Piero Fraternali
Massimo Tisi

POLITECNICO DI MILANO

Mutation Analysis for Model Transformations in ATL

Test set generation for model transformations

- Several existing criteria for test generation/coverage
  - Non-native approaches:
    - Black-box:
      - Object oriented
      - Grammar-based
    - White-box:
      - Code coverage
  - Native approaches:
    - Black-box:
      - Fleaury
      - Fraternali, Tisi
    - White-box:
      - McQuillan,Power (a few minutes ago!)

Mutation analysis for model transformations

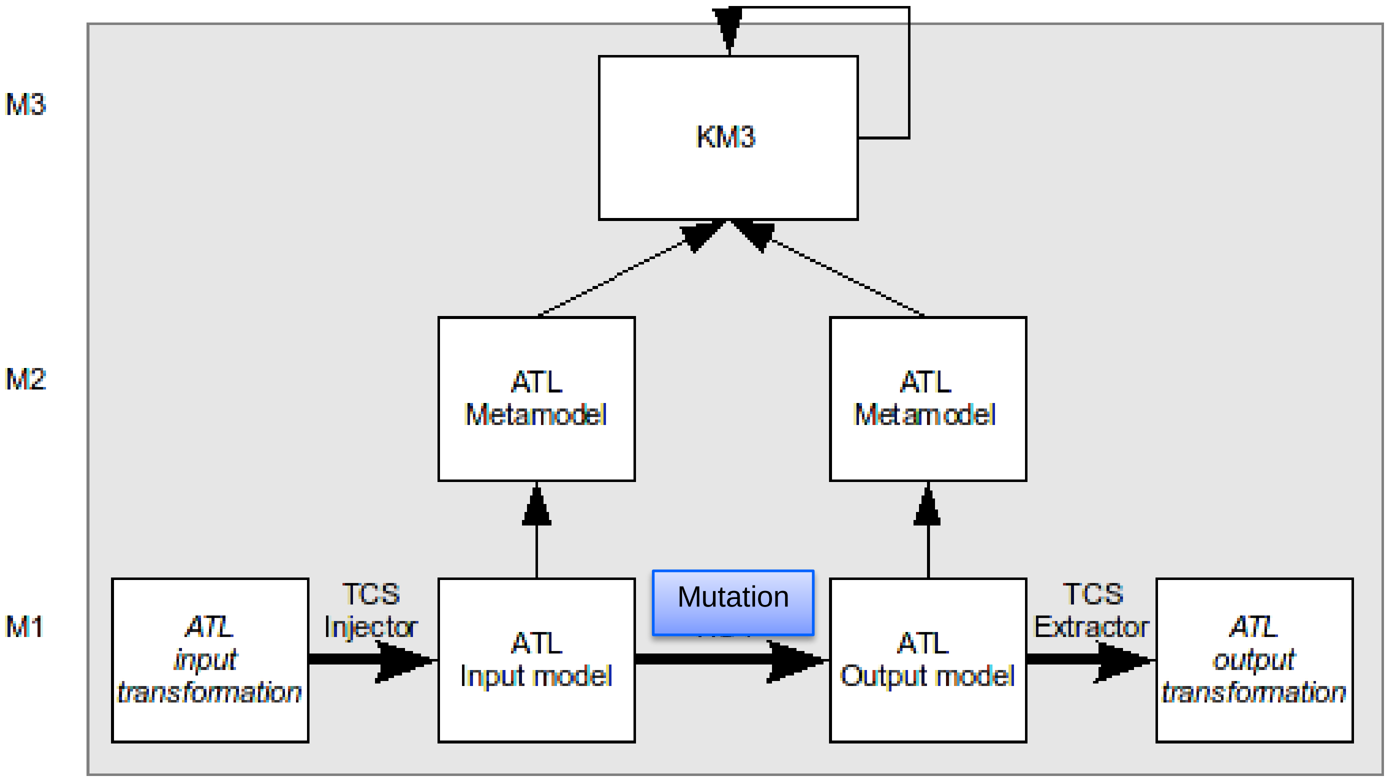
- Given:
  - a test set (manually or automatically built)
  - a (supposedly) correct transformation
- Systematic injection of errors in a transformation
  - creation of mutant transformations
- Estimate of the quality of a test set
  - based on the rate of faulty programs it detects
  - “fault revealing power” of the test set
- Estimate of the quality of a test generation or test coverage criteria

Our proposal

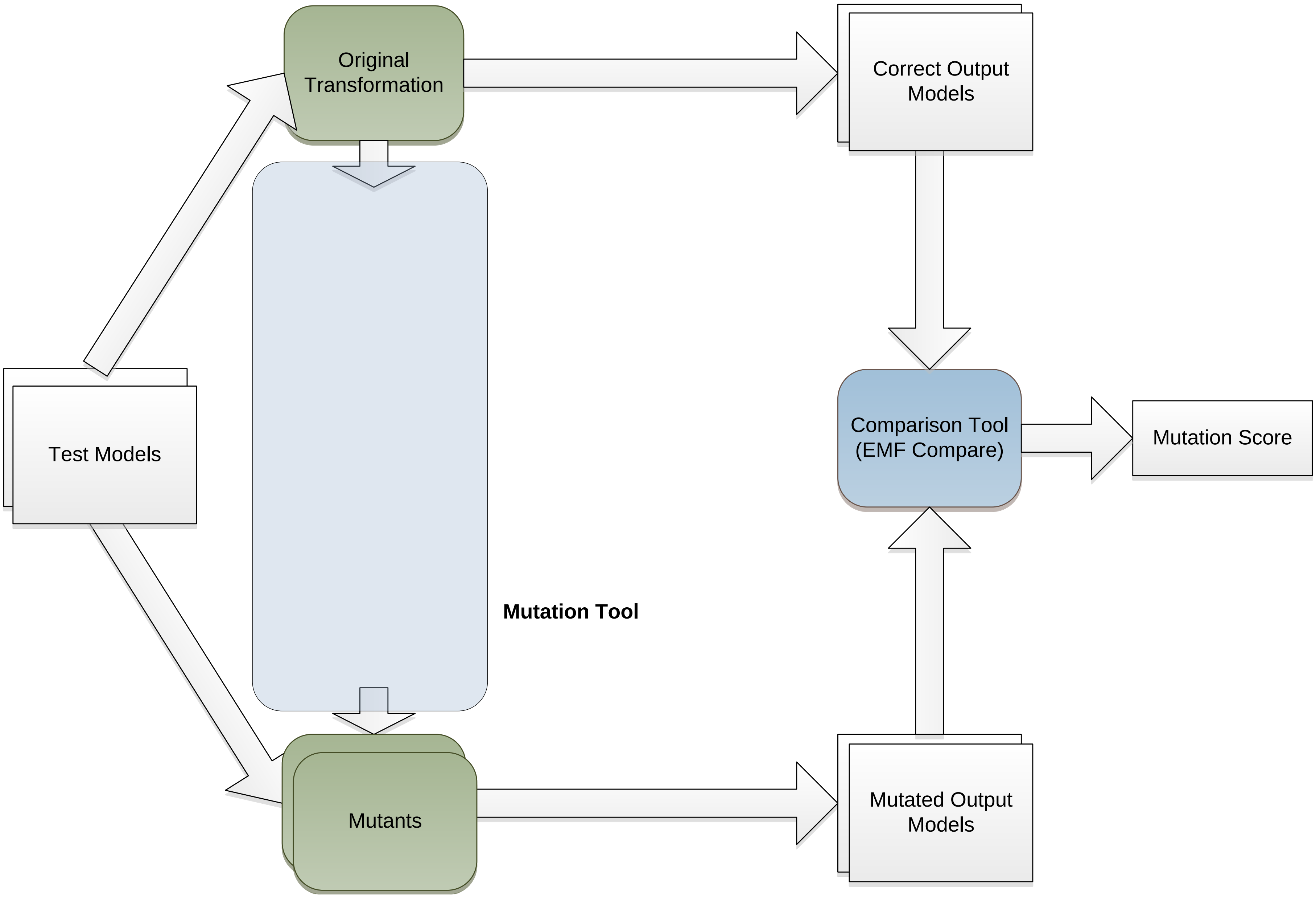
- Framework for mutation analysis (in a model-driven way)
  - Fault injection by means of HOTs
    - Two levels of HOTs for easier specification of the fault
  - Implementation in Java and ATL
    - But support for different transformation languages
- ATL implementation for well-known mutation operators
  - 11 mutation operators (Mottu, Baudry, Le Traon, 2006)
    - Navigation
      - Relation to same class change
      - Relation sequence modification with deletion, ...
    - Filtering
      - Collection filtering change with addition /perturbation / deletion
    - Output model creation
      - Class compatible creation replacement

Higher Order Transformations for mutations

- Mutation in ATL by a HOT in refining mode:



Mutation analysis framework



Example

- Collection filtering change with deletion
- 'Forgetting' filters in input patterns:

```
rule RemoveFilter {
    from
        m : ATL!InPattern (
            not m.filter.oclIsUndefined()
        )
    to
        pp : ATL!InPattern (
            elements <- m.elements,
            rule <- m.rule,
            location <- m.location,
            commentsAfter <- m.commentsAfter,
            commentsBefore <- m.commentsBefore
        )
}
```

```
rule INDEXUNIT {
 from
  element : XML!Tag (name = 'INDEXUNIT')
 to
  result : DSLMM!INDEXUNIT (
   [...]
  )
}
```

↓

```
rule INDEXUNIT {
 from
  element : XML!Tag
 to
  result : DSLMM!INDEXUNIT (
   [...]
  )
}
```

Multiple application points

- Problem
  - the transformation engine would apply it to all the matches at once.

```
rule INDEXUNIT {
 from
  element : XML!Tag (name = 'INDEXUNIT')
 to
  result : DSLMM!INDEXUNIT (
   [...]
  )
}
rule DATAUNIT {
 from
  element : XML!Tag (name = 'DATAUNIT')
 to
  result : DSLMM!DATAUNIT (
   [...]
  )
}
```

→

```
rule INDEXUNIT {
 from
  element : XML!Tag
 to
  result : DSLMM!INDEXUNIT (
   [...]
  )
}

rule DATAUNIT {
 from
  element : XML!Tag
 to
  result : DSLMM!DATAUNIT (
   [...]
  )
}
```

Multiple application points

- What we want:

```
rule INDEXUNIT {
 from
  element : XML!Tag (name = 'INDEXUNIT')
 to
  result : DSLMM!INDEXUNIT (
   [...]
  )
}
rule DATAUNIT {
 from
  element : XML!Tag (name = 'DATAUNIT')
 to
  result : DSLMM!DATAUNIT (
   [...]
  )
}
```

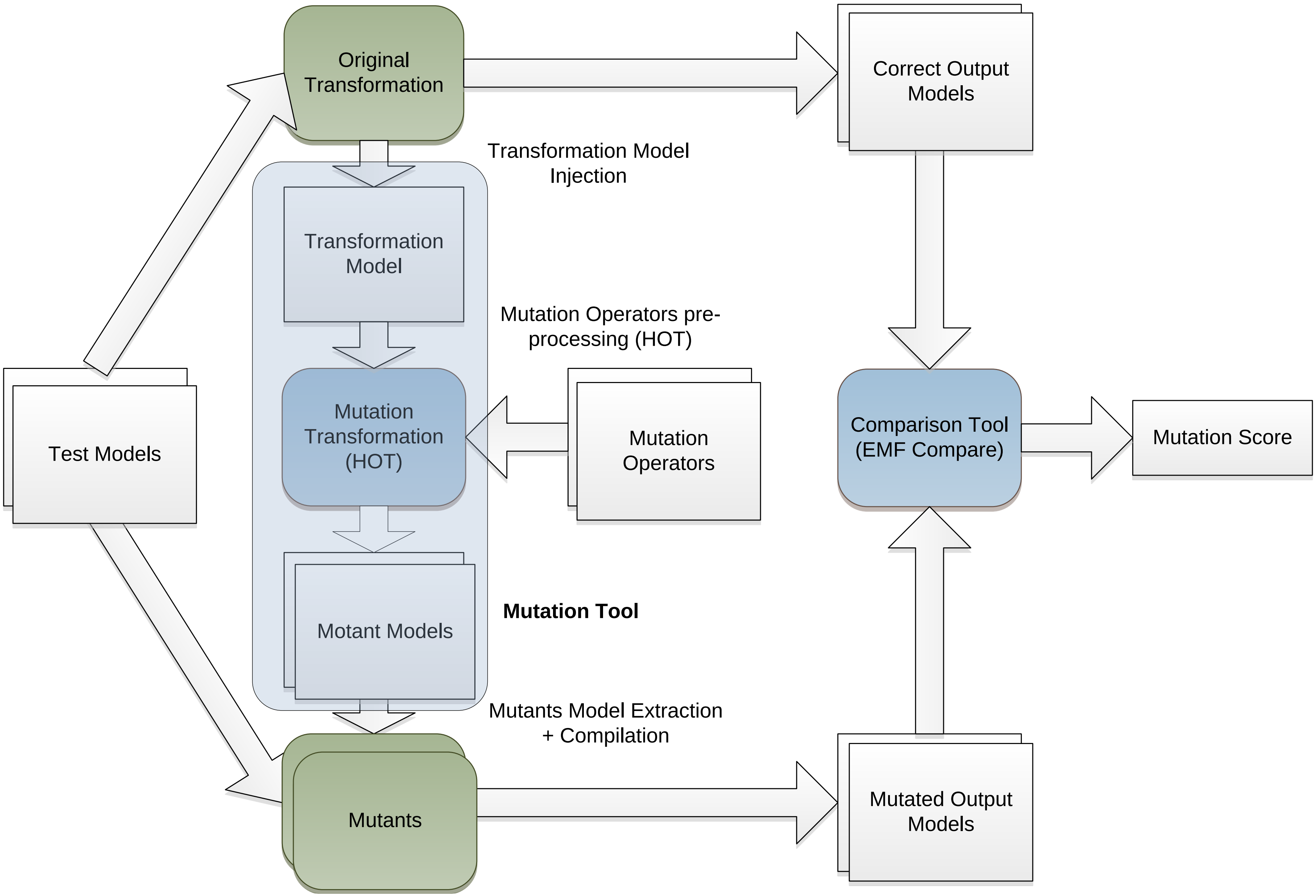
```
rule INDEXUNIT {
 from
  element : XML!Tag
 to
  result : DSLMM!INDEXUNIT (
   [...]
  )
}
rule DATAUNIT {
 from
  element : XML!Tag (name = 'DATAUNIT')
 to
  result : DSLMM!DATAUNIT (
   [...]
  )
}
```

```
rule INDEXUNIT {
 from
  element : XML!Tag (name = 'INDEXUNIT')
 to
  result : DSLMM!INDEXUNIT (
   [...]
  )
}
rule DATAUNIT {
 from
  element : XML!Tag
 to
  result : DSLMM!DATAUNIT (
   [...]
  )
}
```

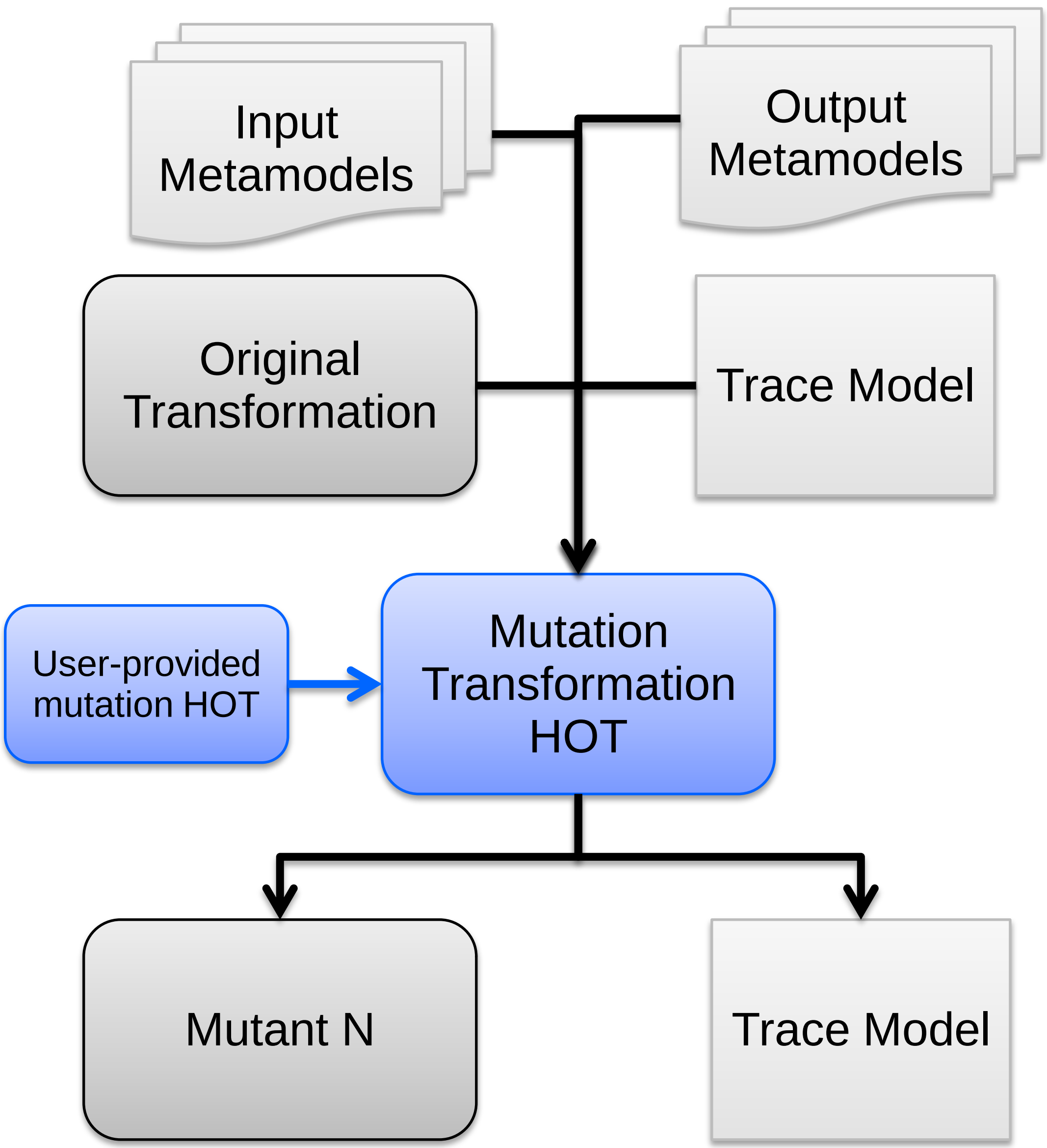
Higher Order Transformations for Mutations

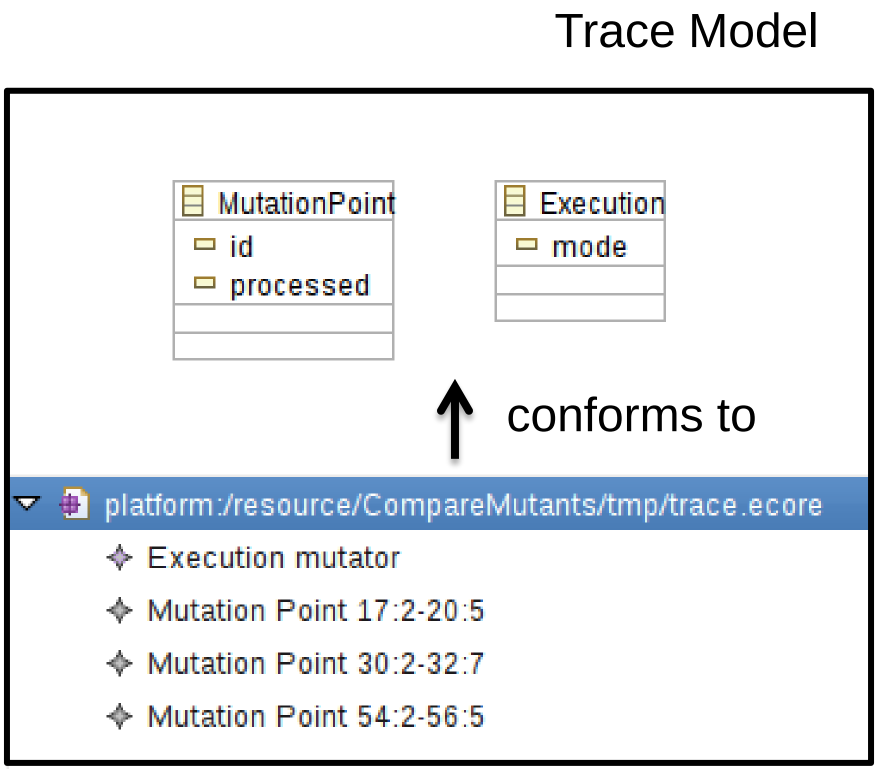
- Our solution should be:
  - Model-driven
  - Transparent for the designer of the mutation
  - Minimal computational cost
    - Apart from running N times a transformation
  - No change to the standard transformation engine
- Our proposal: pre-processing the mutation operator
  - Transforming an HOT
  - Second order (or third order?) HOT

Mutation analysis framework



Higher Order Transformations for Mutations





Higher Order Transformations for Mutations

Two rules are generated from the user provided HOT.
Helpers control the execution (with **constant cost**)

A first rule to identify and record matching points in the trace model at the first run:

```
rule RemoveFilter {
    from
        m : ATL!InPattern (
            not m.filter.oclIsUndefined()
        )
    to
        pp : ATL!InPattern (
            elements <- m.elements,
            rule <- m.rule,
            location <- m.location,
            commentsAfter <- m.commentsAfter,
            commentsBefore <- m.commentsBefore
        )
}
```

→

```
rule NotRemoveFilter {
    from
        m : ATL!InPattern (
            thisModule.isNotNextMatch(m.location)
            and ( not m.filter.oclIsUndefined() )
        )
    to
        m1 : ATL!InPattern (
            elements <- m.elements,
            rule <- m.rule,
            filter <- m.filter,
            location <- m.location,
            commentsAfter <- m.commentsAfter,
            commentsBefore <- m.commentsBefore
        )
    do {
        thisModule.notNextMatchingStep(m.location);
    }
}
```

True at first run

Original LHS

No changes

Recording of the matching point

Higher Order Transformations for Mutations

A second rule to generate single mutations at the next runs:

```
rule RemoveFilter {
    from
        m : ATL!InPattern (
            not m.filter.oclIsUndefined()
        )
    to
        pp : ATL!InPattern (
            elements <- m.elements,
            rule <- m.rule,
            location <- m.location,
            commentsAfter <- m.commentsAfter,
            commentsBefore <- m.commentsBefore
        )
}
```

→

```
rule RemoveFilter {
    from
        m : ATL!InPattern (
            thisModule.isNextMatch(m.location)
        )
    to
        m1 : ATL!InPattern (
            elements <- m.elements,
            rule <- m.rule,
            location <- m.location,
            commentsAfter <- m.commentsAfter,
            commentsBefore <- m.commentsBefore
        )

    do {
        thisModule.nextMatchingStep(m.location);
    }
}
```

True once per run

Original RHS

Trace update

Future Work

- Equivalent mutants
  - What happens if the mutant is ‘correct’?
    - Example: if the filter was superflous
- A validated fault model for transformation languages:
  - Do transformation languages have a common set of mutation operators ?
  - Is there a set of mutation operators for transformation languages that are inherently language specific ?
  - Howto:
    - Develop mutation operators for transformation languages
    - Compare the mutation scores relative to different mutation operators applied to the same test set
    - Identify equivalent mutation operators among different transformation languages

Thanks