Tamarin Workshop
RI.SE – 2019-10-29

Simon BOUGET
(w/ slides from David Basin, Cas Cremers, Jannik Dreier, Ralf Sasse)

Overview

- 2 hours
  - Half lecture, half hands-on
- Adapted from a one-day tutorial
  - 6 hours cut down to 2
  - Removed most of the theory
- Interrupt if you don’t follow

Introduction

- How to know if a protocol is secure?
- With a structured, systemic approach?
- Notion of trace properties
  - When a protocol is run, it generates a “trace”
    - e.g. messages sent, data logged, etc.
- Encode (and prove) properties
  - **Authentication:**
    In all traces, if an initiator completes,
    there exists a responder with…
  - **Secrecy:**
    There is no trace in which Adversary learns k

Tamarin: high-level

- **Modeling** protocol & adversary done using multiset rewriting
  - Specifies transition system; induces set of traces
- **Property** specification using fragment of first-order logic
  - Specifies “good” traces
- Tamarin tries to
  - provide proof that all system traces are good, or
  - construct a counterexample trace of the system (attack)

What can Tamarin do for you?

- Rapid prototyping
- Finding attacks before you start a proof effort
- Provide a symbolic proof
- Explore alternative designs/threat models quickly

Resources & documentation

- Sources on github
- 100+ page manual
- Plenty of examples/case studies
- Algorithm details in theses, papers

Selected case studies

- AKE
  - Naxos
  - Signed DH
  - KEA+
  - UM
  - Tsx
- Group protocols
  - GDH
  - TAK
  - (Sig)Joux
  - STR
- ID-based AKE
  - RYY
  - Scott
  - Chen-Kudla
- Loops
  - TESLA1 & 2
- Non-monotonic global state
  - Keyserver
  - Envelope
  - Exclusive secrets
  - Contract signing
  - Security device
  - YubiKey
  - YubiHSM
- PKI with strong guarantees
  - ARPKI (also global state)
- Transparency
  - DECIM (also global state)
- TLS 1.3
  - Rev 10, 10+, and current

Modelization in Tamarin: Overview

- Modeling protocols and threats:
  1) What is the current state of the system?
     What’s the situation, the state of the world?
  2) How can the system evolve?
     What are the possibilities? How can the system progress?

Modelization in Tamarin: State

- State of the system → multiset / bag
  - **Facts** (think "general statement")
    - parametrized with **Terms**
      (think "specific instance")
  - E.g. **"Agent A owns the secret/public key pair (sk_A, pk_A)"**
    KeyPair(A, sk_A, pk_A)
- Initial state is empty
- **You** can define any arbitrary fact you want
  - Foo(x,y), Toto(a,b,c), KeyPair(A, sk, pk), …
  - Purely symbolic, no meaning attached
    ==> meaning is in the rules,
    in the *relationships between facts*)

Modelization in Tamarin: Change

- Changes → transition rules
  - [*conditions*] --[*actions*]-> [*conclusions*]
  - E.g. **“If A has a secret key sk_A**
    and knows the public key pk_B of B,
    then A can send a message m to B, signed and encrypted.”

```
  [ !SecKey(A, sk_A), !PubKey(B, pk_B) ]
--[ Send(A, B, m) ]->
  [ Out(<aenc(m, pk_B), sign(aenc(m, pk_B), sk_A)>) ]
```

- For a rule to be applied:
  - it must match the conditions (or premises)
  - and it generates the conclusions

‘c’ constant

Semantics: example 1

- **Rules**
  - rule 1: `[      ]--[ Init()  ]-->[ A('5') ]`
  - rule 2: `[A(x)]--[ Step(x) ]-->[  B(x)  ]`
- **ONE possible run**

Rule applied	Current state	Trace
Initial state	[ ]	()
Rule 1	[A(‘5’)]	(Init())
Rule 1	[A(‘5’), A(‘5’)]	(Init(), Init())
Rule 2	[A(‘5’), B(‘5’)]	(Init(), Init(), Step(‘5’))

- **An infinity of other possibilities** ‘c’ constant

Example 2: persistent facts

- **“!Persistent” vs “Linear” facts**
  - Linear facts are consumed by rules that match them,
    i.e. removed from the state of the system when the rule is applied
  - Persistent facts can be matched any number of times
- **Rules**
  - rule1: `[                ]--[ Init()  ]-->[!C('ok'), D('1')]`
  - rule2: `[!C(x), D(y)]--[Step(x,y)]-->[     D(h(y))      ]`
- **Execution example**

Rule applied	Current state	Trace
Initial state	[ ]	()
Rule 1	[!C(‘ok’), D(‘1’)]	(Init())
Rule 2	[!C(‘ok’), D(h(‘1’))]	(Init(), Step(‘ok’, ‘1’))
Rule 2	[!C(‘ok’), D(h(h(‘1’)))]	(Init(), Step(‘ok’, ‘1’), Step(‘ok’, h(‘1’)))

- Now we have the basics…
- … let’s move onto real protocols!
- We still need a few missing pieces
  - How to model the network and the adversary?
  - Randomness?
  - Cryptography?
- Tamarin has built-ins for that.
  Don’t need to start from scratch!

Modelization in Tamarin: Built-In

- 1 special fact: Fr(~x) → get a fresh value x
  - Always matches
  - All instances are distinct,
    i.e. Fr(~a) & Fr(~b) => ~a != ~b
- 2 pre-defined rules controlling network messages

```
1.rule irecv: [ Out( x ) ] --> [ !KD( x ) ]
2.rule isend: [ !KU( x ) ] --[ K( x ) ]-> [ In( x ) ]
```

  - Note the use of **In()**, **Out()**,
    and **K() = the adversary Knows**
    (you can ignore !KU() and !KD())
- You can add more capabilities to the adversary if you want
  - E.g. reveal long term keys

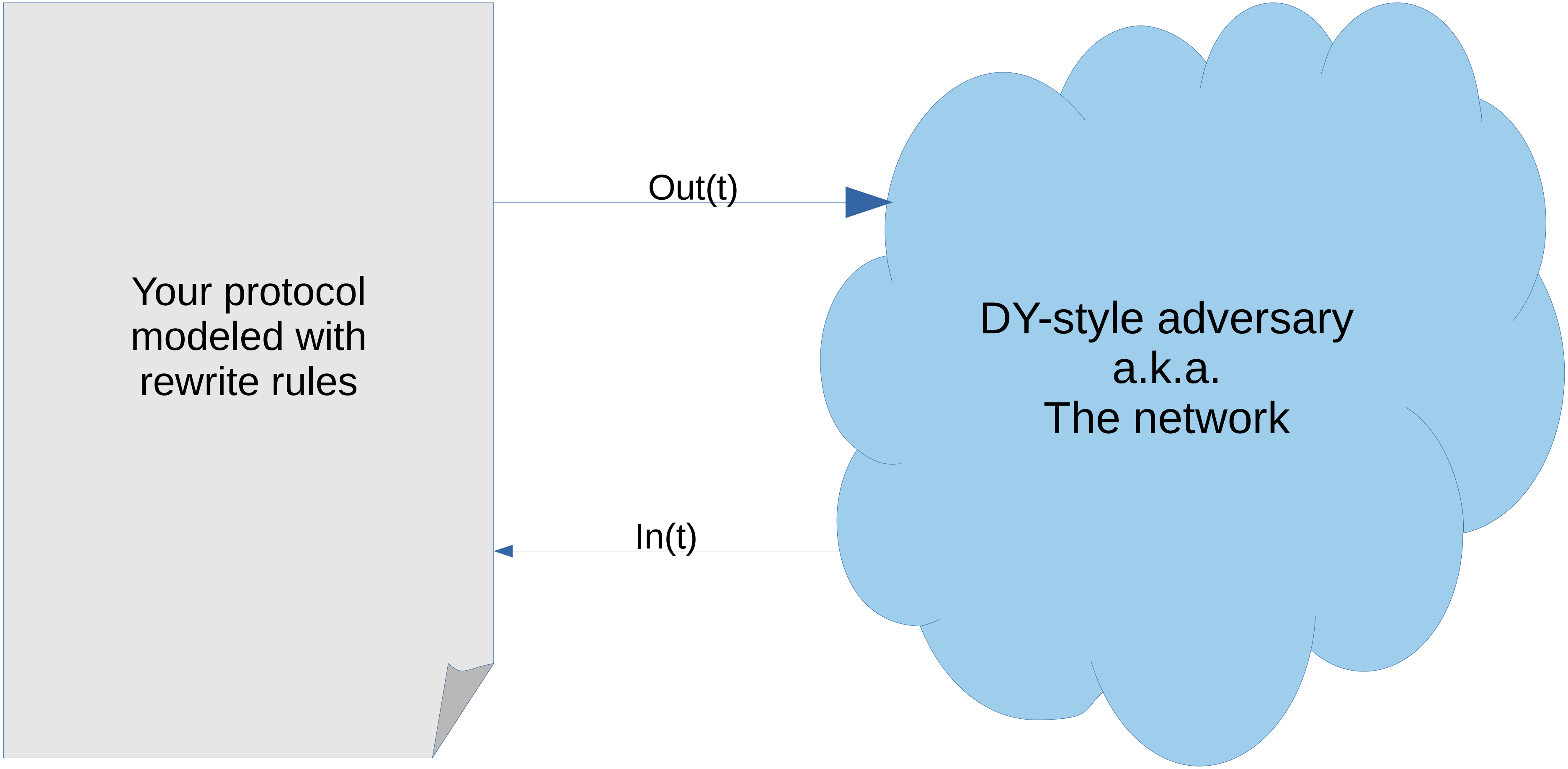
Equational theories

- **Equational theories** are used in symbolic protocol verification to model the **algebraic properties** of the **cryptographic primitives**.
- Example (asymmetric encryption):

```
adec( aenc(m,pk(k), k ) = m
```

- **Built-in:** encryption (sym and asym), (blind) signing, Diffie-Hellman, bilinear pairing, multiset, xor
- You can add your owns (with limitations)
  - Subterm convergent
    - Right-hand side is subterm of left hand side (or constant)
  - Active development

Tamarin explore **ALL** possible interactions with the adversary



Property specification

- Timed first order logic interpreted over a trace
  - `Ex x y i. Fact(x,y) @ #i`
  - `All a b j. Fact(a,b) @ #j`
  - Timepoint ordering: `#i = #j  /  #i < #j`

  - IMPLY, NOT, AND, and OR logic operators:

```
All Client Server k #i #j.
   Complete(Client, Server, k)@i & Secret(k)@j
   ==>
   not K(k) | (Ex l. Revealed(k)@l & l<i)
```

- Demo time
  - Sources
  - Using the tool
    - UI
    - Read graphs

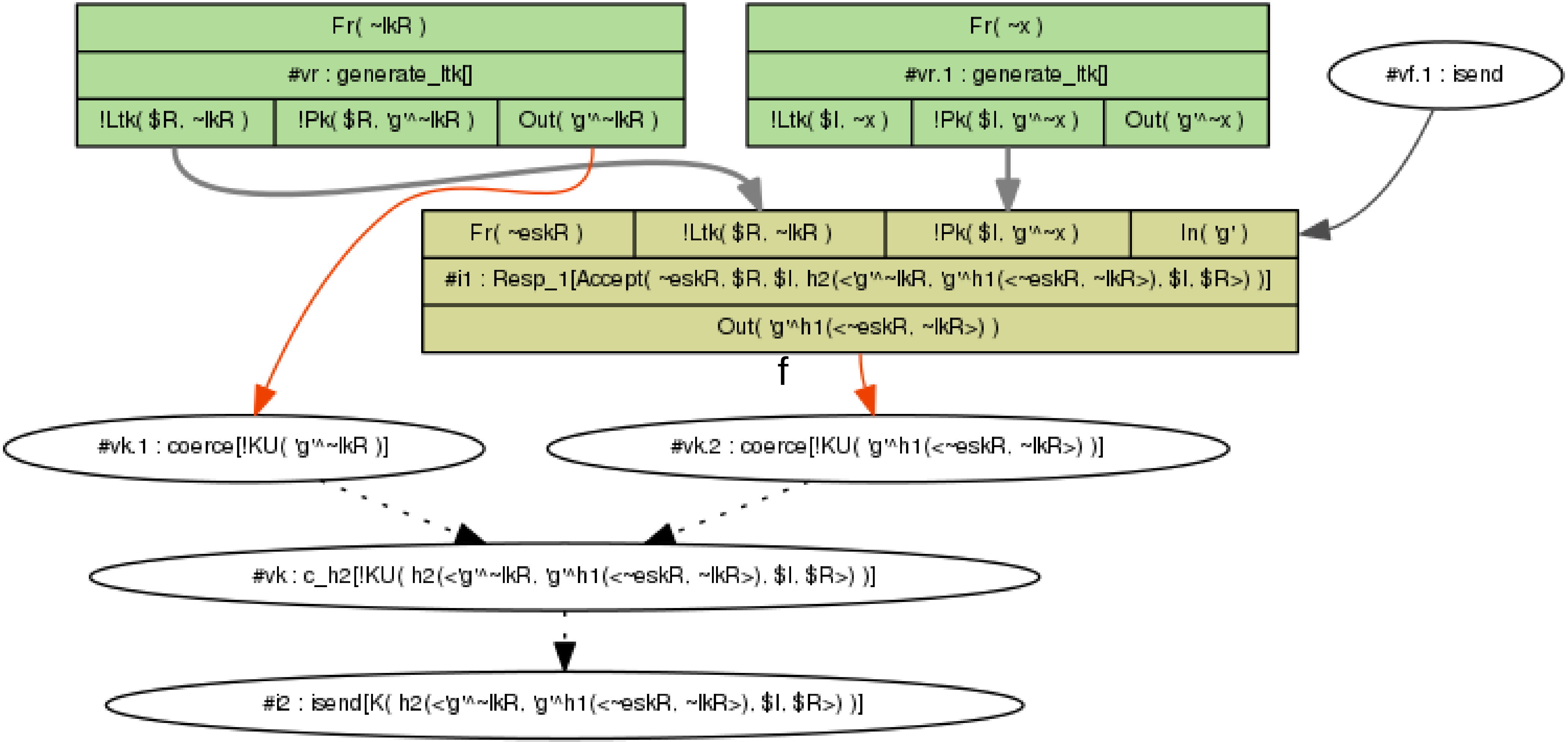
Syntax Issues: Type Annotations

- Mark timepoint (index) variables with a hashmark (#) in quantification.
- Mark fresh values with ~
- Mark public values with $
- Be consistent! If a rule contains ~x, $x, and x that is interpreted as three different variables!
- You do get a warning about it, and should fix it.

Warnings on Loading a theory

- Warnings give good information what is wrong:
  - Mismatch of type: use of $x and x in same rule
  - Using a fact name with different arities
  - Guardedness problems in formula
- Tamarin strict mode stops you from working with warnings, but is optional:
  - Add command-line parameter: --quit-on-warning

Reading Tamarin’s graphs



Basic principles

- Backwards search using **constraint reduction rules** (27!)
- Turn negation of formula into set of constraints
- Case distinctions
  - E.g.: Possible sources of a message or fact
- Try to establish:
  - no solutions exist for constraint system, or
  - there exists a „realizable“ execution (trace)
- If multiple rules can be applied: use heuristics

Heuristics?

- If Tamarin terminates, one of two options:
  - **Proof**, or
  - **counterexample** (in this context: attack)
- At each stage in proof, multiple constraint solving rules might be applicable
  - Similar to “how shall I try to prove this?”
  - Choice influences speed & termination, but not the outcome after termination
- Complex **heuristics choose rule**
  - user can give hints or override

How do I know my model is correct?

- **Lots of ways to cause errors**
- Look at the chains...
  - (requires an understanding of the algorithm)
- Executability
- Break the protocol on purpose
- Much easier to check these things than in manual proofs!

Cocnlusion

Tamarin: Conclusions

- **Tamarin** offers **many unique features**
  - Unbounded analysis, flexible properties, equational theories, global state, …
  - Enables automated analysis in areas previously unexplored
- It has many **other features** I don’t have time to go in details into
  - Many new features planned! Still active dev.
- Tool and sources are **free**;
  development on Github
  mailing list on Google Groups

Bonus: Advanced features

- More accurate models
  - Executability lemmas => sanity checks
  - Restrictions => limit allowed runs
  - Channel models => restrict adversary control of network
  - Custom equational theories
  - Observational Equivalence => stronger notion of secrecy
- Easier proofs
  - Source lemmas => reduce search space
  - Induction => forward instead of backward search
  - Custom Heuristics / Guided proofs

Executability Lemmas

Executability Lemmas

- Executability lemmas are existential properties
- These show the existence of some protocol trace satisfying the formula…
- … instead of the usual case where all traces must satisfy the formula.
- Heuristics tuned for verification
  - Manual intervention needed more often for executability

- lemma exec: exists-trace “...(formula)...”

Channel models

Channel models

- By default, the adversary controls **EVERYTHING** on the network
- You can write custom rules governing the network with alternatives to In() and Out()
- A lot of variants in the manual already
  => don’t try making your owns
- Make sure you REALLY need a secure channel

Channel models: Example

Default rules

```
rule irecv:
  [ Out( x ) ]
-->
  [ !KD( x ) ]

rule isend:
  [ !KU( x ) ]
--[ K( x ) ]->
  [ In( x ) ]
```

Confidential Channel rules

```
rule ChanOut_C:
  [ Out_C($A,$B,x) ]
--[ ChanOut_C($A,$B,x) ]->
  [ !Conf($B,x) ]

rule ChanIn_C:
  [ !Conf($B,x), In($A) ]
--[ ChanIn_C($A,$B,x) ]->
  [ In_C($A,$B,x) ]

rule ChanIn_Cadv:
  [ In(<$A,$B,x>) ]
-->
  [ In_C($A,$B,x) ]
```

Restrictions

Restrictions

- Restrictions exclude undesired traces
  - Take care not to exclude attacks!
- Safe to use for certain checks:
  - Equality
  - Inequality
  - LessThan
  - GreaterThan
  - OnlyOnce
- Use same format as lemmas
- Essentially: Conditional Rewriting

Restriction Example

- **`restriction once:`**

  `“All #i #j. OnlyOnce()@#i & OnlyOnce()@#j ==> #i=#j”`

- **Rules**
  - rule 1: [ ]   –[OnlyOnce()]→ [A('5')]
  - rule 2: [A(x)] –[Step(x)]→ [B(x)]

- **Execution removed by restriction**
  - [ ]
  - –[OnlyOnce()]→ [A('5')]
  - –[OnlyOnce()   ]→ [A('5'), A('5')]
  - –[Step('5')   ]→ [A('5'), B('5')]

- **Execution still allowed**
  - [ ]
  - –[Init()   ]→ [A('5'), A('5')]
  - –[Step('5')   ]→ [A('5'), B('5')]

Restriction Example 2

- `restriction InEq:`

  `“All x #i. Neq(x,x)@#i ==> F”`
- **Rules**
  - rule 1: [ ] –[A1()]→ [A('1')]
  - rule 2: [ ] –[A2()]→ [A('2')]
  - rule 3: [A(x), A(y)] –[Neq(x,y)]→ [B(x,y)]
- **Execution removed by restriction – valid without restriction**
  - [ ]
  - –[A1()]→ [A('1')]
  - –[A1()]→ [A('1'), A('1')]
  - –[Neq('1','1')]→ [B('1','1')]
- **Execution allowed**
  - [ ]
  - –[A1()]→ [A('1')]
  - –[A2()]→ [A('1'), A('2')]
  - –[Neq('1','2')]→ [B('1','1')]

Source lemmas

State space reduction

Pre-computation

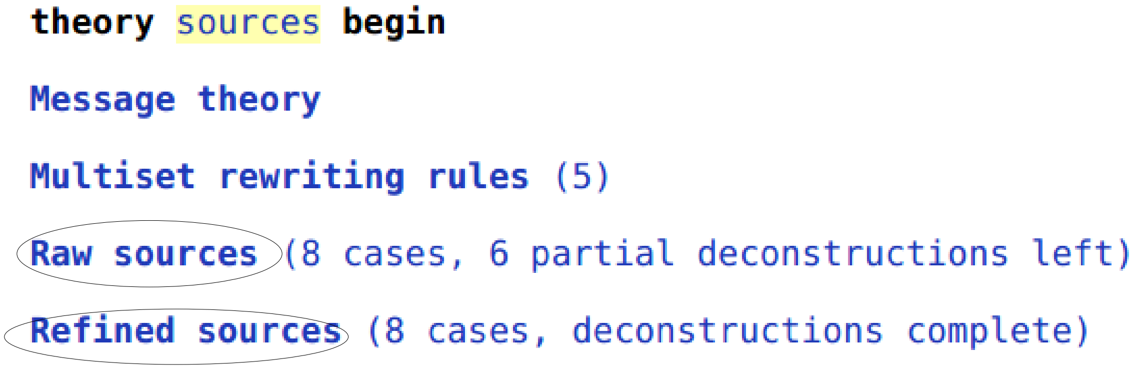
Partial deconstructions

Sources lemmas

Precomputation

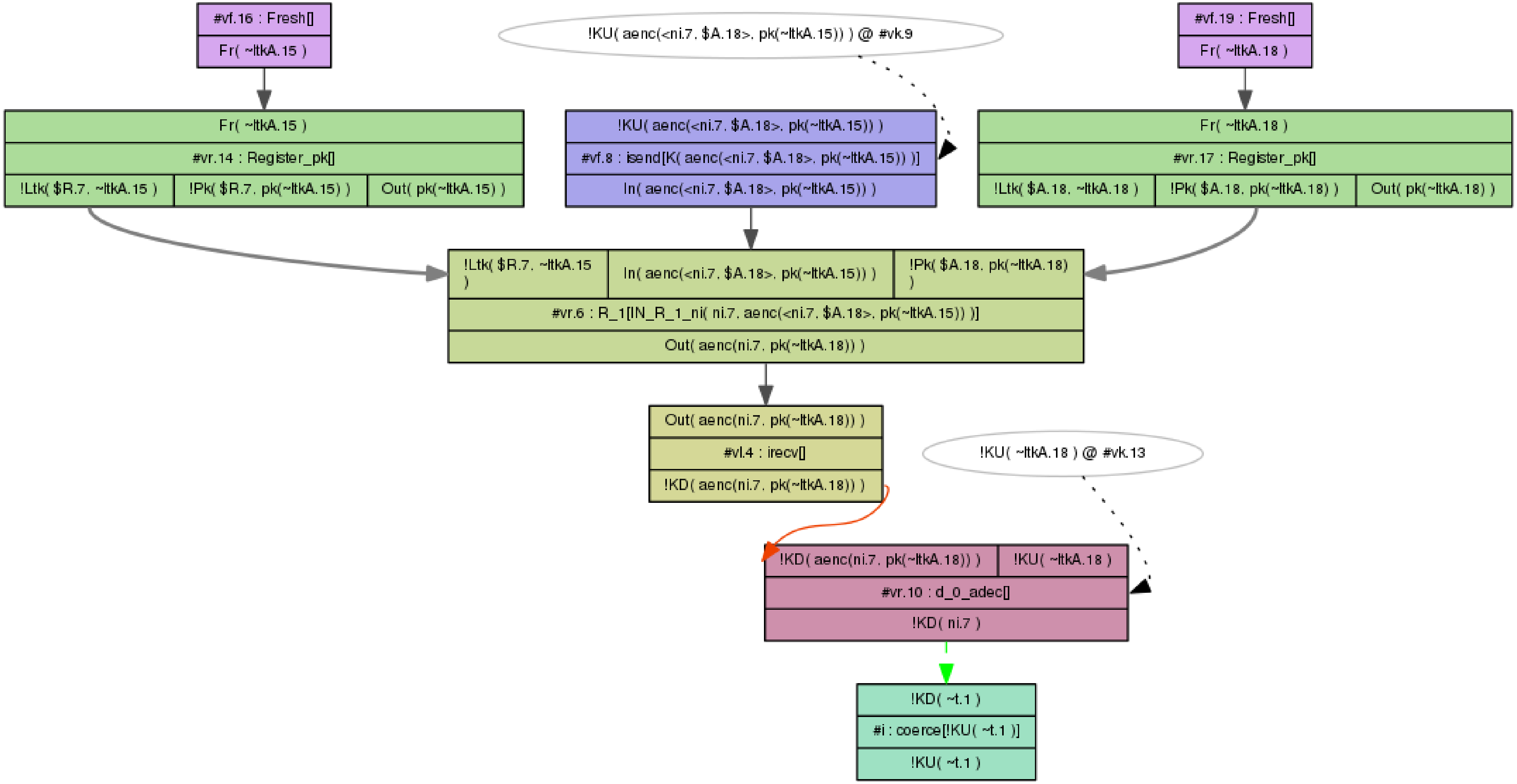
- Idea: for all facts in rule premises compute their possible sources
- sources are (combinations of) rules yielding such a fact as (part of the) result
- Initial precomputations are called raw sources
- Sometimes these precomputations are incomplete, and give partial deconstructions
- GUI shows both raw and refined sources

Demo

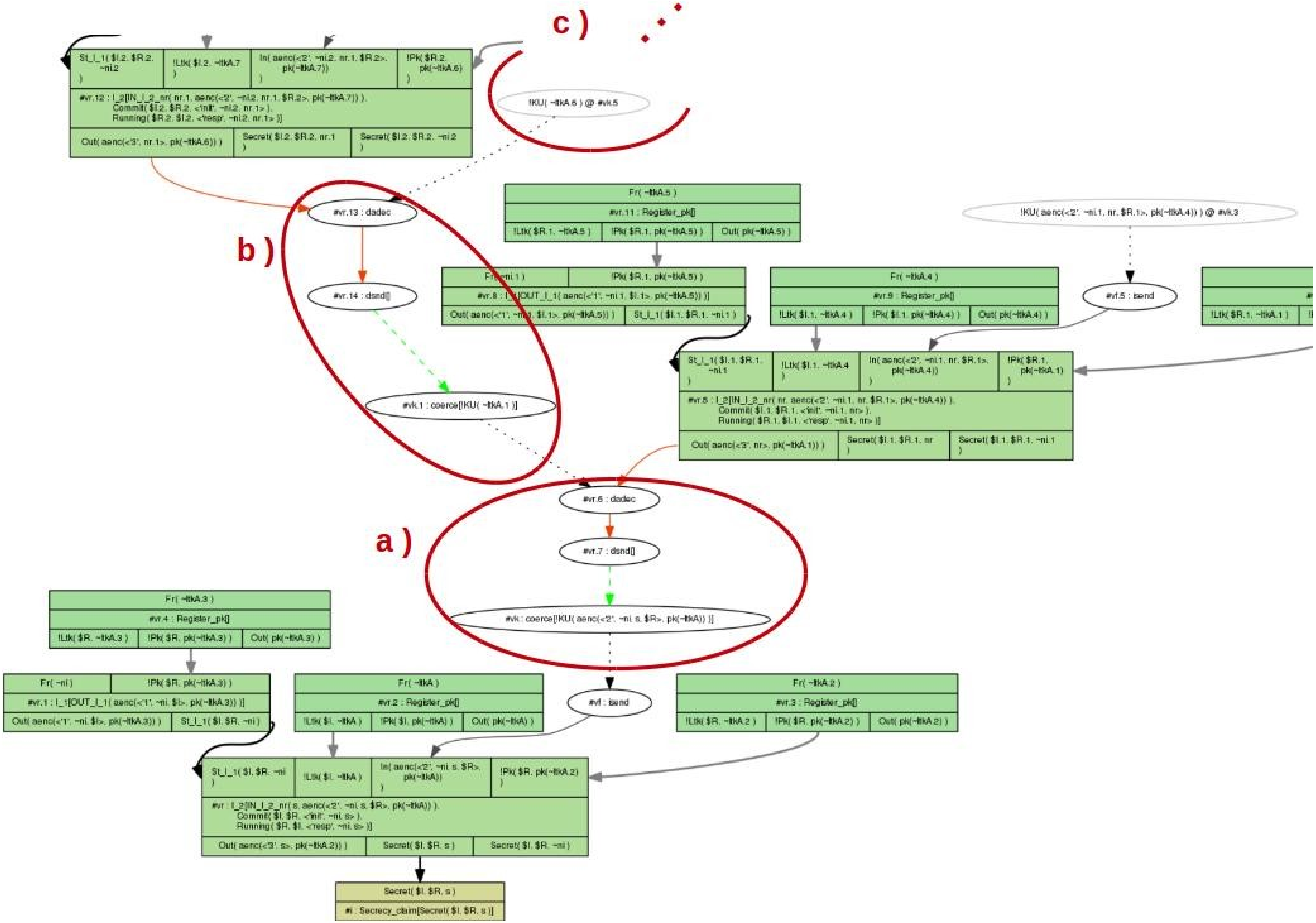


```
theory sources begin

Message theory

Multiset rewriting rules (5)

Raw sources (8 cases, 6 partial deconstructions left)

Refined sources (8 cases, deconstructions complete)
```

Partial deconstruction – derive any value



See demo for detail



Partial deconstructions – issues

- Proofs much more complicated
    - Possibly non-termination due to partial deconstructions
- Need to resolve such partial deconstructions
- Claim (and then prove) such deconstructions are not possible, by sources lemma

Example protocol

```
1. I -> R: {ni,I}pk(R)
2. I <- R: {ni}pk(I)
```

```
rule I_1:
  let m1 = aenc{~ni, $I}pkR  in
    [ Fr(~ni) , !Pk($R, pkR) ]
  --[ OUT_I_1(m1) ]->
    [ Out( m1 ) ]

rule R_1:
  let m1 = aenc{ni, I}pk(ltkR)
      m2 = aenc{ni}pkI    in
    [ !Ltk($R, ltkR) , In( m1 ), !Pk(I, pkI) ]
  --[ IN_R_1_ni( ni, m1 ) ]->
    [ Out( m2 ) ]
```

This looks like a decryption oracle for values ni

Really? Extract everything?

- Realization: only values actually sent by legitimate party (whose private key must be compromised) or adversary-generated terms
  - which are known to the adversary previously

```
lemma types [sources]:
  " (All ni m1 #i.
       IN_R_1_ni( ni, m1) @ i
       ==>
       ( (Ex #j. K(ni) @ j & j < i)
       | (Ex #j. OUT_I_1( m1 ) @ j) ) ) "
```

Demo

Problems with partial deconstructions
Sources lemma removes partial deconstructions for refined sources
Automatic proof of sources lemma

Sources lemmas

- Explain where terms can come from or what their form must be
- Tamarin actions in order:

1) Determine possible sources (raw)
2) Apply sources lemma to raw sources to get refined sources
3) Prove sources lemma WRT raw sources
4) Prove other lemmas WRT refined sources

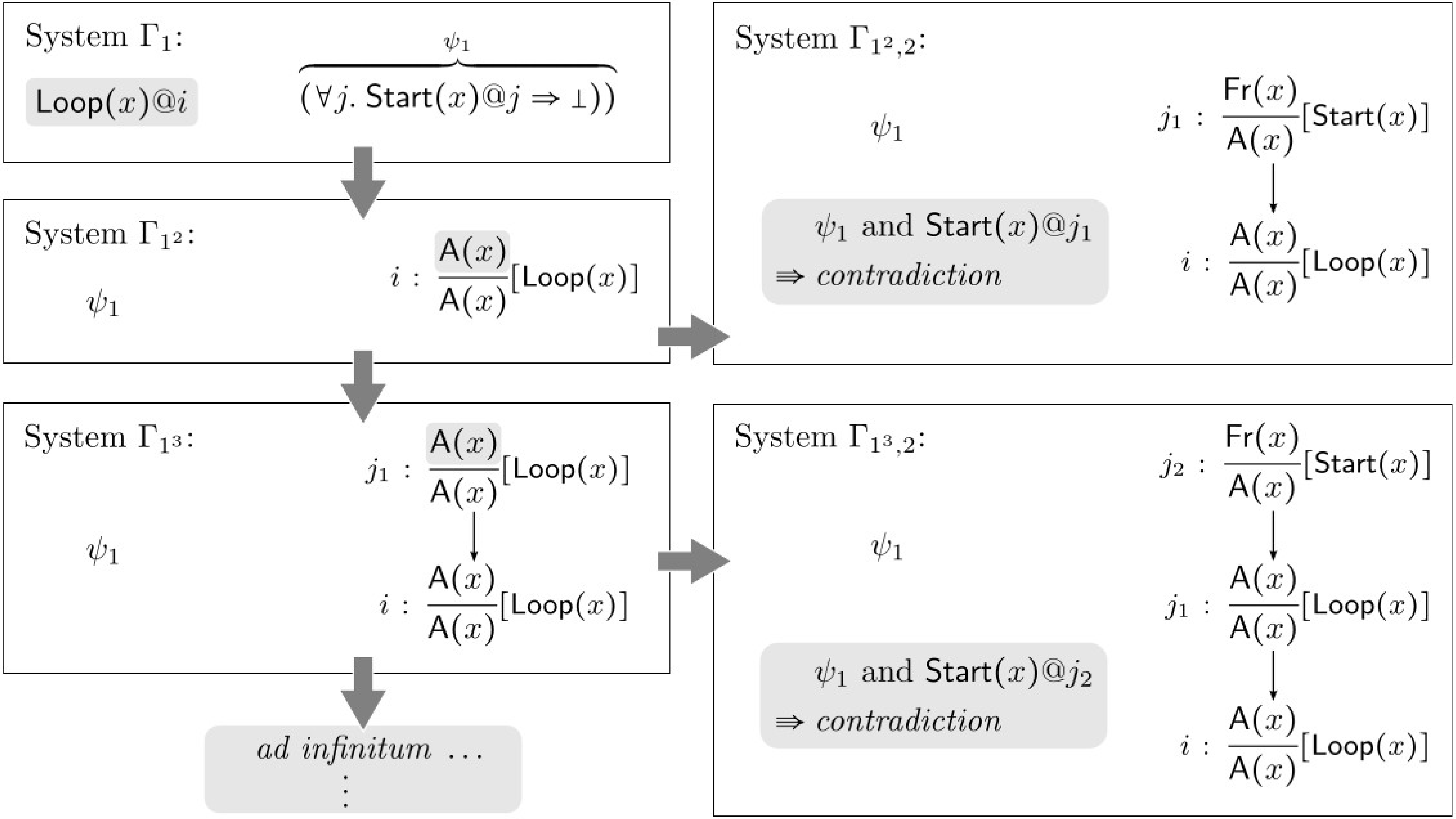
Induction

Induction

$$R_{loop} := \left\{ \frac{\mathsf{Fr}(x)}{\mathsf{A}(x)}[\mathsf{Start}(x)],\ \frac{\mathsf{A}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x)] \right\}$$

- Proof goal: $\forall x\ i.\mathsf{Loop}(x)@i \Rightarrow \exists j.\mathsf{Start}(x)@j$
  - $j < i$? Not needed in formula, but will hold
- Naive constraint solving does not work
- Such properties are needed:
  - “Reuse” lemmas
  - “Sources” lemmas

Constraint solving failure



System Γ_1:

$\text{Loop}(x)@i$

$\overbrace{(\forall j.\ \text{Start}(x)@j \Rightarrow \bot))}^{\psi_1}$

System Γ_{1^2}:

ψ_1

$i : \dfrac{\text{A}(x)}{\text{A}(x)}[\text{Loop}(x)]$

System Γ_{1^3}:

ψ_1

$j_1 : \dfrac{\text{A}(x)}{\text{A}(x)}[\text{Loop}(x)]$

$i : \dfrac{\text{A}(x)}{\text{A}(x)}[\text{Loop}(x)]$

ad infinitum …

⋮

System $\Gamma_{1^2,2}$:

ψ_1

$j_1 : \dfrac{\text{Fr}(x)}{\text{A}(x)}[\text{Start}(x)]$

$i : \dfrac{\text{A}(x)}{\text{A}(x)}[\text{Loop}(x)]$

ψ_1 and $\text{Start}(x)@j_1$

$\Rightarrow$ *contradiction*

System $\Gamma_{1^3,2}$:

ψ_1

$j_2 : \dfrac{\text{Fr}(x)}{\text{A}(x)}[\text{Start}(x)]$

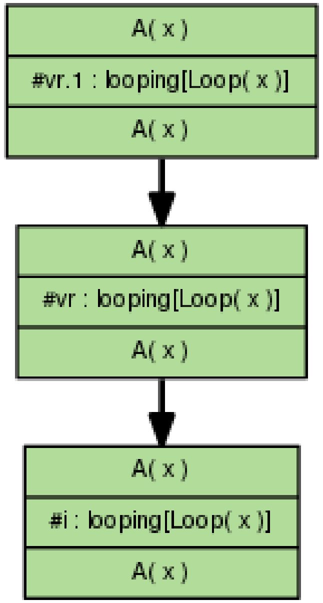
$j_1 : \dfrac{\text{A}(x)}{\text{A}(x)}[\text{Loop}(x)]$

$i : \dfrac{\text{A}(x)}{\text{A}(x)}[\text{Loop}(x)]$

ψ_1 and $\text{Start}(x)@j_2$

$\Rightarrow$ *contradiction*

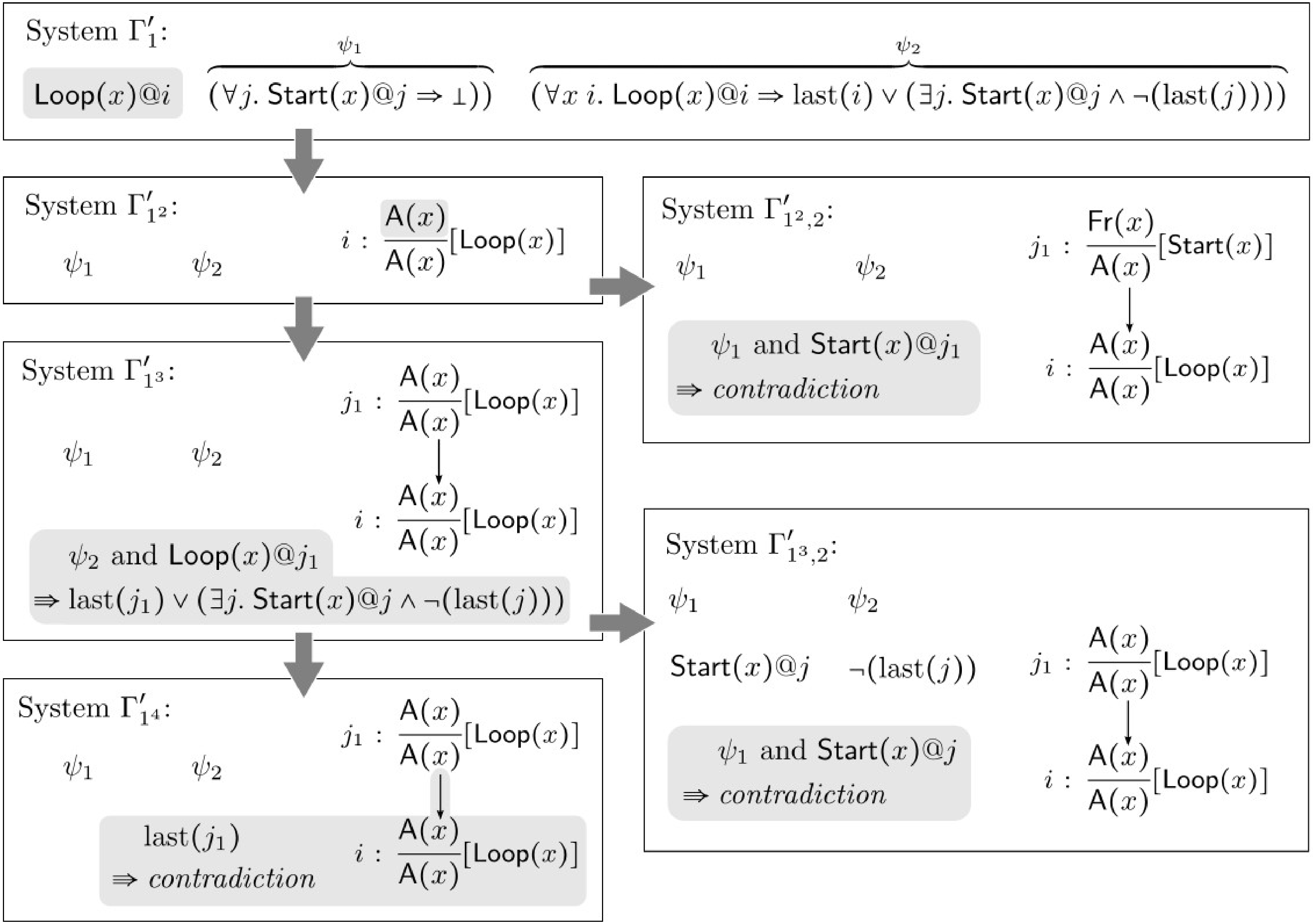
Demo


A(x)
#vr.1 : looping[Loop(x)]
A(x)
A(x)
#vr : looping[Loop(x)]
A(x)
A(x)
#i : looping[Loop(x)]
A(x)

Induction – on time points

- Informally, induction works on previous slide
- Formally, for IH ϕ
  1) Check if ϕ holds for empty trace
  2) Consider special last rule index on trace
     - Assume ϕ holds at all non-last indices, and prove for last
- Added constraint reduction rules for last atoms
- Allows proof of previous example

Example – solved by induction



System Γ'_1:

$\boxed{\mathsf{Loop}(x)@i}$ $\overbrace{(\forall j.\ \mathsf{Start}(x)@j \Rightarrow \bot))}^{\psi_1}$ $\overbrace{(\forall x\ i.\ \mathsf{Loop}(x)@i \Rightarrow \mathrm{last}(i) \vee (\exists j.\ \mathsf{Start}(x)@j \wedge \neg(\mathrm{last}(j))))}^{\psi_2}$

System Γ'_{1^2}:

ψ_1 ψ_2

$i\ :\ \dfrac{\mathsf{A}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x)]$

System $\Gamma'_{1^2,2}$:

ψ_1 ψ_2

$j_1\ :\ \dfrac{\mathsf{Fr}(x)}{\mathsf{A}(x)}[\mathsf{Start}(x)]$

$i\ :\ \dfrac{\mathsf{A}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x)]$

ψ_1 and $\mathsf{Start}(x)@j_1$

$\Rightarrow$ *contradiction*

System Γ'_{1^3}:

ψ_1 ψ_2

$j_1\ :\ \dfrac{\mathsf{A}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x)]$

$i\ :\ \dfrac{\mathsf{A}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x)]$

ψ_2 and $\mathsf{Loop}(x)@j_1$

$\Rightarrow \mathrm{last}(j_1) \vee (\exists j.\ \mathsf{Start}(x)@j \wedge \neg(\mathrm{last}(j)))$

System $\Gamma'_{1^3,2}$:

ψ_1 ψ_2

$\mathsf{Start}(x)@j$ $\neg(\mathrm{last}(j))$

$j_1\ :\ \dfrac{\mathsf{A}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x)]$

$i\ :\ \dfrac{\mathsf{A}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x)]$

ψ_1 and $\mathsf{Start}(x)@j$

$\Rightarrow$ *contradiction*

System Γ'_{1^4}:

ψ_1 ψ_2

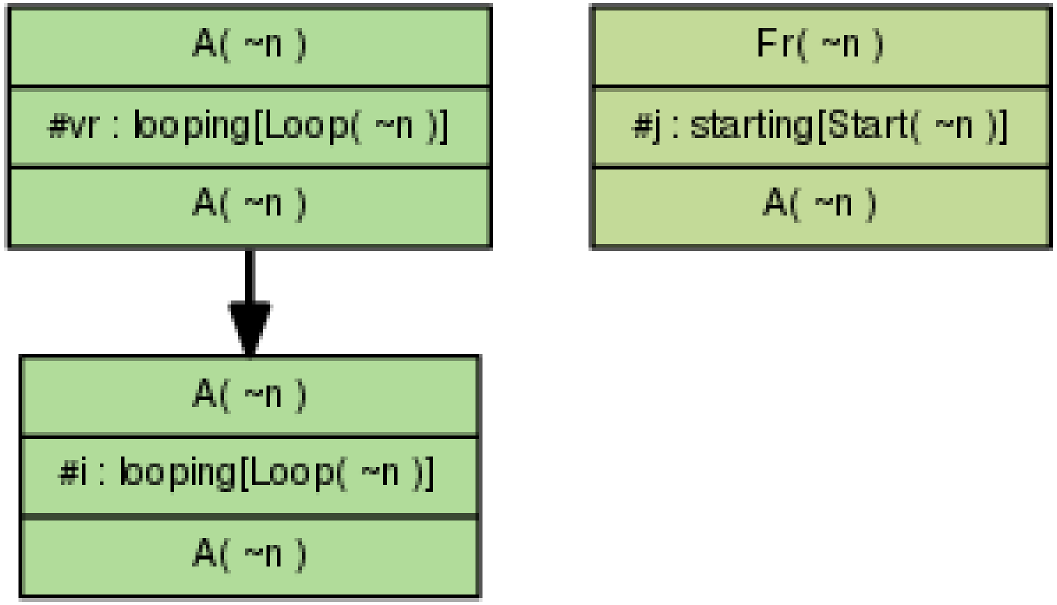
$j_1\ :\ \dfrac{\mathsf{A}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x)]$

$i\ :\ \dfrac{\mathsf{A}(x)}{\mathsf{A}(x)}[\mathsf{Loop}(x)]$

$\mathrm{last}(j_1)$

$\Rightarrow$ *contradiction*

Demo – using induction



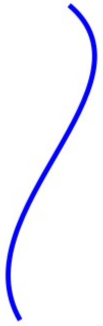
Induction in general

- Required for all “sources” lemmas
- Often required for “reuse” lemmas
- Helps for all looping constructs, used in e.g.:
  - YubiKey
  - TPM
  - PKCS11
  - Group protocols
  - Counters

Observational equivalence

Observational equivalence

Two types of properties:



- Trace properties
  - (Weak) secrecy as reachability
  - Authentication as correspondence

- Observational equivalence $\approx$

Why observational equivalence?

- Consider classic **Dolev-Yao** adversary for deterministic pub-key encryption:

  `adec( aenc(x, k), pk(k) ) = x`

- Adversary can only decrypt if he knows the secret key

  Consider a simple voting system:
  - Voter chooses `v=”Yes”` or `v=”No”`
  - Encrypt v using server’s public key `pk(k)`:

    `c = enc(v,pk(k))`
  - Send `c` to server

  Is the vote secret?
  - Dolev-Yao: **Yes**, adversary does not know server’s secret key
  - Reality: **No**, encryption is deterministic and there are only two choices
    - **Attack**: encrypt `“Yes”`, and compare to `c`

Observational equivalence vs reachability

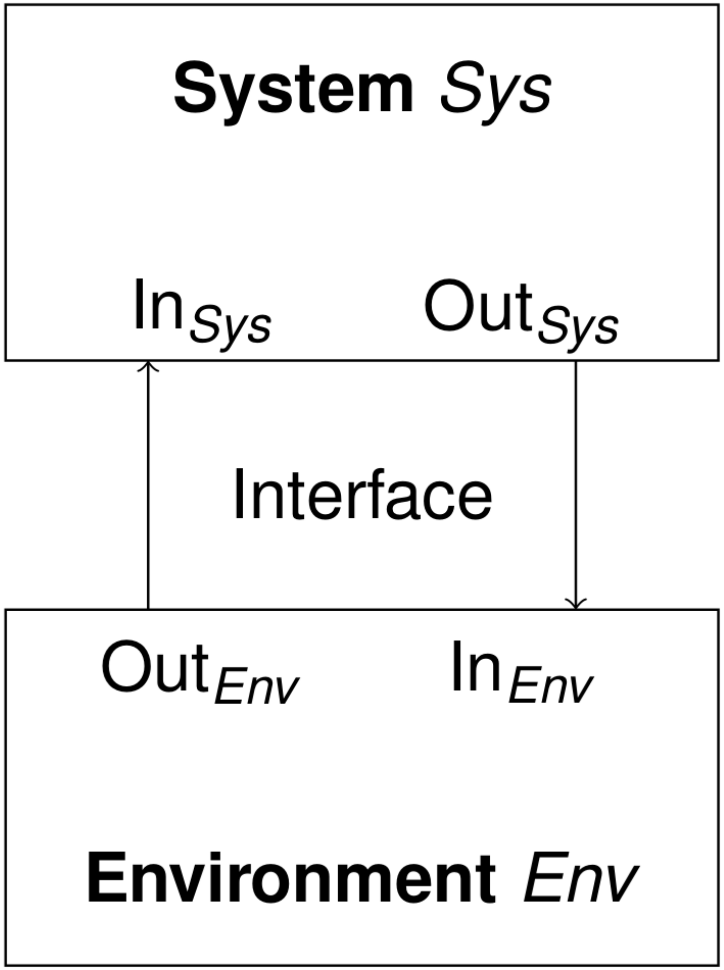
- **Reachability**-based (weak) secrecy is insufficient
- **Stronger** notion: adversary cannot distinguish
  - a system where the voter votes `“Yes”` from
  - a system where the voter votes `“No”`
- **Observational equivalence** between two systems
- Can be used to express
  - Strong secrecy
  - Privacy notions

Running example

- **Auction** system for a **shout-out auction**
- Property: **strong secrecy** of bids
- Property violated:
  - Broadcast bid (e.g., `A` or `B`)
  - Send “`A`” in first system
  - Send “`B`” in second system
  - Observer knows if he is observing first or second system
- Property holds using **shared symmetric key**:
  - Shared symmetric key k between bidder and auctioneer
  - Send “$\{A\}_k$” in first system
  - Send “$\{B\}_k$” in second system
  - Observer has no access to k, does not know which system he observes

System and environment

- We separate **environment** and **system**
  - System: agents running according to protocol
  - Environment: adversary acting according to its capabilities
- Environment can observe:
  - Output of the system
  - If system reacts at all



Defining observational equivalence

- **Two** system **specifications** given as set of rules
  - One rule per role action (send/receive)
  - Running example shout-out auction:

$$\text{System 1: } \frac{}{\mathrm{Out}_{Sys}(A)} \qquad \text{System 2: } \frac{}{\mathrm{Out}_{Sys}(B)}$$

- Interface and environment/adversary rule(s):

$$\frac{\mathrm{Out}_{Sys}(X)}{\mathrm{In}_{Env}(X)} \qquad \frac{\mathrm{Out}_{Env}(X)}{\mathrm{In}_{Sys}(X)} \qquad \frac{\mathrm{In}_{Env}(X) \quad K(X)}{\mathrm{Out}_{Env}(\mathit{true})}$$

  - Last rule models comparison by the adversary
- Each specification yields a labeled transition system
- Observational equivalence is a kind of **bisimulation** accounting for the adversaries’ **viewpoint** and **capabilities**

Diff terms

- General definitions of observational equivalence **difficult** to verify: requires inventing simulation relation
- Idea: **specialize** for cryptographic protocols
  - Consider strong bid secrecy:
    - both systems differ in **secret bid** only, i.e.,
    - both specifications contain **same rule(s)**, which differ only in **some terms**
  - Exploit this similarity in description and proof
- Approach: two systems described by one specification – using **diff**-terms
  - Running example

$$\frac{}{Out_{Sys}(A)} \qquad\qquad \frac{}{Out_{Sys}(B)}$$

  - Is equivalent to one rule with a **diff**-term

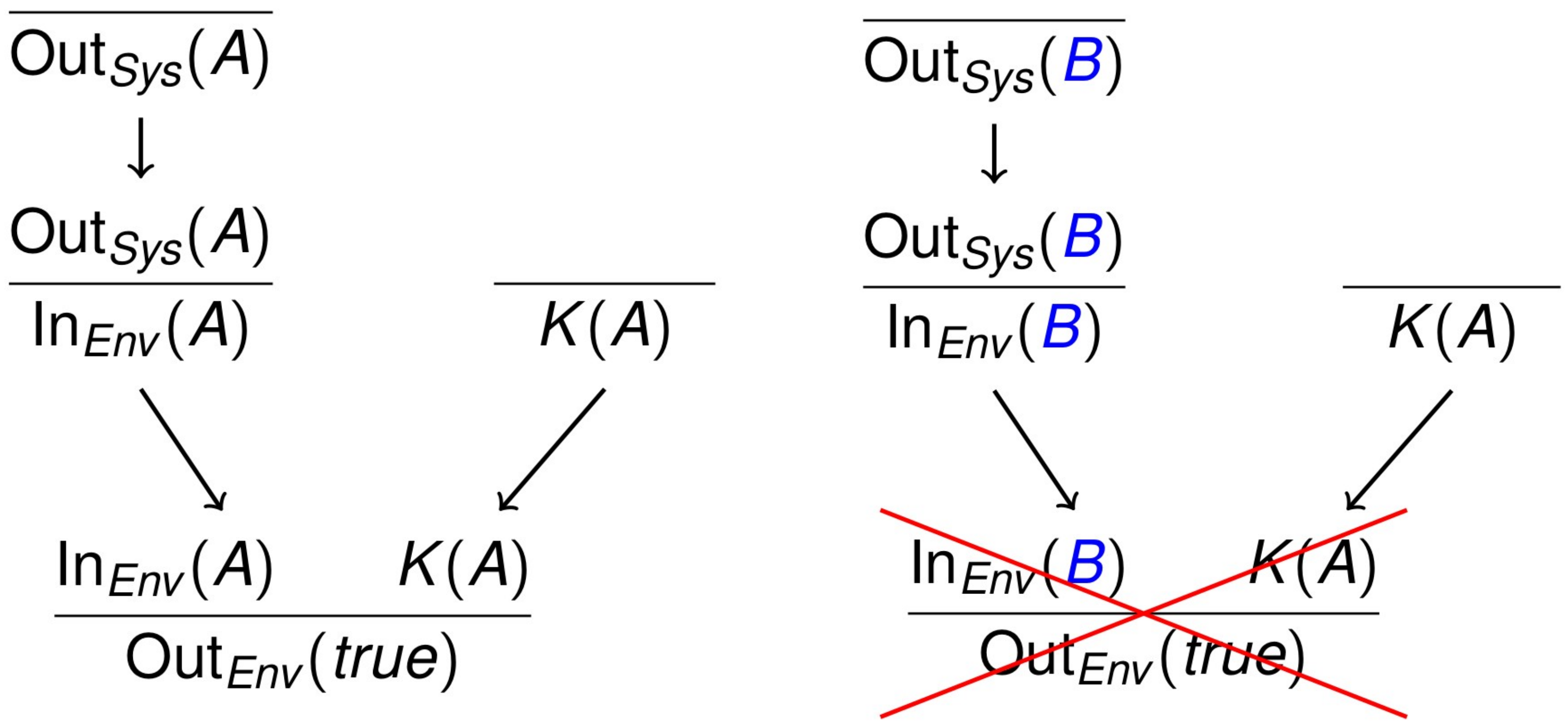
$$\frac{}{Out_{Sys}(\mathbf{diff}(A, B))}$$

Approximating observational equivalence using mirroring

- Both systems contain same rules modulo diff-terms
- Idea: assume that each rule simulates itself
- Compute **mirrors** of each execution into the other system
- If the mirrors are **valid executions**, we have **observational equivalence** (sound approximation)

Invalid mirrors and attacks

Bidder picks A/B, observer compares to public value A



Counter example to observational equivalence

Valid mirror

Observer compares system output to itself

$$\begin{tikzcd}
 & \overline{\mathrm{Out}_{Sys}(A)} \arrow[d] & \\
 & \dfrac{\mathrm{Out}_{Sys}(A)}{\mathrm{In}_{Env}(A)} \arrow[dd] \arrow[rd] & \\
 & & \dfrac{\mathrm{In}_{Env}(A)}{K(A)} \arrow[ld] \\
 & \dfrac{\mathrm{In}_{Env}(A) \qquad K(A)}{\mathrm{Out}_{Env}(true)} &
\end{tikzcd}$$

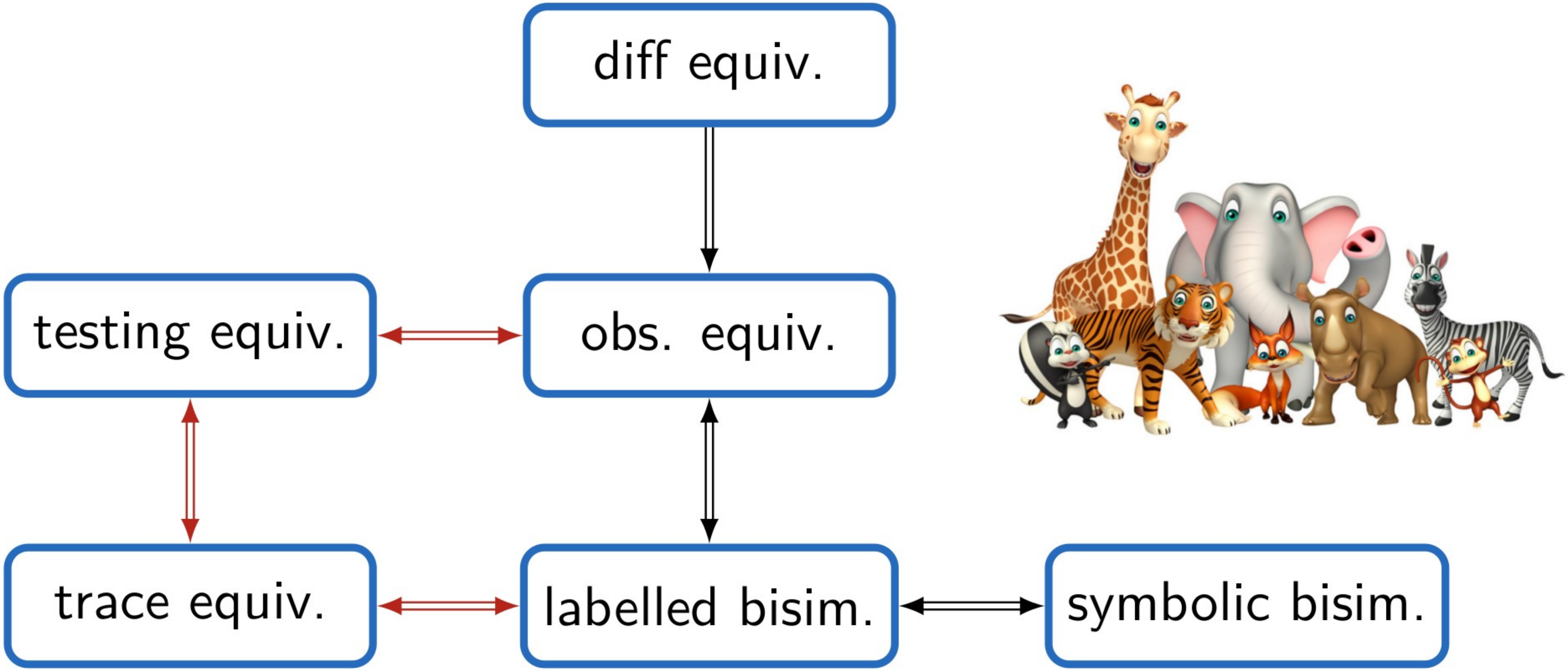
$$\begin{tikzcd}
 & \overline{\mathrm{Out}_{Sys}(B)} \arrow[d] & \\
 & \dfrac{\mathrm{Out}_{Sys}(B)}{\mathrm{In}_{Env}(B)} \arrow[dd] \arrow[rd] & \\
 & & \dfrac{\mathrm{In}_{Env}(B)}{K(B)} \arrow[ld] \\
 & \dfrac{\mathrm{In}_{Env}(B) \qquad K(B)}{\mathrm{Out}_{Env}(true)} &
\end{tikzcd}$$

- **All** mirrors need to be valid for observational equivalence

Dependency graph equivalence

- A **diff**-system is dependency graph equivalent if mirrors of all dependency graphs rooted in any rule on both sides are valid.
  - Sound but incomplete approximation
  - Efficient and sufficient in practice
- Input:
  - Protocol specification
  - Property: equivalence given two choices for some term(s)
    - Example: random value vs expected value
- Output:
  - Yes, observational equivalent
  - No, dependency graph with invalid mirror
  - Non-termination possible

The equivalence zoo



diff equiv.

testing equiv.

obs. equiv.

trace equiv.

labelled bisim.

symbolic bisim.

Red arrows require assumptions: determinate processes + bounded sessions (no replication)

Guided proofs & Custom Heuristics

Lemmas

- When it doesn’t terminate…
- Guide the proof manually; export
- Write **lemmas**
  - “**Hints**” for the prover
    - They don't change the proof obligation, only help finding a proof
  - Specify lemma that can be used to prune proof trees at multiple points

Storing Proofs

- Complete (or partial) proofs can be stored
  - Click the “Download” button in top right
- These can be reloaded like normal theories
  - Proof is rechecked!