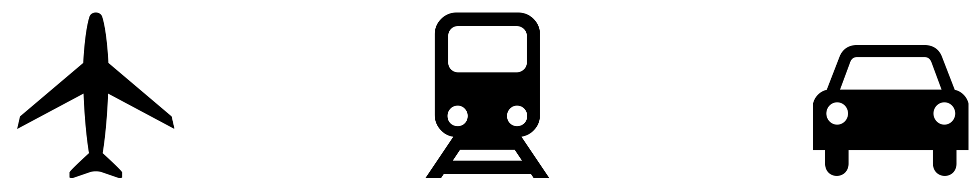
Towards Testing Model Transformation Chains Using Precondition Construction in Algebraic Graph Transformation

Elie Richa	Télécom ParisTech & AdaCore
Etienne Borde	Télécom ParisTech
Laurent Pautet	Télécom ParisTech
Matteo Bordin	AdaCore
José F. Ruiz	AdaCore

September 29, 2014

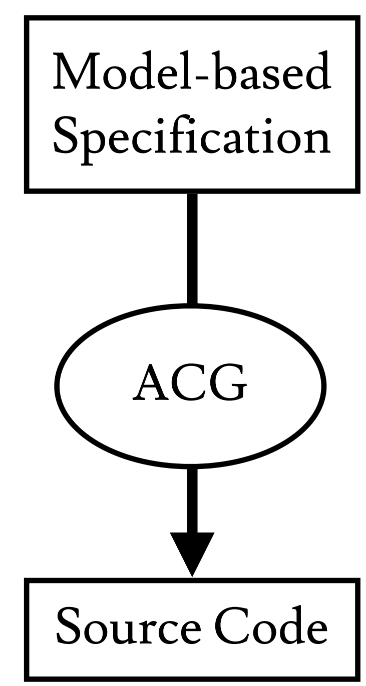
Development of Critical Software

- **Critical** : Software failure can have catastrophic consequences

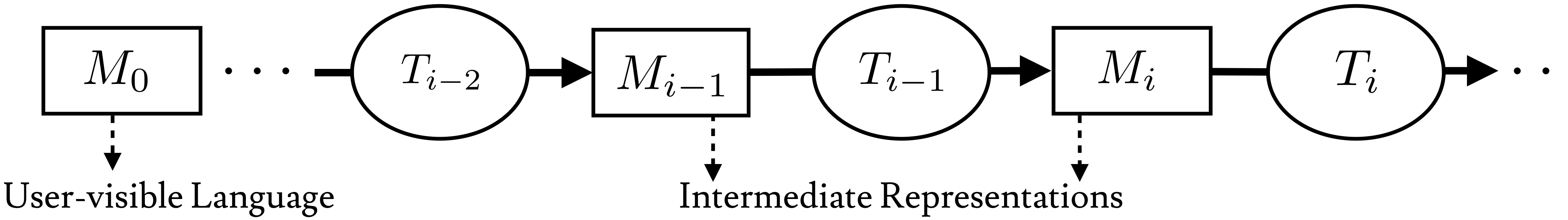


DO178C EN50128 ISO26262

- Certification standards are mandatory and costly to apply
- Model-based development enables Automatic Code Generation (ACG)
- ACGs must be **Qualified.** e.g. SCADE Suite KCG
- Qualification of an ACG is very costly
- Extensive testing of the ACG is required

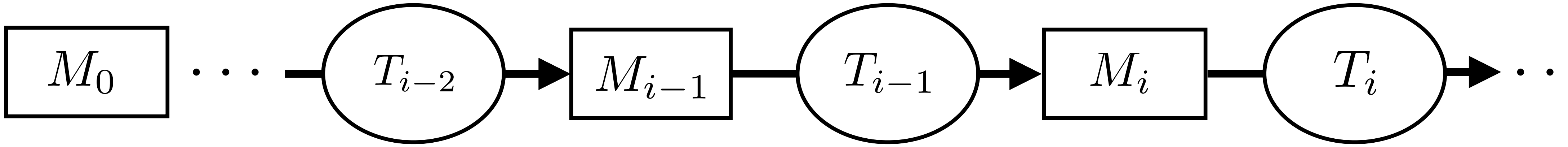


Unit Testing v/s Integration Testing in ACGs



- An ACG is typically a Model Transformation Chain (**MTC**)
- **Unit testing** (Unit = T_i = intermediate transformation)
  - Consider intermediate transformations in isolation
  - Develop test models in **intermediate representations**
- **Integration testing**
  - Consider whole chain
  - Develop test models in the **input language**

Reality Check

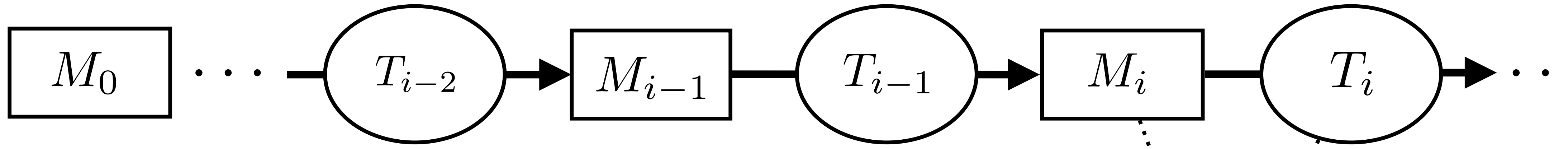


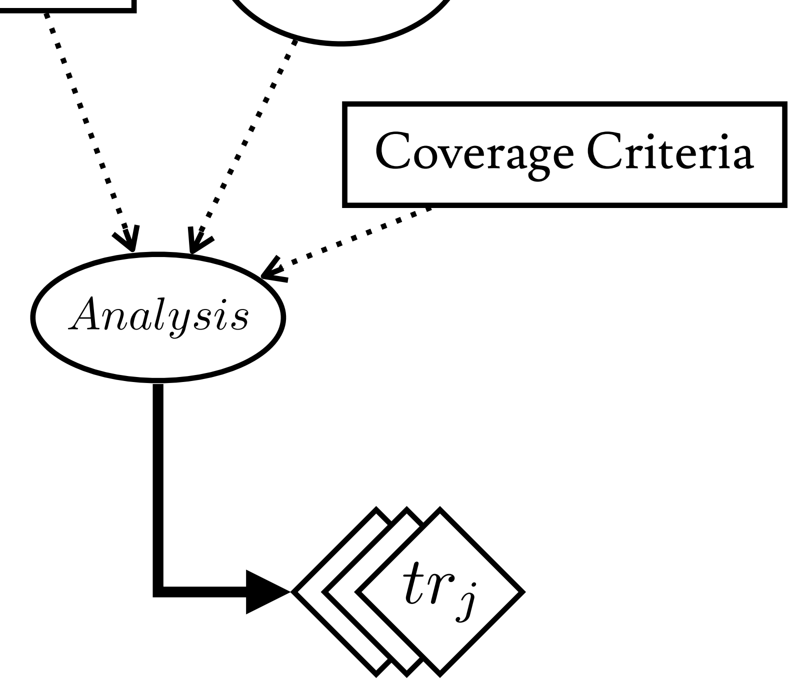
- Feedback from developers of ACGs and the GCC compilation chain
  - 10 ~ 20 intermediate transformations
- Unit testing of intermediate transformation is rarely performed
- Intermediate test models are difficult to produce and maintain
  - Intermediate models increase in size along the chain
  - Internal languages have no dedicated editors
  - Intermediate languages and transformations evolve during the lifecycle

General Problem

Perform only integration tests but cover all unit testing needs

Unit Testing of Model Transformations



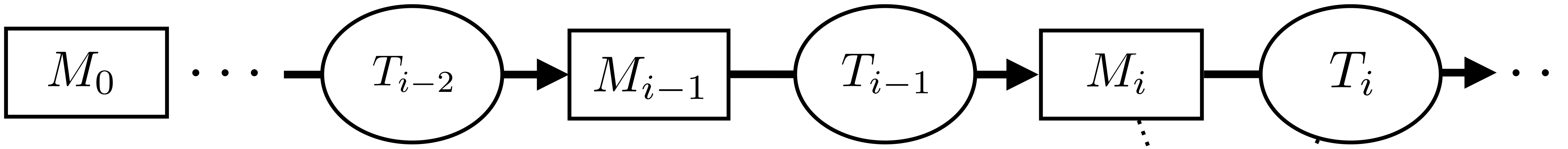


- Coverage
  - Testing all individual cases is unfeasible
  - An analysis identifies sets of equivalent cases

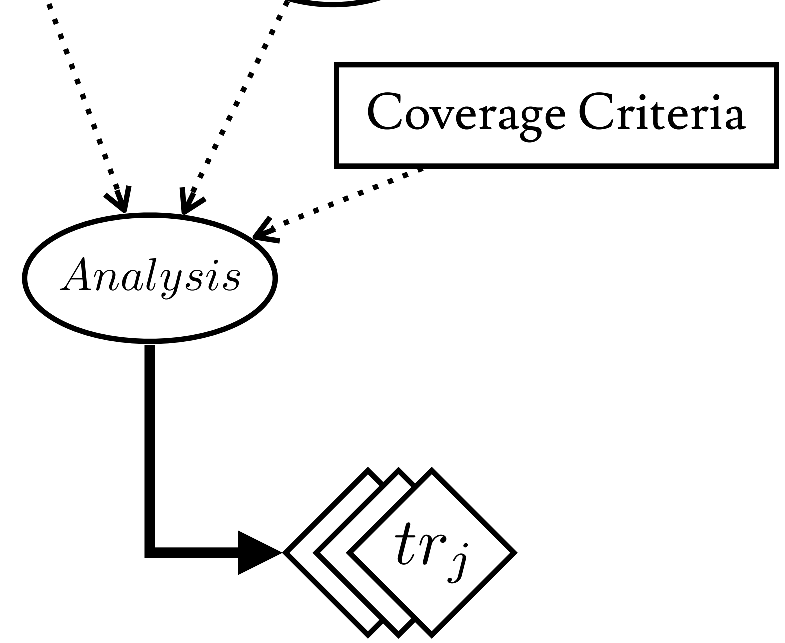
- **Test objective/requirement**
  Constraint over the input language of a transformation
  - Describes a set of input models exhibiting a common property

- One test model is sufficient to cover a test objective

Unit Testing of Model Transformations



- Coverage
  - Testing all individual cases is unfeasible
  - An analysis identifies sets of equivalent cases

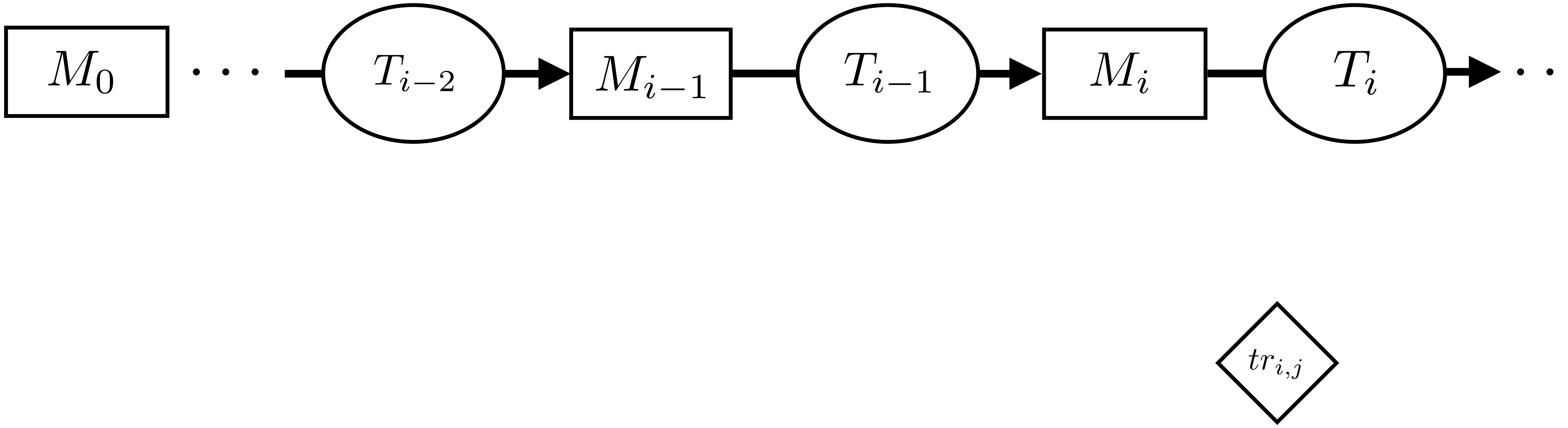


- **Test objective/requirement**
  Constraint over the input language of a transformation
  - Describes a set of input models exhibiting a common property

$$tr_1 : \exists\, p : Param \mid p.value = 0$$
$$tr_2 : \exists\, p : Param \mid 0 < p.value <= 5$$
$$tr_3 : \exists\, p : Param \mid 5 < p.value$$

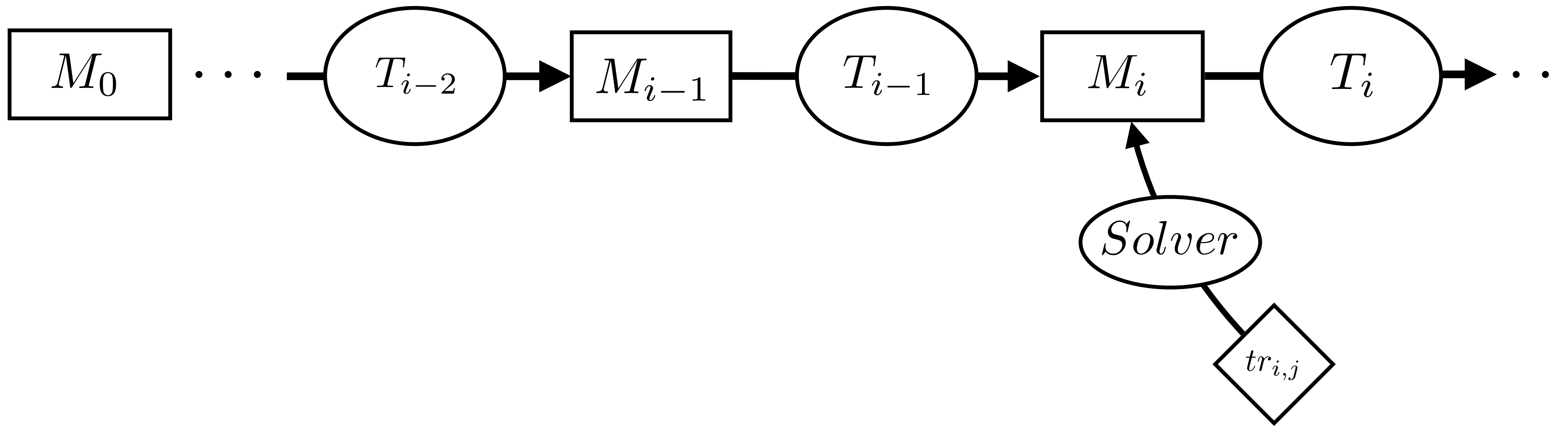
- One test model is sufficient to cover a test objective

Covering Non-Satisfied Test Requirements



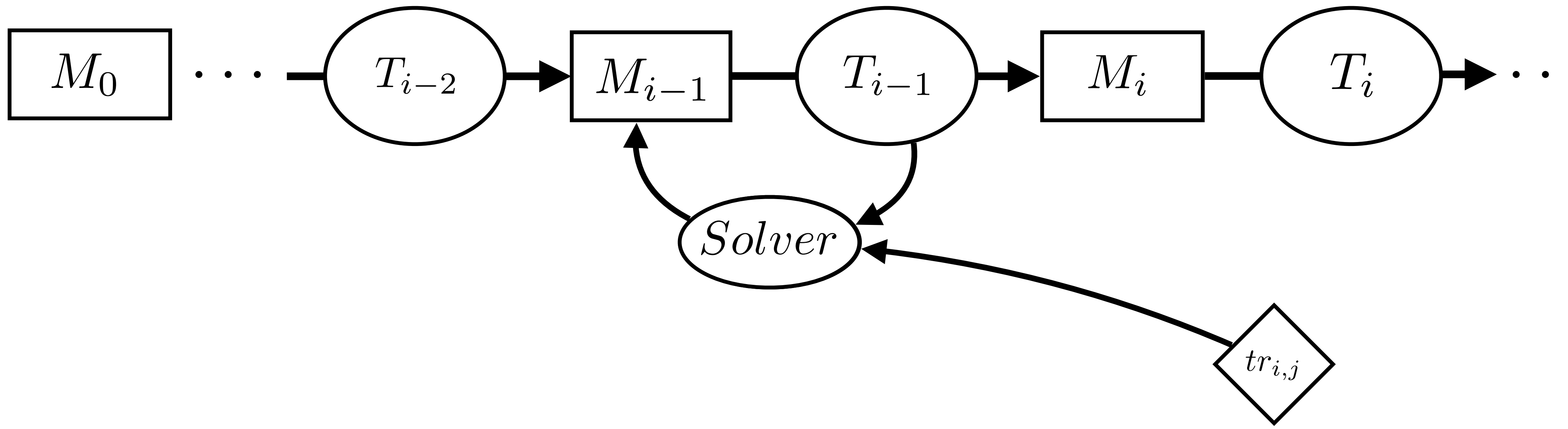
- Constraint Satisfaction Problem (CSP)
- A CSP Solver can theoretically generate a satisfying instance
- Encoding all transformations in the CSP is not scalable
- The result is an **instance** and not a **constraint** which prevents an iterative analysis

Covering Non-Satisfied Test Requirements



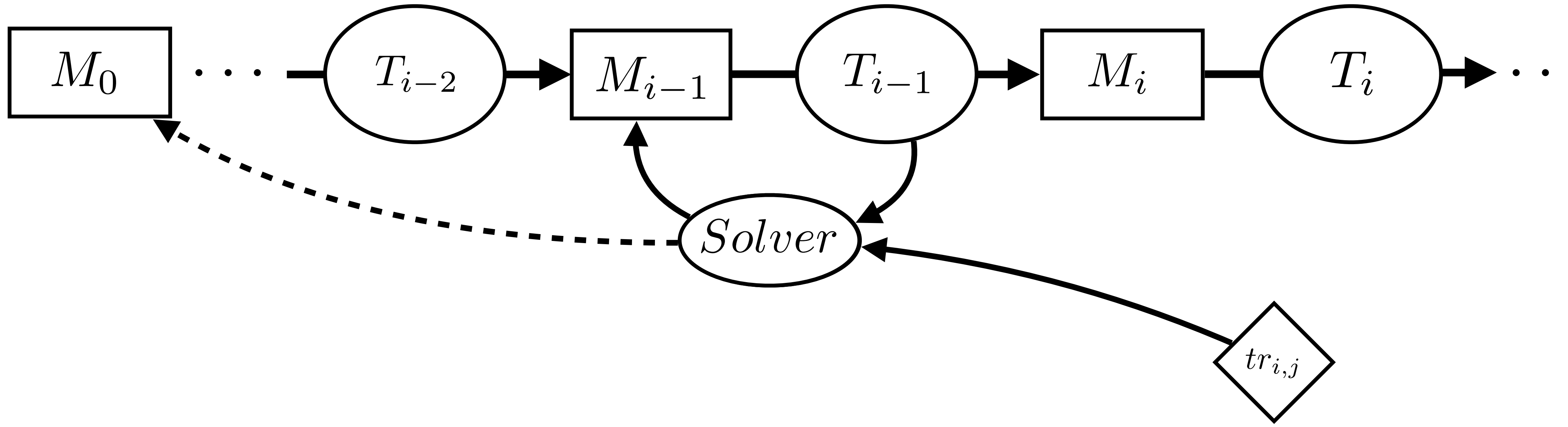
- Constraint Satisfaction Problem (CSP)
- A CSP Solver can theoretically generate a satisfying instance
- Encoding all transformations in the CSP is not scalable
- The result is an **instance** and not a **constraint** which prevents an iterative analysis

Covering Non-Satisfied Test Requirements



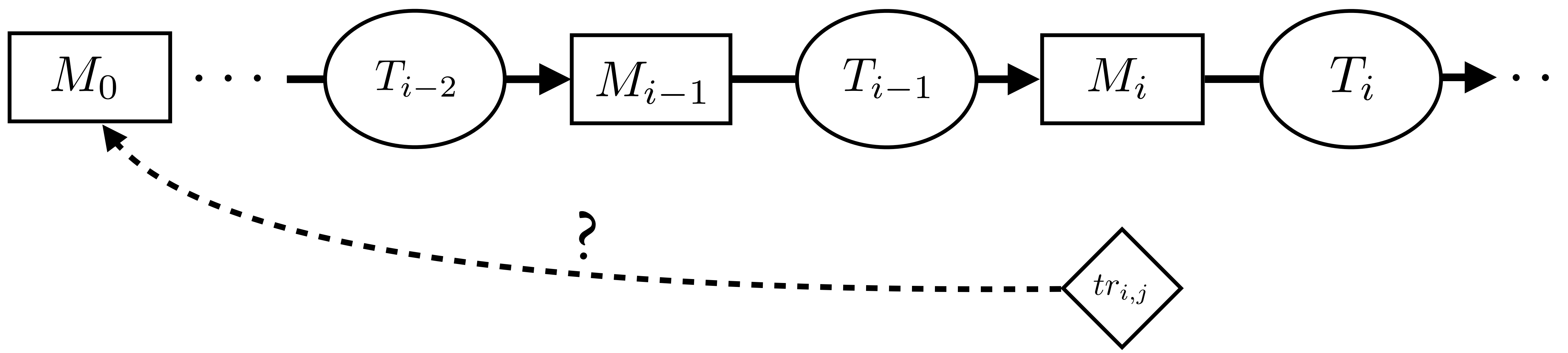
- Constraint Satisfaction Problem (CSP)
- A CSP Solver can theoretically generate a satisfying instance
- Encoding all transformations in the CSP is not scalable
- The result is an **instance** and not a **constraint** which prevents an iterative analysis

Covering Non-Satisfied Test Requirements



- Constraint Satisfaction Problem (CSP)
- A CSP Solver can theoretically generate a satisfying instance
- Encoding all transformations in the CSP is not scalable
- The result is an **instance** and not a **constraint** which prevents an iterative analysis

Satisfying Intermediate Test Requirements

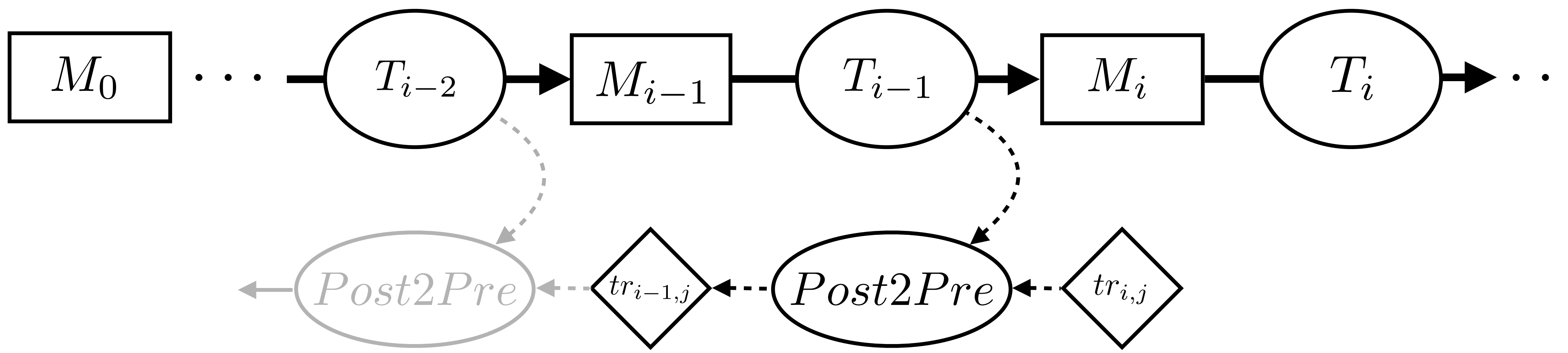


Problem

How to produce a new model in the input language to satisfy a **given** intermediate test requirement

- It is a challenging problem because
  - We have to consider an arbitrary number of preceding transformations
  - Tester has to manually "inverse" transformations
  - Transformations are non-injective and non-surjective
  - We have to reason on constraints and not on instances

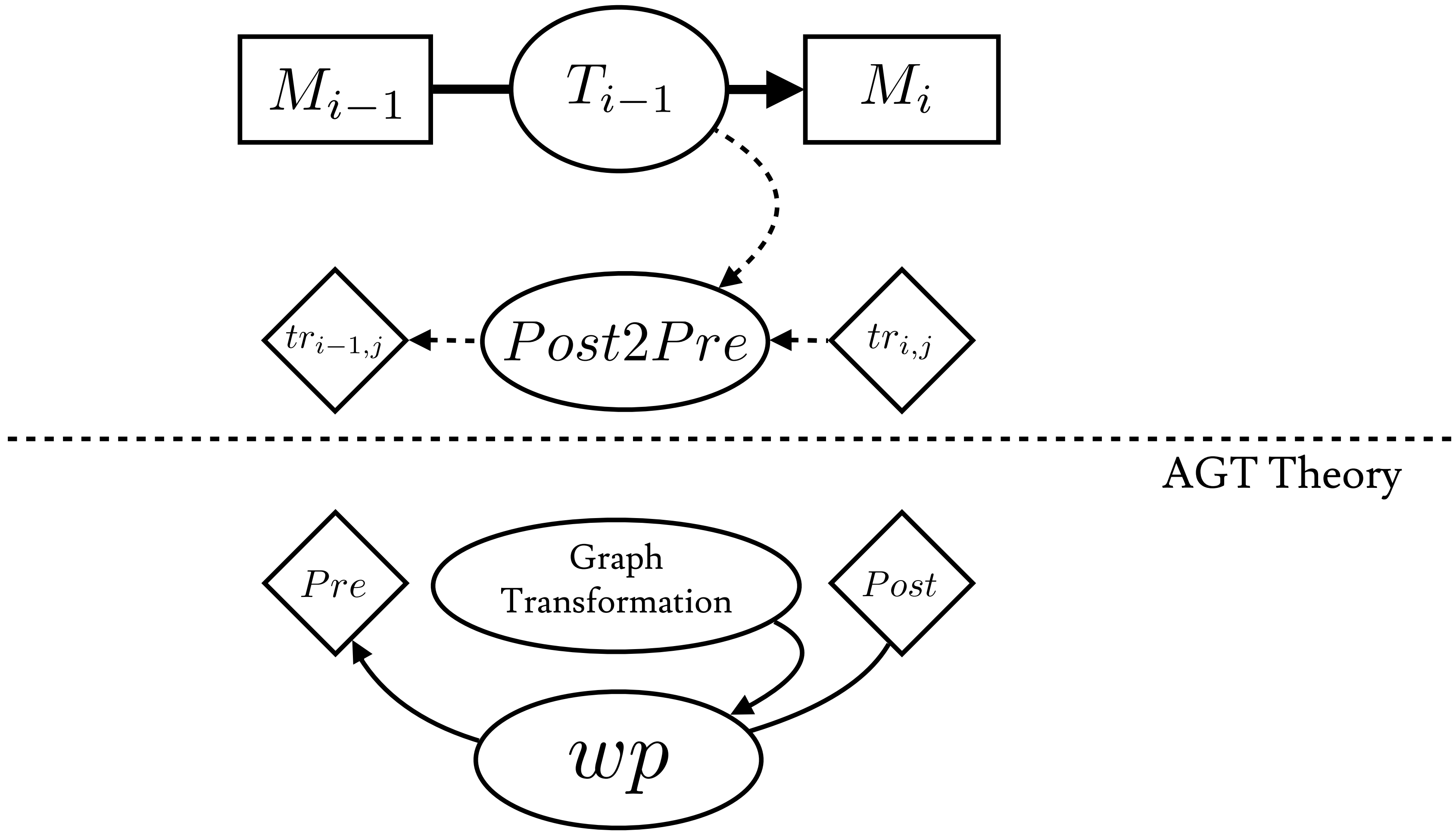
Step-by-step Advancement of Test Requirements



Contribution

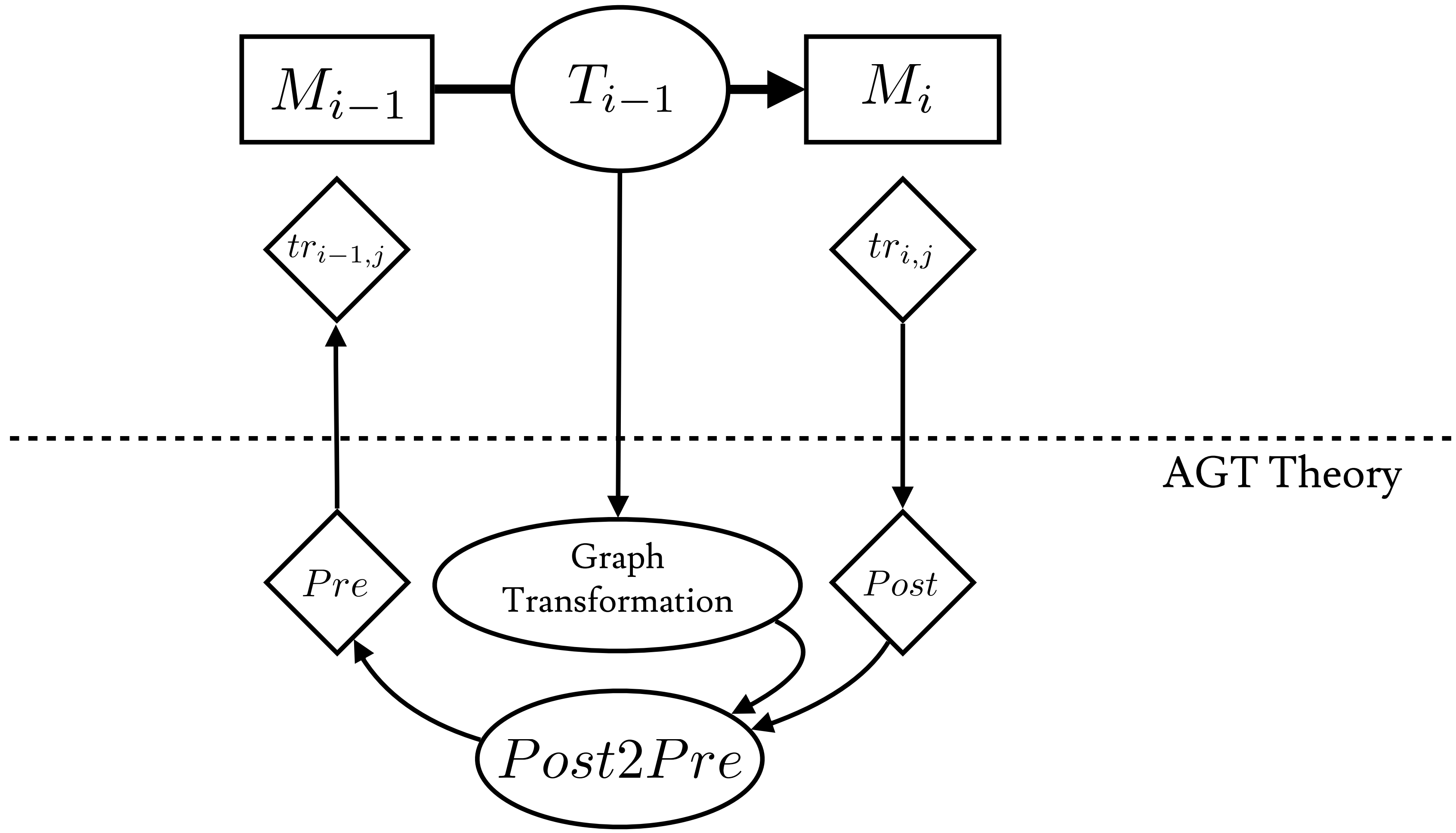
- Step-by-step automatic advancement of test requirements up to input language
  - Test requirement as a postcondition of previous step
  - Transform a postcondition into a sufficient precondition
  - Iterate process up to the input language

Algebraic Graph Transformation



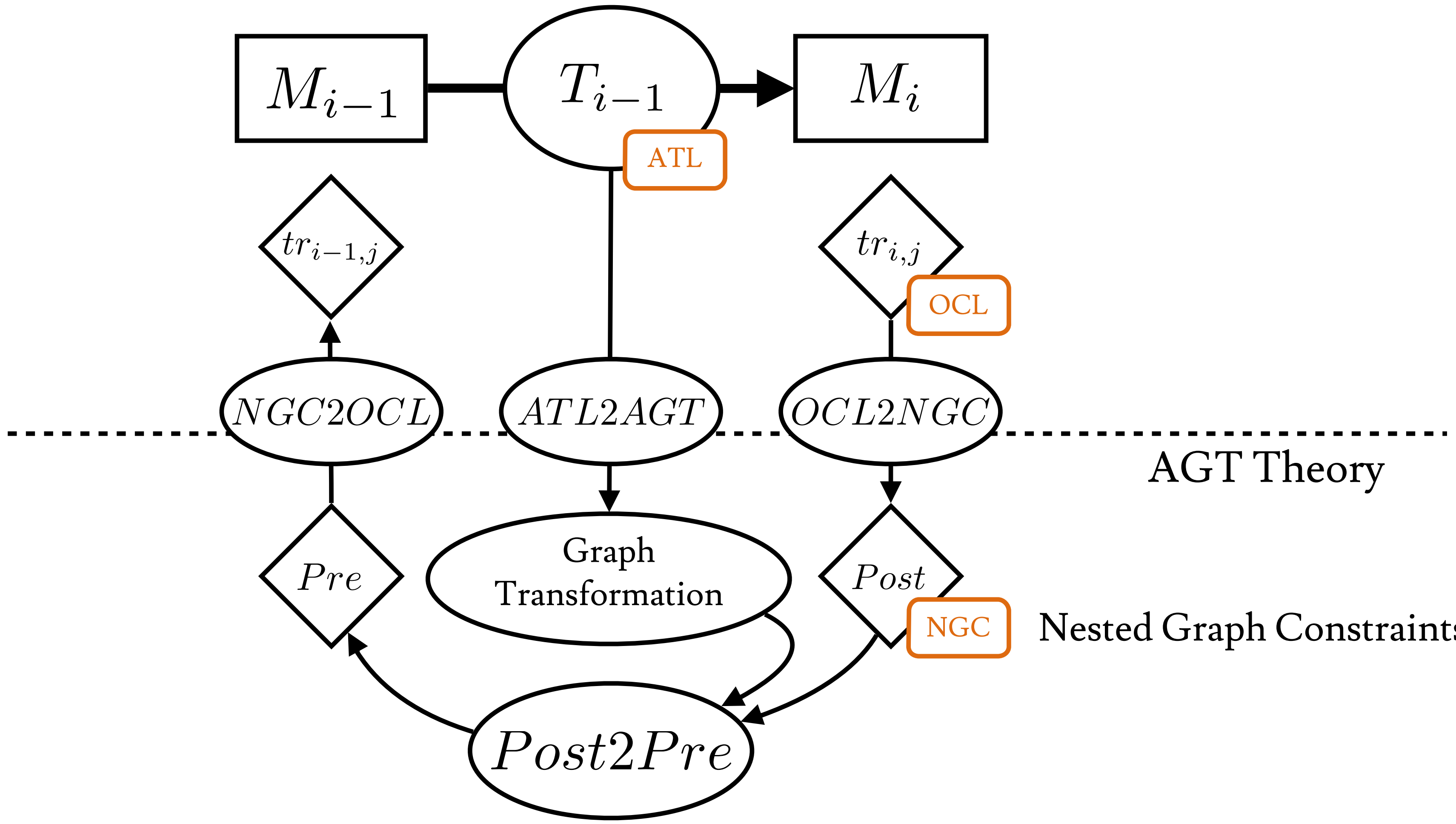
- Algebraic Graph Transformation (AGT)
  - Formal framework based on Category Theory
- Construction of Weakest Precondition wp
  - Constructs a precondition ensuring the satisfaction of the postcondition

Using AGT to Advance Test Requirements



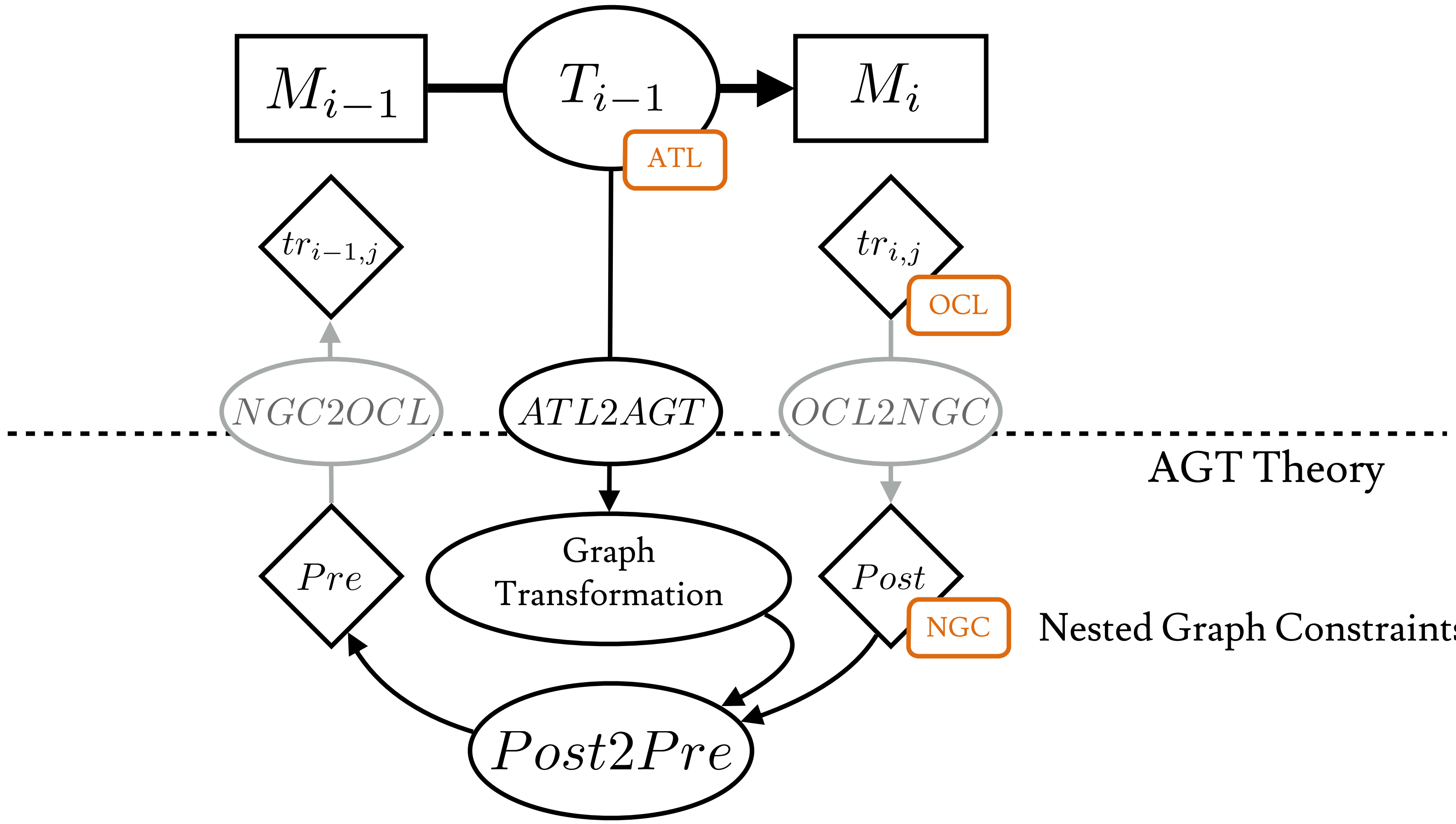
- Transpose our problem into the AGT Theory
- The ACG is specified in industry standard languages
  - ATL and OCL

Using AGT to Advance Test Requirements



- 3 main components
  - Translation of transformations — ATL2AGT
  - Translation of test requirements — OCL2NGC (NGC2OCL)
  - Advancement of constraints — Post2Pre

Using AGT to Advance Test Requirements



- 3 main components
  - Translation of transformations — ATL2AGT
  - Translation of test requirements — OCL2NGC (NGC2OCL)
  - Advancement of constraints — Post2Pre

Translation of Transformations – ATL2AGT

Challenge

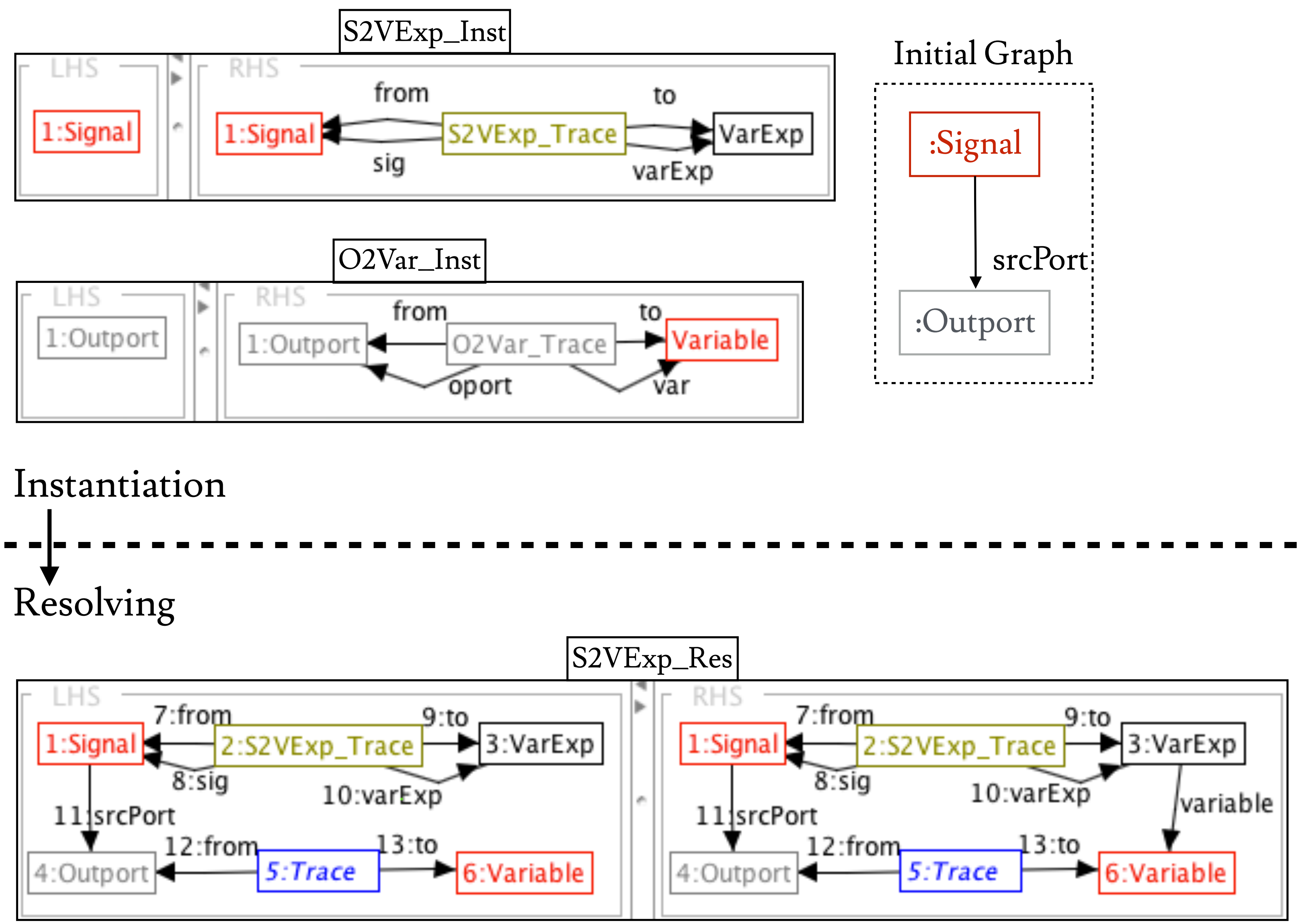
- Semantic gap between Declarative Model Transformation and AGT
- ATL semantics – create a **new** output model
  - Rules executed simultaneously
  - A rule can **resolve** the output of other rules using implicit input to output tracing
- AGT semantics – in-place graph rewriting
  - Rules applied sequentially and atomically
  - No resolve mechanism

Contribution

- Model an ATL transformation as a 2-phase rewriting of the input graph
  - Instantiation phase —> Resolving phase
  - Explicit **Trace nodes**

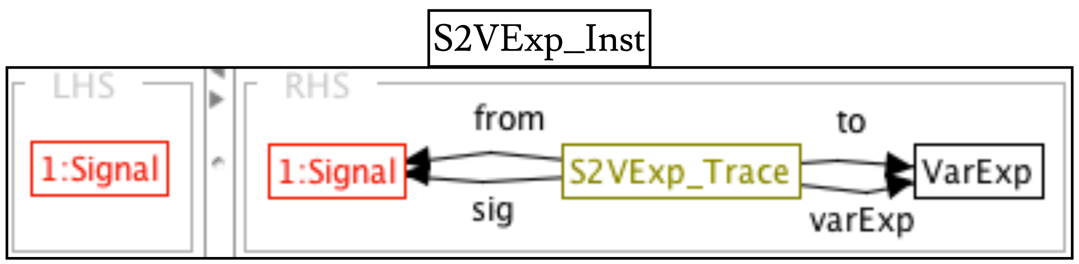
Example — ATL2AGT

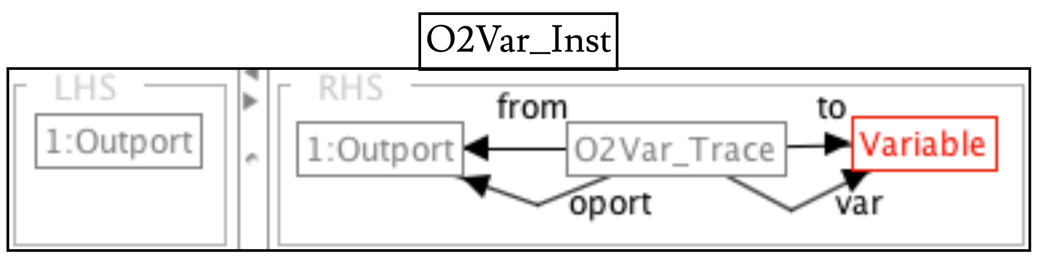
```
rule S2VExp { from sig    : SMM!Signal
              to   varExp : CMM!VarExp (variable <- sig.srcPort) }   -- Resolve

rule O2Var { from oport : SMM!Outport
             to   var   : CMM!Variable }
```

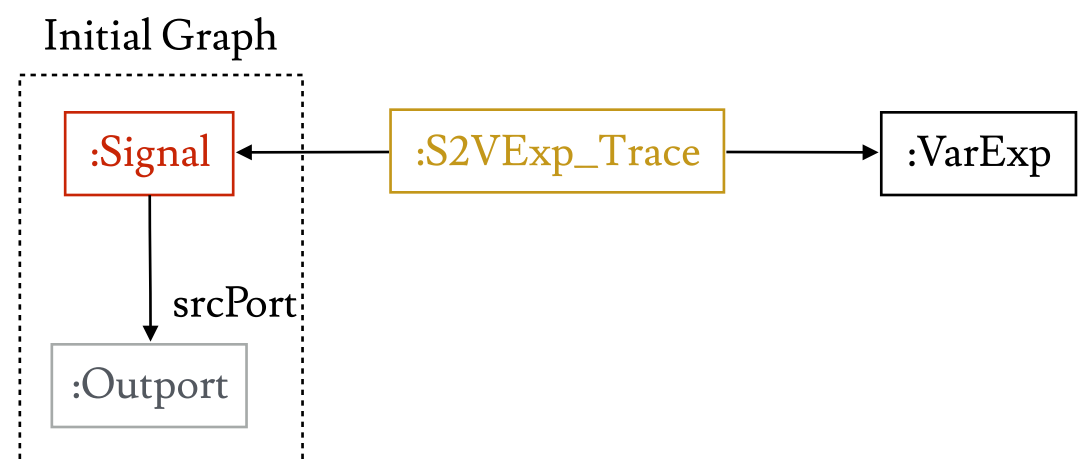


Example — ATL2AGT

```
rule S2VExp { from sig    : SMM!Signal
              to   varExp : CMM!VarExp (variable <- sig.srcPort) }   -- Resolve

rule O2Var { from oport : SMM!Outport
             to   var   : CMM!Variable }
```

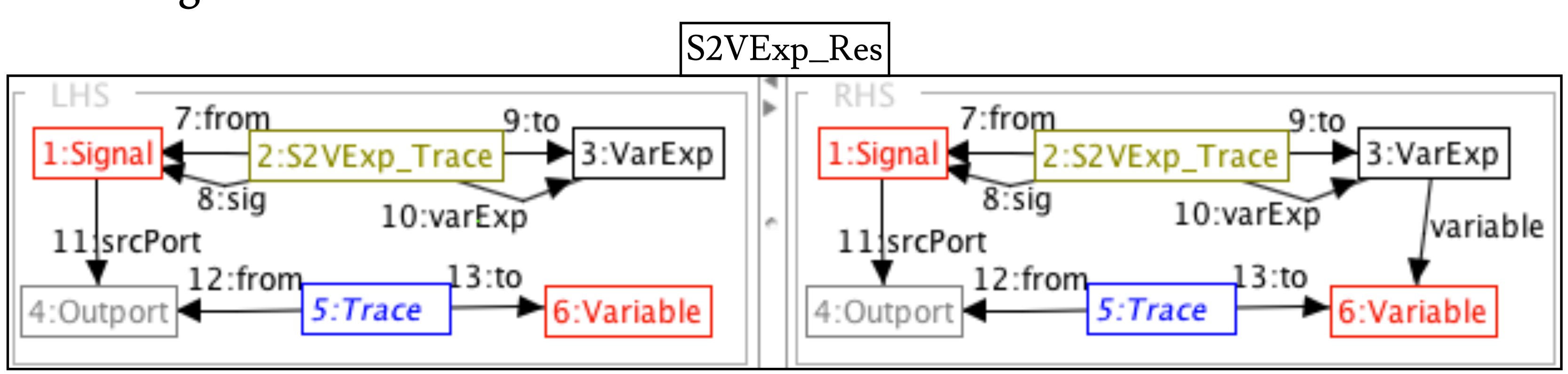






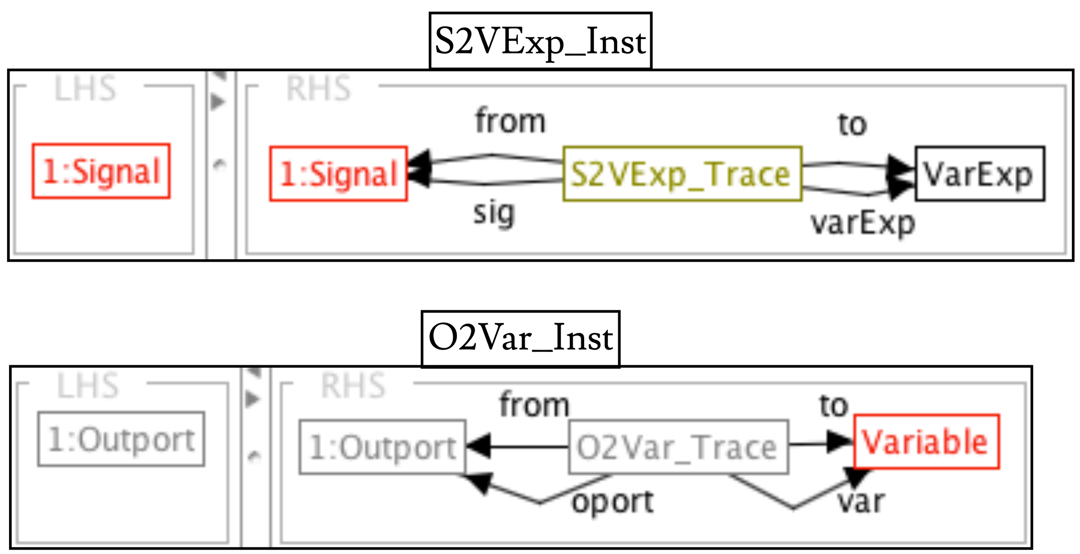
Instantiation

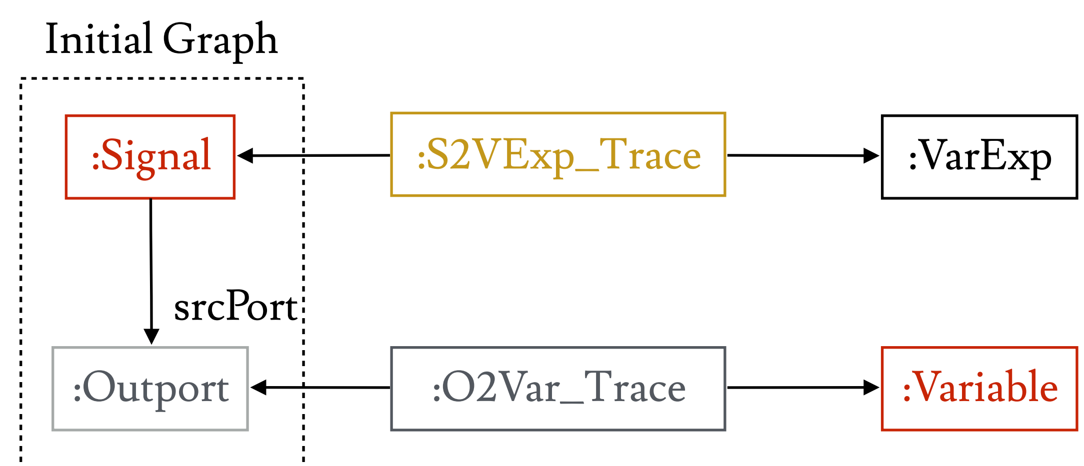
Resolving



Example — ATL2AGT

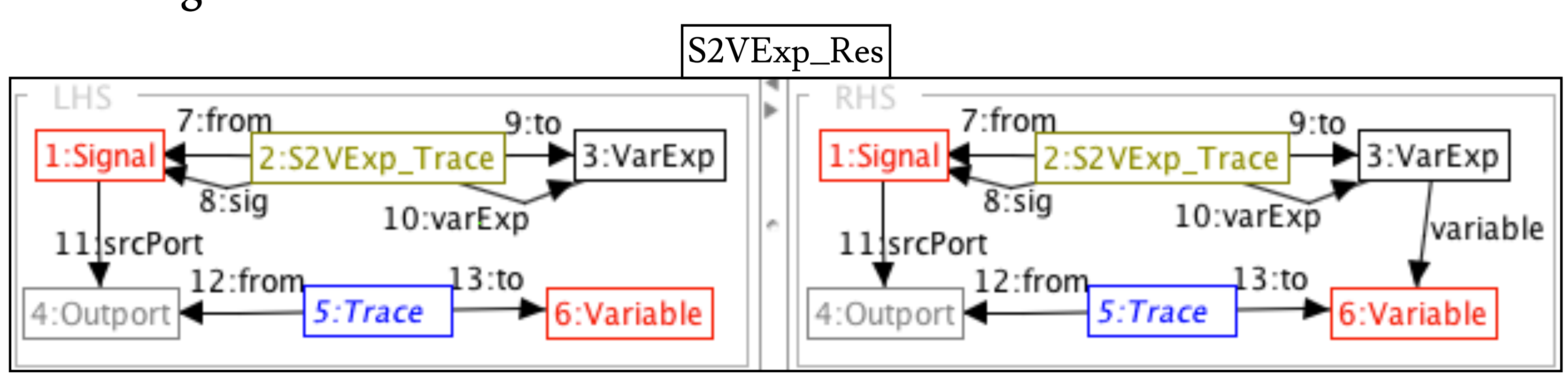
```
rule S2VExp { from sig    : SMM!Signal
              to   varExp : CMM!VarExp (variable <- sig.srcPort) }   -- Resolve

rule O2Var { from oport : SMM!Outport
             to   var   : CMM!Variable }
```





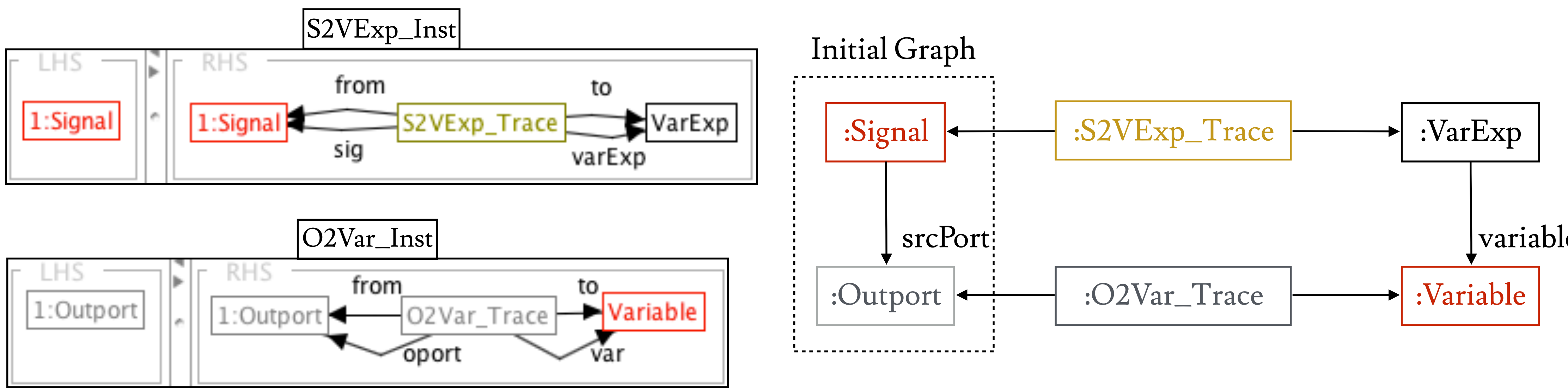
Instantiation

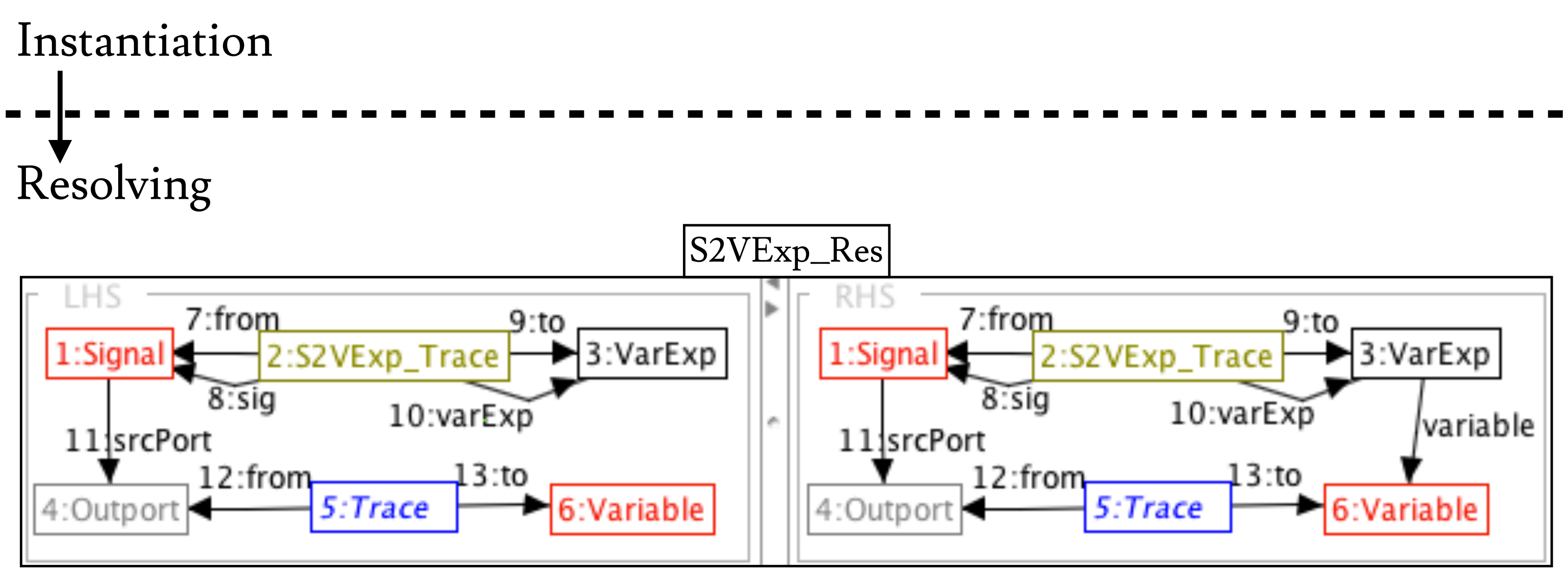
Resolving



Example — ATL2AGT

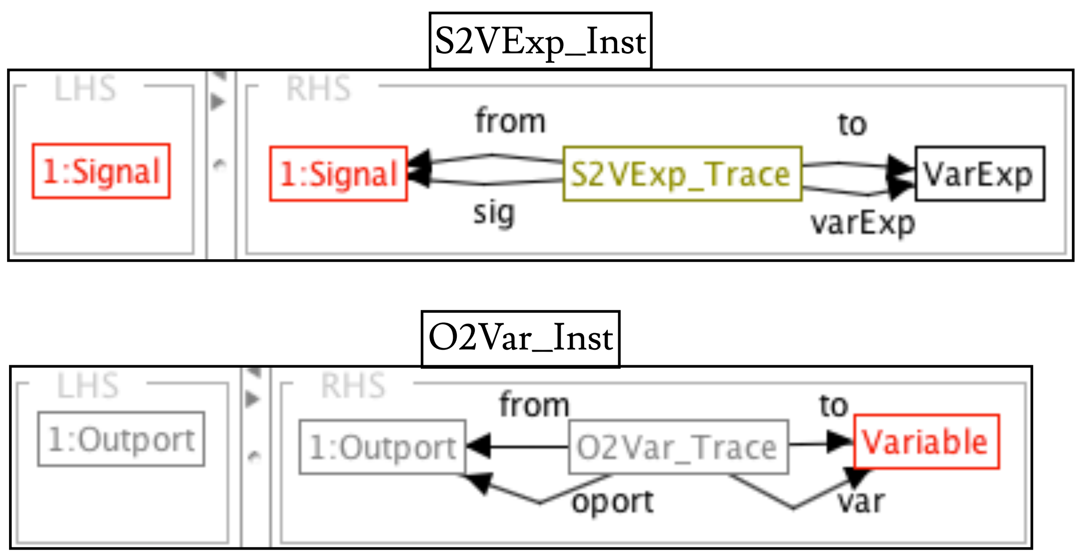
```
rule S2VExp { from sig    : SMM!Signal
              to   varExp : CMM!VarExp (variable <- sig.srcPort) }   -- Resolve

rule O2Var { from oport : SMM!Outport
             to   var   : CMM!Variable }
```

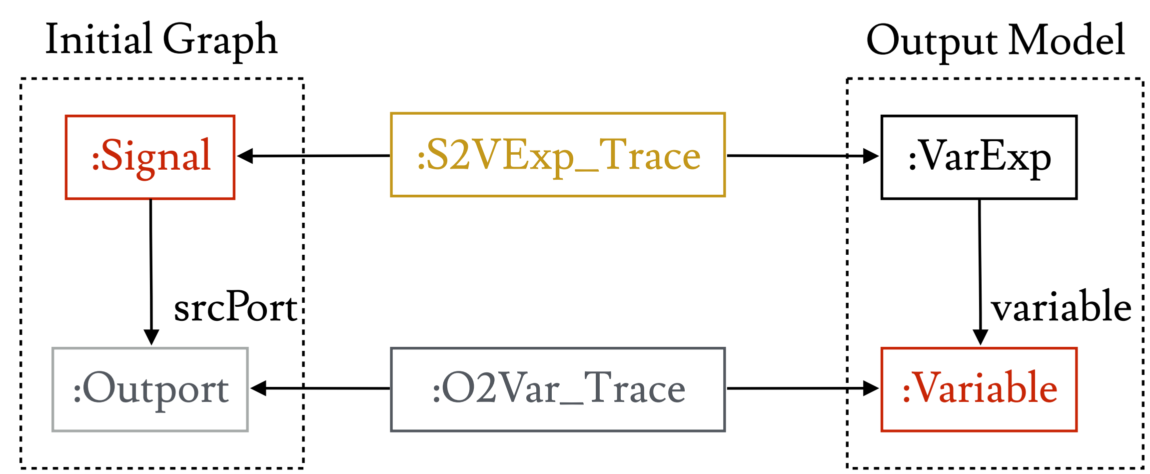




Example — ATL2AGT

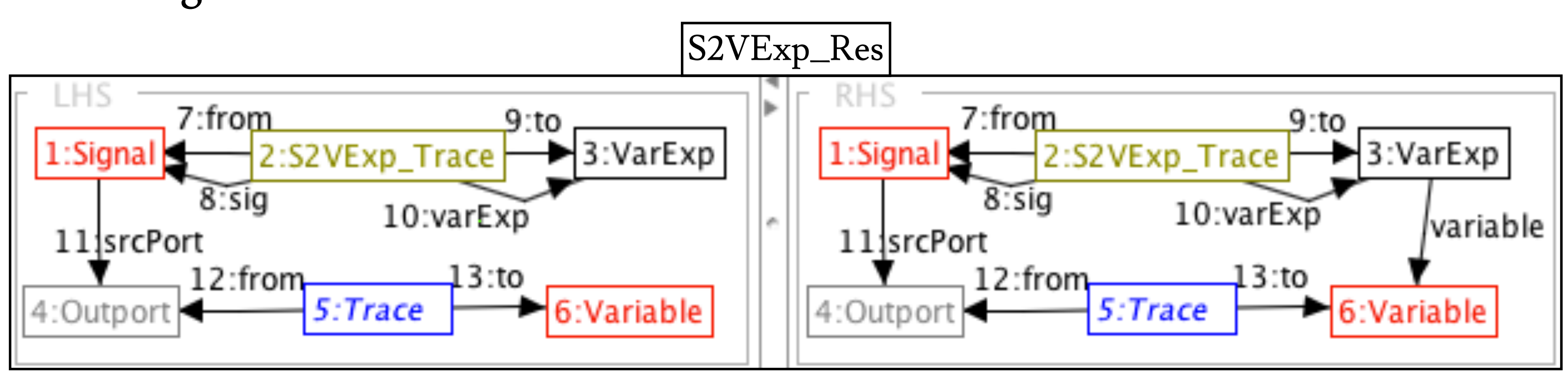
```
rule S2VExp { from sig    : SMM!Signal
              to   varExp : CMM!VarExp (variable <- sig.srcPort) }    -- Resolve

rule O2Var { from oport : SMM!Outport
             to   var   : CMM!Variable }
```





Instantiation

Resolving



Translation of Transformations – ATL2AGT

Challenge

- Semantic gap between Declarative Model Transformation and AGT
- ATL semantics – create a **new** output model
  - Rules executed simultaneously
  - A rule can **resolve** the output of other rules using implicit input to output tracing
- AGT semantics – in-place graph rewriting
  - Rules applied sequentially and atomically
  - No resolve mechanism

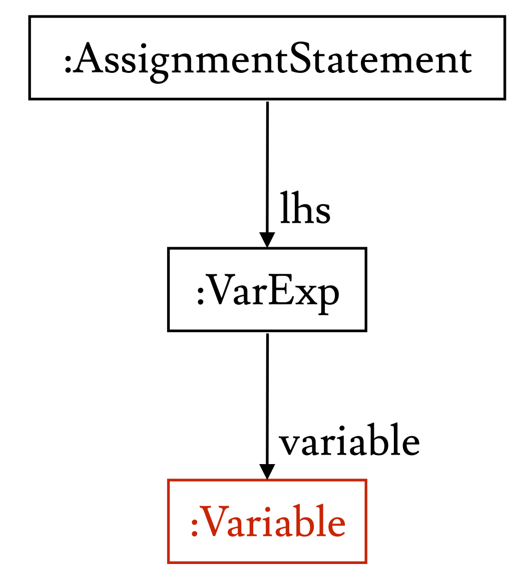
Contribution

- Model an ATL transformation as a 2-phase rewriting of the input graph
  - Instantiation phase —> Resolving phase
  - Explicit **Trace nodes**
- Prototype implementation using Henshin framework (Eclipse-based)
  - Limited to structural aspects

Advancement of Constraints – Post2Pre

Challenge

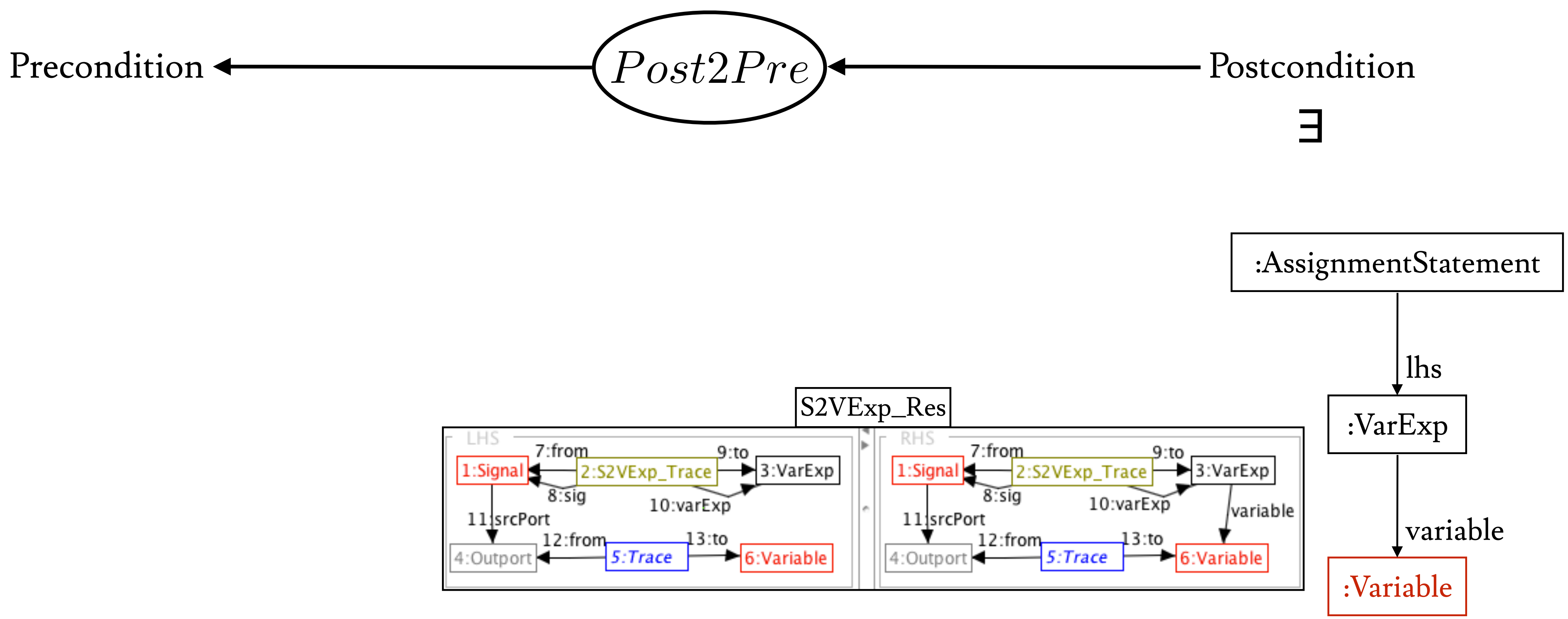
- The theoretical construction in AGT consists in
  1. Enumerate all overlaps of a rule with the postcondition
  2. **Unroll** the effects of applying the rule (reverse application)



Advancement of Constraints – Post2Pre

Challenge

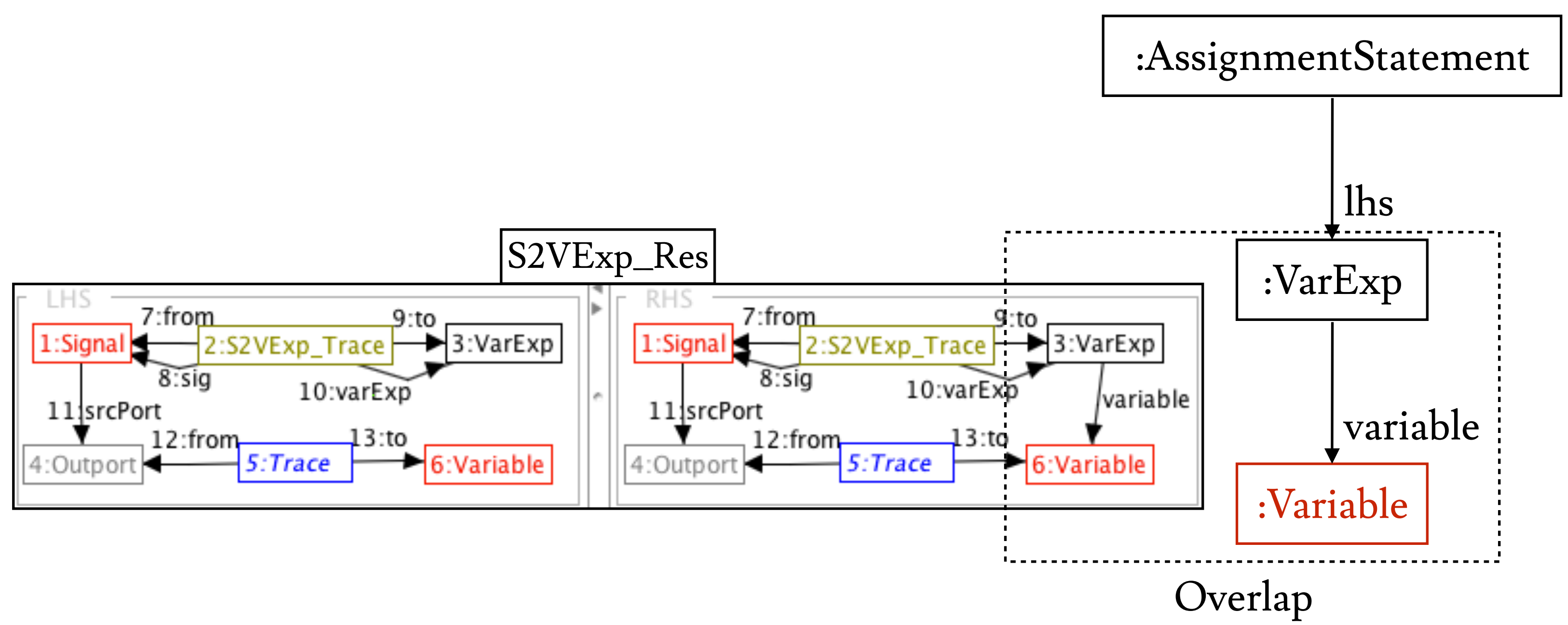
- The theoretical construction in AGT consists in
  1. Enumerate all **overlaps** of a rule with the postcondition
  2. **Unroll** the effects of applying the rule (reverse application)



Advancement of Constraints – Post2Pre

Challenge

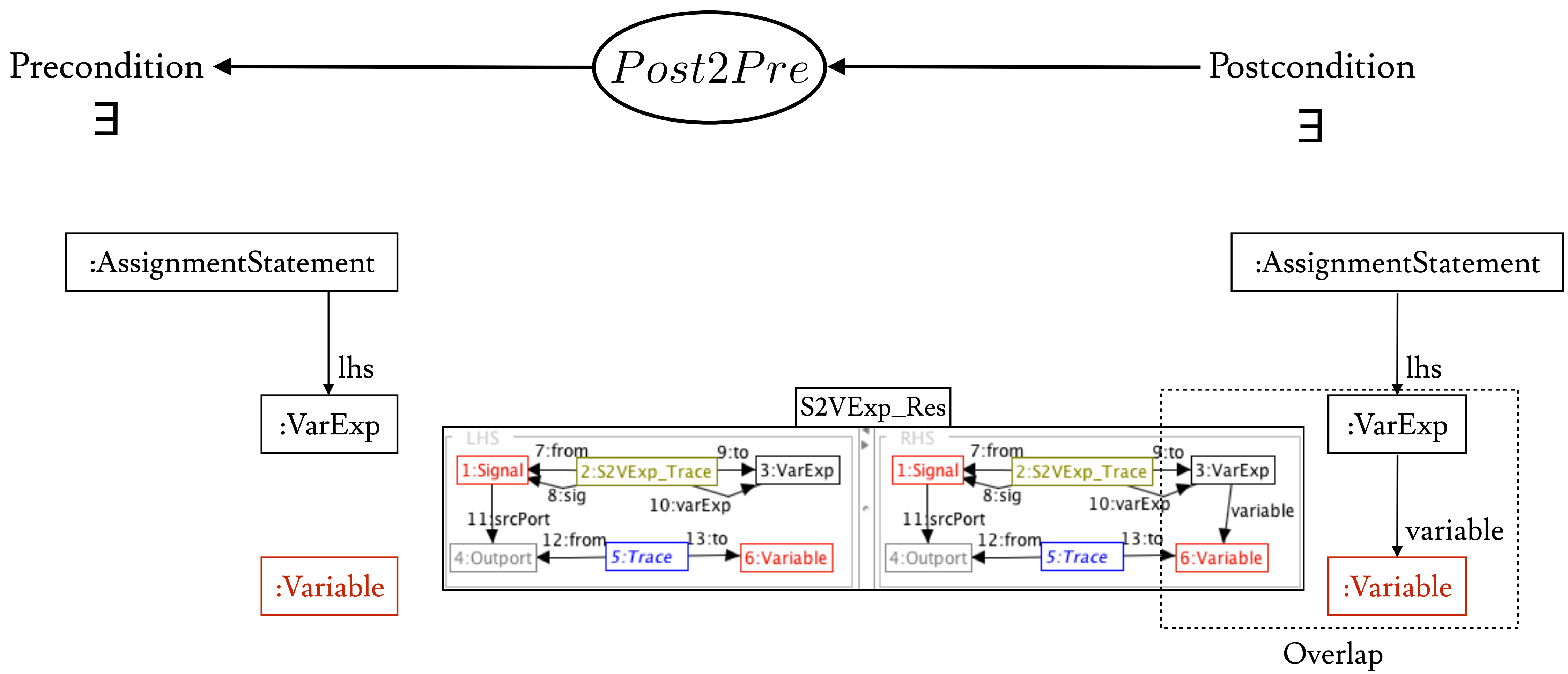
- The theoretical construction in AGT consists in
  1. Enumerate all **overlaps** of a rule with the postcondition
  2. **Unroll** the effects of applying the rule (reverse application)



Advancement of Constraints – Post2Pre

Challenge

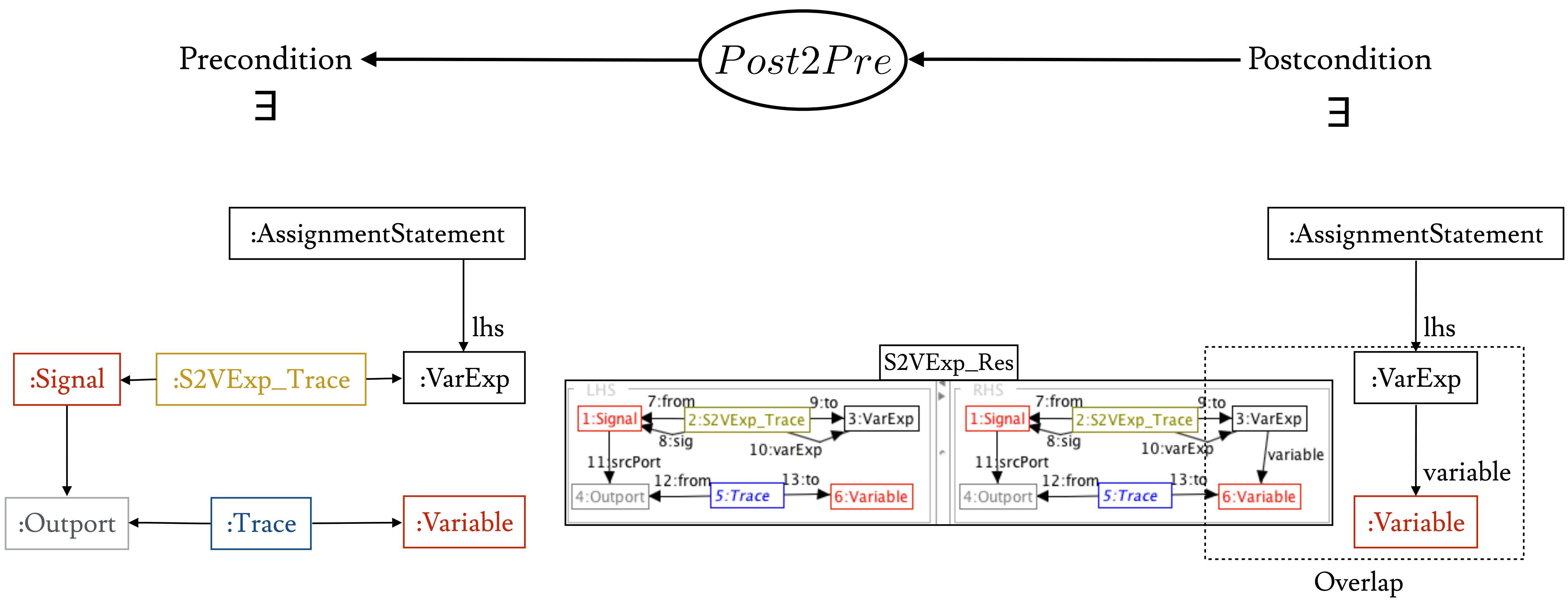
- The theoretical construction in AGT consists in
    1. Enumerate all overlaps of a rule with the postcondition
    2. **Unroll** the effects of applying the rule (reverse application)



Advancement of Constraints – Post2Pre

Challenge

- The theoretical construction in AGT consists in
  1. Enumerate all overlaps of a rule with the postcondition
  2. **Unroll** the effects of applying the rule (reverse application)



Advancement of Constraints – Post2Pre

Challenge

- The theoretical construction in AGT consists in
  1. Enumerate all **overlaps** of a rule with the postcondition
  2. **Unroll** the effects of applying the rule (reverse application) and add the application condition
- Theoretical weakest precondition is **infinite** because of infinite rule iteration

Contribution

- Bound the number of iteration of rule iterations
  - Obtain a **sufficient** precondition instead of the **weakest**
  - Similar to size bounds in CSP-based approaches
- Eliminate overlaps based on knowledge of ATL semantics
- Prototype implementation in AGG framework for basic constraints ∃(graph)
- Validation of Post2Pre and ATL2AGT on a simplified code generation step : 3 ATL rules

  ☑ one-step advancement of simple test requirements

Related Work

- Test suite quality for model transformation chains
  E. Bauer, J. Küster, and G. Engels
  - [x] Cover unit test requirements with integration tests
  - [x] Detect unsatisfied unit test requirements
  - [ ] No support for producing new test models

- Synthesis of OCL pre-conditions for graph transformation rules
  J. Cabot, R. Clariso, E. Guerra, and J. de Lara
  - [x] Construct OCL preconditions from OCL postconditions
  - [ ] No formal proof of completeness and correctness

Future Work

- OCL2NGC — Translation of Test Requirements
  - Very active topic in the community
    - ICGT, July 2014 T. Arendt et al. , "From Core OCL Invariants to Nested Graph Constraints"
    - MODELS, September 2014 G. Bergmann, "Translating OCL to Graph Patterns"
- ATL2AGT
  - Translate OCL embedded in ATL to AGT
  - Realistic ATL transformations
- Post2Pre
  - Handle complete Nested Graph Constraints
  - Investigate performance of overlapping algorithm

Thank you!

Credits

Slide 5: Alert by Juergen Bauer from The Noun Project