatest lower bound
  – $C \land C' = (A \cup A', G \cap G')$
  – The conjunction must accept the environments of both viewpoints
  – The conjunction must enforce the guarantees of both viewpoints
  – If $M$ satisfies both $C$ and $C'$, then $M$ satisfies $C \land C'$
Example: refinement and independent implementability

- For all contracts $C_1$, $C_2$, $C'_1$, $C'_2$, let
  - $C_1$ be compatible with $C_2$
  - $C'_1 \preceq C_1$ and $C'_2 \preceq C_2$

Then $C'_1$ is compatible with $C'_2$ and $C'_1 \otimes C'_2 \preceq C_1 \otimes C_2$

- Example:

\begin{align*}
\text{variables: } & \{ y \} \\
\text{inputs: } & y \\
\text{outputs: } & z \\
C''_1 : & \{ y, z \in \mathbb{R} \} \\
\text{assumptions: } & y \neq 0 \\
\text{guarantees: } & z \in \mathbb{R} \\
\text{variables: } & \{ x \} \\
\text{inputs: } & x \\
\text{outputs: } & y \\
C''_2 : & \{ x, y \in \mathbb{R} \} \\
\text{assumptions: } & \text{T} \\
\text{guarantees: } & y > 0 \\
\text{variables: } & \{ x, y \} \\
\text{inputs: } & x, y \\
\text{outputs: } & z \\
C_1 : & \{ x, y, z \in \mathbb{R} \} \\
\text{assumptions: } & y \neq 0 \\
\text{guarantees: } & z = x/y
\end{align*}

- $C''_1$ compatible with $C''_2$
- $C_1 \preceq C''_1$

Then $C_1 \otimes C''_2 \preceq C''_1 \otimes C''_2$
Example: conjunction

• For all contracts $C_1$, $C_2$ shared refinable, then
  - $C_1 \land C_2 \preceq C_1$ and $C_1 \land C_2 \preceq C_2$
  - for all $C$, if $C \preceq C_1$ and $C \preceq C_2$ then $C \preceq C_1 \land C_2$

• Example:
  - $C_T^2$ shared refinable with $C_1$
  - $C_T^2 \land C_1$ guarantees, in addition to $C_1$, a latency with bound 1.
Platform-Based Design: Horizontal and Vertical Contracts

• So far, contracts for components at the same level of abstraction
  – We refer to this kind of composition as horizontal, and we talk about horizontal contracts

• A component could express assumptions and guarantees w.r.t. another level of abstraction
  – For instance, it may assume an execution environment with certain properties or performance
  – Likewise, it may guarantee certain patterns of activation to the execution environment
  – Contracts that span different levels of abstraction are referred to as vertical contracts
Vertical contracts

- Top-down vertical contracts
  - Capture the desired specifications (gain, bandwidth, carrier frequency, ...)
  - Limit the impact of undesired behaviors (loading, crosstalk, coupling, ...)
  - Take into consideration the approximations of the system model used to reduce complexity

- Bottom-up vertical contracts
  - enforce validity of higher-level macro-models w.r.t. lower level models
Putting it all together

Layer N + 1

Layer N

Layer N - 1

Putting it all together
Circular reasoning in horizontal contracts

- $G_{1H}$ can only be guaranteed for legal design contexts of $S_1$
  - need to establish $A_{1H}$
  - To establish $A_{1H}$, want to involve $G_{3H}$, hence $A_{3H}$

![Diagram showing relationships between $S_1$, $S_2$, $S_3$, $A_{1H}$, $A_{2H}$, $A_{3H}$, $G_{1H}$, $G_{2H}$, and $G_{3H}$ with transition probabilities $p_{S1}$, $p_{S2}$, $p_{12}$, $p_{23}$, $p_{31}$, and $p_{2S}$.]
Circular reasoning in horizontal contracts

- To establish $A_{3H}$, need to involve $G_{2H}$ as a witness, hence $A_{2H}$
Circular reasoning in horizontal contracts

- …but to establish $A_{2H}$, we need to establish $G_{1H}$, which is where we started the chain!
- Seemingly circular arguments are justified for assumptions and guarantees expressed as safety properties
  - can be proven or disproved by finite observations
  - need to observe restrictions on how $A/G$ refer to ports
Cyber-physical control system

Assume execution parameters such as jitter, latency, accuracy, WCET

Assume load and utilization levels, resource usage
Contracts

Controller

• **Assumptions**
  – Overall closed loop latency less than $T$
  – Compute resource availability above $P\%$
  – Computing MIPS power at least $M$

• **Guarantees**
  – Model prediction error bounded by $\Delta$
  – Deadlock state unreachable
  – Response time less than $R$

Platform

• **Assumptions**
  – Task activation interval no more than $I$
  – Task is thread and memory safe (can avoid checks)
  – Task are jitter independent within period

• **Guarantees**
  – Temporal isolation
  – Minimum dedicated computing power
  – Minimum sensor accuracy $S$
Outline: Contracts and compositional methods for system design

- Where and why using contracts?
- Introduction to contracts
- Mathematical meta-theory of contracts
  - Components and composition
  - Contracts
  - Refinement and conjunction
  - Contract composition
  - Quotient
  - Observers
- Overview of concrete contract theories
- Application examples
The meta-theory

- We assume some primitive concepts

<table>
<thead>
<tr>
<th>Component</th>
<th>$M$ open or closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composability</td>
<td>A type property</td>
</tr>
<tr>
<td>Composition</td>
<td>( \times ) commutative &amp; associative</td>
</tr>
<tr>
<td>Environment</td>
<td>$E : E \times M$ closed</td>
</tr>
</tbody>
</table>

- On top of these primitive concepts we define generic operators satisfying generic properties

- How primitive concepts, operators, and properties, are made effective depends on the specific framework

Source: A. Benveniste
The meta-theory

- Generic Relations and Operators:

<table>
<thead>
<tr>
<th>Contract</th>
<th>$C = (\mathcal{E}_c, \mathcal{M}_c) = $ (set of environments, set of implementations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistency</td>
<td>$\mathcal{M}_c \neq \emptyset$</td>
</tr>
<tr>
<td>Compatibility</td>
<td>$\mathcal{E}_c \neq \emptyset$</td>
</tr>
<tr>
<td>Implementation</td>
<td>$M \models^M C$ iff $M \in \mathcal{M}_c$ ; $E \models^E C$ iff $E \in \mathcal{E}_c$</td>
</tr>
</tbody>
</table>
The meta-theory

- Generic Relations and Operators:

<table>
<thead>
<tr>
<th>Contract</th>
<th>( C = (\mathcal{E}_c, \mathcal{M}_c) = (\text{set of environments, set of implementations}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistency</td>
<td>( \mathcal{M}_c \neq \emptyset )</td>
</tr>
<tr>
<td>Compatibility</td>
<td>( \mathcal{E}_c \neq \emptyset )</td>
</tr>
<tr>
<td>Implementation</td>
<td>( M \models^M C \iff M \in \mathcal{M}_c ); ( E \models^E C \iff E \in \mathcal{E}_c )</td>
</tr>
<tr>
<td>Refinement</td>
<td>( C' \subseteq C \iff \mathcal{E}_{c'} \supseteq \mathcal{E}<em>c ) and ( \mathcal{M}</em>{c'} \subseteq \mathcal{M}_c )</td>
</tr>
<tr>
<td>Conjunction</td>
<td>( C_1 \wedge C_2 = \text{GLB for } \leq )</td>
</tr>
</tbody>
</table>

Source: A. Benveniste
Refinement and conjunction: properties

• **Expressible contract family** \( \mathcal{C} \). Every set \( \mathcal{M}' \subseteq \mathcal{M} \) of components can be represented as \( \mathcal{M}' = \mathcal{M}_C \) for some contract \( C \), and similarly for sets of environments.
  
  – Only contracts belonging to this family can be considered

• **Shared refinement**
  
  – Any contract that refines \( C_1 \land C_2 \) also refines \( C_1 \) and \( C_2 \). Any implementation of \( C_1 \land C_2 \) is a shared implementation of \( C_1 \) and \( C_2 \). Any environment of \( C_1 \) and \( C_2 \) is an environment of \( C_1 \land C_2 \).
  
  – For \( \mathcal{C}' \subseteq \mathcal{C} \) subset of contracts, \( \land \mathcal{C}' \) is compatible if and only if there exists a compatible \( C \in \mathcal{C}' \)
## The meta-theory

- **Generic Relations and Operators:**

<table>
<thead>
<tr>
<th>Contract</th>
<th>$C = (\mathcal{E}_c, \mathcal{M}_c) = (\text{set of environments, set of implementations})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistency</td>
<td>$\mathcal{M}_c \neq \emptyset$</td>
</tr>
<tr>
<td>Compatibility</td>
<td>$\mathcal{E}_c \neq \emptyset$</td>
</tr>
<tr>
<td>Implementation</td>
<td>$M \models^M C$ iff $M \in \mathcal{M}_c$; $E \models^E C$ iff $E \in \mathcal{E}_c$</td>
</tr>
<tr>
<td>Refinement</td>
<td>$C' \preceq C$ iff $\mathcal{E}_{c'} \supseteq \mathcal{E}<em>c$ and $\mathcal{M}</em>{c'} \subseteq \mathcal{M}_c$</td>
</tr>
<tr>
<td>Conjunction</td>
<td>$C_1 \land C_2 = \text{GLB for } \preceq$</td>
</tr>
<tr>
<td>Composition</td>
<td>$C_1 \otimes C_2 = \bigwedge \left{ C \left</td>
</tr>
</tbody>
</table>

Source: A. Benveniste
The meta-theory

- **Generic Relations and Operators:**

<table>
<thead>
<tr>
<th>Contract</th>
<th>( C = (\mathcal{E}_c, \mathcal{M}_c) = \text{(set of environments, set of implementations)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistency</td>
<td>( \mathcal{M}_c \neq \emptyset )</td>
</tr>
<tr>
<td>Compatibility</td>
<td>( \mathcal{E}_c \neq \emptyset )</td>
</tr>
<tr>
<td>Implementation</td>
<td>( M \models^M C \text{ iff } M \in \mathcal{M}_c ; \quad E \models^E C \text{ iff } E \in \mathcal{E}_c )</td>
</tr>
<tr>
<td>Refinement</td>
<td>( C' \leq C \text{ iff } \mathcal{E}_{c'} \supseteq \mathcal{E}<em>c \text{ and } \mathcal{M}</em>{c'} \subseteq \mathcal{M}_c )</td>
</tr>
<tr>
<td>Conjunction</td>
<td>( C_1 \land C_2 = \text{GLB for } \leq )</td>
</tr>
<tr>
<td>Composition</td>
<td>( C_1 \otimes C_2 \text{ is defined if } \left{ \begin{array}{l} M_1 \models^M C_1 \ M_2 \models^M C_2 \end{array} \right} \Rightarrow (M_1, M_2) \text{ composable} )</td>
</tr>
<tr>
<td>Quotient</td>
<td>( C_1 / C_2 = \bigvee { C \mid C \otimes C_2 \preceq C_1 } \text{ Least Upper Bound (LUB) for } \leq )</td>
</tr>
</tbody>
</table>

Source: A. Benveniste
The meta-theory

- Generic Properties:

<table>
<thead>
<tr>
<th>Refinement</th>
<th>Substituability of environments</th>
<th>Substituability of implementations</th>
</tr>
</thead>
</table>

Composition

- $(C_1, C_2)$ compatible $C_i' \preceq C_i$ \implies \begin{align*}
  \{ (C_1', C_2') \text{ compatible } \\
  C_1' \otimes C_2' \preceq C_1 \otimes C_2
\end{align*}

  Independent implementability

- $(C_1 \otimes C_2) \otimes (C_3 \otimes C_4) = (C_1 \otimes C_3) \otimes (C_2 \otimes C_4)$

  Compatibility $\implies$ Compatibility

  Associativity

- $[(C_{11} \land C_{21}) \otimes (C_{12} \land C_{22})] \preceq [(C_{11} \otimes C_{12}) \land (C_{21} \otimes C_{22})]$  

  Distributivity (freedom in design processes)

Quotient

- $C \preceq C_1/C_2 \iff C \otimes C_2 \preceq C_1$

Source: A. Benveniste
Independent implementability

\[ (C_1, C_2) \text{ compatible} \quad \begin{cases} C'_i \leq C_i \end{cases} \quad \Rightarrow \quad \begin{cases} (C'_1, C'_2) \text{ compatible} \\ C'_1 \otimes C'_2 \leq C_1 \otimes C_2 \end{cases} \]

• \((C_1, C_2)\) are compatible if their composition is defined and compatible

• Independent implementability says that compatible contracts can be independently refined

• **Corollary**: Compatible contracts can be independently implemented
  
  – Apply independent implementability with \(C'_1, C'_2\) singletons
Independent implementability: proof sketch

\[(C_1, C_2) \text{ compatible} \quad C'_i \preceq C_i \quad \Rightarrow \quad \{ (C'_1, C'_2) \text{ compatible} \quad C'_1 \otimes C'_2 \preceq C_1 \otimes C_2 \}\]

• Assumption 1

Let \(C_{C_1 \otimes C_2} = \{ C \mid M_1 \models^M C_1 \text{ and } M_2 \models^M C_2 \text{ and } E \models^E C \downarrow \}

\[M_1 \times M_2 \models^M C \text{ and } E \times M_2 \models^E C_1 \text{ and } E \times M_1 \models^E C_2 \]

Then \(C_1 \otimes C_2 \in C_{C_1 \otimes C_2}\), i.e. the GLB is in the set.

• Assumption 1 ensures that:

  – Composing implementations of each contract yields an implementation for the composition;

  – Composing an environment for the resulting composition with an implementation for \(C_2\) yields an implementation for \(C_1\) and vice-versa.
Independent implementability: proof sketch

\[(C_1, C_2) \text{ compatible} \quad \{ \quad \Rightarrow \quad (C'_1, C'_2) \text{ compatible} \quad C'_i \preceq C_i \quad \}
\]

- **Lemma 1**: Let \( C'_i \preceq C_i \) and \( C_1 \otimes C_2 \) well defined (respective implementations are composable). Then so is \( C'_1 \otimes C'_2 \) and \( C_{C'_1 \otimes C'_2} \supseteq C_{C_1 \otimes C_2} \).

- **Proof of Lemma 1**

  - \( C_1 \otimes C_2 \) well defined implies that every \((M_1, M_2)\) implementing the contracts are composable. Hence, \( C'_1 \otimes C'_2 \) well defined.

  - Any respective implementations of \( C'_1 \) and \( C'_2 \) are also implementations of \( C_1 \) and \( C_2 \)

  - Similarly any environment for \( C_1 \) and \( C_2 \) satisfies also \( C'_1 \) and \( C'_2 \)

  - Replace in the composition formula \( C_1 \) by \( C'_1 \) and \( C_2 \) by \( C'_2 \),...
Independent implementability: proof sketch

\( (C_1, C_2) \) compatible \( C'_i \preceq C_i \) \( \Rightarrow \) \( (C'_1, C'_2) \) compatible \( C'_1 \otimes C'_2 \preceq C_1 \otimes C_2 \)

- **Lemma 1**: Let \( C'_i \preceq C_i \) and \( C_1 \otimes C_2 \) well defined (respective implementations are composable). Then so is \( C'_1 \otimes C'_2 \) and \( C_{C'_1 \otimes C'_2} \supseteq C_{C_1 \otimes C_2} \).

- **Proof of Lemma 1**
  - Replace in the composition formula \( C_i \) by \( C'_i \) and \( C_2 \) by \( C'_2 \)

\[
C_{C'_1 \otimes C'_2} = \left\{ C \left| \begin{array}{l}
M_1 \models^M C_1 \text{ and } M_2 \models^M C_2 \text{ and } E \models^E C \\
M_1 \times M_2 \models^M C \text{ and } E \times M_2 \models^E C_1 \text{ and } E \times M_1 \models^E C_2
\end{array} \right. \right\}
\]

\( C_0 \in C_{C'_1 \otimes C'_2} \Rightarrow C_0 \in C_{C'_1 \otimes C'_2} \)

- Independent implementability becomes a direct corollary of Lemma 1
Meta-theory versus concrete theories

- The meta-theory offers some fundamental properties (incremental development, independent implementability,…)

- Need concrete definitions for components, component compositions

- Need effective means to implement (or approximate) the notions of contracts, refinement, conjunction, composition,…

- Observers provide a generic approach to recover effectiveness
Observers

- Originate from the basic notion of test for programs
- **Definition.** An observer for a contract \( C \) is a pair \((b^E_C, b^M_C)\) of non-deterministic Boolean functions called **verdicts**, such that:
  - \( b^E_C(M) \) outputs \( F \) \( \rightarrow \) \( M \notin \mathcal{E}_C \)
  - \( b^M_C(M) \) outputs \( F \) \( \rightarrow \) \( M \notin \mathcal{M}_C \)
- Non-determinism accounts for dependence of test on the environment stimuli and internal non-determinisms of the component
- Tests only provide **semi-decision** (arrow in one direction)
Mirroring the algebra of contracts with observers

<table>
<thead>
<tr>
<th>Notion</th>
<th>Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C = (\mathcal{E}_C, \mathcal{M}_C) )</td>
<td>((b^E_C, b^M_C))</td>
</tr>
<tr>
<td>( C = C_1 \land C_2 )</td>
<td>( b^E_C = b^E_{C_1} \lor b^E_{C_2} ), ( b^M_C = b^M_{C_1} \land b^M_{C_2} )</td>
</tr>
<tr>
<td>( C = C_1 \lor C_2 )</td>
<td>( b^E_C = b^E_{C_1} \land b^E_{C_2} ), ( b^M_C = b^M_{C_1} \lor b^M_{C_2} )</td>
</tr>
<tr>
<td>( C = C_1 \otimes C_2 )</td>
<td>( b^E_C(E) = \begin{cases} b^M_{C_1}(M_1) \land b^M_{C_2}(M_2) \ \downarrow b^E_{C_2}(E \times M_1) \land b^E_{C_1}(E \times M_2) \ b^M_C(M_1 \times M_2) = b^M_{C_1}(M_1) \land b^M_{C_2}(M_2) \end{cases} )</td>
</tr>
</tbody>
</table>

- Nothing can be said about the relation of the observers for contracts in a refinement ordering
- Nothing can be inferred about contract refinement from observers in a refinement ordering
Observers: properties

• **Lemma 2**: \((b^E_C, b^M_C)\) be an observer for \(C\) and let \(C' \leq C\). Then, any \((b^E, b^M)\) satisfying \(b^E \geq b^E_C\) and \(b^M \leq b^M_C\) is an observer for \(C'\).

• **Lemma 3**

  – If \(b^E_C(E)\) outputs F for all tested environment \(E\), then \(C\) is incompatible

  – If \(b^M_C(M)\) outputs F for all tested component \(M\), then \(C\) is inconsistent

• Still need to exercise all components or environments

  – Non-effective unless notion of “strongest” component or environment is provided in concrete theories
Observers: survey

• Widely studied for software and system testing

• Synchronous languages are a formalism of choice
  – E.g. Esterel, Lustre, Signal, …, Scade V6
  – Support only discrete dynamics
  – Benefit from a solid mathematical semantics
    – Results independent of the type of simulator
    – Consistency between simulated and generated code

• RT-Builder supports the combination of functional and timing viewpoints on top of Signal
Observers: survey

- **Simulink/Stateflow**
  - Mathematical semantics is less firmly defined
  - Supports CT dynamics in the forms of ODE
  - Simulink+SimScape allows including physical system models in observers
  - Similar considerations for Modelica

- **Advocated in the context of Lustre, Scade, Esterel and Signal**
  - In Scade, tests can be evaluated at run time while executing a program
Observers: survey

• Property Specification Language (PSL)
  – An industrial standard for functional properties targeted to digital HW
  – Well-suited specification language for expressing functional requirements involving sequential causality of actions and events
  – Suitable in the contract-based design using observers because of the existing tool support
    – E.g. FoCS translate PSL into checkers to be attached to the design
    – Used for generating transactors that adapt high-level requirements in TLM to RTL implementations
    – Exist a methodology for user-guided automated property exploration
Observers: survey

- **Live Sequence Charts**
  - Graphical specification language based on scenarios
  - Pre-chart versus Main-chart semantics
  - Multi-modal: cold ("may happen") or hot ("must happen")

- **Abstract interpretation techniques can offer finite and effective representation of contracts**
  - Non-computable objects are under- or over-approximated by computable ones
  - It is possible to prove consistency and compatibility for concrete contracts, based on corresponding abstractions (observers can only disprove properties)
Outline: Contracts and compositional methods for system design

- Where and why using contracts?
- Introduction to contracts
- Mathematical meta-theory of contracts
- Overview of concrete contract theories
- Application examples