The Coinductive Approach to Verifying Cryptographic Protocols

Jesse Hughes

joint work with **Martijn Warnier**

jesseh@cs.kun.nl

University of Nijmegen

Outline

I. Cryptographic protocols in general

Outline

I. Cryptographic protocols in general

II. An example protocol

Outline

I. Cryptographic protocols in general

II. An example protocol

III. Coalgebra primer

Outline

I. Cryptographic protocols in general

II. An example protocol

III. Coalgebra primer

IV. Temporal operators/Galois algebras

Outline

I. Cryptographic protocols in general

II. An example protocol

III. Coalgebra primer

IV. Temporal operators/Galois algebras

V. The specification language CCSL

Outline

I. Cryptographic protocols in general
II. An example protocol
III. Coalgebra primer
IV. Temporal operators/Galois algebras
V. The specification language CCSL
VI. The CCSL compiler

Outline

I. Cryptographic protocols in general
II. An example protocol
III. Coalgebra primer
IV. Temporal operators/Galois algebras
V. The specification language CCSL
VI. The CCSL compiler
VII. Security protocols revisited

Outline

I. Cryptographic protocols in general
II. An example protocol
III. Coalgebra primer
IV. Temporal operators/Galois algebras
V. The specification language CCSL
VI. The CCSL compiler
VII. Security protocols revisited
VIII. Paulson’s inductive method

Outline

I. Cryptographic protocols in general
II. An example protocol
III. Coalgebra primer
IV. Temporal operators/Galois algebras
V. The specification language CCSL
VI. The CCSL compiler
VII. Security protocols revisited
VIII. Paulson's inductive method

Outline

I. Cryptographic protocols in general
II. An example protocol
III. Coalgebra primer
IV. Temporal operators/Galois algebras
V. The specification language CCSL
VI. The CCSL compiler
VII. Security protocols revisited
VIII. Paulson's inductive method

Outline

Outline

I. Cryptographic protocols in general
II. An example protocol
III. Coalgebra primer
IV. Temporal operators/Galois algebras
V. The specification language CCSL
VI. The CCSL compiler
VII. Security protocols revisited
VIII. Paulson's inductive method

Part One: The Background

Cryptographic Protocols

Cryptographic Protocols, Abstract representation for:

Cryptographic Protocols

Cryptographic Protocols, Abstract representation for:

- Distributing secret keys over an open (insecure) network.

Cryptographic Protocols

Cryptographic Protocols, Abstract representation for:

- Distributing secret keys over an open (insecure) network.
- Authenticating principals to each other.

Cryptographic Protocols

Cryptographic Protocols, Abstract representation for:

- Distributing secret keys over an open (insecure) network.
- Authenticating principals to each other.
- Assuring secrecy of message content.

Cryptographic Protocols

Cryptographic Protocols, Abstract representation for:

- Distributing secret keys over an open (insecure) network.
- Authenticating principals to each other.
- Assuring secrecy of message content.
- Assuring integrity of messages.

Cryptographic Protocols

Cryptographic Protocols, Abstract representation for:

- Distributing secret keys over an open (insecure) network.
- Authenticating principals to each other.
- Assuring secrecy of message content.
- Assuring integrity of messages.
- A combination of all of the above.

An example Protocol

Bilateral Key Exchange with Public Key Protocol: a simple protocol for distributing a symmetric key.

An example Protocol

Bilateral Key Exchange with Public Key Protocol: a simple protocol for distributing a symmetric key.

1. $\mathbf{B} \rightarrow \mathbf{A}: \quad B, \{N_B, B\}_{\mathrm{pk}_A}$

Principal **B** sends to Principal **A** a message containing

- his name, B,

An example Protocol

Bilateral Key Exchange with Public Key Protocol: a simple protocol for distributing a symmetric key.

$$1.\quad \mathbf{B} \rightarrow \mathbf{A}: \quad B, \{N_B, B\}_{\mathrm{pk}_A}$$

Principal **B** sends to Principal **A** a message containing

- his name, B,
- and a nonce, N_B.

An example Protocol

Bilateral Key Exchange with Public Key Protocol: a simple protocol for distributing a symmetric key.

$$1. \quad \mathbf{B} \rightarrow \mathbf{A} : \quad B, \{N_B, B\}_{\mathrm{pk}_A}$$

Principal **B** sends to Principal **A** a message containing

- his name, B,
- and a nonce, N_B.

This is (mostly) encrypted with **A**'s public key, pk_A.

An example Protocol

Bilateral Key Exchange with Public Key Protocol: a simple protocol for distributing a symmetric key.

1. $\mathbf{B} \rightarrow \mathbf{A} : \quad B, \{N_B, B\}_{\mathrm{pk}_A}$
2. $\mathbf{A} \rightarrow \mathbf{B} : \quad \{\mathrm{Sha}(N_B), N_A, A, K_{AB}\}_{\mathrm{pk}_B}$

A replies with a message containing

- a hash of **B**’s nonce,

An example Protocol

Bilateral Key Exchange with Public Key Protocol: a simple protocol for distributing a symmetric key.

1. $\mathbf{B} \rightarrow \mathbf{A}: \quad B, \{N_B, B\}_{\mathrm{pk}_A}$
2. $\mathbf{A} \rightarrow \mathbf{B}: \quad \{\mathrm{Sha}(N_B), N_A, A, K_{AB}\}_{\mathrm{pk}_B}$

A replies with a message containing

- a hash of **B**'s nonce,
- a fresh nonce, N_A,

An example Protocol

Bilateral Key Exchange with Public Key Protocol: a simple protocol for distributing a symmetric key.

$$
\begin{array}{lll}
1. & \mathbf{B} \rightarrow \mathbf{A}: & B, \{N_B, B\}_{\mathrm{pk}_A} \\
2. & \mathbf{A} \rightarrow \mathbf{B}: & \{\mathrm{Sha}(N_B), N_A, A, K_{AB}\}_{\mathrm{pk}_B}
\end{array}
$$

A replies with a message containing

- a hash of **B**'s nonce,
- a fresh nonce, N_A,
- **A**'s name, A,

An example Protocol

Bilateral Key Exchange with Public Key Protocol: a simple protocol for distributing a symmetric key.

$$\begin{array}{lll} 1. & \mathbf{B} \rightarrow \mathbf{A}: & B, \{N_B, B\}_{\mathrm{pk}_A} \\ 2. & \mathbf{A} \rightarrow \mathbf{B}: & \{\mathrm{Sha}(N_B), N_A, A, K_{AB}\}_{\mathrm{pk}_B} \end{array}$$

A replies with a message containing

- a hash of **B**'s nonce,
- a fresh nonce, N_A,
- **A**'s name, A,
- and a key, K_{AB}.

An example Protocol

Bilateral Key Exchange with Public Key Protocol: a simple protocol for distributing a symmetric key.

1. $\mathbf{B} \rightarrow \mathbf{A}: \quad B, \{N_B, B\}_{\mathrm{pk}_A}$
2. $\mathbf{A} \rightarrow \mathbf{B}: \quad \{\mathrm{Sha}(N_B), N_A, A, K_{AB}\}_{\mathrm{pk}_B}$

A replies with a message containing

- a hash of **B**'s nonce,
- a fresh nonce, N_A,
- **A**'s name, A,
- and a key, K_{AB}.

All of this is encrypted with **B**'s public key, pk_B.

An example Protocol

Bilateral Key Exchange with Public Key Protocol: a simple protocol for distributing a symmetric key.

1. $\mathbf{B} \rightarrow \mathbf{A} : \quad B, \{N_B, B\}_{\mathrm{pk}_A}$
2. $\mathbf{A} \rightarrow \mathbf{B} : \quad \{\mathrm{Sha}(N_B), N_A, A, K_{AB}\}_{\mathrm{pk}_B}$
3. $\mathbf{B} \rightarrow \mathbf{A} : \quad \{\mathrm{Sha}(N_A)\}_{K_{AB}}$

B replies with a hash of N_A

An example Protocol

Bilateral Key Exchange with Public Key Protocol: a simple protocol for distributing a symmetric key.

1. $\mathbf{B} \to \mathbf{A} :\quad B, \{N_B, B\}_{\mathrm{pk}_A}$
2. $\mathbf{A} \to \mathbf{B} :\quad \{\mathrm{Sha}(N_B), N_A, A, K_{AB}\}_{\mathrm{pk}_B}$
3. $\mathbf{B} \to \mathbf{A} :\quad \{\mathrm{Sha}(N_A)\}_{K_{AB}}$

B replies with a hash of N_A encrypted with the session key K_{AB}.

Analysis

It would be nice to know that, if two participants use a protocol, the outcome is good.

Analysis

It would be nice to know that, if two participants use a protocol, the outcome is good.

- No one learns the key they agree to use;

Analysis

It would be nice to know that, if two participants use a protocol, the outcome is good.

- No one learns the key they agree to use;
- Both of them know the key;

Analysis

It would be nice to know that, if two participants use a protocol, the outcome is good.

- No one learns the key they agree to use;
- Both of them know the key;
- Each is aware the other knows the key.

Analysis

It would be nice to know that, if two participants use a protocol, the outcome is good.

- No one learns the key they agree to use;
- Both of them know the key;
- Each is aware the other knows the key.

For this, we need an appropriate model in which to reason about the protocols.

Analysis

It would be nice to know that, if two participants use a protocol, the outcome is good.

- No one learns the key they agree to use;
- Both of them know the key;
- Each is aware the other knows the key.

For this, we need an appropriate model in which to reason about the protocols.
We analyze the protocol using a Dolev-Yao security model.
That is, we create a model consisting of

- any number of “normal” agents and

Analysis

It would be nice to know that, if two participants use a protocol, the outcome is good.

- No one learns the key they agree to use;
- Both of them know the key;
- Each is aware the other knows the key.

For this, we need an appropriate model in which to reason about the protocols.
We analyze the protocol using a Dolev-Yao security model.
That is, we create a model consisting of

- any number of “normal” agents and
- one very powerful spy.

Analysis

It would be nice to know that, if two participants use a protocol, the outcome is good.

- No one learns the key they agree to use;
- Both of them know the key;
- Each is aware the other knows the key.

For this, we need an appropriate model in which to reason about the protocols.
We analyze the protocol using a Dolev-Yao security model.
That is, we create a model consisting of

- any number of “normal” agents and
- one very powerful spy.

We then prove that the conditions above hold.

Analysis

Requirements:

needed for:

- to model abstract data types messages

Analysis

Requirements:

	needed for:
• to model abstract data types	messages
• to model dynamic systems	users' knowledge

Analysis

Requirements:

	needed for:
• to model abstract data types	messages
• to model dynamic systems	users' knowledge
• to use temporal reasoning	correctness conditions

Analysis

Requirements:

	needed for:
• to model abstract data types	messages
• to model dynamic systems	users' knowledge
• to use temporal reasoning	correctness conditions

The language CCSL allows all of this.

Analysis

Requirements:

theory:

- to model abstract data types algebra
- to model dynamic systems
- to use temporal reasoning

The language CCSL allows all of this.

CCSL is built upon an abstract mathematical foundation.

Analysis

Requirements:

	theory:
• to model abstract data types	algebra
• to model dynamic systems	coalgebra
• to use temporal reasoning	

The language CCSL allows all of this.

CCSL is built upon an abstract mathematical foundation.

Analysis

Requirements:

	theory:
• to model abstract data types	algebra
• to model dynamic systems	coalgebra
• to use temporal reasoning	Galois algebra

The language CCSL allows all of this.

CCSL is built upon an abstract mathematical foundation.

Part Two: The Theory

Algebra primer

Let Σ be a signature, i.e.,

$$\Sigma = \{f_i^{(n_i)} \mid i \in I\}.$$

Algebra primer

Let Σ be a signature, i.e.,

$$\Sigma = \{f_i^{(n_i)} \mid i \in I\}.$$

A Σ-algebra is a set A together with an interpretation for each f_i.

Algebra primer

Example: $\Sigma = \{e, -^{-1}, \times\}$.

$$\begin{tikzcd}
1 \arrow[dr, "e"'] & A \arrow[d, "-^{-1}" description] & A \times A \arrow[dl, "\times"] \\
 & A &
\end{tikzcd}$$

Algebra primer

Example: $\Sigma = \{e, -^{-1}, \times\}$.

$$\begin{array}{c} 1 \; + \; A \; + \; A \times A \\ \downarrow \\ A \end{array}$$

Algebra primer

Example: $\Sigma = \{e, -^{-1}, \times\}$.

$$
\begin{array}{ccc}
1 + A + A \times A & \qquad & FA \\
\downarrow & & \downarrow \\
A & & A
\end{array}
$$

Let F:**SET**$\rightarrow$**SET** be given. An F-algebra is a set A with a structure

$$
\begin{array}{c}
FA \\
\downarrow \\
A
\end{array}
$$

Algebra primer

Example: $\Sigma = \{e, -^{-1}, \times\}$.

$$
\begin{array}{ccc}
1 + A + A \times A & \qquad & FA \\
\downarrow & & \downarrow \\
A & & A
\end{array}
$$

For polynomial functors, an F-algebra is a universal algebra.

Coalgebra primer

Example:

$$
\begin{array}{ccc}
1 + A + A \times A & \qquad & FA \\
\downarrow & & \downarrow \\
A & & A
\end{array}
$$

Coalgebra primer

Example:

$$\begin{tikzcd}[column sep=small] 1 + & A \arrow[from=2-2, u] & + A \times A & & FA \\ & A & & & A \arrow[u] \end{tikzcd}$$

An F-coalgebra is a set A with a structure

$$\begin{tikzcd} FA \\ A \arrow[u] \end{tikzcd}$$

Coalgebra primer

Example:

$$\begin{array}{ccc} 1 + A + A \times A & \qquad & FA \\ \uparrow & & \uparrow \\ A & & A \end{array}$$

An F-coalgebra is a set A with a structure

$$\begin{array}{c} FA \\ \uparrow \\ A \end{array}$$

Think: a coalgebra is a set in which each element can be decomposed as elements of a structured set.

Coalgebra primer

Example:

$$\begin{array}{ccc} 1 + A + A \times A & \qquad & FA \\ \uparrow & & \uparrow \\ A & & A \end{array}$$

Coalgebras model non-well-founded structures, including infinitary trees, streams, etc.

Coalgebra primer

Example:

$$
\begin{array}{ccc}
1 + A + A \times A & \qquad & FA \\
\uparrow & & \uparrow \\
A & & A
\end{array}
$$

Coalgebras can also represent dynamic systems.

Coalgebra primer

Example:

$$
\begin{tikzcd}
1 + A + A \times A & & F A \\
A \arrow[u] & & A \arrow[u]
\end{tikzcd}
$$

Coalgebras can also represent dynamic systems.

In security protocols, the principals' knowledge changes over time as messages are sent and received.

Coalgebra primer

Example:

$$
\begin{array}{ccc}
1 + A + A \times A & \qquad & FA \\
\uparrow & & \uparrow \\
A & & A
\end{array}
$$

Coalgebras can also represent dynamic systems.

In security protocols, the principals' knowledge changes over time as messages are sent and received.

Hence, we use a coalgebraic model.

Coalgebraic signatures

An algebraic signature is given by declarations:

$$f_i : X^{n_i} \longrightarrow X$$

Coalgebraic signatures

An algebraic signature is given by declarations:

$$f_i : F_i X \longrightarrow X$$

Coalgebraic signatures

An algebraic signature is given by declarations:

$$f_i : F_i X \longrightarrow X$$

Equivalently,

$$f : \coprod_i F_i X \longrightarrow X$$

Coalgebraic signatures

An algebraic signature is given by declarations:

$$f_i : F_i X \longrightarrow X$$

Equivalently,

$$f : \coprod_i F_i X \longrightarrow X$$

A coalgebraic signature is given by declarations

$$f_i : X \longrightarrow F_i X$$

Coalgebraic signatures

An algebraic signature is given by declarations:

$$f_i : F_i X \longrightarrow X$$

Equivalently,

$$f : \coprod_i F_i X \longrightarrow X$$

A coalgebraic signature is given by declarations

$$f_i : X \longrightarrow F_i X$$

Equivalently,

$$f : X \longrightarrow \prod_i F_i X$$

Examples

FX	Initial algebra	Final coalgebra
$Z \times X$	$\emptyset$	infinite streams

Examples

FX	Initial algebra	Final coalgebra
$Z \times X$	$\emptyset$	infinite streams
$1 + Z \times X$	finite streams	finite and infinite streams

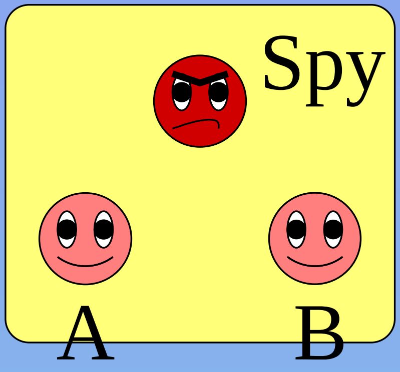
Examples

FX	Initial algebra	Final coalgebra
$Z \times X$	$\emptyset$	infinite streams
$1 + Z \times X$	finite streams	finite and infinite streams
$1 + X \times X$	finite trees	finite and infinite trees

Examples

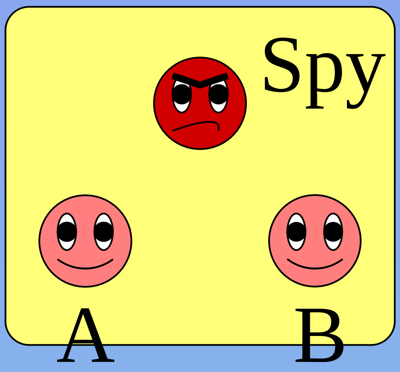
FX	Initial algebra	Final coalgebra
$Z \times X$	$\emptyset$	infinite streams
$1 + Z \times X$	finite streams	finite and infinite streams
$1 + X \times X$	finite trees	finite and infinite trees
$\mathcal{P}_\omega X$	finite, arb. branching trees	Kripke frame

Our coalgebra



Consider a run with three principals: **A**, **B** and the **Spy**.

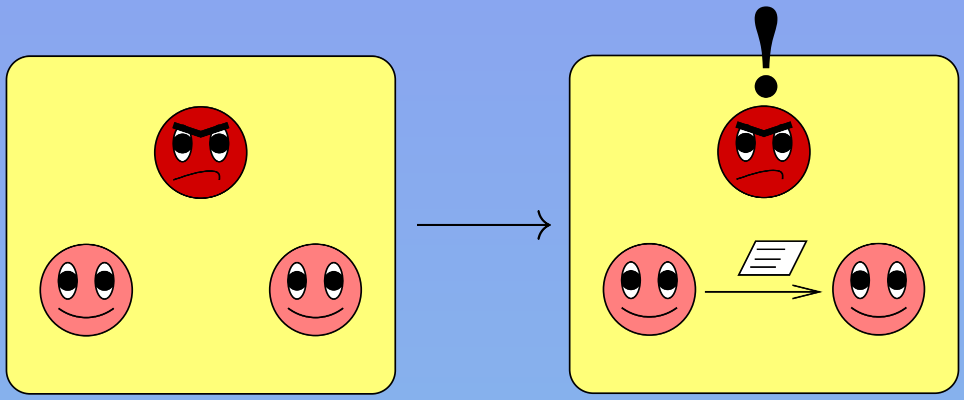
Our coalgebra



Consider a run with three principals: **A**, **B** and the **Spy**.

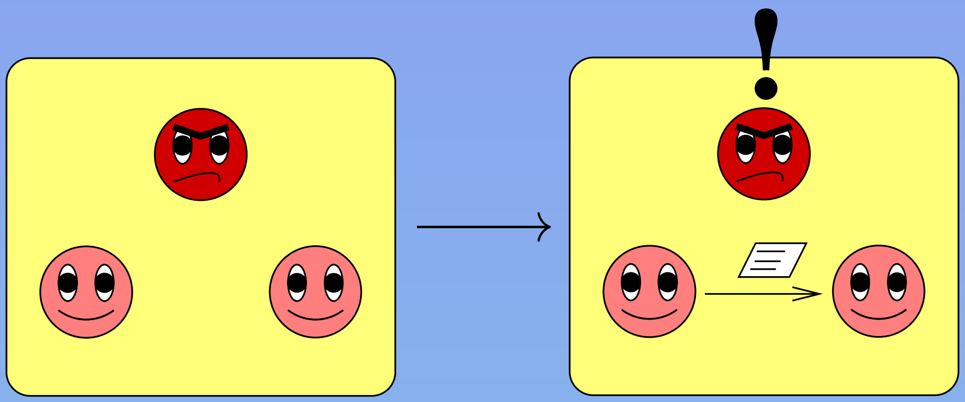
Suppose that **A** sends a message to **B**.

Our coalgebra



Then, in the next instant, the **Spy** learns the message.

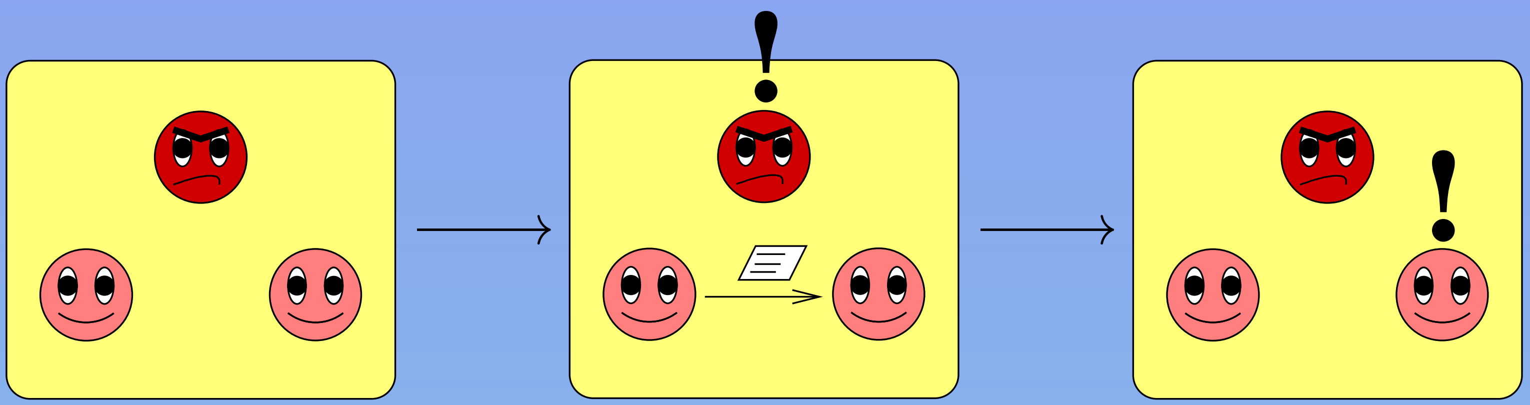
Our coalgebra



Then, in the next instant, the **Spy** learns the message.

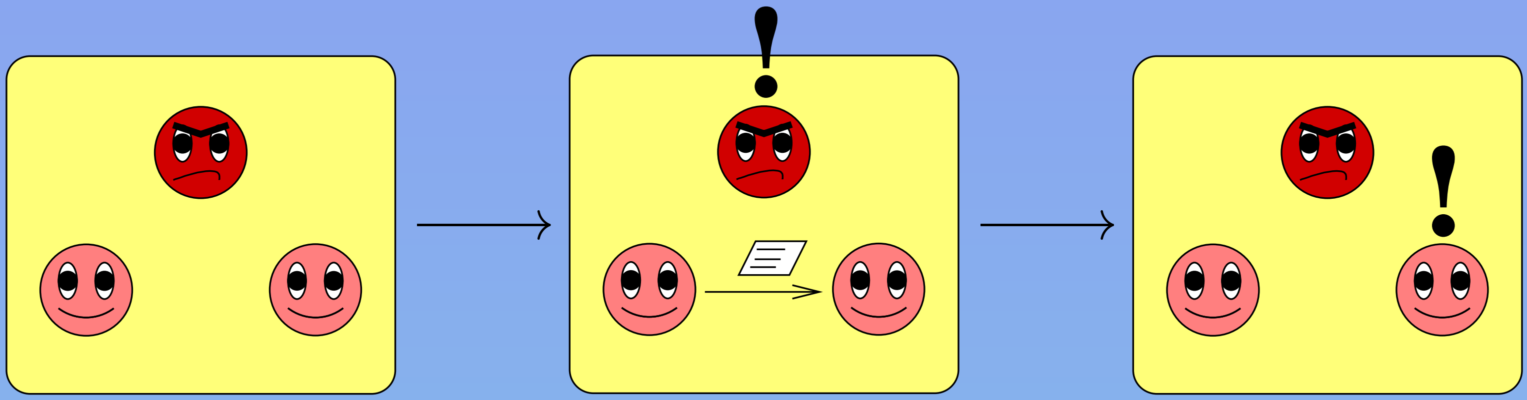
Supposing that the message arrives at that time, then…

Our coalgebra



…the next instant, **B** learns the message, too.

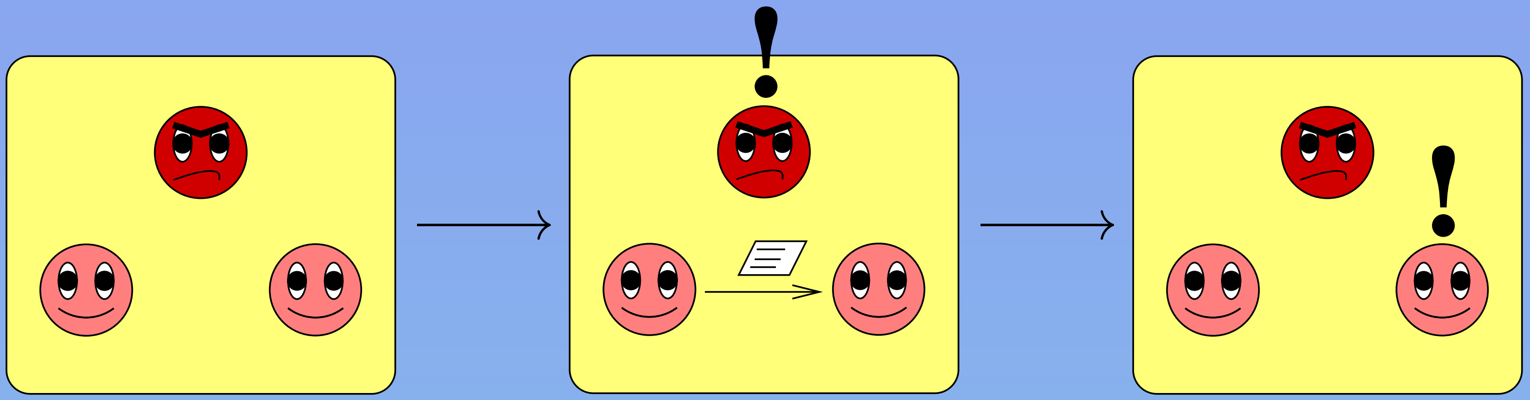
Our coalgebra



So, to describe this system, we use a coalgebra with

- a method giving the next state,

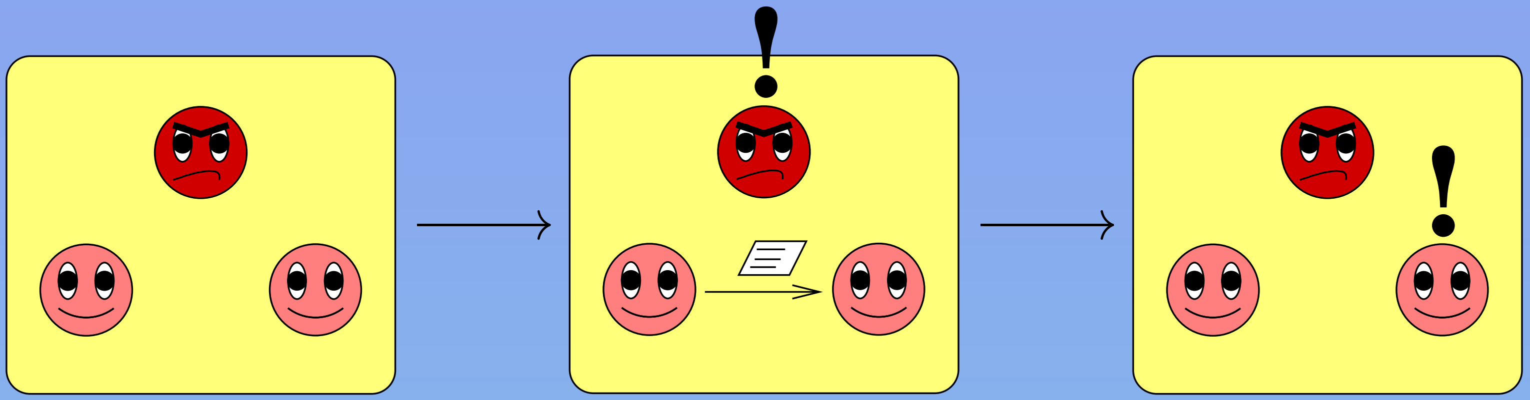
Our coalgebra



So, to describe this system, we use a coalgebra with

- a method giving the next state,
- attributes describing the action occurring,

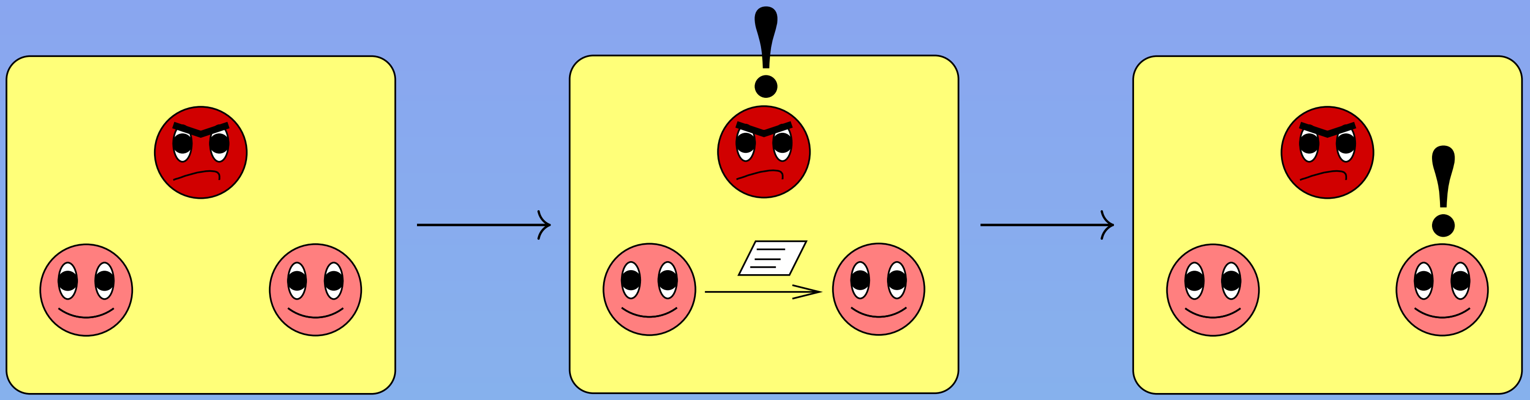
Our coalgebra



So, to describe this system, we use a coalgebra with

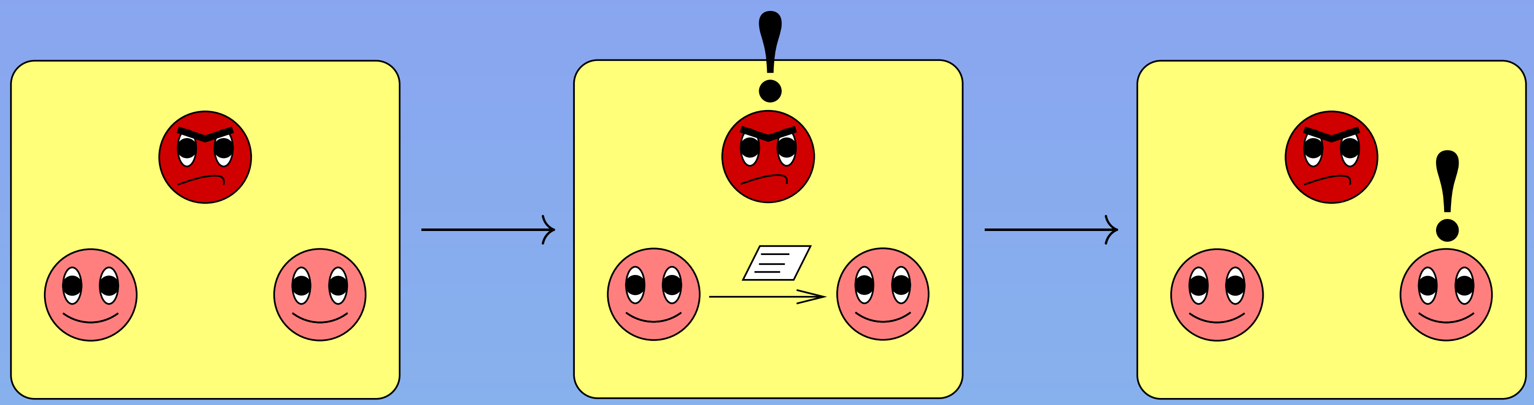
- a method giving the next state,
- attributes describing the action occurring,
- attributes describing the participants' knowledge.

Our coalgebra



$$
\begin{array}{l}
\mathsf{MsgContext} : \textbf{CLASSSPEC} \\
\quad \textbf{METHOD} \\
\qquad \texttt{next} \colon \mathsf{Self} \longrightarrow \mathsf{Self} \\
\qquad \texttt{action} \colon \mathsf{Self} \longrightarrow \{\textbf{idle}, \textbf{sent}, \textbf{received}\} \\
\qquad \texttt{knows} \colon \mathsf{Self} \times \mathsf{Princ} \longrightarrow [\mathsf{Message} \longrightarrow \mathsf{Bool}]
\end{array}
$$

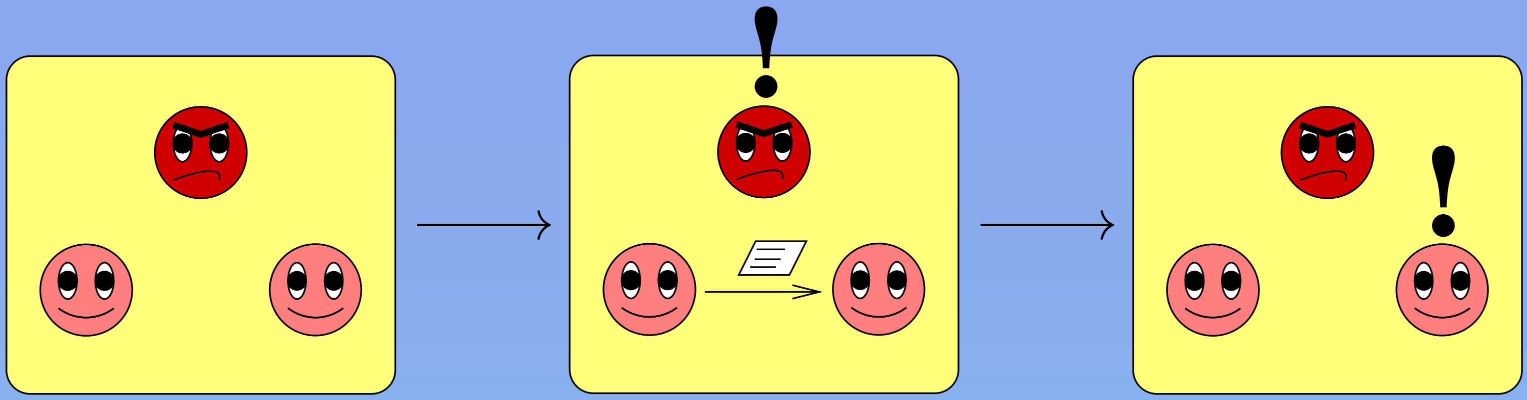
Our coalgebra



We would like to prove, e.g., that

*The **Spy** never learns the session key.*

Our coalgebra

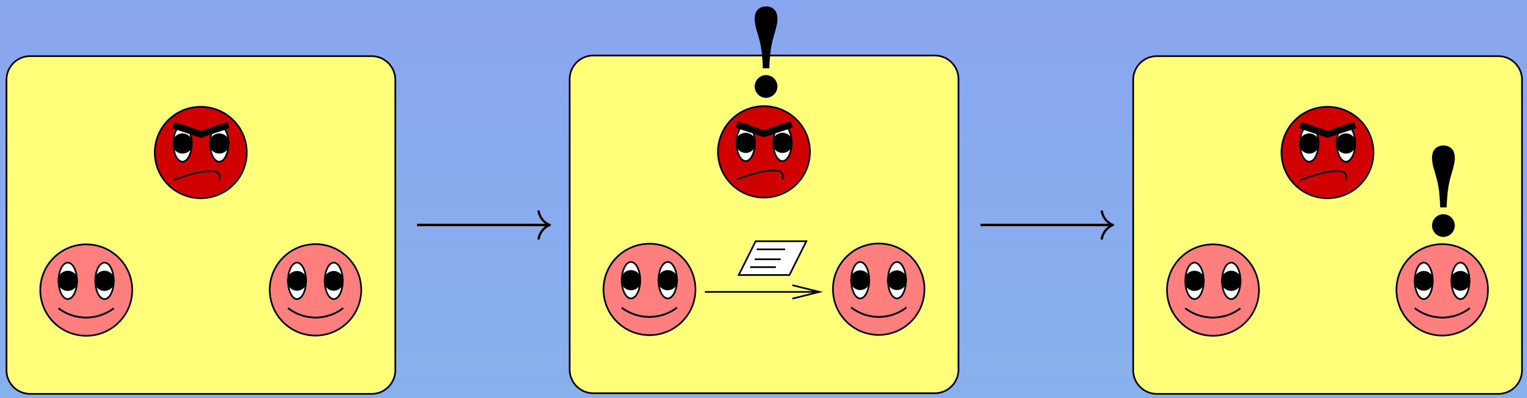


We would like to prove, e.g., that

*The **Spy** never learns the session key.*

For this, we need to reason temporally.

Our coalgebra



We would like to prove, e.g., that

*The **Spy** never learns the session key.*

For this, we need to reason temporally.

Categories of coalgebras come with temporal operators, which we can understand in terms of Galois algebras.

Galois algebras

A Galois algebra is a complete, Boolean algebra $\mathbb{P}$ together with an operation

$$[\,] : \mathbb{P} \longrightarrow \mathbb{P}$$

which preserves meets.

Galois algebras

A Galois algebra is a complete, Boolean algebra $\mathbb{P}$ together with an operation

$$[\,] : \mathbb{P} \longrightarrow \mathbb{P}$$

which preserves meets.

Think: $[\,]P(x)$ means P holds for all successor states of x.

Galois algebras

A Galois algebra is a complete, Boolean algebra $\mathbb{P}$ together with an operation

$$[\,] : \mathbb{P} \longrightarrow \mathbb{P}$$

which preserves meets.

Think: $[\,]P(x)$ means P holds for all successor states of x.

With just these assumptions, we can develop a remarkable amount of temporal logic.

Galois algebras

$$\langle\ \rangle^{\leftarrow} \quad \dashv \quad [\ ]$$

$[\ ]$ is part of a Galois connection, with left adjoint $\langle\ \rangle^{\leftarrow}$.

Galois algebras

$$\begin{array}{ccc} \langle\ \rangle^{\leftarrow} & \dashv & [\ ] \\ [\ ]^{\leftarrow} & & \langle\ \rangle \end{array}$$

Each operator has a conjugate,

$$[\ ]^{\leftarrow} = \neg\langle\ \rangle^{\leftarrow}\neg$$
$$\langle\ \rangle = \neg[\ ]\neg$$

Galois algebras

$$\langle\;\rangle^{\leftarrow} \quad \dashv \quad [\;]$$

$$[\;]^{\leftarrow} \quad \vdash \quad \langle\;\rangle$$

This yields another Galois connection.

Galois algebras

$$\langle\,\rangle^{\leftarrow} \quad \dashv \quad [\,] \qquad \text{Next time}$$

$$[\,]^{\leftarrow} \quad \vdash \quad \langle\,\rangle$$

In our interpretation, $[\,]$ means "in every next state".

$$[\,]P = \{p \mid \forall p \to r \,.\, P(r)\}$$

Galois algebras

$$\langle\;\rangle^{\leftarrow} \quad \dashv \quad [\;] \qquad \text{Next time}$$

$$[\;]^{\leftarrow} \quad \vdash \quad \langle\;\rangle$$

In our interpretation, $[\;]$ means “in every next state”.

$$[\;]P = \{p \mid \forall p \to r \,.\, P(r)\}$$

A proposition P such that P implies $[\;]P$ is called an $invariant$.

Galois algebras

$$\langle\,\rangle^{\leftarrow} \quad \dashv \quad [\,] \qquad \text{Next time}$$

$$[\,]^{\leftarrow} \quad \vdash \quad \langle\,\rangle$$

In our interpretation, $[\,]$ means “in every next state”.

$$[\,]P = \{p \mid \forall p \rightarrow r \,.\, P(r)\}$$

A proposition P such that P implies $[\,]P$ is called an $invariant$.
Invariants are the coalgebraic analogues to inductive predicates.

Galois algebras

Some time preceding	$\langle\,\rangle^{\leftarrow}$	$\dashv$	$[\,]$	Next time
Always preceding	$[\,]^{\leftarrow}$	$\vdash$	$\langle\,\rangle$	Some next time

This induces the remaining interpretations.

Galois algebras

Some time preceding	$\langle\ \rangle^{\leftarrow}$	$\dashv$	$[\ ]$	Next time
Always preceding	$[\ ]^{\leftarrow}$	$\vdash$	$\langle\ \rangle$	Some next time

This induces the remaining interpretations.

Galois algebras

Some time preceding	$\langle\ \rangle^{\leftarrow}$	$\dashv$	$[\ ]$	Next time
Always preceding	$[\ ]^{\leftarrow}$	$\vdash$	$\langle\ \rangle$	Some next time

This induces the remaining interpretations.

Galois algebras

Some time preceding	$\langle\ \rangle^{\leftarrow}$	$\dashv$	$[\ ]$	Next time
Always preceding	$[\ ]^{\leftarrow}$	$\vdash$	$\langle\ \rangle$	Some next time

This allows us to represent statements like

> *If **B** receives a message at time t, then **B** knows the message at $t+1$.*

Galois algebras

Some time preceding	$\overleftarrow{\langle\ \rangle}$	$\dashv$	$[\ ]$	Next time
Always preceding	$\overleftarrow{[\ ]}$	$\vdash$	$\langle\ \rangle$	Some next time

Note: from just a complete partial order with a meet-preserving operator, we get the remaining three operators.

Galois algebras

Some time preceding	$\langle\,\rangle^{\leftarrow}$	$\dashv$	$[\,]$	Next time
Always preceding	$[\,]^{\leftarrow}$	$\vdash$	$\langle\,\rangle$	Some next time

Note: from just a complete partial order with a meet-preserving operator, we get the remaining three operators.

But wait! There's more…

Fixed point operators

$\Box$ Always

We can define an "always" operator via a fixed point construction:

$$\Box P = \nu Z \,.\, P \wedge [\,]Z$$

Fixed point operators

$\square$ Always

We can define an "always" operator via a fixed point construction:

$$\square P = \nu Z \,.\, P \wedge [\,]Z$$

$\square P$ is the greatest invariant contained in P.

Fixed point operators

$\Box$ Always

We can define an "always" operator via a fixed point construction:

$$\Box P = \nu Z \,.\, P \wedge [\,]Z$$

$\Box P$ is the greatest invariant contained in P.

This operator preserves meets, so we have *another* Galois algebra.

Fixed point operators

$$\text{Once} \quad \overleftarrow{\Diamond} \quad \dashv \quad \Box \quad \text{Always}$$

$$\text{Previously} \quad \overleftarrow{\Box} \quad \vdash \quad \Diamond \quad \text{Eventually}$$

This yields the remaining operators and interpretations.

Fixed point operators

Once	$\overleftarrow{\Diamond}$	$\dashv$	$\Box$	Always
Previously	$\overleftarrow{\Box}$	$\vdash$	$\Diamond$	Eventually

Now, we can represent statements like

> *The **Spy** never learns the private keys of the other principals.*

Fixed point operators

Once	$\overleftarrow{\Diamond}$	$\dashv$	$\Box$	Always
Previously	$\overleftarrow{\Box}$	$\vdash$	$\Diamond$	Eventually

All of this structure just comes from the presence of the "next time" operator, $[\,]$.

Part Three: CCSL

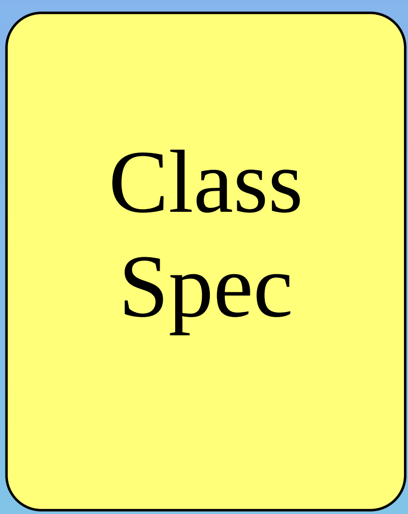
Overview

The mathematical theories of algebra, coalgebra and Galois algebras give us a number of tools for reasoning about class specifications.

Overview

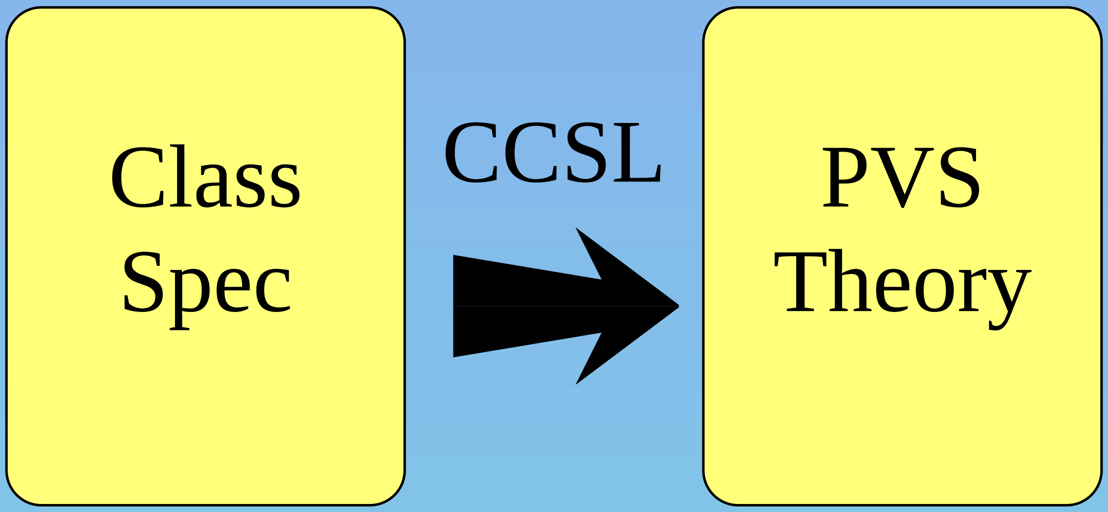
The mathematical theories of algebra, coalgebra and Galois algebras give us a number of tools for reasoning about class specifications.

CCSL provides a means for expressing a class specification in terms of these theories.



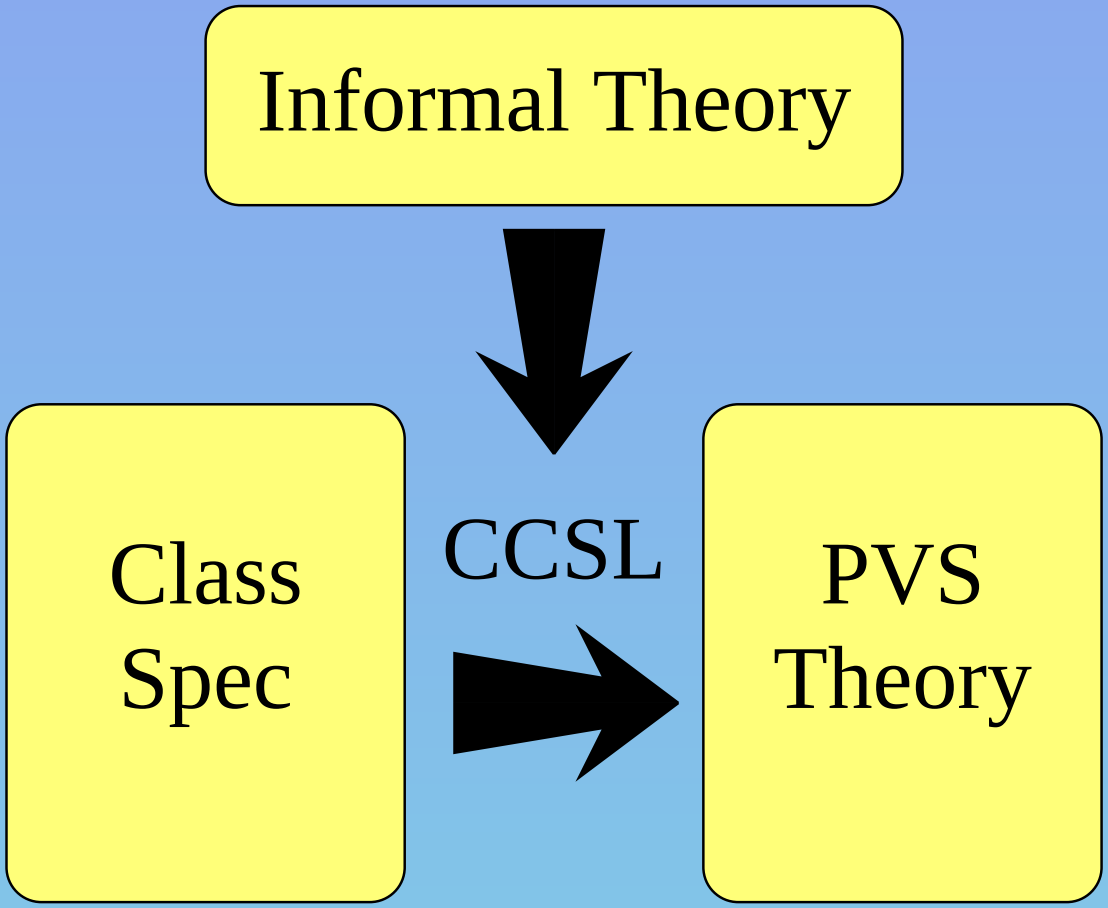
Overview

The compiler translates a specification into a formal, logical theory (in PVS/Isabelle).



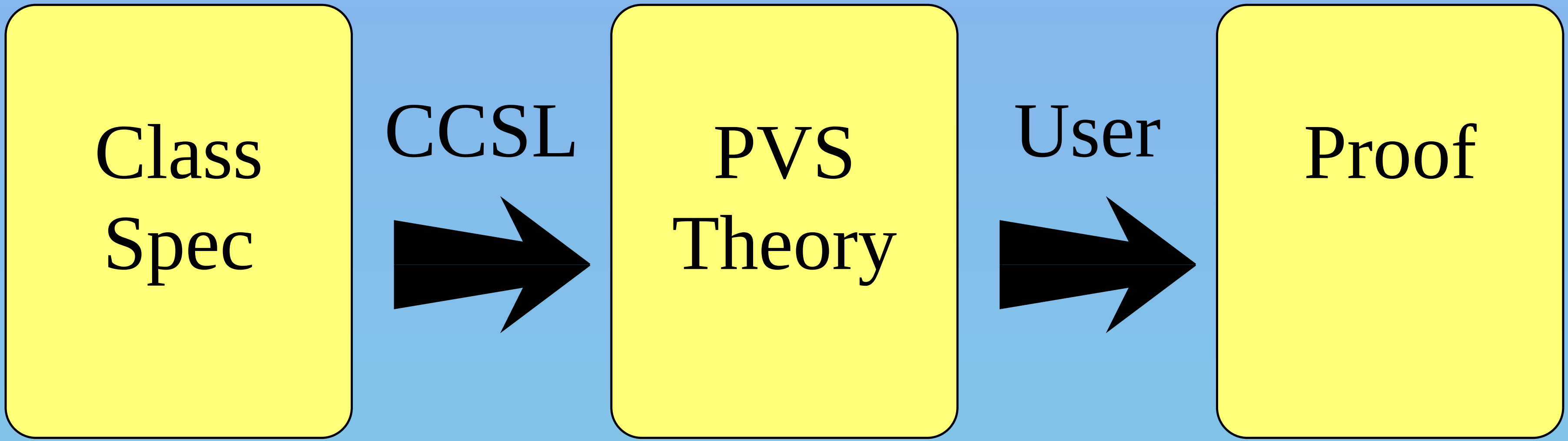
Overview

This theory includes induction (algebra), coinduction (coalgebra), temporal axioms (Galois algebra), etc.



Overview

The user then proves the correctness of the specification in the theorem prover.



The specification language CCSL

The **C**oalgebraic **C**lass **S**pecification **L**anguage is a formal language for writing specifications of:

The specification language CCSL

The **C**oalgebraic **C**lass **S**pecification **L**anguage is a formal language for writing specifications of:

- Object oriented classes

The specification language CCSL

The **C**oalgebraic **C**lass **S**pecification **L**anguage is a formal language for writing specifications of:

- Object oriented classes
- Abstract data types

The specification language CCSL

The **C**oalgebraic **C**lass **S**pecification **L**anguage is a formal language for writing specifications of:

	Models
• Object oriented classes	Coalgebras
• Abstract data types	Algebras

The specification language CCSL

The **C**oalgebraic **C**lass **S**pecification **L**anguage is a formal language for writing specifications of:

	Models
• Object oriented classes	Greatest fixed point
• Abstract data types	Least fixed point

The specification language CCSL

The **C**oalgebraic **C**lass **S**pecification **L**anguage is a formal language for writing specifications of:

	Reasoning
• Object oriented classes	Coinductive
• Abstract data types	Inductive

The specification language CCSL

The **C**oalgebraic **C**lass **S**pecification **L**anguage is a formal language for writing specifications of:

- Object oriented classes
- Abstract data types

In our setting, we represent:

The specification language CCSL

The **C**oalgebraic **C**lass **S**pecification **L**anguage is a formal language for writing specifications of:

- Object oriented classes
- Abstract data types

In our setting, we represent:

- static structure by an abstract data type (e.g. the set of messages);

The specification language CCSL

The **C**oalgebraic **C**lass **S**pecification **L**anguage is a formal language for writing specifications of:

- Object oriented classes
- Abstract data types

In our setting, we represent:

- static structure by an abstract data type (e.g. the set of messages);
- dynamic structure by a class (e.g. principal's current knowledge).

CCSL class specs

A class specification consists of:

CCSL class specs

A class specification consists of:

- Coalgebraic method declarations;

CCSL class specs

A class specification consists of:

- Coalgebraic method declarations;
- Assertions (axioms);

CCSL class specs

A class specification consists of:

- Coalgebraic method declarations;
- Assertions (axioms);
- Theorems (to be proved).

CCSL class specs

A class specification consists of:

- Coalgebraic method declarations;
- Assertions (axioms);
- Theorems (to be proved).

The method declarations define a coalgebraic signature.

CCSL class specs

A class specification consists of:

- Coalgebraic method declarations;
- Assertions (axioms);
- Theorems (to be proved).

The method declarations define a coalgebraic signature.

$$
\begin{array}{l}
\mathsf{MsgContext} : \mathbf{CLASSSPEC} \\
\quad \mathbf{METHOD} \\
\qquad \mathtt{next} \colon \mathsf{Self} \longrightarrow \mathsf{Self} \\
\qquad \mathtt{action} \colon \mathsf{Self} \longrightarrow \{\mathbf{idle}, \mathbf{sent}, \mathbf{received}\} \\
\qquad \mathtt{knows} \colon \mathsf{Self} \times \mathsf{Princ} \longrightarrow [\mathsf{Message} \longrightarrow \mathsf{Bool}]
\end{array}
$$

CCSL class specs

A class specification consists of:

- Coalgebraic method declarations;
- Assertions (axioms);
- Theorems (to be proved).

The assertions restrict the models of the signature.

CCSL class specs

A class specification consists of:

- Coalgebraic method declarations;
- Assertions (axioms);
- Theorems (to be proved).

The assertions restrict the models of the signature.

Assertions are axioms for the specification.

CCSL class specs

A class specification consists of:

- Coalgebraic method declarations;
- Assertions (axioms);
- Theorems (to be proved).

The assertions restrict the models of the signature.

Assertions are axioms for the specification.

Here's where the assumptions come in!

CCSL class specs

A class specification consists of:

- Coalgebraic method declarations;
- Assertions (axioms);
- Theorems (to be proved).

The theorems are claims to be proved (by the user).

CCSL class specs

A class specification consists of:

- Coalgebraic method declarations;
- Assertions (axioms);
- Theorems (to be proved).

The theorems are claims to be proved (by the user).

Correctness conditions for a specification are represented as theorems.

The CCSL compiler

Input: class and abstract data specifications.

The CCSL compiler

Input: class and abstract data specifications.

Output: PVS theories including axioms, definitions, etc.

The CCSL compiler

Input: class and abstract data specifications.

Output: PVS theories including axioms, definitions, etc.

This includes:

- definitions of invariant predicate, homomorphism, etc.,

The CCSL compiler

Input: class and abstract data specifications.

Output: PVS theories including axioms, definitions, etc.

This includes:

- definitions of invariant predicate, homomorphism, etc.,
- principles of induction, coinduction, etc.,

The CCSL compiler

Input: class and abstract data specifications.

Output: PVS theories including axioms, definitions, etc.

This includes:

- definitions of invariant predicate, homomorphism, etc.,
- principles of induction, coinduction, etc.,
- basic theory of temporal operators.

Part Four: The Application

Back to security protocols

Considering protocols like:

$$\begin{array}{lll} 1. & \mathbf{B} \rightarrow \mathbf{A}: & B, \{N_B, B\}_{\mathrm{pk}_A} \\ 2. & \mathbf{A} \rightarrow \mathbf{B}: & \{\mathrm{Sha}(N_B), N_A, A, K_{AB}\}_{\mathrm{pk}_B} \\ 3. & \mathbf{B} \rightarrow \mathbf{A}: & \{\mathrm{Sha}(N_A)\}_{K_{AB}} \end{array}$$

Back to security protocols

Considering protocols like:

$$\begin{array}{lll} 1. & \mathbf{B} \rightarrow \mathbf{A}: & B, \{N_B, B\}_{\mathrm{pk}_A} \\ 2. & \mathbf{A} \rightarrow \mathbf{B}: & \{\mathrm{Sha}(N_B), N_A, A, K_{AB}\}_{\mathrm{pk}_B} \\ 3. & \mathbf{B} \rightarrow \mathbf{A}: & \{\mathrm{Sha}(N_A)\}_{K_{AB}} \end{array}$$

We make a number of assumptions:

- Perfect cryptography assumption

Back to security protocols

Considering protocols like:

$$
\begin{array}{lll}
1. & \mathbf{B} \rightarrow \mathbf{A}: & B, \{N_B, B\}_{\mathrm{pk}_A} \\
2. & \mathbf{A} \rightarrow \mathbf{B}: & \{\mathrm{Sha}(N_B), N_A, A, K_{AB}\}_{\mathrm{pk}_B} \\
3. & \mathbf{B} \rightarrow \mathbf{A}: & \{\mathrm{Sha}(N_A)\}_{K_{AB}}
\end{array}
$$

We make a number of assumptions:

- Perfect cryptography assumption
- Dolev-Yao model : **Spy** can read (but not nec. decrypt) any message in the network

Back to security protocols

Considering protocols like:

$$\begin{array}{lll} 1. & \mathbf{B} \rightarrow \mathbf{A}: & B, \{N_B, B\}_{\mathrm{pk}_A} \\ 2. & \mathbf{A} \rightarrow \mathbf{B}: & \{\mathrm{Sha}(N_B), N_A, A, K_{AB}\}_{\mathrm{pk}_B} \\ 3. & \mathbf{B} \rightarrow \mathbf{A}: & \{\mathrm{Sha}(N_A)\}_{K_{AB}} \end{array}$$

We make a number of assumptions:

- Perfect cryptography assumption
- Dolev-Yao model : **Spy** can read (but not nec. decrypt) any message in the network
- Other assumptions: freshness, "perfect" hashes, true randomness of nonces.

Back to security protocols

Considering protocols like:

$$\begin{array}{lll} 1. & \mathbf{B} \rightarrow \mathbf{A}: & B, \{N_B, B\}_{\mathrm{pk}_A} \\ 2. & \mathbf{A} \rightarrow \mathbf{B}: & \{\mathrm{Sha}(N_B), N_A, A, K_{AB}\}_{\mathrm{pk}_B} \\ 3. & \mathbf{B} \rightarrow \mathbf{A}: & \{\mathrm{Sha}(N_A)\}_{K_{AB}} \end{array}$$

We do not assume:

- a fixed number of participants

Back to security protocols

Considering protocols like:

$$\begin{array}{lll} 1. & \mathbf{B} \rightarrow \mathbf{A}: & B, \{N_B, B\}_{\mathrm{pk}_A} \\ 2. & \mathbf{A} \rightarrow \mathbf{B}: & \{\mathrm{Sha}(N_B), N_A, A, K_{AB}\}_{\mathrm{pk}_B} \\ 3. & \mathbf{B} \rightarrow \mathbf{A}: & \{\mathrm{Sha}(N_A)\}_{K_{AB}} \end{array}$$

We do not assume:

- a fixed number of participants
- a limited number of parallel protocol runs

Back to security protocols

Considering protocols like:

$$\begin{array}{lll} 1. & \mathbf{B} \rightarrow \mathbf{A}: & B, \{N_B, B\}_{\mathrm{pk}_A} \\ 2. & \mathbf{A} \rightarrow \mathbf{B}: & \{\mathrm{Sha}(N_B), N_A, A, K_{AB}\}_{\mathrm{pk}_B} \\ 3. & \mathbf{B} \rightarrow \mathbf{A}: & \{\mathrm{Sha}(N_A)\}_{K_{AB}} \end{array}$$

We do not assume:

- a fixed number of participants
- a limited number of parallel protocol runs
- participants send only protocol messages

The Message Context class

The assumptions common to all security protocols go into the Message Context class, MsgContext. This class:

The Message Context class

The assumptions common to all security protocols go into the Message Context class, MsgContext. This class:

- Represents the state of the system at a point in time;

The Message Context class

The assumptions common to all security protocols go into the Message Context class, MsgContext. This class:

- Represents the state of the system at a point in time;
- Axiomatizes the effects of sending and receiving messages;

The Message Context class

The assumptions common to all security protocols go into the Message Context class, MsgContext. This class:

- Represents the state of the system at a point in time;
- Axiomatizes the effects of sending and receiving messages;
- Restricts the possible actions of the participants.

MsgContext: sample methods

$$
\begin{array}{l}
\mathsf{MsgContext} : \textbf{CLASSSPEC} \\
\quad \textbf{METHOD} \\
\qquad \texttt{next} \colon \mathsf{Self} \longrightarrow \mathsf{Self} \\
\qquad \texttt{action} \colon \mathsf{Self} \longrightarrow \{\textbf{idle}, \textbf{sent}, \textbf{received}\} \\
\qquad \texttt{knows} \colon \mathsf{Self} \times \mathsf{Princ} \longrightarrow [\mathsf{Message} \longrightarrow \mathsf{Bool}]
\end{array}
$$

The basic methods represent

- the flow of time (`next`),

MsgContext: sample methods

$\mathsf{MsgContext} : \textbf{CLASSSPEC}$
$\quad \textbf{METHOD}$
$\qquad \texttt{next} \colon \mathsf{Self} \longrightarrow \mathsf{Self}$
$\qquad \texttt{action} \colon \mathsf{Self} \longrightarrow \{\textbf{idle}, \textbf{sent}, \textbf{received}\}$
$\qquad \texttt{knows} \colon \mathsf{Self} \times \mathsf{Princ} \longrightarrow [\mathsf{Message} \longrightarrow \mathsf{Bool}]$

The basic methods represent

- the flow of time (`next`),
- the action occurring (`action`),

MsgContext: sample methods

$$\begin{array}{l}
\mathsf{MsgContext} : \textbf{CLASSSPEC} \\
\quad \textbf{METHOD} \\
\qquad \mathtt{next} : \mathsf{Self} \longrightarrow \mathsf{Self} \\
\qquad \mathtt{action} : \mathsf{Self} \longrightarrow \{\textbf{idle}, \textbf{sent}, \textbf{received}\} \\
\qquad \mathtt{knows} : \mathsf{Self} \times \mathsf{Princ} \longrightarrow [\mathsf{Message} \longrightarrow \mathsf{Bool}]
\end{array}$$

The basic methods represent

- the flow of time (`next`),
- the action occurring (`action`),
- the state of the principals' knowledge (`knows`).

MsgContext: sample assertion

$$
\begin{array}{l}
\mathsf{MsgContext} : \textbf{CLASSSPEC} \\
\quad \textbf{METHOD} \\
\qquad \mathtt{next} \colon \mathsf{Self} \longrightarrow \mathsf{Self} \\
\qquad \mathtt{action} \colon \mathsf{Self} \longrightarrow \{\textbf{idle}, \textbf{sent}, \textbf{received}\} \\
\qquad \mathtt{knows} \colon \mathsf{Self} \times \mathsf{Princ} \longrightarrow [\mathsf{Message} \longrightarrow \mathsf{Bool}] \\
\quad \textbf{ASSERTION} \\
\qquad \mathtt{action}(x) = \textbf{idle} \Rightarrow \\
\qquad\quad \forall (P : \mathsf{Princ}) : \\
\qquad\qquad x.\mathtt{knows}(P) = x.\mathtt{next}.\mathtt{knows}(P)
\end{array}
$$

Knowledge does not change if idle.

MsgContext: sample theorem

$\mathsf{MsgContext} : \textbf{CLASSSPEC}$

METHOD

$\texttt{next} \colon \mathsf{Self} \longrightarrow \mathsf{Self}$

$\texttt{action} \colon \mathsf{Self} \longrightarrow \{\textbf{idle}, \textbf{sent}, \textbf{received}\}$

$\texttt{knows} \colon \mathsf{Self} \times \mathsf{Princ} \longrightarrow [\mathsf{Message} \longrightarrow \mathsf{Bool}]$

ASSERTION

$$\begin{aligned} & \texttt{action}(x) = \textbf{idle} \Rightarrow \\ & \quad \forall (P : \mathsf{Princ}) : \\ & \qquad x.\texttt{knows}(P) = x.\texttt{next}.\texttt{knows}(P) \end{aligned}$$

THEOREM

$$\begin{aligned} & \forall (P : \mathsf{Princ}, m : \mathsf{Message}) : \\ & \qquad x.\texttt{knows}(P)(m) \Rightarrow \\ & \qquad\quad x.\texttt{next}.\texttt{knows}(P)(m). \end{aligned}$$

Inheritance

The CCSL language supports class inheritance.

Inheritance

The CCSL language supports class inheritance.
We use:

- A generic MsgContext class

Inheritance

The CCSL language supports class inheritance.
We use:

- A generic MsgContext class
  - general model for learning, message passing, etc.

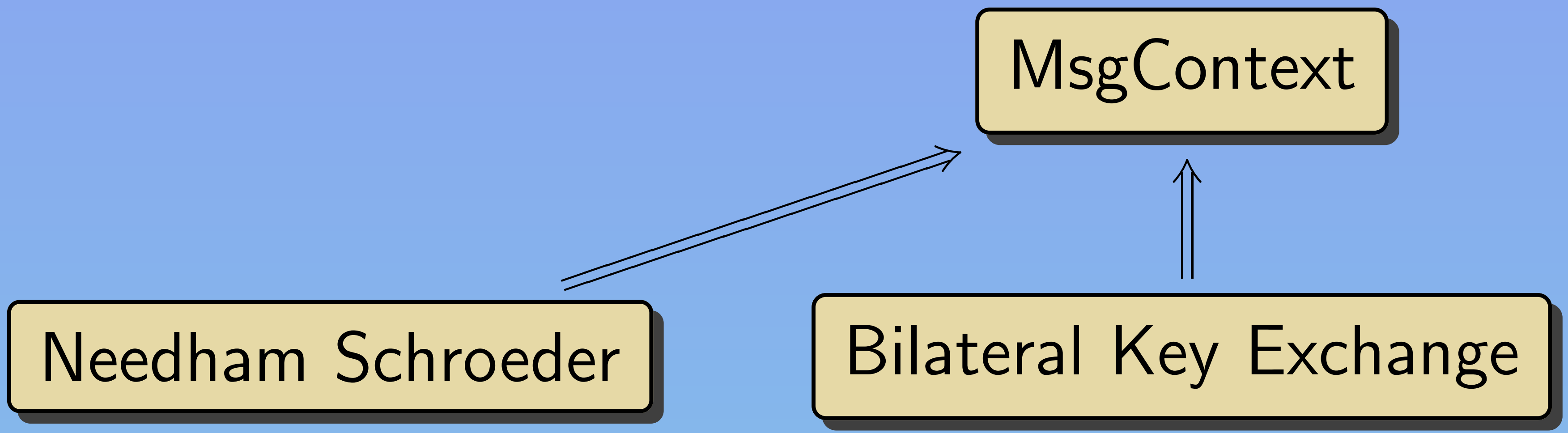
Inheritance

The CCSL language supports class inheritance.
We use:

- A generic MsgContext class
  - general model for learning, message passing, etc.
  - our security model assumptions.

Inheritance

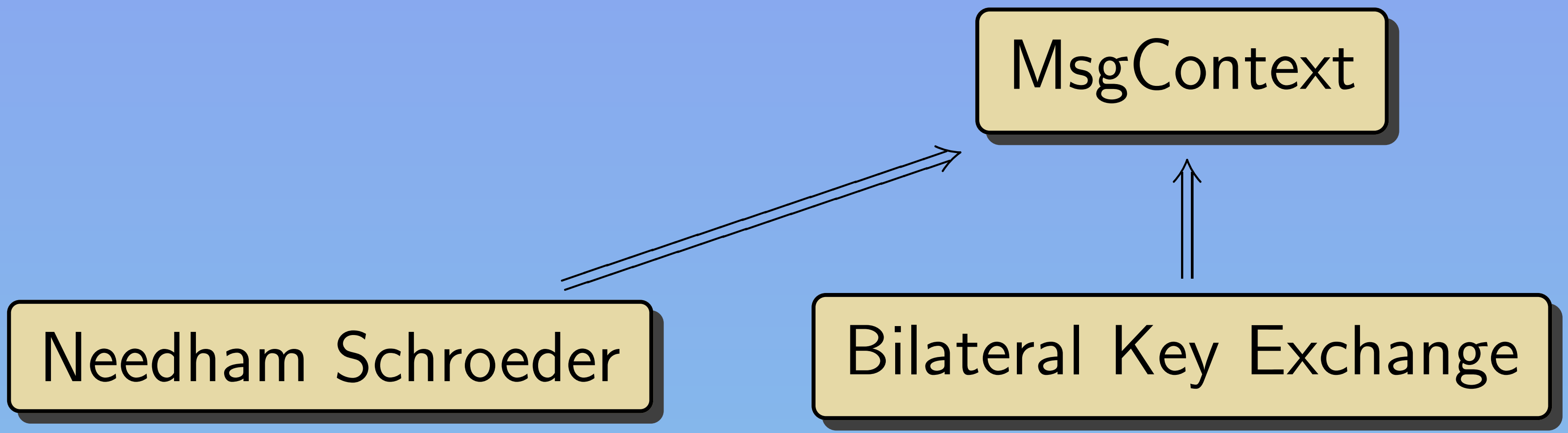
The CCSL language supports class inheritance.
We use:



- A generic MsgContext class
- Specific protocol classes containing:

Inheritance

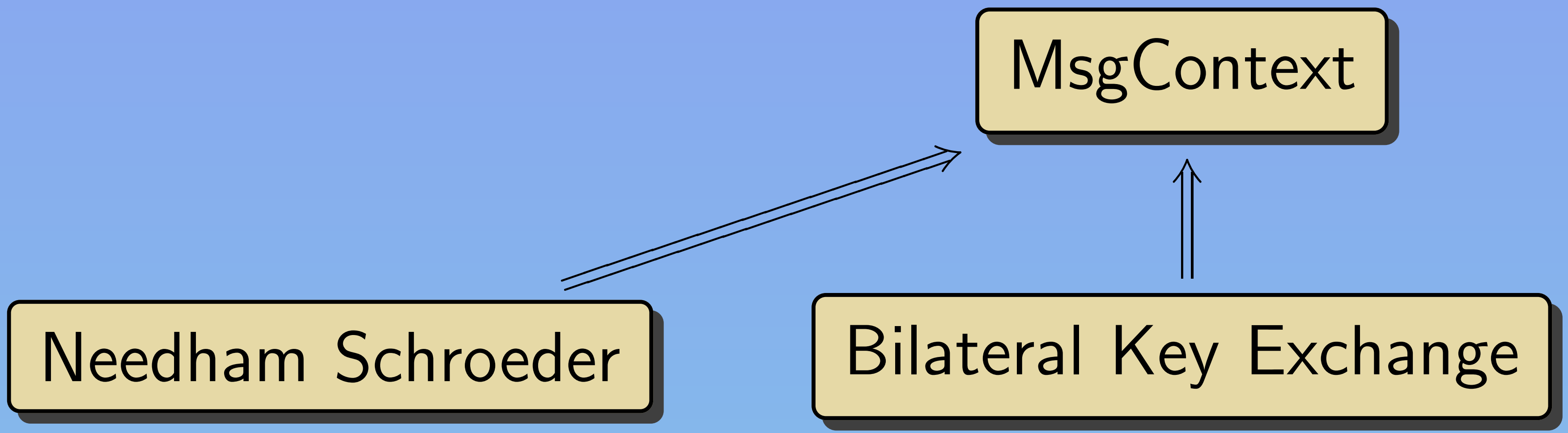
The CCSL language supports class inheritance.
We use:



- A generic MsgContext class
- Specific protocol classes containing:
  - Axioms describing the protocol,

Inheritance

The CCSL language supports class inheritance.
We use:



- A generic MsgContext class
- Specific protocol classes containing:
  - Axioms describing the protocol,
  - Correctness theorems.

Correctness

So, what do we want to prove about a protocol (say, Bilateral Key Exchange)?

Correctness

So, what do we want to prove about a protocol (say, Bilateral Key Exchange)?

We want to prove:

If ***A*** *invites* ***B*** *to start the protocol …*

Correctness

So, what do we want to prove about a protocol (say, Bilateral Key Exchange)?

We want to prove:

> *If **A** invites **B** to start the protocol and **A** and **B** respond as the protocol dictates …*

Correctness

So, what do we want to prove about a protocol (say, Bilateral Key Exchange)?

We want to prove:

> *If **A** invites **B** to start the protocol and **A** and **B** respond as the protocol dictates then there is a key **K** such that*
>
> - *eventually **A** and **B** know **K**;*

Correctness

So, what do we want to prove about a protocol (say, Bilateral Key Exchange)?

We want to prove:

> *If **A** invites **B** to start the protocol and **A** and **B** respond as the protocol dictates then there is a key **K** such that*
>
> - *eventually **A** and **B** know **K**;*
> - *eventually **A** and **B** believe they each know **K**;*

Correctness

So, what do we want to prove about a protocol (say, Bilateral Key Exchange)?

We want to prove:

> *If **A** invites **B** to start the protocol and **A** and **B** respond as the protocol dictates then there is a key **K** such that*
>
> - *eventually **A** and **B** know **K**;*
> - *eventually **A** and **B** believe they each know **K**;*
> - *no one else knows K.*

Correctness

So, what do we want to prove about a protocol (say, Bilateral Key Exchange)?

We want to prove:

> *If **A** invites **B** to start the protocol and **A** and **B** respond as the protocol dictates then there is a key **K** such that*
>
> - *eventually **A** and **B** know **K**;*
> - *eventually **A** and **B** believe they each know **K**;*
> - *no one else knows K.*

All of this is easily expressible in CCSL, using our MsgContext protocol.

Correctness

So, what do we want to prove about a protocol (say, Bilateral Key Exchange)?

We want to prove:

> *If **A** invites **B** to start the protocol and **A** and **B** respond as the protocol dictates then there is a key **K** such that*
>
> - *eventually **A** and **B** know **K**;*
> - *eventually **A** and **B** believe they each know **K**;*
> - *no one else knows K.*

Admittedly, proving it is not so easy.

Paulson's Inductive Approach

Lawrence Paulson uses a similar approach to analyzing security protocols.

Paulson's Inductive Approach

Lawrence Paulson uses a similar approach to analyzing security protocols.

However, his models are inherently algebraic, rather than coalgebraic.

Paulson's Inductive Approach

Lawrence Paulson uses a similar approach to analyzing security protocols.

However, his models are inherently algebraic, rather than coalgebraic.

He considers the set of finite traces for a protocol. This set can be given by a least fixed point construction, i.e., by an initial algebra.

Paulson's Inductive Approach

His basic proof principle is induction. To prove P always holds, he shows

- $P[]$ holds and ...

Paulson's Inductive Approach

His basic proof principle is induction. To prove P always holds, he shows

- $P[]$ holds and
- if $P(evs)$, then $P(ev \# evs)$.

Paulson's Inductive Approach

His basic proof principle is induction. To prove P always holds, he shows

- $P[]$ holds and
- if $P(evs)$, then $P(ev \# evs)$.

This is analogous to showing that P is an invariant, in the coalgebraic sense.

Paulson's Inductive Approach

His basic proof principle is induction. To prove P always holds, he shows

- $P[]$ holds and
- if $P(evs)$, then $P(ev \# evs)$.

This is analogous to showing that P is an invariant, in the coalgebraic sense.

The main theoretical difference is that we consider infinite traces as models, while Paulson considers finite traces.

Comparison

There are a number of practical differences in Paulson's work and our own.

Our approach	Paulson's

Comparison

There are a number of practical differences in Paulson's work and our own.

Our approach	Paulson's
Separate specification language (CCSL)	Specified directly in Isabelle

Comparison

There are a number of practical differences in Paulson's work and our own.

Our approach	Paulson's
Separate specification language (CCSL)	Specified directly in Isabelle
Temporal reasoning	Inductive reasoning

Comparison

There are a number of practical differences in Paulson's work and our own.

Our approach	Paulson's
Separate specification language (CCSL)	Specified directly in Isabelle
Temporal reasoning	Inductive reasoning
Inheritance	No inheritance

Comparison

There are a number of practical differences in Paulson's work and our own.

Our approach	Paulson's
Separate specification language (CCSL)	Specified directly in Isabelle
Temporal reasoning	Inductive reasoning
Inheritance	No inheritance

As well, our specification places fewer restrictions on the behavior of the participants ...

Comparison

There are a number of practical differences in Paulson's work and our own.

Our approach	Paulson's
Separate specification language (CCSL)	Specified directly in Isabelle
Temporal reasoning	Inductive reasoning
Inheritance	No inheritance

As well, our specification places fewer restrictions on the behavior of the participants but we pay for this generality!

Summary

Summarizing:

Summary

Summarizing:

Specify a protocol in CCSL, using temporal operators.

Summary

Summarizing:

Specify a protocol in CCSL, using temporal operators.

The protocol inherits from a general MsgContext class.

Summary

Summarizing:

Specify a protocol in CCSL, using temporal operators.

The protocol inherits from a general MsgContext class.

Compile the CCSL specification into a PVS theory.

Summary

Summarizing:

Specify a protocol in CCSL, using temporal operators.

The protocol inherits from a general MsgContext class.

Compile the CCSL specification into a PVS theory.

Prove the correctness conditions.

Summary

Summarizing:

Specify a protocol in CCSL, using temporal operators.

The protocol inherits from a general MsgContext class.

Compile the CCSL specification into a PVS theory.

Prove the correctness conditions.

More on CCSL can be found here:
`http://wwwtcs.inf.tu-dresden.de/~tews/ccsl/`