Specification and Analysis of Contracts
Lecture 1
Introduction

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

1. Formal Methods
2. Contracts ‘and’ Informatics

Plan

1. Formal Methods
2. Contracts ‘and’ Informatics

How to Guarantee Correctness?

Is it possible at all?

- How to show a system is correct?
  - It is not enough to show that it can meet its requirement
  - We should show that a system cannot fail to meet its requirement
  - By testing? Dijkstra wrote (1972): "Program testing can be used to show the presence of bugs, but never to show their absence"
  - By other kind of "proof"? Dijkstra again (1965): "One can never guarantee that a proof is correct, the best one can say is: 'I have not discovered any mistakes"'
  - What about automatic proof? It is impossible to construct a general proof procedure for arbitrary programs[1]
- Any hope?

[1] Undecidability of the halting problem, by Turing.

How to Guarantee Correctness?

Is it possible at all?

- How to show a system is correct?
  - It is not enough to show that it can meet its requirement
  - We should show that a system cannot fail to meet its requirement
  - By testing? Dijkstra wrote (1972): "Program testing can be used to show the presence of bugs, but never to show their absence"
  - By other kind of "proof"? Dijkstra again (1965): "One can never guarantee that a proof is correct, the best one can say is: 'I have not discovered any mistakes'"
  - What about automatic proof? It is impossible to construct a general proof procedure for arbitrary programs[1]
- Any hope?

[1] Undecidability of the halting problem, by Turing.

How to Guarantee Correctness?

Is it possible at all?

- How to show a system is correct?
  - It is not enough to show that it can meet its requirement
  - We should show that a system cannot fail to meet its requirement
  - By testing? Dijkstra wrote (1972): "Program testing can be used to show the presence of bugs, but never to show their absence"
  - By other kind of "proof"? Dijkstra again (1965): "One can never guarantee that a proof is correct, the best one can say is: 'I have not discovered any mistakes"'
  - What about automatic proof? It is impossible to construct a general proof procedure for arbitrary programs[1]
- Any hope? In some cases it is possible to mechanically verify correctness;

[1] Undecidability of the halting problem, by Turing.

How to Guarantee Correctness?

Is it possible at all?

- How to show a system is correct?
  - It is not enough to show that it can meet its requirement
  - We should show that a system cannot fail to meet its requirement
  - By testing? Dijkstra wrote (1972): "Program testing can be used to show the presence of bugs, but never to show their absence"
  - By other kind of "proof"? Dijkstra again (1965): "One can never guarantee that a proof is correct, the best one can say is: 'I have not discovered any mistakes"'
  - What about automatic proof? It is impossible to construct a general proof procedure for arbitrary programs[1]
- Any hope? In some cases it is possible to mechanically verify correctness; in other cases... we try to do our best

[1] Undecidability of the halting problem, by Turing.

System Correctness

- A system is correct if it meets its design requirements

System Correctness

- A system is correct if it meets its design requirements

Example

- **System:** A telephone system
  Requirement: If user A want to call user B, then eventually (s)he will manage to establish a connection

---

a

System Correctness

- A system is correct if it meets its design requirements

Example

- **System:** A telephone system
  Requirement: If user A want to call user B, then eventually (s)he will manage to establish a connection
- **System:** An operating system
  Requirement: A deadly embrace[a] will never happen

[a] A deadly embrace is entered when two processes obtain access to two mutually dependent shared resources and each decide to wait indefinitely for the other

System Correctness

- A system is correct if it meets its design requirements

Example

- **System:** A telephone system
  Requirement: If user A want to call user B, then eventually (s)he will manage to establish a connection
- **System:** An operating system
  Requirement: A deadly embrace[a] will never happen
- **System:** A contract for Internet services
  Requirement: Signatory A will never be obliged to pay more than a certain amount of money

[a] A deadly embrace is entered when two processes obtain access to two mutually dependent shared resources and each decide to wait indefinitely for the other

System Correctness

- A system is correct if it meets its design requirements

Example

- **System:** A telephone system
  Requirement: If user A want to call user B, then eventually (s)he will manage to establish a connection
- **System:** An operating system
  Requirement: A deadly embrace[a] will never happen
- **System:** A contract for Internet services
  Requirement: Signatory A will never be obliged to pay more than a certain amount of money

[a] A deadly embrace is entered when two processes obtain access to two mutually dependent shared resources and each decide to wait indefinitely for the other

Saying 'a program is correct' is only meaningful w.r.t. a given specification!

What is Validation?

- In general, validation is the process of checking if something satisfies a certain criterion
- Some authors differentiate **validation** and **verification**:

Validation: "Are we building the right product?", i.e., does the product do what the user really requires

Verification: "Are we building the product right?", i.e., does the product conform to the specifications

What is Validation?

- In general, validation is the process of checking if something satisfies a certain criterion
- Some authors differentiate **validation** and **verification**:

Validation: "Are we building the right product?", i.e., does the product do what the user really requires

Verification: "Are we building the product right?", i.e., does the product conform to the specifications

What is Validation?

- In general, validation is the process of checking if something satisfies a certain criterion
- Some authors differentiate **validation** and **verification**:

Validation: "Are we building the right product?", i.e., does the product do what the user really requires

Verification: "Are we building the product right?", i.e., does the product conform to the specifications

Remark

Some authors define verification as a validation technique, others talk about V & V –Validation & Verification– as being complementary techniques. In this tutorial I consider verification as a validation technique

Usual Approaches for Validation

The following techniques are used in industry for validation:

- Testing
  - Check the actual system rather than a model
  - Focused on sampling executions according to some coverage criteria – Not exhaustive
  - It is usually informal, though there are some formal approaches
- Simulation
  - A model of the system is written in a PL, which is run with different inputs – Not exhaustive
- Verification
  - "Is the process of applying a manual or automatic technique for establishing whether a given system satisfies a given property or behaves in accordance to some abstract description (specification) of the system"[2]

[2] From Peled's book "Software reliability methods".

Usual Approaches for Validation

The following techniques are used in industry for validation:

- Testing
  - Check the actual system rather than a model
  - Focused on sampling executions according to some coverage criteria – Not exhaustive
  - It is usually informal, though there are some formal approaches
- Simulation
  - A model of the system is written in a PL, which is run with different inputs – Not exhaustive
- Verification
  - "Is the process of applying a manual or automatic technique for establishing whether a given system satisfies a given property or behaves in accordance to some abstract description (specification) of the system"[2]

[2] From Peled's book "Software reliability methods".

Usual Approaches for Validation

The following techniques are used in industry for validation:

- Testing
  - Check the actual system rather than a model
  - Focused on sampling executions according to some coverage criteria – Not exhaustive
  - It is usually informal, though there are some formal approaches
- Simulation
  - A model of the system is written in a PL, which is run with different inputs – Not exhaustive
- Verification
  - "Is the process of applying a manual or automatic technique for establishing whether a given system satisfies a given property or behaves in accordance to some abstract description (specification) of the system"[2]

[2] From Peled's book "Software reliability methods".

Usual Approaches for Validation

The following techniques are used in industry for validation:

- Testing
  - Check the actual system rather than a model
  - Focused on sampling executions according to some coverage criteria – Not exhaustive
  - It is usually informal, though there are some formal approaches
- Simulation
  - A model of the system is written in a PL, which is run with different inputs – Not exhaustive
- Verification
  - "Is the process of applying a manual or automatic technique for establishing whether a given system satisfies a given property or behaves in accordance to some abstract description (specification) of the system"[2]

[2] From Peled's book "Software reliability methods".

What are Formal Methods?

- "Formal methods are a collection of notations and techniques for describing and analyzing systems"[3]
- Formal means the methods used are based on mathematical theories, such as logic, automata, graph or set theory
- Formal specification techniques are used to unambiguously describe the system itself or its properties
- Formal analysis/verification techniques serve to verify that a system satisfies its specification (or to help finding out why it is not the case)

[3] From D.Peled's book "Software Reliability Methods".

What are Formal Methods?

- "Formal methods are a collection of notations and techniques for describing and analyzing systems"[3]
- Formal means the methods used are based on mathematical theories, such as logic, automata, graph or set theory
- Formal specification techniques are used to unambiguously describe the system itself or its properties
- Formal analysis/verification techniques serve to verify that a system satisfies its specification (or to help finding out why it is not the case)

[3] From D.Peled's book "Software Reliability Methods".

What are Formal Methods?

- "Formal methods are a collection of notations and techniques for describing and analyzing systems"[3]
- Formal means the methods used are based on mathematical theories, such as logic, automata, graph or set theory
- Formal specification techniques are used to unambiguously describe the system itself or its properties
- Formal analysis/verification techniques serve to verify that a system satisfies its specification (or to help finding out why it is not the case)

[3] From D.Peled's book "Software Reliability Methods".

Limitations

- Software verification methods do not guarantee, in general, the correctness of the code itself but rather of an abstract model of it
- It cannot identify fabrication faults (e.g. in digital circuits)
- If the specification is incomplete or wrong, the verification result will also be wrong
- The implementation of verification tools may be faulty
- The bigger the system (number of possible states) more difficult is to analyze it (*state explosion problem*)

Any advantage?

Any advantage?

OF COURSE!!

Formal methods are not intended to guarantee absolute reliability but to *increase* the confidence on system reliability. They help minimizing the number of errors and in many cases allow to find errors impossible to find manually

Using Formal Methods

Formal methods are used in different stages of the development process, giving a classification of formal methods

1. We describe the system giving a formal specification
2. We can then prove some properties about the specification (formal verification)
3. We can proceed by:
   - Deriving a program from its specification (formal synthesis)
   - Verifying the specification w.r.t. implementation (formal verification)

Formal Specification

- A specification formalism must be unambiguous: it should have a precise syntax and semantics (e.g., natural languages are not suitable)
- A trade-off must be found between expressiveness and analysis feasibility
    - More expressive the specification formalism more difficult its analysis (if possible at all)

Proving Properties about the Specification

To gain confidence about the correctness of a specification it is useful to:

- Prove some properties of the specification to check that it really means what it is supposed to
- Prove the equivalence of different specifications

Proving Properties about the Specification

To gain confidence about the correctness of a specification it is useful to:

- Prove some properties of the specification to check that it really means what it is supposed to
- Prove the equivalence of different specifications

Example

- a should be true for the first two points of time, and then oscillates
  - First attempt: $a(0) \wedge a(1) \wedge \forall t \cdot a(t+1) = \neg a(t)$

Proving Properties about the Specification

To gain confidence about the correctness of a specification it is useful to:

- Prove some properties of the specification to check that it really means what it is supposed to
- Prove the equivalence of different specifications

Example

- *a* should be true for the first two points of time, and then oscillates
  - First attempt: $a(0) \wedge a(1) \wedge \forall t \cdot a(t+1) = \neg a(t)$
    INCORRECT! - The error may be found when trying to prove some properties

Proving Properties about the Specification

To gain confidence about the correctness of a specification it is useful to:

- Prove some properties of the specification to check that it really means what it is supposed to
- Prove the equivalence of different specifications

Example

- a should be true for the first two points of time, and then oscillates
  - First attempt: $a(0) \wedge a(1) \wedge \forall t \cdot a(t+1) = \neg a(t)$
    INCORRECT! - The error may be found when trying to prove some properties
  - Correct specification: $a(0) \wedge a(1) \wedge \forall t \geq 0 \cdot a(t+2) = \neg a(t+1)$

Formal synthesis

- It would be helpful to automatically obtain an implementation from the specification of a system
- Difficult since most specifications are *declarative* and not *constructive*
  - They usually describe **what** the system should do; not **how** it can be achieved

Formal synthesis

- It would be helpful to automatically obtain an implementation from the specification of a system
- Difficult since most specifications are *declarative* and not *constructive*
  - They usually describe **what** the system should do; not **how** it can be achieved

Example

1. Specify the operational semantics of a programming language in a constructive logic (Calculus of Constructions)
2. Prove the correctness of a given property w.r.t. the operational semantics in Coq
3. Extract an OCAML code from the correctness proof (using Coq's extraction mechanism)

Verifying Specifications w.r.t. Implementations

There are mainly two approaches:

- Deductive approach (automated theorem proving)
  - Describe the specification Φ_{spec} in a formal model (logic)
  - Describe the system's model Φ_{imp} in the same formal model
  - Prove that $\Phi_{imp} \implies \Phi_{spec}$
- Algorithmic approach
  - Describe the specification Φ_{spec} as a formula of a logic
  - Describe the system as an interpretation M_{imp} of the given logic (e.g. as a finite automaton)
  - Prove that M_{imp} is a "model" (in the logical sense) of Φ_{spec}

Verifying Specifications w.r.t. Implementations

There are mainly two approaches:

- Deductive approach (automated theorem proving)
  - Describe the specification Φ_{spec} in a formal model (logic)
  - Describe the system's model Φ_{imp} in the same formal model
  - Prove that $\Phi_{imp} \implies \Phi_{spec}$
- Algorithmic approach
  - Describe the specification Φ_{spec} as a formula of a logic
  - Describe the system as an interpretation M_{imp} of the given logic (e.g. as a finite automaton)
  - Prove that M_{imp} is a "model" (in the logical sense) of Φ_{spec}

Remark

The same technique may be used to prove properties about the specification

When and Which Formal Method to Use?

- It depends on the problem, the underlying system and the property we want to prove
  Examples:
  - Digital circuits ... (BDDs, model checking)
  - Communication protocol with unbounded number of processes.... (verification of infinite-state systems)
  - Overflow in programs (static analysis and abstract interpretation)
  - ...
- Open distributed concurrent systems with unbounded number of processes interacting through shared variables and with real-time constraints => VERY DIFFICULT!!
  Need the combination of different techniques

When and Which Formal Method to Use?

- It depends on the problem, the underlying system and the property we want to prove
  Examples:
  - Digital circuits ... (BDDs, model checking)
  - Communication protocol with unbounded number of processes.... (verification of infinite-state systems)
  - Overflow in programs (static analysis and abstract interpretation)
  - ...
- Open distributed concurrent systems with unbounded number of processes interacting through shared variables and with real-time constraints => VERY DIFFICULT!!
  Need the combination of different techniques

When and Which Formal Method to Use?

- It depends on the problem, the underlying system and the property we want to prove
  Examples:
  - Digital circuits ... (BDDs, model checking)
  - Communication protocol with unbounded number of processes.... (verification of infinite-state systems)
  - Overflow in programs (static analysis and abstract interpretation)
  - ...
- Open distributed concurrent systems with unbounded number of processes interacting through shared variables and with real-time constraints => VERY DIFFICULT!!
  Need the combination of different techniques

Remark

In this tutorial: Specification and verification of contracts using logics and *model checking* techniques

Plan

1. Formal Methods

2. Contracts ‘and’ Informatics

Contracts

- "A contract is a binding agreement between two or more persons that is enforceable by law." [Webster on-line]

- "A contract is a binding agreement between two or more persons that is enforceable by law." [Webster on-line]

This deed of **Agreement** is made between:
1. **[name]**, from now on referred to as **Provider** and
2. the **Client**.

INTRODUCTION

3. The **Provider** is obliged to provide the **Internet Services** as stipulated in this **Agreement**.
4. DEFINITIONS

a) **Internet traffic** may be measured by both **Client** and **Provider** by means of Equipment and may take the two values **high** and **normal**.

OPERATIVE PART

1. The **Client** shall not supply false information to the Client Relations Department of the **Provider**.
2. Whenever the Internet Traffic is **high** then the **Client** must pay [*price*] immediately, or the **Client** must notify the **Provider** by sending an e-mail specifying that he will pay later.
3. If the **Client** delays the payment as stipulated in 2, after notification he must immediately lower the Internet traffic to the **normal** level, and pay later twice ($2 *$ [*price*]).
4. If the **Client** does not lower the Internet traffic immediately, then the **Client** will have to pay $3 *$ [*price*].
5. The **Client** shall, as soon as the Internet Service becomes operative, submit within seven (7) days the Personal Data Form from his account on the **Provider**'s web page to the Client Relations Department of the **Provider**.

Contracts and Informatics

1. Conventional contracts
    - Traditional commercial and judicial domain

Contracts and Informatics

1. Conventional contracts
    - Traditional commercial and judicial domain
2. "Programming by contract" or "Design by contract" (e.g., Eiffel)
    - Relation between pre- and post-conditions of routines, method calls, invariants, temporal dependencies, etc

Contracts and Informatics

1. Conventional contracts
   - Traditional commercial and judicial domain
2. "Programming by contract" or "Design by contract" (e.g., Eiffel)
   - Relation between pre- and post-conditions of routines, method calls, invariants, temporal dependencies, etc
3. In the context of web services (SOA)
   - Service-Level Agreement, usually written in an XML-like language (e.g. WSLA)

Contracts and Informatics

1. Conventional contracts
   - Traditional commercial and judicial domain
2. "Programming by contract" or "Design by contract" (e.g., Eiffel)
   - Relation between pre- and post-conditions of routines, method calls, invariants, temporal dependencies, etc
3. In the context of web services (SOA)
   - Service-Level Agreement, usually written in an XML-like language (e.g. WSLA)
4. Behavioral interfaces
   - Specify the sequence of interactions between different participants. The allowed interactions are captured by legal (sets of) traces

Contracts and Informatics

1. Conventional contracts
   - Traditional commercial and judicial domain
2. "Programming by contract" or "Design by contract" (e.g., Eiffel)
   - Relation between pre- and post-conditions of routines, method calls, invariants, temporal dependencies, etc
3. In the context of web services (SOA)
   - Service-Level Agreement, usually written in an XML-like language (e.g. WSLA)
4. Behavioral interfaces
   - Specify the sequence of interactions between different participants. The allowed interactions are captured by legal (sets of) traces
5. Contractual protocols
   - To specify the interaction between communicating entities

Contracts and Informatics

1. Conventional contracts
    - Traditional commercial and judicial domain
2. "Programming by contract" or "Design by contract" (e.g., Eiffel)
    - Relation between pre- and post-conditions of routines, method calls, invariants, temporal dependencies, etc
3. In the context of web services (SOA)
    - Service-Level Agreement, usually written in an XML-like language (e.g. WSLA)
4. Behavioral interfaces
    - Specify the sequence of interactions between different participants. The allowed interactions are captured by legal (sets of) traces
5. Contractual protocols
    - To specify the interaction between communicating entities
6. "Social contracts": Multi-agent systems

Contracts and Informatics

1. Conventional contracts
   - Traditional commercial and judicial domain
2. "Programming by contract" or "Design by contract" (e.g., Eiffel)
   - Relation between pre- and post-conditions of routines, method calls, invariants, temporal dependencies, etc
3. In the context of web services (SOA)
   - Service-Level Agreement, usually written in an XML-like language (e.g. WSLA)
4. Behavioral interfaces
   - Specify the sequence of interactions between different participants. The allowed interactions are captured by legal (sets of) traces
5. Contractual protocols
   - To specify the interaction between communicating entities
6. "Social contracts": Multi-agent systems
7. "Deontic e-contracts": representing Obligations, Permissions, Prohibitions, Power, etc

Conventional contracts

- Traditional commercial and judicial domain
- Research on Law and Informatics
- Interesting questions:
  - Is it possible translate conventional contracts into formal languages/logics? Which "properties" are preserved?
  - How to analyze traditional contracts? Detect superfluous clauses, cross references, inconsistencies, etc?
  - Tools
- A nice successful story
  - Jean-Marc Eber and Simon Peyton-Jones' paper "How to write a financial contract"
  - Eber's startup: Lexifi Technologies

Conventional contracts

- Traditional commercial and judicial domain
- Research on Law and Informatics
- Interesting questions:
  - Is it possible translate conventional contracts into formal languages/logics? Which “properties” are preserved?
  - How to analyze traditional contracts? Detect superfluous clauses, cross references, inconsistencies, etc?
  - Tools
- A nice successful story
  - Jean-Marc Eber and Simon Peyton-Jones’ paper “How to write a financial contract”
  - Eber’s startup: Lexifi Technologies

Conventional contracts

- Traditional commercial and judicial domain
- Research on Law and Informatics
- Interesting questions:
  - Is it possible translate conventional contracts into formal languages/logics? Which "properties" are preserved?
  - How to analyze traditional contracts? Detect superfluous clauses, cross references, inconsistencies, etc?
  - Tools
- A nice successful story
  - Jean-Marc Eber and Simon Peyton-Jones' paper "How to write a financial contract"
  - Eber's startup: Lexifi Technologies

"Programming by contract"

- The term originated with the language Eiffel ("Programming by contract" or "Design by contract") —Used in the context of Object-Oriented Programming
- Software designers should define precise verifiable interface specifications for software components
  - Based on the theory of abstract data types
  - Inspired by business contracts
- Impose an obligation to be guaranteed when calling a module: the routine's precondition
  - An obligation for the client, and a benefit for the supplier (of the routine)
- Guarantee a property on exit: the routine's postcondition
  - An obligation for the supplier, and a benefit for the client
- Maintain a property, assumed on entry and guaranteed on exit: the class invariant

“Programming by contract”

- The term originated with the language Eiffel (“Programming by contract” or “Design by contract”) —Used in the context of Object-Oriented Programming
- Software designers should define precise verifiable interface specifications for software components
  - Based on the theory of abstract data types
  - Inspired by business contracts
- Impose an obligation to be guaranteed when calling a module: the routine’s precondition
  - An obligation for the client, and a benefit for the supplier (of the routine)
- Guarantee a property on exit: the routine’s postcondition
  - An obligation for the supplier, and a benefit for the client
- Maintain a property, assumed on entry and guaranteed on exit: the class invariant

"Programming by contract"

- The term originated with the language Eiffel ("Programming by contract" or "Design by contract") —Used in the context of Object-Oriented Programming
- Software designers should define precise verifiable interface specifications for software components
  - Based on the theory of abstract data types
  - Inspired by business contracts
- Impose an obligation to be guaranteed when calling a module: the routine's precondition
  - An obligation for the client, and a benefit for the supplier (of the routine)
- Guarantee a property on exit: the routine's postcondition
  - An obligation for the supplier, and a benefit for the client
- Maintain a property, assumed on entry and guaranteed on exit: the class invariant

Contracts in the Context of SOA

- Several SOA standards provide a way to describe 'contractual' aspects
- Usually written in an XML-like language (e.g. WSLA)
- These service contracts act at different levels, specific to different aspects
  - Between service provider and consumer
  - Orchestration of services
  - Functional aspects
  - Describe collaboration between business partners
- Service-Level Agreement
  - It describes different levels of service
    - Availability, serviceability, performance, operation, other attributes like billing and even penalties in the case of violation

Contracts in the Context of SOA

- Several SOA standards provide a way to describe 'contractual' aspects
- Usually written in an XML-like language (e.g. WSLA)
- These service contracts act at different levels, specific to different aspects
  - Between service provider and consumer
  - Orchestration of services
  - Functional aspects
  - Describe collaboration between business partners
- Service-Level Agreement
  - It describes different levels of service
    - Availability, serviceability, performance, operation, other attributes like billing and even penalties in the case of violation

Contracts in the Context of SOA

- Several SOA standards provide a way to describe 'contractual' aspects
- Usually written in an XML-like language (e.g. WSLA)
- These service contracts act at different levels, specific to different aspects
  - Between service provider and consumer
  - Orchestration of services
  - Functional aspects
  - Describe collaboration between business partners
- Service-Level Agreement
  - It describes different levels of service
    - Availability, serviceability, performance, operation, other attributes like billing and even penalties in the case of violation

Contracts in the Context of SOA

- Several SOA standards provide a way to describe 'contractual' aspects
- Usually written in an XML-like language (e.g. WSLA)
- These service contracts act at different levels, specific to different aspects
  - Between service provider and consumer
  - Orchestration of services
  - Functional aspects
  - Describe collaboration between business partners
- Service-Level Agreement
  - It describes different levels of service
    - Availability, serviceability, performance, operation, other attributes like billing and even penalties in the case of violation

Note

- We will expand on a taxonomy of SOA contract specification languages in next lecture

Behavioral interfaces

- Specify the sequence of interactions between different participants
  - Such interface represents a "contract" between the participants
- The allowed interactions are captured by legal (sets of) traces
- The behavior of objects and components can be completely defined in terms of their reaction to incoming message sequences
- Advantages:
  - Different objects and component implementations can be compared based on their behavior
  - It helps analyzing compositionality of components
  - It is possible to analyze component implementations without knowing the context

Behavioral interfaces

- Specify the sequence of interactions between different participants
  - Such interface represents a "contract" between the participants
- The allowed interactions are captured by legal (sets of) traces
- The behavior of objects and components can be completely defined in terms of their reaction to incoming message sequences
- Advantages:
  - Different objects and component implementations can be compared based on their behavior
  - It helps analyzing compositionality of components
  - It is possible to analyze component implementations without knowing the context

Contractual protocols

- Protocols may be seen as "contracts" regulating the parties' ideal mode of interaction
- Other names for the same kind of "contracts"
  - Trade procedures
  - Business protocols
  - Specifications
- This definition of a contract could be mostly seen as a specification
- Usually one specify the point of view of each party as a Finite State Machine or Petri net
  - Sometimes the 'contract' is not explicit, but implicit in the interaction of the parties

Contractual protocols

- Protocols may be seen as "contracts" regulating the parties' ideal mode of interaction
- Other names for the same kind of "contracts"
  - Trade procedures
  - Business protocols
  - Specifications
- This definition of a contract could be mostly seen as a specification
- Usually one specify the point of view of each party as a Finite State Machine or Petri net
  - Sometimes the 'contract' is not explicit, but implicit in the interaction of the parties

"Social contracts"

- Based on different modal logics
- In the context of multi-agent systems
- Aim at simulate/specify "social" behavior
  - Interaction between agents who can decide based on knowledge and trust on other agents
  - Agents act according to certain normative rules prescribing:
    - Proper and acceptable behavior (moral)
    - Acceptable behavior (legal)
- Very expressive: It considers various types of norms
  - What *ought* to be
  - Expectation on what *will* be
  - Particular *reactions* to behavior
  - Sanctions to be applied, or how to induce a particular kind of conduct

"Social contracts"

- Based on different modal logics
- In the context of multi-agent systems
- Aim at simulate/specify "social" behavior
  - Interaction between agents who can decide based on knowledge and trust on other agents
  - Agents act according to certain normative rules prescribing:
    - Proper and acceptable behavior (moral)
    - Acceptable behavior (legal)
- Very expressive: It considers various types of norms
  - What *ought* to be
  - Expectation on what *will* be
  - Particular *reactions* to behavior
  - Sanctions to be applied, or how to induce a particular kind of conduct

"Social contracts"

- Based on different modal logics
- In the context of multi-agent systems
- Aim at simulate/specify "social" behavior
  - Interaction between agents who can decide based on knowledge and trust on other agents
  - Agents act according to certain normative rules prescribing:
    - Proper and acceptable behavior (moral)
    - Acceptable behavior (legal)
- Very expressive: It considers various types of norms
  - What *ought* to be
  - Expectation on what *will* be
  - Particular *reactions* to behavior
  - Sanctions to be applied, or how to induce a particular kind of conduct

Electronic "deontic" contracts

- Based on deontic logic and combined with other modal logics
- It contains constructs to specify at least
  - Obligations, Permissions, and Prohibitions
- A contract can be obtained
  - From a conventional contract
  - Written directly in a formal specification language
- It allows formal reasoning

Contracts

In this tutorial

- We will see 'deontic' e-contracts

Two scenarios:

1. Obtain an e-contract from a conventional contract
   - Context: legal (e.g. financial) contracts
2. Write the e-contract directly in a formal language
   - Context: web services, components, OO, etc

Contracts

In this tutorial

- We will see ‘deontic’ e-contracts

Two scenarios:

1. Obtain an e-contract from a conventional contract
   - Context: legal (e.g. financial) contracts
2. Write the e-contract directly in a formal language
   - Context: web services, components, OO, etc

Definition

A contract is a document which engages several parties in a transaction and stipulates their (conditional) obligations, rights, and prohibitions, as well as penalties in case of contract violations.

Links and Papers

- Introduction to Formal Methods: See first lecture of the course "Specification and verification of parallel systems" (INF5140) and references therein: `http://www.uio.no/studier/emner/matnat/ifi/INF5140/v07/undervisningsmateriale/1-formal-methods.pdf`