Specification and Analysis of Contracts
Tutorial

Gerardo Schneider

gerardo@ifi.uio.no

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

Department of Informatics,
University of Oslo

What Is This Tutorial About?

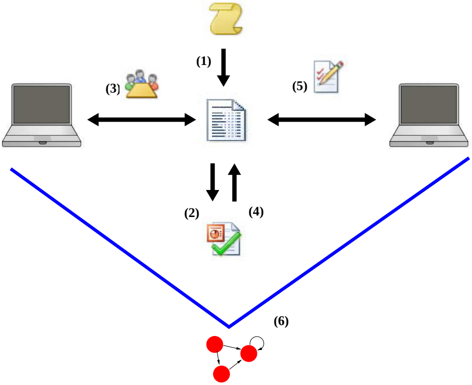
- Specification and analysis of contracts for Services
- Many of the material is state-of-the-art and on-going research
- It is not an exhaustive exposition of
  - Service-Oriented Architecture (SOA)
  - Components
- We will see how to use (a special kind of) contracts in the context of Services and Components

What Is This Tutorial About?

- Contracts and Service-Oriented Computing (SOC)

What Is This Tutorial About?

- Contracts and Service-Oriented Computing (SOC)

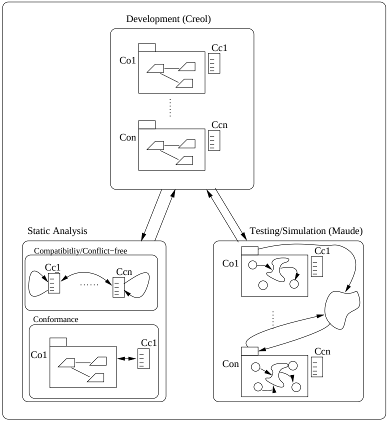


What Is This Tutorial About?

- Contracts and Components

What Is This Tutorial About?

- Contracts and Components



What Is This Tutorial About?

We will see:

- A bit of formal methods
- SOC and components
- Deontic logic
- A formal language for writing contracts
- How to analyze contracts using *model checking*

Outline

1. Lesson 1: Introduction
   - Formal Methods
   - Contracts 'and' Informatics
2. Lesson 2: Components, Services and Contracts
   - Components
   - Service-Oriented Computing
3. Lesson 3: Deontic Logic
   - Deontic Logic
   - Paradoxes in Deontic Logic
4. Lesson 4: Specification and Analysis of Contracts
   - The Contract Language $\mathcal{CL}$
   - Properties of the Language
   - Verification of Contracts

Outline

Outline

1. Lesson 1: Introduction
   - Formal Methods
   - Contracts ‘and’ Informatics
2. Lesson 2: Components, Services and Contracts
   - Components
   - Service-Oriented Computing
3. Lesson 3: Deontic Logic
   - Deontic Logic
   - Paradoxes in Deontic Logic
4. Lesson 4: Specification and Analysis of Contracts
   - The Contract Language $\mathcal{CL}$
   - Properties of the Language
   - Verification of Contracts

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?

[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?

[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] 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

[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
- Do not confuse validation with *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
- Do not confuse validation with *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
- Do not confuse validation with *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".

- The term verification is used in different ways
  - Sometimes used only to refer the process of obtaining the formal correctness proof of a system (deductive verification)
  - In other cases, used to describe any action taken for finding errors in a program (including model checking and testing)
  - Sometimes testing is not considered to be a verification technique

What are Formal Methods?

Some Terminology

- The term verification is used in different ways
  - Sometimes used only to refer the process of obtaining the formal correctness proof of a system (deductive verification)
  - In other cases, used to describe any action taken for finding errors in a program (including model checking and testing)
  - Sometimes testing is not considered to be a verification technique

We will use the following definition (reminder):

Definition

Formal verification is the process of applying a manual or automatic *formal* technique for establishing whether a given system satisfies a given property or behaves in accordance to some abstract description (*formal* specification) of the system

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)

Do not confuse the specification of the system itself with the specification of some of its properties

- Both kinds of specifications may use the same formalism but not necessarily.
  For example:
  - The system specification can be given as a program or as a state machine
  - System properties can be formalized using some logic

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)

Do not confuse the specification of the system itself with the specification of some of its properties

- Both kinds of specifications may use the same formalism but not necessarily.

  For example:

  - The system specification can be given as a program or as a state machine
  - System properties can be formalized using some logic

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)

Do not confuse the specification of the system itself with the specification of some of its properties

- Both kinds of specifications may use the same formalism but not necessarily.
  For example:
  - The system specification can be given as a program or as a state machine
  - System properties can be formalized using some logic

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: $a(0) \wedge a(1) \wedge \forall t \geq 0 \cdot a(t+3) = \neg a(t+2)$

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: $a(0) \wedge a(1) \wedge \forall t \geq 0 \cdot a(t+3) = \neg a(t+2)$
- In the last lesson we will see how to verify contracts

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

Outline

1. Lesson 1: Introduction
   - Formal Methods
   - Contracts 'and' Informatics
2. Lesson 2: Components, Services and Contracts
   - Components
   - Service-Oriented Computing
3. Lesson 3: Deontic Logic
   - Deontic Logic
   - Paradoxes in Deontic Logic
4. Lesson 4: Specification and Analysis of Contracts
   - The Contract Language $\mathcal{CL}$
   - Properties of the Language
   - Verification of Contracts

Contracts

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

Contracts

- "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
   - 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
   - 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
   - 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
   - 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
   - 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
   - Inspired from a conventional contract
   - Written directly in a formal specification language

Contracts

- In this tutorial: '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: '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`

Outline

What is a Component?

- We will concentrate only on *software components*
- 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

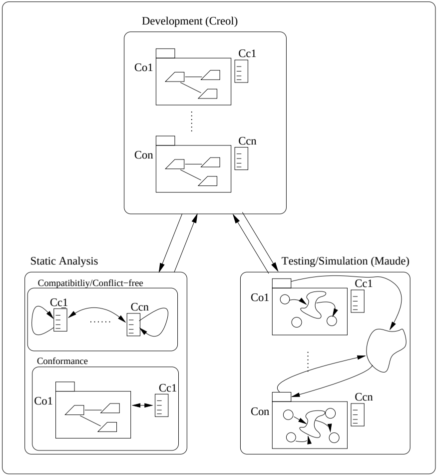
Components and Contracts I

- In "traditional" component-based development, contracts are understood as specification attached to interfaces
  - Behavioral interfaces instead of static interfaces

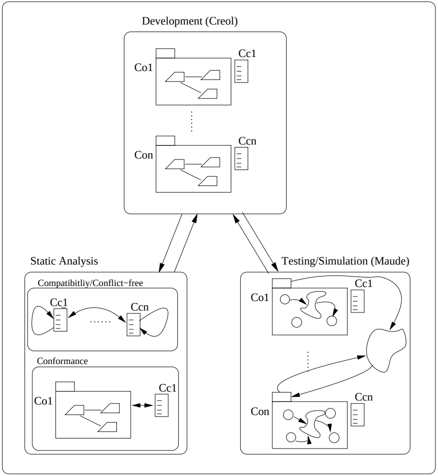
Observation

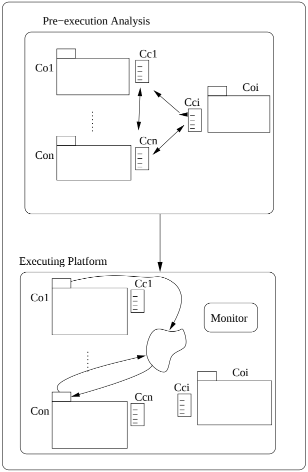
We propose the use of 'deontic' e-contracts to help verification of and reasoning about components

Components and Contracts II



Components and Contracts II





Outline

What is a Service?

- A service is a self-describing, platform-independent computational element
  - It supports rapid, low-cost composition of distributed applications
  - It allows organizations to offer their core competences over intra-nets or the Internet using standard languages (e.g., XML-based) and protocols

What is a Service?

- A service is a self-describing, platform-independent computational element
  - It supports rapid, low-cost composition of distributed applications
  - It allows organizations to offer their core 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?

- A service is a self-describing, platform-independent computational element
  - It supports rapid, low-cost composition of distributed applications
  - It allows organizations to offer their core 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

What is a Web Service?

- Def. 1: A web service is a web site to be used by software instead of by humans
- Def. 2: A web service is a specific kind of service identified by a URI (Uniform Resource Indicator), that:
  - It is exposed over Internet using standard languages and protocols
  - It can be implemented via a self-describing interface based on open Internet standards (e.g. XML)
- Web services require special consideration since they use a public, insecure, low-fidelity mechanism for inter-service interactions
- Service descriptions are usually expressed using WSDL (Web Services Description Language)
- UDDI (Universal Description, Discovery and Integration)
  - Providing registry and repository services for storing and retrieving web service interfaces

What is a Web Service?

- Def. 1: A web service is a web site to be used by software instead of by humans
- Def. 2: A web service is a specific kind of service identified by a URI (Uniform Resource Indicator), that:
  - It is exposed over Internet using standard languages and protocols
  - It can be implemented via a self-describing interface based on open Internet standards (e.g. XML)
- Web services require special consideration since they use a public, insecure, low-fidelity mechanism for inter-service interactions
- Service descriptions are usually expressed using WSDL (Web Services Description Language)
- UDDI (Universal Description, Discovery and Integration)
  - Providing registry and repository services for storing and retrieving web service interfaces

Services vs. 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

Services and Contracts I

- In web services, a service contract is usually understood as service-level agreement (SLA)
  - Example: how much the client might pay for the service; guarantees from the provider: minimal performance, capacity, etc

Services and Contracts I

- In web services, a service contract is usually understood as service-level agreement (SLA)
  - Example: how much the client might pay for the service; guarantees from the provider: minimal performance, capacity, etc
- Challenges:
  - How to reason about service contracts
  - How to address (automatic) negotiation
  - How to enforce the fulfillment of the contract

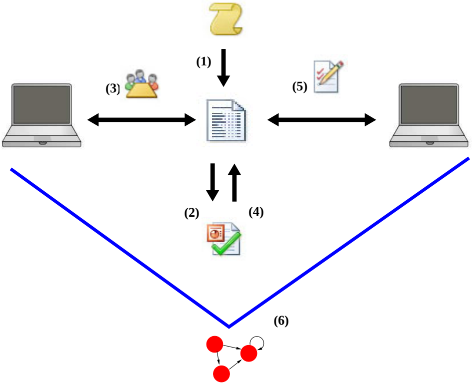
Services and Contracts I

- In web services, a service contract is usually understood as service-level agreement (SLA)
  - Example: how much the client might pay for the service; guarantees from the provider: minimal performance, capacity, etc
- Challenges:
  - How to reason about service contracts
  - How to address (automatic) negotiation
  - How to enforce the fulfillment of the contract

Observation

We propose the use of 'deontic' e-contracts to help verification of and reasoning about services.
Such contracts may also be useful in the negotiation process.

Services and Contracts II



Links and Papers

- M. Papazoglou. **Service-Oriented Computing: Concepts, Characteristics and Directions**
- E. Newcomer. **Understanding Web Services**
- C. Szyperski. **Component Technology - What, Where, and How?**
- O. Owe, G. Schneider and M. Steffen. **Objects, Components and Contracts**
- COSoDIS project: `http://www.ifi.uio.no/cosodis`

Outline

Outline

Why Deontic Logic?

- We propose the use of 'deontic' e-contracts in the context of Service-Oriented Computing and Components
- We need then some knowledge of deontic logic
  - Though we only get inspiration from deontic logic and not build upon its standard formalization

(Standard) Deontic Logic
In One Slide

- Concerned with moral and normative notions
  - *obligation*, *permission*, *prohibition*, *optionality*, *power*, *indifference*, *immunity*, etc
- Focus on
  - The logical consistency of the above notions
  - The faithful representation of their intuitive meaning in law, moral systems, business organizations and security systems
- Difficult to avoid *puzzles* and *paradoxes*
  - Logical paradoxes, where we can deduce contradictory actions
  - "Practical oddities", where we can get counterintuitive conclusions
- Approaches
  - ought-to-do: expressions consider *names of actions*
    - "The Internet Provider *must send* a password to the Client"
  - ought-to-be: expressions consider state of affairs (results of actions)
    - "The average bandwidth *must be* more than 20kb/s"

(Standard) Deontic Logic
In One Slide

- Concerned with moral and normative notions
  - *obligation*, *permission*, *prohibition*, *optionality*, *power*, *indifference*, *immunity*, etc
- Focus on
  - The logical consistency of the above notions
  - The faithful representation of their intuitive meaning in law, moral systems, business organizations and security systems
- Difficult to avoid *puzzles* and *paradoxes*
  - Logical paradoxes, where we can deduce contradictory actions
  - "Practical oddities", where we can get counterintuitive conclusions
- Approaches
  - ought-to-do: expressions consider *names of actions*
    - "The Internet Provider *must send* a password to the Client"
  - ought-to-be: expressions consider state of affairs (results of actions)
    - "The average bandwidth *must be* more than 20kb/s"

(Standard) Deontic Logic
In One Slide

- Concerned with moral and normative notions
  - *obligation*, *permission*, *prohibition*, *optionality*, *power*, *indifference*, *immunity*, etc
- Focus on
  - The logical consistency of the above notions
  - The faithful representation of their intuitive meaning in law, moral systems, business organizations and security systems
- Difficult to avoid *puzzles* and *paradoxes*
  - Logical paradoxes, where we can deduce contradictory actions
  - "Practical oddities", where we can get counterintuitive conclusions
- Approaches
  - ought-to-do: expressions consider *names of actions*
    - "The Internet Provider *must send* a password to the Client"
  - ought-to-be: expressions consider state of affairs (results of actions)
    - "The average bandwidth *must be* more than 20kb/s"

(Standard) Deontic Logic

In One Slide

- Concerned with moral and normative notions
  - *obligation*, *permission*, *prohibition*, *optionality*, *power*, *indifference*, *immunity*, etc
- Focus on
  - The logical consistency of the above notions
  - The faithful representation of their intuitive meaning in law, moral systems, business organizations and security systems
- Difficult to avoid *puzzles* and *paradoxes*
  - Logical paradoxes, where we can deduce contradictory actions
  - "Practical oddities", where we can get counterintuitive conclusions
- Approaches
  - ought-to-do: expressions consider *names of actions*
    - "The Internet Provider *must send* a password to the Client"
  - ought-to-be: expressions consider state of affairs (results of actions)
    - "The average bandwidth *must be* more than 20kb/s"

A Bit of Prehistory

- Since Aristotle (384 BC–322 BC) there were some philosophers' writing on obligation, permission and prohibition
- Leibniz (1646–1716) related obligation, permission and prohibition with logical modalities of necessity, possibility and impossibility
- Ernst Mally (1926) used the term *deontik* for his "Logic of the Will"
    - Also called it: The logic of what ought to be
    - No mention of Leibniz nor of relation between modal and normative notions
- A lot of discussions in the late 1930s and early 1940s
    - Jørgen Jørgensen and Alf Ross

The Beginnings

- It is accepted that the deontic logic was born as discipline from the following (independent) works
  - G.H. von Wright published the paper "Deontic Logic" (1951)
  - O. Becker (1952, in German)
  - J. Kalinowski (1953, in French)
- All 3 authors explored the analogy between normative and modal concepts
- von Wright (1951)
  - Started by exploring the formal analogy between the modalities "possible", "impossible" and "necessary" with the quantifiers "some", "no" and "all"
  - Extended his study to the analogy with the normative notions (the 1951 paper)
- A. Prior (1954) criticized von Wright's paper
  - How to obtain derived obligations, i.e. *conditional obligations*?
  - von Wright's answer by adding relative permission:
    - $P(p/q)$: "it is permitted that p on the condition that q"
- Much more followed...

The Beginnings

- It is accepted that the deontic logic was born as discipline from the following (independent) works
  - G.H. von Wright published the paper "Deontic Logic" (1951)
  - O. Becker (1952, in German)
  - J. Kalinowski (1953, in French)
- All 3 authors explored the analogy between normative and modal concepts
- von Wright (1951)
  - Started by exploring the formal analogy between the modalities "possible", "impossible" and "necessary" with the quantifiers "some", "no" and "all"
  - Extended his study to the analogy with the normative notions (the 1951 paper)
- A. Prior (1954) criticized von Wright's paper
  - How to obtain derived obligations, i.e. *conditional obligations*?
  - von Wright's answer by adding relative permission:
    - $P(p/q)$: "it is permitted that p on the condition that q"
- Much more followed...

Ought-to-do vs. Ought-to-be

- Ought-to-do: expressions consider *names of actions*
  - "One ought to close the window"
- Ought-to-be: expressions consider state of affairs (results of actions)
  - "The window ought to be closed"

Ought-to-do vs. Ought-to-be

- Ought-to-do: expressions consider *names of actions*
  - "One ought to close the window"
- Ought-to-be: expressions consider state of affairs (results of actions)
  - "The window ought to be closed"

Why is this so important?

- Some things are easier to represent in one approach and others in the other
  - "The average bandwidth *must be* more than 20kb/s"
  - Sergot's example on the "strict University code"
- The logical system may have some nicer properties in one or the other approach
  - Paradoxes...

Why Is This All So Complicated?

- *Norms* as prescriptions for conduct, are not **true** or **false**
  - If norms have no truth-value, how can we reason about them and detect contradictions and define logical consequence?
- According to von Wright: norms and valuations are still subject to logical view
- Consequence: Logic has a wider reach than truth!
- *Prescriptive* vs. *descriptive* view
- Conditional norms
- Meta-norms
- How to represent what happens when an obligation is not fulfilled or a prohibition is violated?
- Paradoxes
- A lot more...

Outline

Standard Deontic Logic

- Takes different modal logics and makes analogies between “necessity” and “possibility”, with “obligation” and “permission”
- It turns out to be difficult!
  - Many of the rules in modal logic do not extrapolate to deontic logic

Standard Deontic Logic

- Takes different modal logics and makes analogies between "necessity" and "possibility", with "obligation" and "permission"
- It turns out to be difficult!
  - Many of the rules in modal logic do not extrapolate to deontic logic

Example

In modal logic:

- If $\Box p$ then p (if it is necessary that p, then p is true)
- If p then $\Diamond p$ (if p is true, then it is possible)

The deontic analogs:

- If $O(p)$ then p (if it is obligatory that p, then p is true)
- If p then $P(p)$ (if p is true, then it is permissible)

Paradoxes and Practical Oddities

- Deontic paradoxes. A paradox is an apparently true statement that leads to a contradiction, or a situation which is counter-intuitive.
  - *The Gentle Murderer Paradox*
    1. It is obligatory that John does not kill his mother;
    2. If John does kill his mother, then it is obligatory that John kills her gently;
    3. John does kill his mother.

    It could be possible to infer that John is obliged to kill his mother (contradicting 1 above)
- Practical oddities. A situation where you can infer two assertions which are contradictory from the intuitive practical point of view, though they might not represent a logical contradiction
  - Assume you have the following norms and facts:
    1. Keep your promise;
    2. If you haven't kept your promise, apologize;
    3. You haven't kept your promise.

    It could be possible to deduce that you are both obliged to keep your promise and to apologize for not keeping it

Paradoxes and Practical Oddities

- Deontic paradoxes. A paradox is an apparently true statement that leads to a contradiction, or a situation which is counter-intuitive.
  - *The Gentle Murderer Paradox*
    1. It is obligatory that John does not kill his mother;
    2. If John does kill his mother, then it is obligatory that John kills her gently;
    3. John does kill his mother.

    It could be possible to infer that John is obliged to kill his mother (contradicting 1 above)
- Practical oddities. A situation where you can infer two assertions which are contradictory from the intuitive practical point of view, though they might not represent a logical contradiction
  - Assume you have the following norms and facts:
    1. Keep your promise;
    2. If you haven't kept your promise, apologize;
    3. You haven't kept your promise.

    It could be possible to deduce that you are both obliged to keep your promise and to apologize for not keeping it

1. It is obligatory that one mails the letter
2. It is obligatory that one mails the letter or one destroys the letter

In Standard Deontic Logic (SDL) these are expressed as:

1. $O(p)$
2. $O(p \vee q)$

- Problem: in SDL one can infer that $O(p) \Rightarrow O(p \vee q)$

1. You may either sleep on the sofa or sleep on the bed.
2. You may sleep on the sofa and you may sleep on the bed.

In SDL this is:

1. $P(p \vee q)$
2. $P(p) \wedge P(q)$

- The natural intuition tells that $P(p \vee q) \Rightarrow P(p) \wedge P(q)$
- In SDL this would lead to $P(p) \Rightarrow P(p \vee q)$ which is $P(p) \Rightarrow P(p) \wedge P(q)$,
- so $P(p) \Rightarrow P(q)$
- Thus: *If one is permitted something, then one is permitted anything*.

1. It is obligatory I now meet Jones (as promised to Jones).
2. It is obligatory I now do not meet Jones (as promised to Smith).

In SDL this is:

1. $O(p)$
2. $O(\neg p)$

- The problem is that in the natural language the two obligations are intuitive and often happen
- But the logical formulae are inconsistent when put together (in conjunction) in SDL
- (In SDL, $O(p) \Rightarrow \neg O(\neg p)$ and we get a contradiction.)

Paradoxes

The Good Samaritan Paradox

1. It ought to be the case that Jones helps Smith who has been robbed.
2. It ought to be the case that Smith has been robbed.

And one naturally infers that:

> Jones helps Smith who has been robbed if and only if Jones helps Smith and Smith has been robbed.

In SDL the first two are expressed as:

1. $O(p \wedge q)$
2. $O(q)$

- The problem is that in SDL one can derive that $O(p \wedge q) \Rightarrow O(q)$ which is counter intuitive in the natural language

Paradoxes

Chisholm's Paradox

1. John ought to go to the party.
2. If John goes to the party then he ought to tell them he is coming.
3. If John does not go to the party then he ought not to tell them he is coming.
4. John does not go to the party.

In Standard Deontic Logic (SDL) these are expressed as:

1. $O(p)$
2. $O(p \Rightarrow q)$
3. $\neg p \Rightarrow O(\neg q)$
4. $\neg p$

- The problem is that in SDL one can infer $O(q) \wedge O(\neg q)$ (due to statement 2)

1. It is obligatory that John does not kill his mother.
2. If John does kill his mother, then it is obligatory that John kills her gently.
3. John does kill his mother.

In Standard Deontic Logic (SDL) these are expressed as:

1. $O(\neg p)$
2. $p \Rightarrow O(q)$
3. p

- The problem is that when adding a natural inference like $q \Rightarrow p$, one can infer that $O(p)$ (contradicting 1 above)

Role of Deontic Logic in Services

Reminder

- We want to use deontic e-contracts to specify and reason about contracts in Internet services
- We need a formal system to relate the normative notions of obligation, permission and prohibition
- We want to represent (nested) "exceptions": Can we represent and reason about what happens when an obligation is not fulfilled or a prohibition is violated?
- We want to avoid the philosophical problems of deontic logic (restrict its use to our application domain)

Links and Papers

- G.H. von Wright. **Deontic Logic: A personal view**.
- P. McNamara. **Deontic Logic**. See the entry at the Stanford Encyclopedia of Philosophy (`http://plato.stanford.edu/entries/logic-deontic`)
- J.-J. Ch. Meyer, F.P.M. Dignum and R.J. Wieringa. **The Paradoxes of Deontic Logic Revisited: A Computer Science Perspective.**

Outline

Aim and Motivation

- Use deontic e-contracts to 'rule' services exchange (e.g., web services and component-based development)

1. Give a formal language for specifying/writing contracts
2. Analyze contracts "internally"
   - Detect contradictions/inconsistencies statically
   - Determine the obligations (permissions, prohibitions) of a signatory
   - Detect superfluous contract clauses
3. Tackle the negotiation process (automatically?)
4. Develop a theory of contracts
   - Contract composition
   - Subcontracting
   - Conformance between a contract and the governing policies
   - *Meta-contracts* (policies)
5. Monitor contracts
   - Run-time system to ensure the contract is respected
   - In case of contract violations, act accordingly

Aim and Motivation

- Use deontic e-contracts to ‘rule’ services exchange (e.g., web services and component-based development)

1. Give a formal language for specifying/writing contracts
2. Analyze contracts “internally”
   - Detect contradictions/inconsistencies statically
   - Determine the obligations (permissions, prohibitions) of a signatory
   - Detect superfluous contract clauses
3. Tackle the negotiation process (automatically?)
4. Develop a theory of contracts
   - Contract composition
   - Subcontracting
   - Conformance between a contract and the governing policies
   - *Meta-contracts* (policies)
5. Monitor contracts
   - Run-time system to ensure the contract is respected
   - In case of contract violations, act accordingly

Aim and Motivation

- Use deontic e-contracts to 'rule' services exchange (e.g., web services and component-based development)

1. Give a formal language for specifying/writing contracts
2. Analyze contracts "internally"
   - Detect contradictions/inconsistencies statically
   - Determine the obligations (permissions, prohibitions) of a signatory
   - Detect superfluous contract clauses
3. Tackle the negotiation process (automatically?)
4. Develop a theory of contracts
   - Contract composition
   - Subcontracting
   - Conformance between a contract and the governing policies
   - *Meta-contracts* (policies)
5. Monitor contracts
   - Run-time system to ensure the contract is respected
   - In case of contract violations, act accordingly

Aim and Motivation

- Use deontic e-contracts to 'rule' services exchange (e.g., web services and component-based development)

1. Give a formal language for specifying/writing contracts
2. Analyze contracts "internally"
   - Detect contradictions/inconsistencies statically
   - Determine the obligations (permissions, prohibitions) of a signatory
   - Detect superfluous contract clauses
3. Tackle the negotiation process (automatically?)
4. Develop a theory of contracts
   - Contract composition
   - Subcontracting
   - Conformance between a contract and the governing policies
   - *Meta-contracts* (policies)
5. Monitor contracts
   - Run-time system to ensure the contract is respected
   - In case of contract violations, act accordingly

Aim and Motivation

- Use deontic e-contracts to 'rule' services exchange (e.g., web services and component-based development)

1. Give a formal language for specifying/writing contracts
2. Analyze contracts "internally"
   - Detect contradictions/inconsistencies statically
   - Determine the obligations (permissions, prohibitions) of a signatory
   - Detect superfluous contract clauses
3. Tackle the negotiation process (automatically?)
4. Develop a theory of contracts
   - Contract composition
   - Subcontracting
   - Conformance between a contract and the governing policies
   - *Meta-contracts* (policies)
5. Monitor contracts
   - Run-time system to ensure the contract is respected
   - In case of contract violations, act accordingly

Aim and Motivation

- Use deontic e-contracts to 'rule' services exchange (e.g., web services and component-based development)

1. Give a formal language for specifying/writing contracts
2. Analyze contracts "internally"
   - Detect contradictions/inconsistencies statically
   - Determine the obligations (permissions, prohibitions) of a signatory
   - Detect superfluous contract clauses
3. Tackle the negotiation process (automatically?)
4. Develop a theory of contracts
   - Contract composition
   - Subcontracting
   - Conformance between a contract and the governing policies
   - *Meta-contracts* (policies)
5. Monitor contracts
   - Run-time system to ensure the contract is respected
   - In case of contract violations, act accordingly

A Formal Language for Contracts

- A precise and concise syntax and a formal semantics
- Expressive enough as to capture natural contract clauses
- Restrictive enough to avoid (deontic) paradoxes and be amenable to formal analysis
  - Model checking
  - Deductive verification
- Allow representation of complex clauses: conditional obligations, permissions, and prohibitions
- Allow specification of (nested) contrary-to-duty (CTD) and contrary-to-prohibition (CTP)
  - CTD: when an obligation is not fulfilled
  - CTP: when a prohibition is violated
- We want to combine
  - The logical approach (e.g., dynamic, temporal, deontic logic)
  - The automata-like approach (labelled Kripke structures)

A Formal Language for Contracts

- A precise and concise syntax and a formal semantics
- Expressive enough as to capture natural contract clauses
- Restrictive enough to avoid (deontic) paradoxes and be amenable to formal analysis
  - Model checking
  - Deductive verification
- Allow representation of complex clauses: conditional obligations, permissions, and prohibitions
- Allow specification of (nested) contrary-to-duty (CTD) and contrary-to-prohibition (CTP)
  - CTD: when an obligation is not fulfilled
  - CTP: when a prohibition is violated
- We want to combine
  - The logical approach (e.g., dynamic, temporal, deontic logic)
  - The automata-like approach (labelled Kripke structures)

The Contract Specification Language $\mathcal{CL}$

Definition ($\mathcal{CL}$)

$$
\begin{array}{rcl}
\mathcal{C}ontract & := & \mathcal{D} \; ; \; \mathcal{C} \\
\mathcal{C} & := & \mathcal{C}_O \mid \mathcal{C}_P \mid \mathcal{C}_F \mid \mathcal{C} \wedge \mathcal{C} \mid [\alpha]\mathcal{C} \mid \langle\alpha\rangle\mathcal{C} \mid \mathcal{C}\,\mathcal{U}\,\mathcal{C} \mid \bigcirc\,\mathcal{C} \mid \Box\mathcal{C} \\
\mathcal{C}_O & := & O(\alpha) \mid \mathcal{C}_O \oplus \mathcal{C}_O \\
\mathcal{C}_P & := & P(\alpha) \mid \mathcal{C}_P \oplus \mathcal{C}_P \\
\mathcal{C}_F & := & F(\alpha) \mid \mathcal{C}_F \vee [\alpha]\mathcal{C}_F
\end{array}
$$

- $O(\alpha)$, $P(\alpha)$, $F(\alpha)$ specify obligation, permission (rights), and prohibition (forbidden) over actions
- α are actions given in the definition part $\mathcal{D}$
  - $+$ choice
  - $\cdot$ concatenation (sequencing)
  - $\&$ concurrency
  - $\phi?$ test
- $\wedge$, $\vee$, and $\oplus$ are conjunction, disjunction, and exclusive disjunction
- $[\alpha]$ and $\langle\alpha\rangle$ are the action parameterized modalities of dynamic logic
- $\mathcal{U}$, $\bigcirc$, and $\Box$ correspond to temporal logic operators

The Contract Specification Language $\mathcal{CL}$

Definition ($\mathcal{CL}$)

$$
\begin{array}{rcl}
Contract & := & \mathcal{D} ; \mathcal{C} \\
\mathcal{C} & := & \mathcal{C}_O \mid \mathcal{C}_P \mid \mathcal{C}_F \mid \mathcal{C} \wedge \mathcal{C} \mid [\alpha]\mathcal{C} \mid \langle\alpha\rangle\mathcal{C} \mid \mathcal{C}\,\mathcal{U}\,\mathcal{C} \mid \bigcirc\mathcal{C} \mid \Box\mathcal{C} \\
\mathcal{C}_O & := & O(\alpha) \mid \mathcal{C}_O \oplus \mathcal{C}_O \\
\mathcal{C}_P & := & P(\alpha) \mid \mathcal{C}_P \oplus \mathcal{C}_P \\
\mathcal{C}_F & := & F(\alpha) \mid \mathcal{C}_F \vee [\alpha]\mathcal{C}_F
\end{array}
$$

- $O(\alpha)$, $P(\alpha)$, $F(\alpha)$ specify obligation, permission (rights), and prohibition (forbidden) over actions
- α are actions given in the definition part $\mathcal{D}$
  - $+$ choice
  - $\cdot$ concatenation (sequencing)
  - $\&$ concurrency
  - $\phi?$ test
- $\wedge$, $\vee$, and $\oplus$ are conjunction, disjunction, and exclusive disjunction
- $[\alpha]$ and $\langle\alpha\rangle$ are the action parameterized modalities of dynamic logic
- $\mathcal{U}$, $\bigcirc$, and $\Box$ correspond to temporal logic operators

The Contract Specification Language $\mathcal{CL}$

Definition ($\mathcal{CL}$)

$$
\begin{array}{rcl}
\mathcal{C}ontract & := & \mathcal{D} \; ; \; \mathcal{C} \\
\mathcal{C} & := & \mathcal{C}_O \mid \mathcal{C}_P \mid \mathcal{C}_F \mid \mathcal{C} \wedge \mathcal{C} \mid [\alpha]\mathcal{C} \mid \langle\alpha\rangle\mathcal{C} \mid \mathcal{C}\,\mathcal{U}\,\mathcal{C} \mid \bigcirc\,\mathcal{C} \mid \Box\mathcal{C} \\
\mathcal{C}_O & := & O(\alpha) \mid \mathcal{C}_O \oplus \mathcal{C}_O \\
\mathcal{C}_P & := & P(\alpha) \mid \mathcal{C}_P \oplus \mathcal{C}_P \\
\mathcal{C}_F & := & F(\alpha) \mid \mathcal{C}_F \vee [\alpha]\mathcal{C}_F
\end{array}
$$

- $O(\alpha)$, $P(\alpha)$, $F(\alpha)$ specify obligation, permission (rights), and prohibition (forbidden) over actions
- α are actions given in the definition part $\mathcal{D}$
  - $+$ choice
  - $\cdot$ concatenation (sequencing)
  - $\&$ concurrency
  - $\phi?$ test
- $\wedge$, $\vee$, and $\oplus$ are conjunction, disjunction, and exclusive disjunction
- $[\alpha]$ and $\langle\alpha\rangle$ are the action parameterized modalities of dynamic logic
- $\mathcal{U}$, $\bigcirc$, and $\Box$ correspond to temporal logic operators

The Contract Specification Language $\mathcal{CL}$

Definition ($\mathcal{CL}$)

$$
\begin{array}{rcl}
\mathcal{C}ontract & := & \mathcal{D} \,;\, \mathcal{C} \\
\mathcal{C} & := & \mathcal{C}_O \mid \mathcal{C}_P \mid \mathcal{C}_F \mid \mathcal{C} \wedge \mathcal{C} \mid [\alpha]\mathcal{C} \mid \langle\alpha\rangle\mathcal{C} \mid \mathcal{C}\,\mathcal{U}\,\mathcal{C} \mid \bigcirc\,\mathcal{C} \mid \Box\mathcal{C} \\
\mathcal{C}_O & := & O(\alpha) \mid \mathcal{C}_O \oplus \mathcal{C}_O \\
\mathcal{C}_P & := & P(\alpha) \mid \mathcal{C}_P \oplus \mathcal{C}_P \\
\mathcal{C}_F & := & F(\alpha) \mid \mathcal{C}_F \vee [\alpha]\mathcal{C}_F
\end{array}
$$

- $O(\alpha)$, $P(\alpha)$, $F(\alpha)$ specify obligation, permission (rights), and prohibition (forbidden) over actions
- α are actions given in the definition part $\mathcal{D}$
  - $+$ choice
  - $\cdot$ concatenation (sequencing)
  - $\&$ concurrency
  - $\phi?$ test
- $\wedge$, $\vee$, and $\oplus$ are conjunction, disjunction, and exclusive disjunction
- $[\alpha]$ and $\langle\alpha\rangle$ are the action parameterized modalities of dynamic logic
- $\mathcal{U}$, $\bigcirc$, and $\Box$ correspond to temporal logic operators

The Contract Specification Language $\mathcal{CL}$

Definition ($\mathcal{CL}$)

$$
\begin{array}{rcl}
\mathcal{C}ontract & := & \mathcal{D} \; ; \; \mathcal{C} \\
\mathcal{C} & := & \mathcal{C}_O \mid \mathcal{C}_P \mid \mathcal{C}_F \mid \mathcal{C} \wedge \mathcal{C} \mid [\alpha]\mathcal{C} \mid \langle\alpha\rangle\mathcal{C} \mid \mathcal{C}\,\mathcal{U}\,\mathcal{C} \mid \bigcirc\mathcal{C} \mid \Box\mathcal{C} \\
\mathcal{C}_O & := & O(\alpha) \mid \mathcal{C}_O \oplus \mathcal{C}_O \\
\mathcal{C}_P & := & P(\alpha) \mid \mathcal{C}_P \oplus \mathcal{C}_P \\
\mathcal{C}_F & := & F(\alpha) \mid \mathcal{C}_F \vee [\alpha]\mathcal{C}_F
\end{array}
$$

- $O(\alpha)$, $P(\alpha)$, $F(\alpha)$ specify obligation, permission (rights), and prohibition (forbidden) over actions
- α are actions given in the definition part $\mathcal{D}$
  - $+$ choice
  - $\cdot$ concatenation (sequencing)
  - $\&$ concurrency
  - $\phi?$ test
- $\wedge$, $\vee$, and $\oplus$ are conjunction, disjunction, and exclusive disjunction
- $[\alpha]$ and $\langle\alpha\rangle$ are the action parameterized modalities of dynamic logic
- $\mathcal{U}$, $\bigcirc$, and $\Box$ correspond to temporal logic operators

- Tests as actions: $\phi?$
  - The behaviour of a test is like a *guard*; e.g. $\varphi? \cdot a$ *if the test succeeds then action a is performed*
  - Tests are used to model implication: $[\varphi?]\mathcal{C}$ is the same as $\varphi \Rightarrow \mathcal{C}$
- Action negation $\overline{\alpha}$
  - It represents all immediate traces that take us outside the trace of α
  - Involves the use of a *canonic form* of actions
  - E.g.: consider two atomic actions a and b then $\overline{a \cdot b}$ is $b + a \cdot a$

- Tests as actions: $\phi?$
  - The behaviour of a test is like a *guard*; e.g. $\varphi? \cdot a$ *if the test succeeds then action a is performed*
  - Tests are used to model implication: $[\varphi?]\mathcal{C}$ is the same as $\varphi \Rightarrow \mathcal{C}$
- Action negation $\overline{\alpha}$
  - It represents all immediate traces that take us outside the trace of α
  - Involves the use of a *canonic form* of actions
  - E.g.: consider two atomic actions a and b then $\overline{a \cdot b}$ is $b + a \cdot a$

- $a \& b$
- "The client must pay immediately, or the client must notify the service provider by sending an e-mail specifying that he delays the payment"

$$O(p) \oplus O(d \& n)$$

- $O(d \& n) \equiv O(d) \wedge O(n)$
- Action algebra enriched with a conflict relation to represent incompatible actions
  - $a =$ "increase Internet traffic" and $b =$ "decrease Internet traffic"
    - $a \#_{\mathcal{C}}\, b$
    - $O(a) \wedge O(b)$ gives an inconsistency

Actions

Concurrent actions

- $a \& b$
- "The client must pay immediately, or the client must notify the service provider by sending an e-mail specifying that he delays the payment"

$$O(p) \oplus O(d \& n)$$

- $O(d \& n) \equiv O(d) \wedge O(n)$
- Action algebra enriched with a conflict relation to represent incompatible actions
  - $a =$ "increase Internet traffic" and $b =$ "decrease Internet traffic"
    - $a \#_{\mathcal{C}} b$
    - $O(a) \wedge O(b)$ gives an inconsistency

- Expressing contrary-to-duty (CTD)

$$O_{\mathcal{C}}(\alpha) = O(\alpha) \wedge [\overline{\alpha}]\mathcal{C}$$

- Expressing contrary-to-duty (CTD)

$$O_{\mathcal{C}}(\alpha) = O(\alpha) \wedge [\overline{\alpha}]\mathcal{C}$$

- Expressing contrary-to-prohibition (CTP)

$$F_{\mathcal{C}}(\alpha) = F(\alpha) \wedge [\alpha]\mathcal{C}$$

- Expressing contrary-to-duty (CTD)

$$O_{\mathcal{C}}(\alpha) = O(\alpha) \wedge [\overline{\alpha}]\mathcal{C}$$

- Expressing contrary-to-prohibition (CTP)

$$F_{\mathcal{C}}(\alpha) = F(\alpha) \wedge [\alpha]\mathcal{C}$$

Example

"[...] the client must immediately lower the Internet traffic to the *low* level, and pay . If the client does not lower the Internet traffic immediately, then the client will have to pay three times the price"

In $\mathcal{CL}$: $\Box(O_{\mathcal{C}}(l) \wedge [l]\Diamond(O(p\&p)))$

where $\mathcal{C} = \Diamond O(p\&p\&p)$

- Translation into a variant of μ-calculus ($\mathcal{C}\mu$)
- The syntax of the $\mathcal{C}\mu$ logic
  $\varphi \ := \ P \mid Z \mid P_c \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid [\gamma]\varphi \mid \mu Z.\varphi(Z)$

Main differences with respect to the classical μ-calculus:

1. P_c is set of propositional constants O_a and $\mathcal{F}_a$, one for each basic action a
2. Multisets of basic actions: i.e. $\gamma = \{a, a, b\}$ is a label

- Translation into a variant of μ-calculus ($\mathcal{C}\mu$)
- The syntax of the $\mathcal{C}\mu$ logic
  $\varphi \ := \ P \mid Z \mid P_c \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid [\gamma]\varphi \mid \mu Z.\varphi(Z)$

Main differences with respect to the classical μ-calculus:

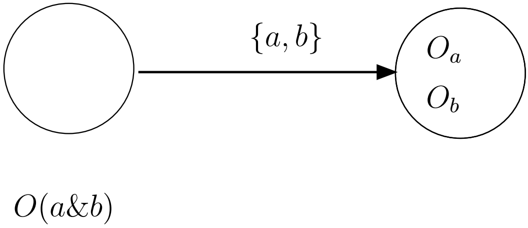
1. P_c is set of propositional constants O_a and $\mathcal{F}_a$, one for each basic action a
2. Multisets of basic actions: i.e. $\gamma = \{a, a, b\}$ is a label

- Obligation

$$f^{\mathcal{T}}(O(a\&b)) = \langle\{a, b\}\rangle(O_a \wedge O_b)$$

- Obligation

$$f^{\mathcal{T}}(O(a\&b)) = \langle\{a,b\}\rangle(O_a \wedge O_b)$$



Outline

Properties of the contract language

Theorem

The following paradoxes are avoided in $\mathcal{CL}$:

- *Ross's paradox*
- *The Free Choice Permission paradox*
- *Sartre's dilemma*
- *The Good Samaritan paradox*
- *Chisholm's paradox*
- *The Gentle Murderer paradox*

Properties of the contract language (II)

Theorem

The following hold in $\mathcal{CL}$:

- $P(\alpha) \equiv \neg F(\alpha)$
- $O(\alpha) \Rightarrow P(\alpha)$
- $P(a) \not\Rightarrow P(a\&b)$
- $F(a) \not\Rightarrow F(a\&b)$
- $F(a\&b) \not\Rightarrow F(a)$
- $P(a\&b) \not\Rightarrow P(a)$

Outline

Model Checking in a Nutshell

A model checker is a software tool that given:

- A model M (usually a *Kripke model*)
- A property ϕ (usually a *temporal logic formula*)

It decides whether

$$M \models \phi$$

- It returns YES if the property is satisfied,
- Otherwise returns NO and provides a counterexample

It is completely automatic!

Model Checking in a Nutshell

A model checker is a software tool that given:

- A model M (usually a *Kripke model*)
- A property ϕ (usually a *temporal logic formula*)

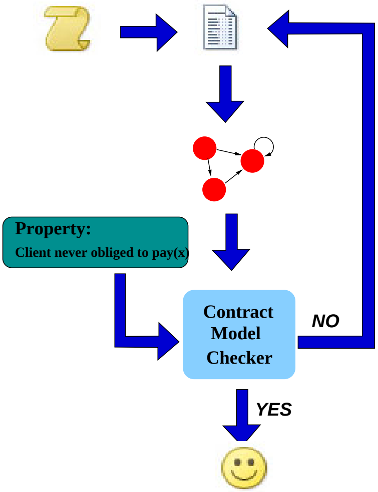
It decides whether

$$M \models \phi$$

- It returns YES if the property is satisfied,
- Otherwise returns NO and provides a counterexample

It is completely automatic!

Model Checking Contracts



Model Checking Contracts

1. Model the conventional contract (in English) as a $\mathcal{CL}$ expression
2. Translate the $\mathcal{CL}$ specification into $\mathcal{C}\mu$
3. Obtain a Kripke-like model (LTS) from the $\mathcal{C}\mu$ formulas
4. Translate the LTS into the input language of NuSMV
5. Perform model checking using NuSMV
   - Check the model is 'good'
   - Check some properties about the client and the provider
6. In case of a counter-example given by NuSMV, interpret it as a $\mathcal{CL}$ clause and repeat the model checking process until the property is satisfied
7. In some cases rephrase the original contract

Model Checking Contracts

1. Model the conventional contract (in English) as a $\mathcal{CL}$ expression
2. Translate the $\mathcal{CL}$ specification into $\mathcal{C}\mu$
3. Obtain a Kripke-like model (LTS) from the $\mathcal{C}\mu$ formulas
4. Translate the LTS into the input language of NuSMV
5. Perform model checking using NuSMV
    - Check the model is 'good'
    - Check some properties about the client and the provider
6. In case of a counter-example given by NuSMV, interpret it as a $\mathcal{CL}$ clause and repeat the model checking process until the property is satisfied
7. In some cases rephrase the original contract

Model Checking Contracts

1. Model the conventional contract (in English) as a $\mathcal{CL}$ expression
2. Translate the $\mathcal{CL}$ specification into $\mathcal{C}\mu$
3. Obtain a Kripke-like model (LTS) from the $\mathcal{C}\mu$ formulas
4. Translate the LTS into the input language of NuSMV
5. Perform model checking using NuSMV
    - Check the model is 'good'
    - Check some properties about the client and the provider
6. In case of a counter-example given by NuSMV, interpret it as a $\mathcal{CL}$ clause and repeat the model checking process until the property is satisfied
7. In some cases rephrase the original contract

Model Checking Contracts

1. Model the conventional contract (in English) as a $\mathcal{CL}$ expression
2. Translate the $\mathcal{CL}$ specification into $\mathcal{C}\mu$
3. Obtain a Kripke-like model (LTS) from the $\mathcal{C}\mu$ formulas
4. Translate the LTS into the input language of NuSMV
5. Perform model checking using NuSMV
    - Check the model is 'good'
    - Check some properties about the client and the provider
6. In case of a counter-example given by NuSMV, interpret it as a $\mathcal{CL}$ clause and repeat the model checking process until the property is satisfied
7. In some cases rephrase the original contract

Model Checking Contracts

1. Model the conventional contract (in English) as a $\mathcal{CL}$ expression
2. Translate the $\mathcal{CL}$ specification into $\mathcal{C}\mu$
3. Obtain a Kripke-like model (LTS) from the $\mathcal{C}\mu$ formulas
4. Translate the LTS into the input language of NuSMV
5. Perform model checking using NuSMV
    - Check the model is 'good'
    - Check some properties about the client and the provider
6. In case of a counter-example given by NuSMV, interpret it as a $\mathcal{CL}$ clause and repeat the model checking process until the property is satisfied
7. In some cases rephrase the original contract

Model Checking Contracts

1. Model the conventional contract (in English) as a $\mathcal{CL}$ expression
2. Translate the $\mathcal{CL}$ specification into $\mathcal{C}\mu$
3. Obtain a Kripke-like model (LTS) from the $\mathcal{C}\mu$ formulas
4. Translate the LTS into the input language of NuSMV
5. Perform model checking using NuSMV
   - Check the model is 'good'
   - Check some properties about the client and the provider
6. In case of a counter-example given by NuSMV, interpret it as a $\mathcal{CL}$ clause and repeat the model checking process until the property is satisfied
7. In some cases rephrase the original contract

Model Checking Contracts

1. Model the conventional contract (in English) as a $\mathcal{CL}$ expression
2. Translate the $\mathcal{CL}$ specification into $\mathcal{C}\mu$
3. Obtain a Kripke-like model (LTS) from the $\mathcal{C}\mu$ formulas
4. Translate the LTS into the input language of NuSMV
5. Perform model checking using NuSMV
   - Check the model is 'good'
   - Check some properties about the client and the provider
6. In case of a counter-example given by NuSMV, interpret it as a $\mathcal{CL}$ clause and repeat the model checking process until the property is satisfied
7. In some cases rephrase the original contract

Case Study
A Contract Example

1. The **Client** shall not:

a) 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**.

6. **Provider** may, at its sole discretion, without notice or giving any reason or incurring any liability for doing so:

a) Suspend Internet Services immediately if **Client** is in breach of Clause 1;

Case Study
A Contract Example

1. The **Client** shall not:

a) 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**.

6. **Provider** may, at its sole discretion, without notice or giving any reason or incurring any liability for doing so:

a) Suspend Internet Services immediately if **Client** is in breach of Clause 1;

Case Study

Translating into $\mathcal{CL}$ syntax

1. $\Box F(fi)$

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**.
6. **Provider** may, at its sole discretion, without notice or giving any reason or incurring any liability for doing so:

a) Suspend Internet Services immediately if **Client** is in breach of Clause 1;

1. $\Box F(fi)$
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**.
6. **Provider** may, at its sole discretion, without notice or giving any reason or incurring any liability for doing so:

a) Suspend Internet Services immediately if **Client** is in breach of Clause 1;

Case Study

Translating into $\mathcal{CL}$ syntax

1. $\Box F_{P(s)}(fi)$
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**.

Case Study

Translating into $\mathcal{CL}$ syntax

1. $\Box F_{P(s)}(fi)$

2. $\Box[h](\phi \Rightarrow O(p + (d\&n)))$

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**.

Case Study

Translating into $\mathcal{CL}$ syntax

1. $\Box F_{P(s)}(fi)$
2. $\Box[h](\phi \Rightarrow O(p + (d\&n)))$
3. $\Box([d\&n](O(l) \wedge [l]\Diamond O(p\&p)))$
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**.

Case Study

Translating into $\mathcal{CL}$ syntax

1. $\Box F_{P(s)}(fi)$
2. $\Box[h](\phi \Rightarrow O(p + (d\&n)))$
3. $\Box([d\&n](O(l) \wedge [l]\Diamond O(p\&p)))$
4. $\Box([d\&n \cdot \bar{l}\,]\Diamond O(p\&p\&p))$
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**.

Case Study

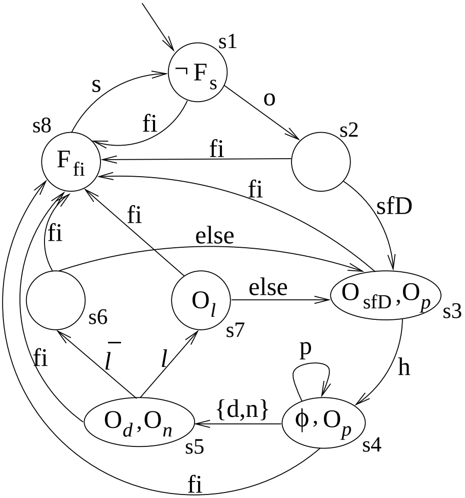
Translating into $\mathcal{CL}$ syntax

1. $\Box F_{P(s)}(fi)$
2. $\Box[h](\phi \Rightarrow O(p + (d\&n)))$
3. $\Box([d\&n](O(l) \wedge [l]\Diamond O(p\&p)))$
4. $\Box([d\&n \cdot \bar{l}\,]\Diamond O(p\&p\&p))$
5. $\Box([o]O(sfD))$

Case Study
Handcrafting the model

$$
\begin{aligned}
\phi &= \text{the Internet traffic is high}\\
\mathit{fi} &= \text{client supplies false information}\\
&\quad\ \text{to Client Relations Department}\\
h &= \text{client increases Internet traffic}\\
&\quad\ \text{to } \textit{high} \text{ level}\\
p &= \text{client pays [price]}\\
d &= \text{client delays payment}\\
n &= \text{client notifies by e-mail}\\
l &= \text{client lowers the Int. traffic}\\
\mathit{sfD} &= \text{client sends the Personal}\\
&\quad\ \text{Data Form to Client Relations}\\
&\quad\ \text{Department}\\
o &= \text{provider activates the Internet}\\
&\quad\ \text{Service (it becomes operative)}\\
s &= \text{provider suspends service}
\end{aligned}
$$

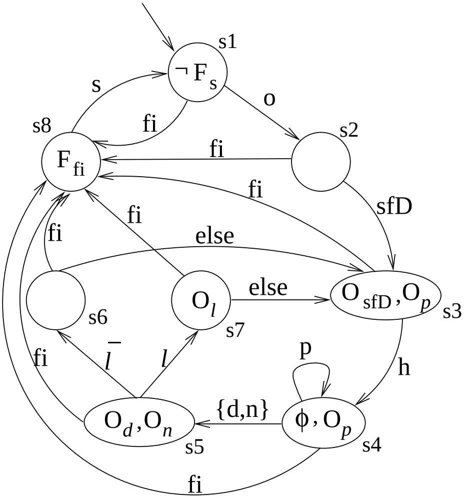
ϕ = the Internet traffic is high
fi = client supplies false information to Client Relations Department
h = client increases Internet traffic to *high* level
p = client pays [price]
d = client delays payment
n = client notifies by e-mail
l = client lowers the Int. traffic
sfD = client sends the Personal Data Form to Client Relations Department
o = provider activates the Internet Service (it becomes operative)
s = provider suspends service



Case Study

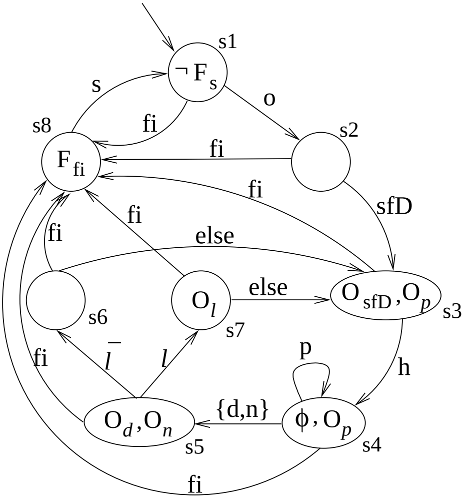
Checking the contract on the model

1. $\Box F_{P(s)}(fi)$
2. $\Box[h](\phi \Rightarrow O(p + (d\&n)))$
3. $\Box([d\&n](O(l) \wedge [l]\Diamond O(p\&p)))$
4. $\Box([d\&n \cdot \bar{l}]\Diamond O(p\&p\&p))$
5. $\Box([o]O(sfD))$



1. $\Box F_{P(s)}(fi)$
2. $\Box[h](\phi \Rightarrow O(p + (d\&n)))$
3. $\Box([d\&n](O(l) \wedge [l]\Diamond O(p\&p)))$
4. $\Box([d\&n \cdot \bar{l}]\Diamond O(p\&p\&p))$
5. $\Box([o]O(sfD))$

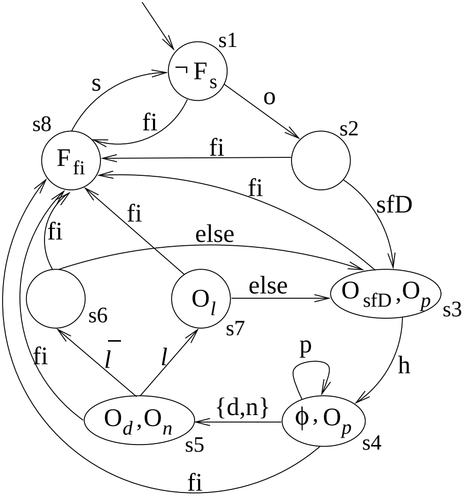
1, 2, and 4: OK



1. $\Box F_{P(s)}(fi)$
2. $\Box[h](\phi \Rightarrow O(p + (d\&n)))$
3. $\Box([d\&n](O(l) \wedge [l]\Diamond O(p\&p)))$
4. $\Box([d\&n \cdot \bar{l}\,]\Diamond O(p\&p\&p))$
5. $\Box([o]O(sfD))$

1, 2, and 4: OK

3 and 5: FAIL!



Failure of 3. It fails since there is a dependency with clause 2

- We need to combine clauses 2 and 3: it model checks!

Failure of 3. It fails since there is a dependency with clause 2

- We need to combine clauses 2 and 3: it model checks!

Failure on our formalization in $\mathcal{CL}$!

Failure of 3. It fails since there is a dependency with clause 2

- We need to combine clauses 2 and 3: it model checks!

Failure on our formalization in $\mathcal{CL}$!

Failure of 5. $(\Box([o]O(sfD)))$

- The system should become operative only once

Failure of 3. It fails since there is a dependency with clause 2

- We need to combine clauses 2 and 3: it model checks!

Failure on our formalization in $\mathcal{CL}$!

Failure of 5. $(\Box([o]O(sfD)))$

- The system should become operative only once

1. We rewrite the original contract
2. This is formulated in $\mathcal{CL}$, written in NuSMV, and it model checks!

Failure of 3. It fails since there is a dependency with clause 2

- We need to combine clauses 2 and 3: it model checks!

Failure on our formalization in $\mathcal{CL}$!

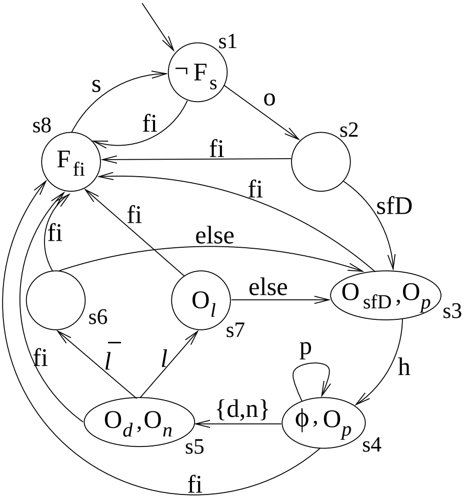
Failure of 5. $(\Box([o]O(sfD)))$

- The system should become operative only once

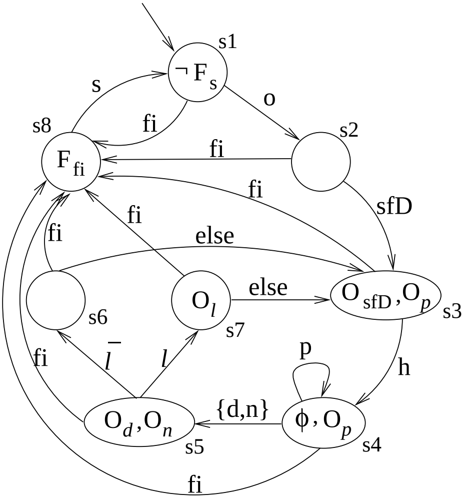
1. We rewrite the original contract
2. This is formulated in $\mathcal{CL}$, written in NuSMV, and it model checks!

'Failure' on the original contract!

- "It is always the case that whenever the Internet traffic is high, if the clients pays immediately, then the client is *not* obliged to pay again immediately afterward"



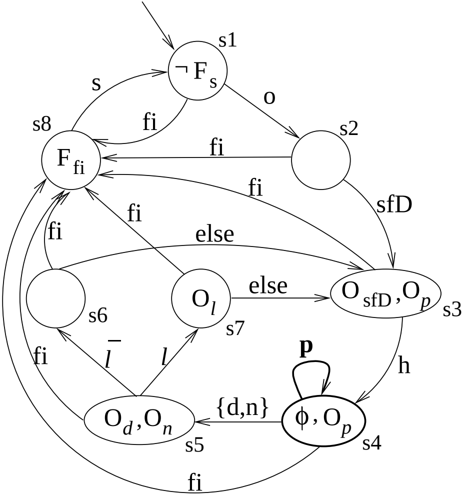
- "It is always the case that whenever the Internet traffic is high, if the clients pays immediately, then the client is *not* obliged to pay again immediately afterward"
- It fails!



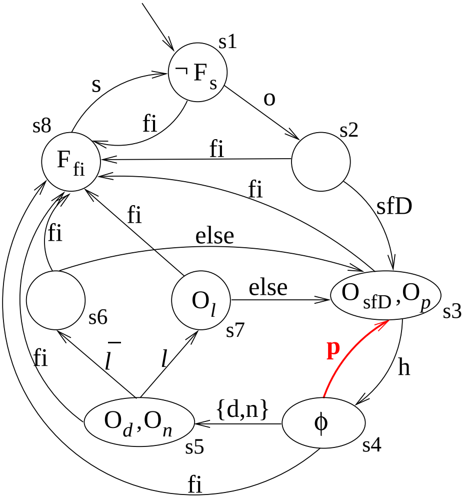
Case Study

Verifying a property about client obligations

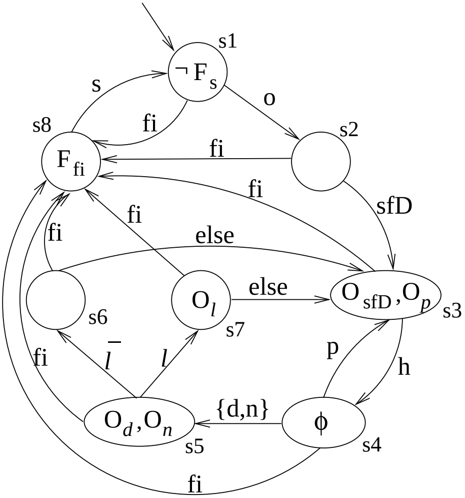
- "It is always the case that whenever the Internet traffic is high, if the clients pays immediately, then the client is *not* obliged to pay again immediately afterward"
- It fails!
- We get a counter-example –Problem: state $s4$



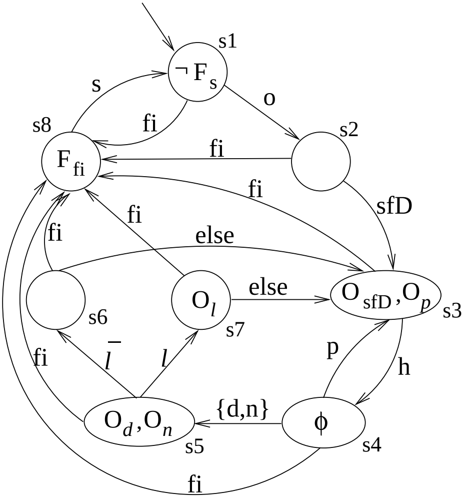
- "It is always the case that whenever the Internet traffic is high, if the clients pays immediately, then the client is *not* obliged to pay again immediately afterward"
- It fails!
- We get a counter-example –Problem: state *s4*
- We modify the original contract to capture the above more precisely



- "It is always the case that whenever Internet traffic is high, if the client delays payment and notifies, and afterward lowers the Internet traffic, then the client is forbidden to increase Internet traffic until he pays twice"



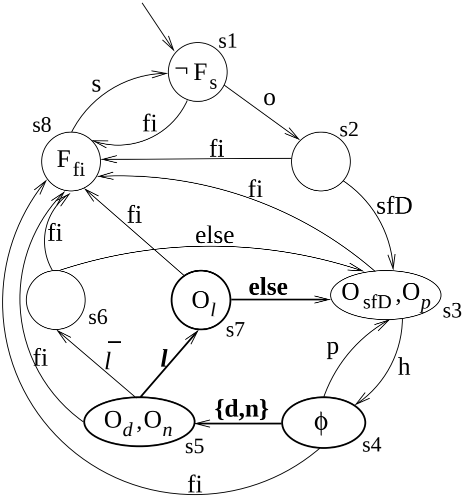
- "It is always the case that whenever Internet traffic is high, if the client delays payment and notifies, and afterward lowers the Internet traffic, then the client is forbidden to increase Internet traffic until he pays twice"
- It fails!



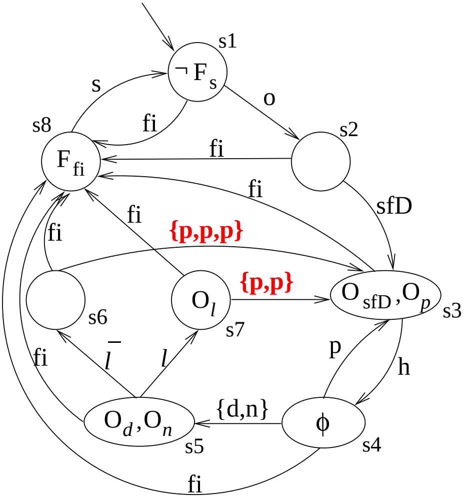
Case Study

Verifying a property about payment in case of increasing Internet traffic

- "It is always the case that whenever Internet traffic is high, if the client delays payment and notifies, and afterward lowers the Internet traffic, then the client is forbidden to increase Internet traffic until he pays twice"
- It fails!
- Counter-example: From s_4 (ϕ holds), after $d\&n \cdot l$, it is possible to increase Internet traffic in state s_7, so neither $F(h)$ nor $\mathrm{done}_{p\&p}$ hold



- "It is always the case that whenever Internet traffic is high, if the client delays payment and notifies, and afterward lowers the Internet traffic, then the client is forbidden to increase Internet traffic until he pays twice"
- It fails!
- Counter-example: From s_4 (ϕ holds), after $d\&n \cdot l$, it is possible to increase Internet traffic in state s_7, so neither $F(h)$ nor $\mathrm{done}_{p\&p}$ hold
- Add to the original contract the clause above!



Links and Papers

- COSoDIS: "Contract-Oriented Software Development for Internet Services" –A Nordunet3 project (`http://www.ifi.uio.no/cosodis`)

- FLACOS'07 – 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (`http://www.ifi.uio.no/flacos07/`)
  - Oslo, 9-10 October 2007

- C. Prisacariu and G. Schneider. **A formal language for electronic contracts.** In FMOODS'07, vol. 4468 of LNCS, pages 174-189, Cyprus. June 2007
- G. Pace, C. Prisacariu, and G. Schneider. **Model checking contracts -a case study.** In ATVA'07, vol. 4762 of LNCS, pages 82-97, Tokyo, Japan. October 2007.

Research Topics

- Improve the language/logic $\mathcal{CL}$ for contracts
- Develop a proof system for $\mathcal{CL}$
- Develop a theory of contracts
- Case studies
- Find other application domains to use contracts (e.g., financial)
- Implement algorithms for model checking
- Implement web services/components including contracts as presented in this tutorial

Thank you!