Specification and Analysis of Contracts
Lecture 2
Components, Services and Contracts

Gerardo Schneider

gerardo@ifi.uio.no

http://folk.uio.no/gerardo/

Department of Informatics,
University of Oslo

SEFM School, Oct. 27 - Nov. 7, 2008
Cape Town, South Africa

Plan of the Course

Plan

Plan

What is a Component?

- We will concentrate only on *software components*

Definition (?!)

- A component has to be a unit of deployment
  - It has to be an executable deliverable for a (virtual) machine
- A component has to be a unit of versioning and replacement
  - It has to remain invariant in different contexts
  - It lives at the level of packages, modules, or classes, and not at the level of objects
- It is useful to see software components as a collection of modules and resources

1. Acquisition is the process of obtaining a software component
2. Deployment is the process of readying the component for installation in a specific environment
3. Installation is the process of making the component available in the specific environment
4. Loading is the process of enabling an installed component in a particular runtime context

- Deployment is not a development activity: it does not happen at the supplier's site

Components Vs. Objects

1. Components are supposed to be self-contained units, platform independent, and independently deployable.
   - Objects are usually not executable by themselves

Components Vs. Objects

1. Components are supposed to be self-contained units, platform independent, and independently deployable.
   - Objects are usually not executable by themselves
2. A component may contain many objects which are encapsulated and thus are not accessible from the outer of the components
   - If an object creates another object inside a component, this new object is not visible from the outside unless explicitly allowed by the interface

Components Vs. Objects

1. Components are supposed to be self-contained units, platform independent, and independently deployable.
   - Objects are usually not executable by themselves
2. A component may contain many objects which are encapsulated and thus are not accessible from the outer of the components
   - If an object creates another object inside a component, this new object is not visible from the outside unless explicitly allowed by the interface
3. Components are unique and cannot be copied within one system
   - There might exists many copies of an object

Components Vs. Objects

1. Components are supposed to be self-contained units, platform independent, and independently deployable.
   - Objects are usually not executable by themselves
2. A component may contain many objects which are encapsulated and thus are not accessible from the outer of the components
   - If an object creates another object inside a component, this new object is not visible from the outside unless explicitly allowed by the interface
3. Components are unique and cannot be copied within one system
   - There might exists many copies of an object
4. Components are static entities representing the main elements of the run-time structure
   - Classes can be instantiated dynamically in any number
   - A purely class-oriented program does not identify the main elements of a system

Why Components?

Four main "levels" of reasons:

1. "Make and buy"
   - Balance between purpose-built software and standard software
2. Reuse partial design and implementation fragments across multiple solutions or products
3. Use components from multiple sources, and integrate them on site (i.e., not part of the software build process)
   - The integration is called *deployment*
   - The matching components are called *deployable components*
4. Achieve highly dynamic servicing, upgrading, extension, and integration of deployed systems

Challenges

- Practical use of components stop in the third reason above
  - Truly dynamic components needs to address correctness, robustness and efficiency
- Components can be combined in many ways
  - No possibility to perform exhaustive and final integration tests at the component supplier's site
  - Verification of component properties are crucial
  - A compositional reasoning at all levels is required

Challenges

- Practical use of components stop in the third reason above
  - Truly dynamic components needs to address correctness, robustness and efficiency
- Components can be combined in many ways
  - No possibility to perform exhaustive and final integration tests at the component supplier's site
  - Verification of component properties are crucial
  - A compositional reasoning at all levels is required

Remark

A correct component is 100% reliable

A component with a very slight defect is 100% unreliable!

Components and Contracts I

- In "traditional" component-based development, contracts are understood as specification attached to interfaces
  - Behavioral interfaces instead of static interfaces
- A four-level approach for contract awareness has been proposed in [BJP+99]
  1. Basic contracts
  2. Behavioral contracts
  3. Synchronization contracts
  4. Quality-of-service contracts

[BJP+99] A. Beugnard, J.-M. Jézequel, N. Plouzeau and D. Watkins. "Making Components Contract Aware".

- These basic contracts specify static behavior
  - It determines the **signature** or the **interface**
- The designer specify
  - The operations a component can perform
  - The input and output parameters
  - Possible exceptions raised during operation

- Contract on static properties are limited and it does not deal with dynamic interactions
- Behavioral contracts use invariants, pre- and post-conditions, as in the "design-by-contract" approach
- The contract carries mutual obligations and benefits for both provider and user of a routine/method
- The behavioral specification could be seen as the contract itself

- Level 2 (behavioral) contracts assume interactions are atomic or executed as transactions
- Synchronization contracts specify global behavior of components
  - In terms of synchronizations between method calls
  - It describes dependencies: sequence, parallelism, etc
- In a (concurrent) multi-client setting, the contract guarantees that whatever is requested it will be executed correctly
  - It requires a synchronization policy
  - E.g. when mutual exclusion is necessary

- The previous levels allow to qualify behavioral contractual properties
- Quality-of-Service Contracts allows to specify quantitative contractual issues
- Examples of quality-of-service parameters
  - Maximum response delay
  - Average response
  - Precision of quality of a result
  - Statistics
- The problem is how to enforce such contracts
  - "Observing" such quantitative issues may involve the use of monitors affecting the behavior

Components and Contracts II

- What we have seen was a hierarchical **classification** of contracts
- There is no mention of how to analyze such contracts

Our Proposal

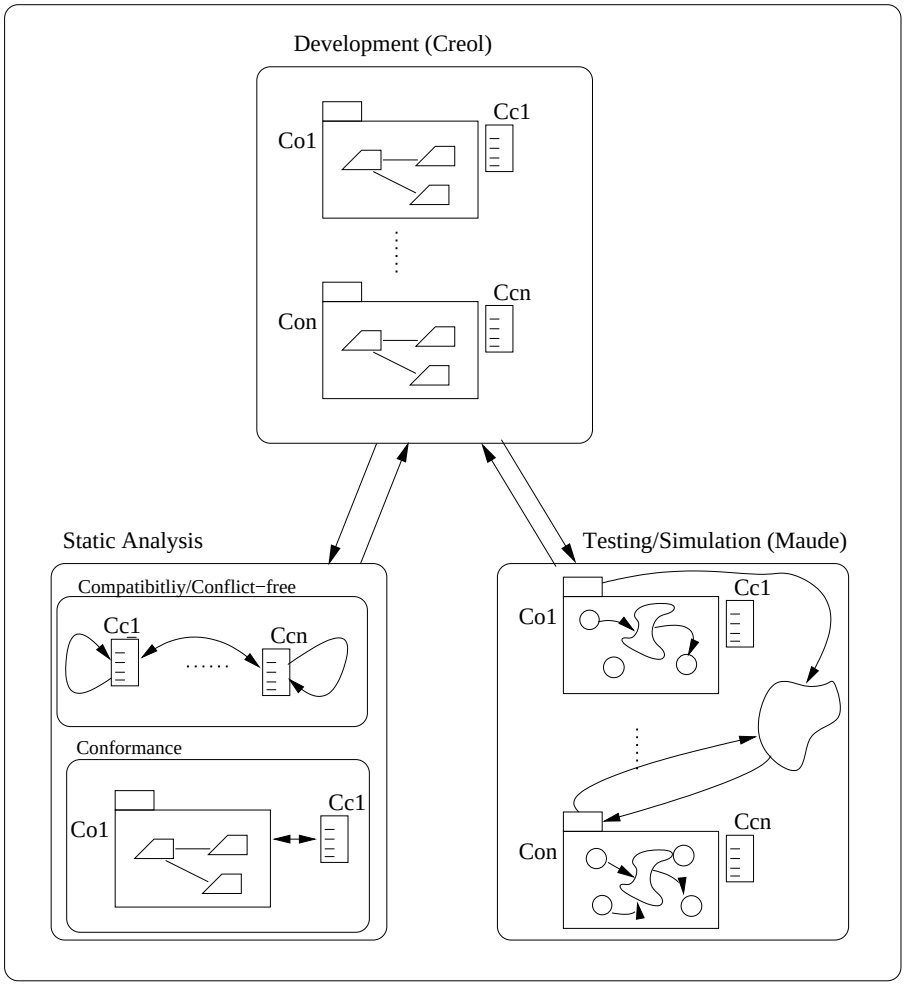
We propose the use of 'deontic' e-contracts to help verification of and reasoning about components

- To be used both at the development and deployment phases

Components and Contracts II

Development Phase

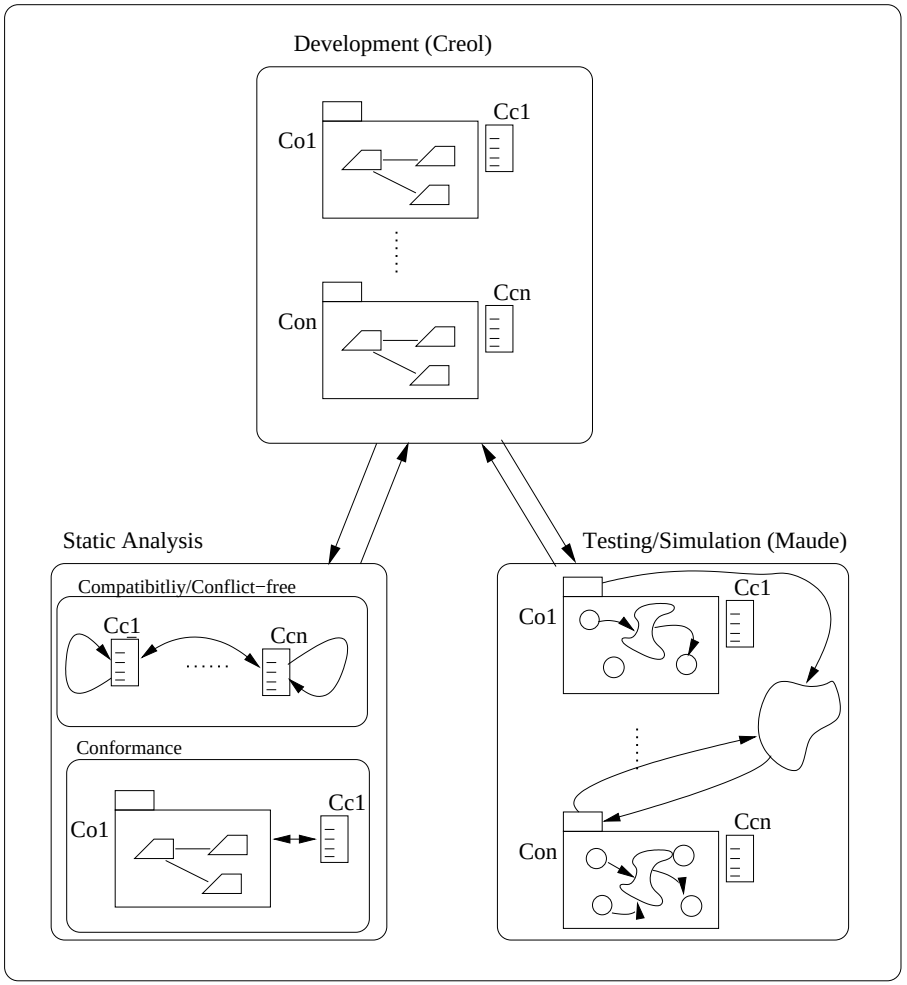
- Development: Associate one or more contracts to each component, specifying the obligations, permissions, and prohibitions in the component's interacting behavior



Components and Contracts II

Development Phase

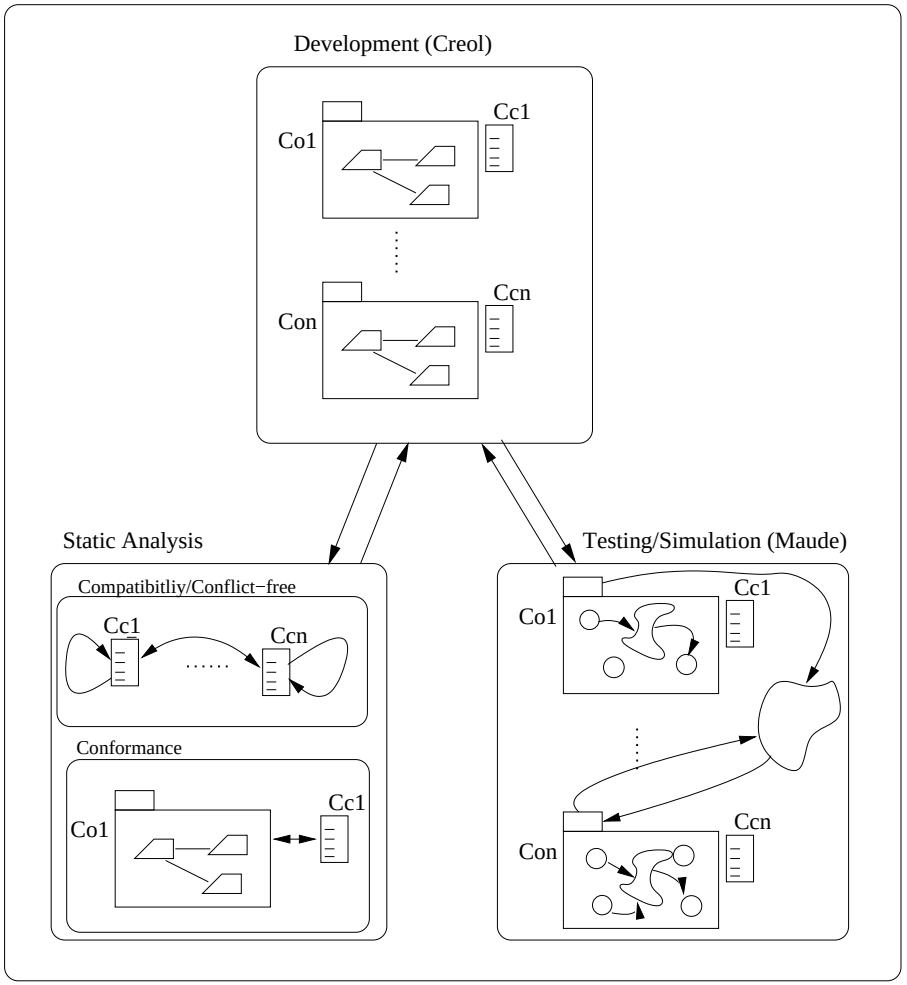
- Development: Associate one or more contracts to each component, specifying the obligations, permissions, and prohibitions in the component's interacting behavior
- Static Analysis: The contract is formally analyzed to guarantee that it is contradiction free. Static conformance between the component and its contract is also proved.



Components and Contracts II

Development Phase

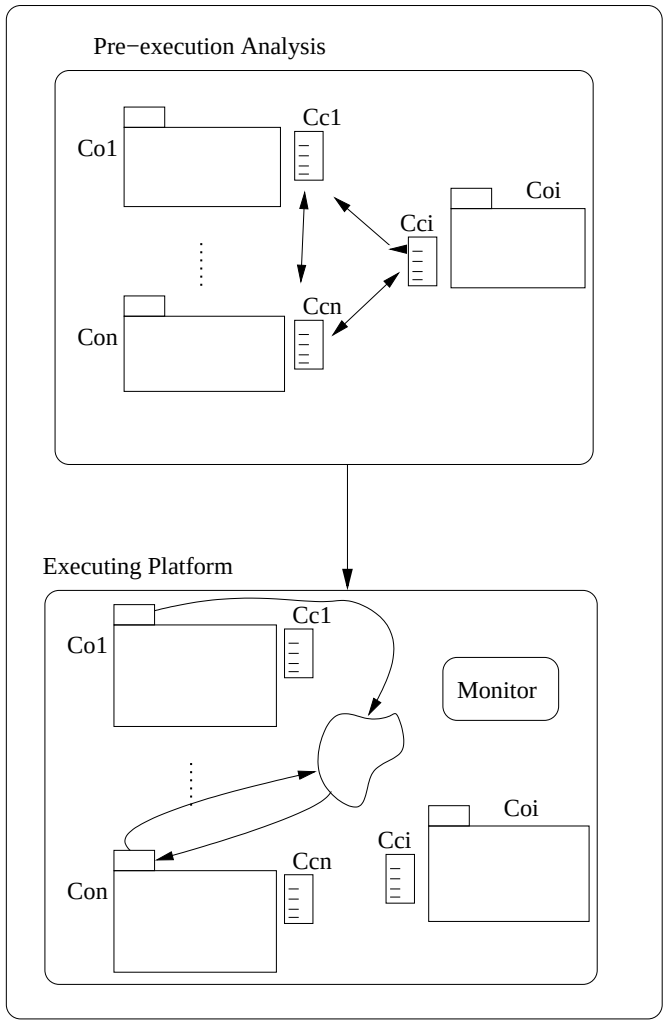
- Development: Associate one or more contracts to each component, specifying the obligations, permissions, and prohibitions in the component's interacting behavior
- Static Analysis: The contract is formally analyzed to guarantee that it is contradiction free. Static conformance between the component and its contract is also proved.
- Testing/Simulation: Simulate and test each component separately and its interaction with other components being developed



Components and Contracts II

Deployment Phase

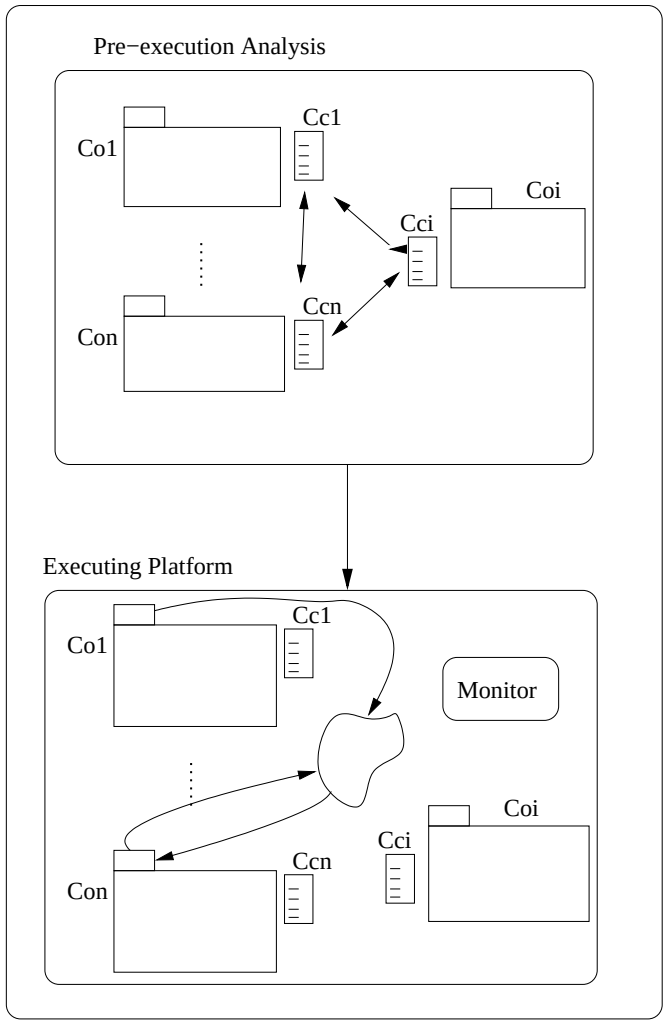
- Pre-execution Analysis:
  – Before composition the contracts are checked to guarantee compatibility
  – If disagreement: a phase of negotiation may start, or the component is simply rejected
  – Kind of static analysis on the side of the execution platform



Components and Contracts II

Deployment Phase

- Pre-execution Analysis:
  – Before composition the contracts are checked to guarantee compatibility
  – If disagreement: a phase of negotiation may start, or the component is simply rejected
  – Kind of static analysis on the side of the execution platform
- Execution:
  – If accepted, component is deployed
  – A monitor guarantees that the components behave according to the contract
  – In case of contract violation, the monitor acts as stipulated in the contract, or cancel the contract and disable the component



Plan

What is a Service?

Definition

- A service is a self-describing, platform-independent computational element

- It supports rapid, low-cost composition of distributed applications
- It allows to offer competences over intra-nets or the Internet using standard languages (e.g., XML-based) and protocols

What is a Service?

Definition

- A service is a self-describing, platform-independent computational element

- It supports rapid, low-cost composition of distributed applications
- It allows to offer competences over intra-nets or the Internet using standard languages (e.g., XML-based) and protocols
- Services must be
  - Technology neutral: Invocation mechanisms should comply with standards
  - Loosely coupled: Not require any knowledge, internal structure, nor context at the client or service side
  - Locally transparent: Have their definition and local information stored in repositories accessible independent of their location

What is a Service?

Definition

- A service is a self-describing, platform-independent computational element

- It supports rapid, low-cost composition of distributed applications
- It allows to offer competences over intra-nets or the Internet using standard languages (e.g., XML-based) and protocols
- Services must be
  - Technology neutral: Invocation mechanisms should comply with standards
  - Loosely coupled: Not require any knowledge, internal structure, nor context at the client or service side
  - Locally transparent: Have their definition and local information stored in repositories accessible independent of their location
- Services may be
  - Simple
  - Composite

Service-Oriented Computing

Definition

"Service-Oriented Computing (SOC) is the computing paradigm that utilizes services as fundamental elements for developing applications / solutions."

"To build the service model, SOC relies on the Service-Oriented Architecture (SOA), which is a way of reorganizing software applications and infrastructure into a set of interacting services."

(*) From "Service-Oriented Computing: Concepts, Characteristics and Directions", by Mike P. Papazoglou

Services vs. Components

Services and Components

- Payment of services is on execution basis (*per-use value*) for the delivery of the service
  - In components, there is a one-time payment for the implementation of the software
- Services may be a non-component implementation
  - A deployed component may offer one or more services

A Taxonomy of SOA Contract Specification Languages

- We will follow the taxonomy proposed in [OR08]
  - Services seen abstractly as Mealy machines
- Three broad families of languages and standards to deal with service contracts —Those dealing with:
  1. Web services
  2. Semantic Web services
  3. Electronic business

[OR08] J.C. Okika and A.P. Ravn. A Taxonomy of SOA Contract Specification Languages.

A Taxonomy of SOA Contract Specification Languages

- We will follow the taxonomy proposed in [OR08]
  - Services seen abstractly as Mealy machines
- Three broad families of languages and standards to deal with service contracts —Those dealing with:
  1. Web services
  2. Semantic Web services
  3. Electronic business

Some Preliminaries:

Definition

An ontology is a formal representation of a set of concepts within a domain and the relationships between those concepts. It is used to reason about the properties of that domain, and may be used to define the domain

[OR08] J.C. Okika and A.P. Ravn. A Taxonomy of SOA Contract Specification Languages.

A Taxonomy of SOA Contract Specification Languages

Examples of the three families: (1) Web services; (2) Semantic Web services; (3) Electronic business

Example

1. Web Service Definition Language (WSDL)
   - An XML-based language
   - Describes capabilities of WS through its interface description
   - Others: WS-BPEL, WS-CDL, WS-Security, WSLA, WS-Policy
2. Semantic Markup for Web Services (OWL-S)
   - Built on top of the Ontology Web Language (OWL)
   - An ontology of services for to discover, invoke, compose, and monitor Web resources offering particular services
3. Business Process Specification Schema (BPSS)
   - A framework to support execution of business collaborations consisting of business transactions
   - It supports the specification of business transactions
   - Other examples: ebXML, CPP, CPA

- Interface: Defines the (syntactic) interaction between services (or between a service provider and consumer)
- Functionality: What the service can do for a user
- Preconditions: What must be true when the service is called
- Post-conditions: Which guarantees hold when the service is done
- Protocol: Describes the input events, the response of the service to those events, signals and messages
- Security: Techniques and practices ensuring confidentiality properties for a service
- Extra functional properties
  - Performance: Measure in terms of:
    - Throughput: Nr. of requests served a at a given time period
    - Latency: Round-trip time between sending-receiving
  - Reliability: Capability of keeping the service in operation (and service quality)
  - Availability: Whether the service is ready for immediate use

A Taxonomy of SOA Contract Specification Languages

	Web Services	Semantic Web	Electronic Business
Interface	WSDL	OWL-S	ebBSI
Functionality	WS-BPEL, WSOL	OWL-S, WSMO	ebBPSS
Protocol	WS-BPEL, WS-CDL	OWL-S, WSMO	ebBPSS
Security	WS-Security	OWL-S	ebCPA
Policy	WS-Policy	OWL-S	
Trust	WS-Trust		ebCPP (XMLDSIG)
Availability	WSOL		
Performance	WSLA, WSOL	WSMO, WSML	ebCPA
Response Time			
Throughput			

[OR08] J.C. Okika and A.P. Ravn. A Taxonomy of SOA Contract Specification Languages.

Services and Contracts

- In the above taxonomy languages were classified according to many aspects
- None of them covers all the aspects

Services and Contracts

- In the above taxonomy languages were classified according to many aspects
- None of them covers all the aspects

Challenges

- How to obtain a general language for describing service contracts
- How to reason about service contracts
- How to address (automatic) negotiation
- How to enforce the fulfillment of the contract
- How to describe normal and exceptional behavior

Services and Contracts

- In the above taxonomy languages were classified according to many aspects
- None of them covers all the aspects

Challenges

- How to obtain a general language for describing service contracts
- How to reason about service contracts
- How to address (automatic) negotiation
- How to enforce the fulfillment of the contract
- How to describe normal and exceptional behavior

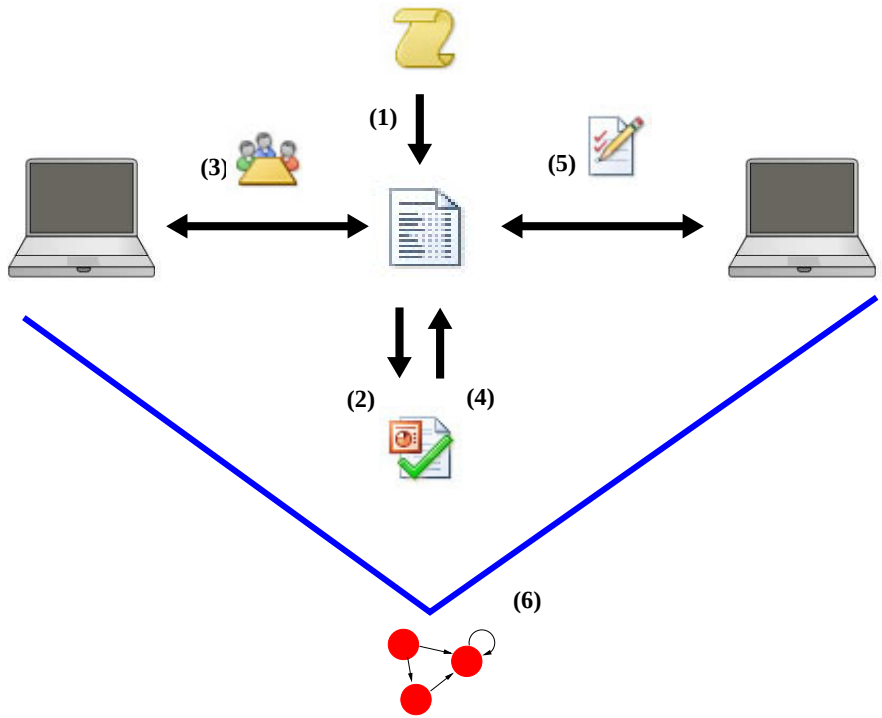
Observation

We propose the use of 'deontic' e-contracts to help specification of and reasoning about services

Such contracts may also be useful in the negotiation process

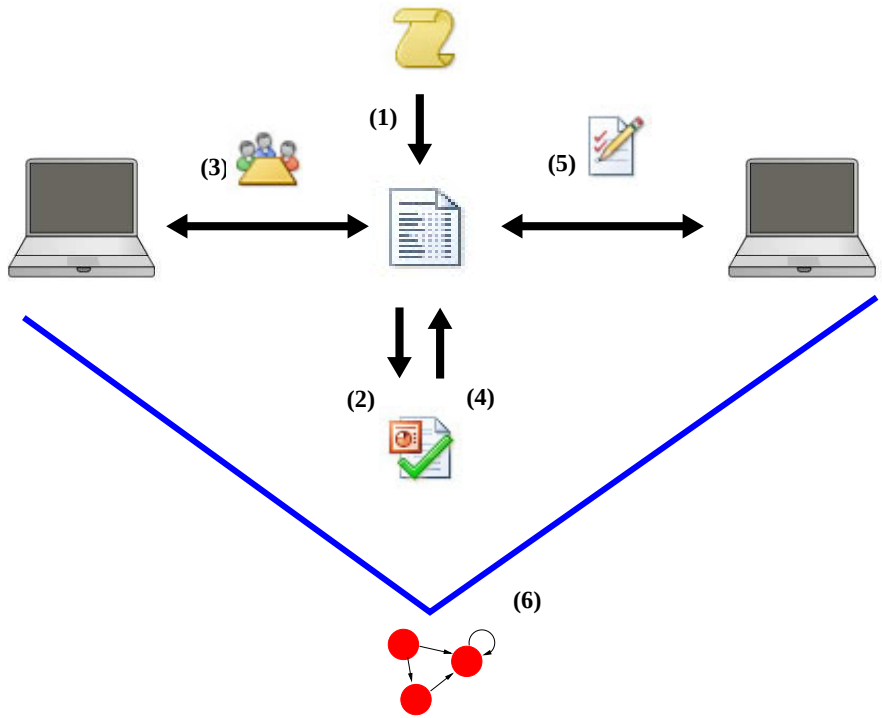
Services and Contracts

1. Translate the informal contract into a formal language



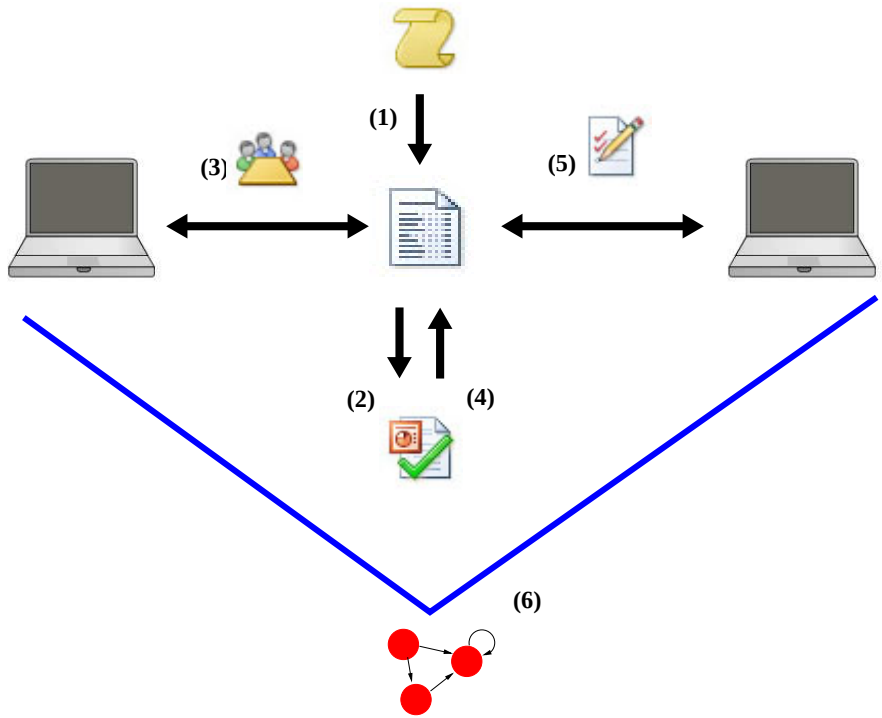
Services and Contracts

1. Translate the informal contract into a formal language
2. Verify the contract (e-g-, that it is contradiction-free)



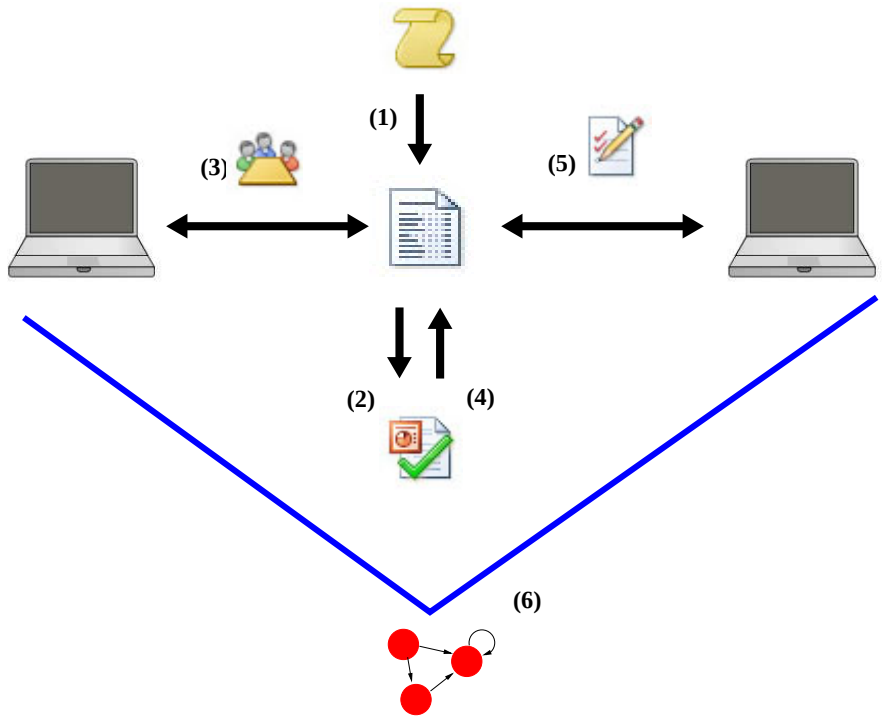
Services and Contracts

1. Translate the informal contract into a formal language
2. Verify the contract (e-g-, that it is contradiction-free)
3. Negotiate the contract



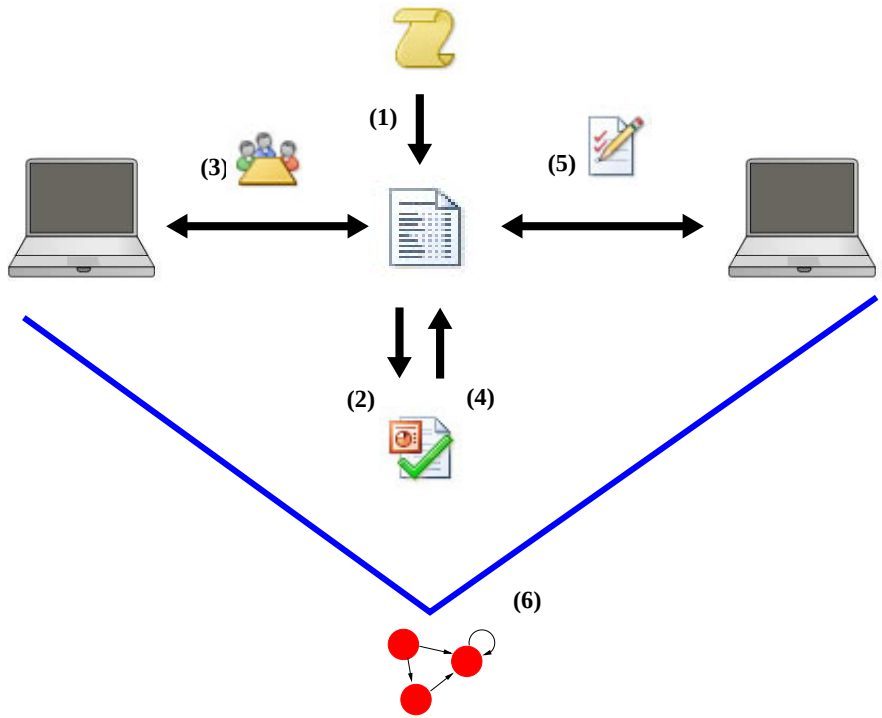
Services and Contracts

1. Translate the informal contract into a formal language
2. Verify the contract (e-g-, that it is contradiction-free)
3. Negotiate the contract
4. After negotiation verify the contract again



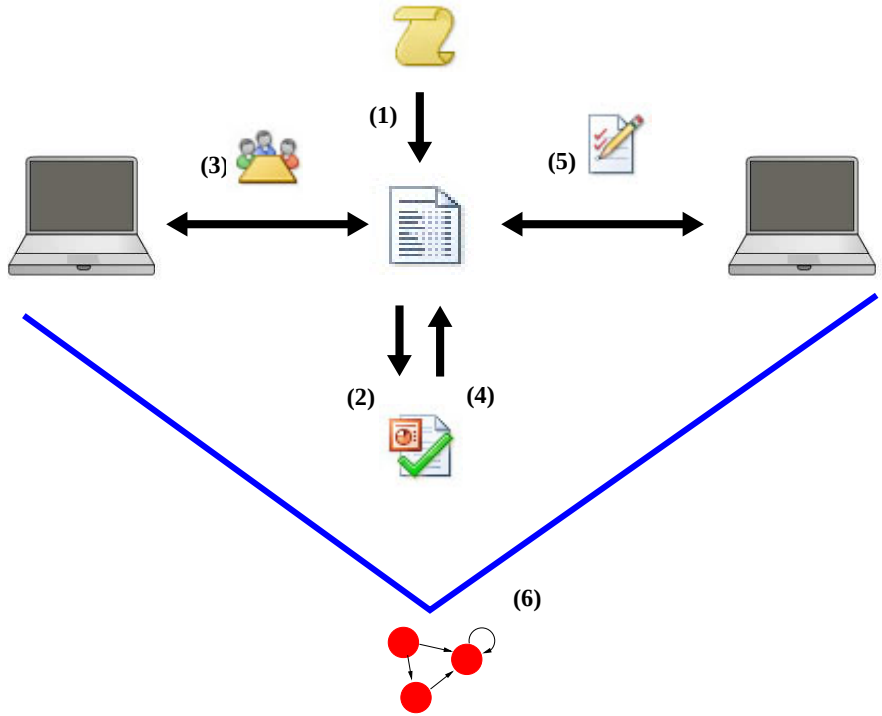
Services and Contracts

1. Translate the informal contract into a formal language
2. Verify the contract (e-g-, that it is contradiction-free)
3. Negotiate the contract
4. After negotiation verify the contract again
5. Obtain the final contract and "sign" it



Services and Contracts

1. Translate the informal contract into a formal language
2. Verify the contract (e-g-, that it is contradiction-free)
3. Negotiate the contract
4. After negotiation verify the contract again
5. Obtain the final contract and "sign" it
6. Monitor/enforce contract fulfillment



Further Reading

- C. Szyperski. **Component Technology - What, Where, and How?**
- A. Beugnard, J.-M. Jézequel, N. Plouzeau and D. Watkins. **Making Components Contract Aware**
- M. Papazoglou. **Service-Oriented Computing: Concepts, Characteristics and Directions**
- O. Owe, G. Schneider and M. Steffen. **Objects, Components and Contracts**
- J.C. Okika and A.P. Ravn. **A Taxonomy of SOA Contract Specification Languages**
- COSoDIS project: `http://www.ifi.uio.no/cosodis`