Fully Verifying Transformation Contracts for Declarative ATL

Bentley James Oakes, Javier Troya, Levi Lúcio, Manuel Wimmer

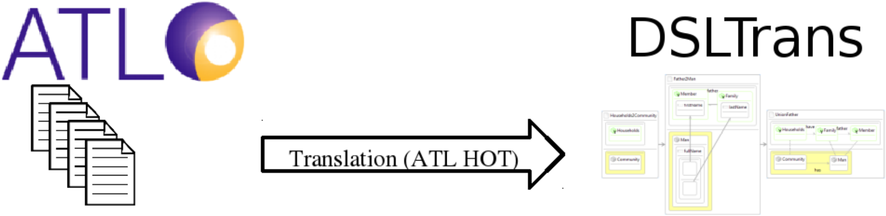
McGill University, Canada
Vienna University of Technology, Austria

Motivation

- Model transformations are at the heart of model-based engineering
- Atlas Transformation Language (ATL) is increasingly used in industry
  - Example: Generating code to/from models
- Want to verify correctness for ATL transformation specifications
  - Verify visual contracts
  - Input independence - verification for all input models
  - Examine combinations of transformation rules

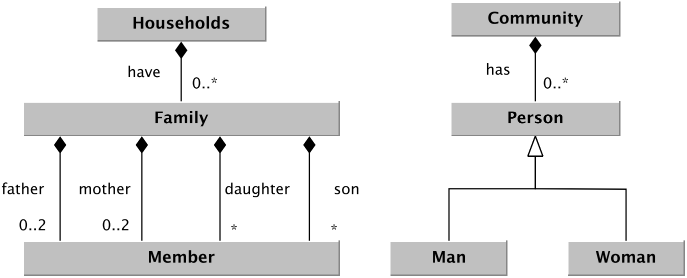
Overview

- Translating ATL transformation into DSLTrans language
- Verify visual contracts on DSLTrans



- Performed through a higher-order transformation
  - Specified in ATL

Transformation Metamodels



- Transform *Members* to *Men* and *Women*
- NB: Metamodels are not representative of today's society!

ATL Transformation

```
module Families2Persons;
create OUT : Persons from IN : Families;












rule Father2Man { -- R2
 from
  mem : Families!Member, fam : Families!Family
        (fam.father=mem)
 to
  m : Persons!Man (
   fullName <- mem.firstName + fam.lastName --B2
  )}
```

ATL Transformation

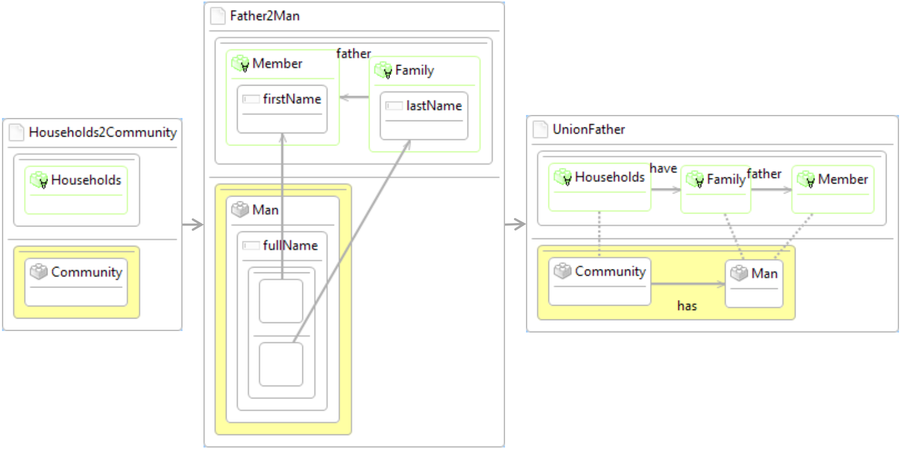
```
module Families2Persons;
create OUT : Persons from IN : Families;

rule Households2Community { -- R1
 from
  hh: Families!Households
 to
  c : Persons!Community (
   has <- hh.have->collect(f | thisModule.
    resolveTemp(Tuple{mem=f.father,fam=f}, 'm')), --B11
   has <- hh.have->collect(f | thisModule.
    resolveTemp(Tuple{mem=f.mother,fam=f}, 'w')) --B12
 )}

rule Father2Man { -- R2
 from
  mem : Families!Member, fam : Families!Family
        (fam.father=mem)
 to
  m : Persons!Man (
   fullName <- mem.firstName + fam.lastName --B2
  )}
```

- Implicit resolution mechanism of ATL
- Through collect operation

DSLTrans Transformation

- Visual language for model transformations
- Graph-based, contains rules arranged in layers
- Out-place so no rewriting performed, only production
  - Suited for ‘translation’ transformations
- All DSLTrans computations are terminating and confluent
- Unbounded loops during execution are not allowed

DSLTrans



- Rules arranged in layers
- Match graph on top of rules
- Apply graph on bottom
  - Produced when match graph is found

Mapping - Part One

- Higher-order transformation written in ATL
- Creates a DSLTrans transformation from declarative ATL
  - Informal testing: less than 20 seconds
- Available on our website: `http://msdl.cs.mcgill.ca/people/levi/files/MODELS2015`

Mapping - Part Two

TABLE I
FEATURES OF DECLARATIVE ATL CONSIDERED

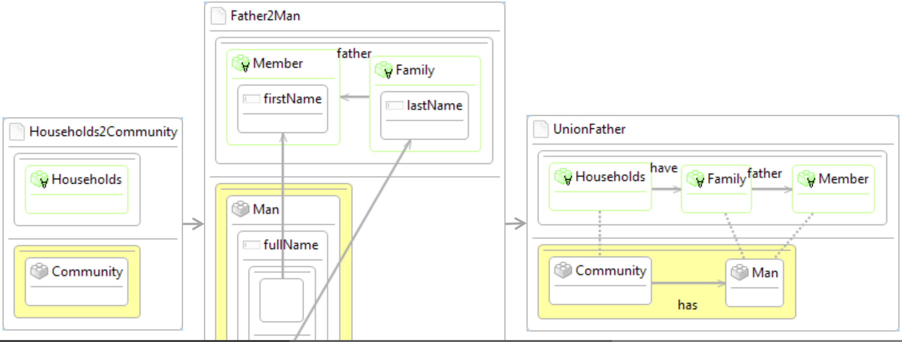
Matched Rules	✓	Filters	✓
Lazy Rules	✓	OCL Expressions	✓
Several Bindings	✓	Helpers	×
Several *InPatternElements*	✓	Conditions	×
Several *OutPatternElements*	✓	Using Block	×

- Covers declarative ATL
- Transformation can be rewritten to avoid missing features

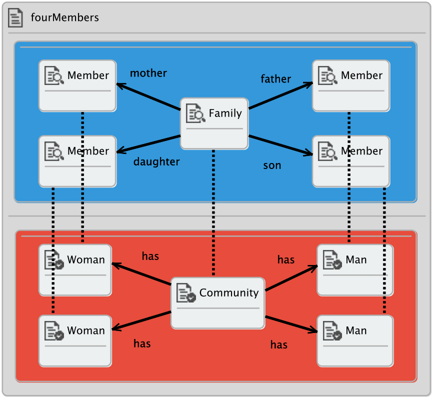
Mapping - Part Three

- Two steps for higher-order transformation
- First, each from/to part of an ATL rule is transformed into match/apply graphs in DSLTrans
- Attributes will also be set in these rules
- Second, DSLTrans rules are produced for any bindings in the ATL rule

```
rule Households2Community { -- R1
 from
  hh: Families!Households
 to
  c : Persons!Community (
   has <- hh.have->collect(f | thisModule.
    resolveTemp(Tuple{mem=f.father,fam=f}, 'm')), --B11
   has <- hh.have->collect(f | thisModule.
    resolveTemp(Tuple{mem=f.mother,fam=f}, 'w')) --B12
 )}
```

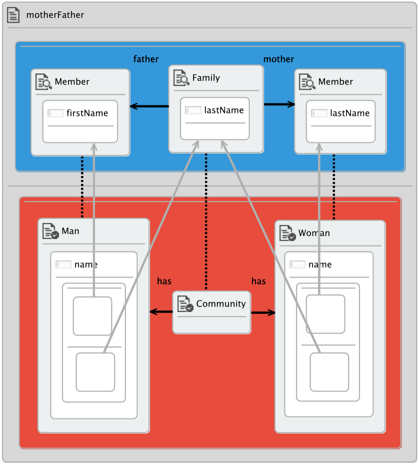


Contracts



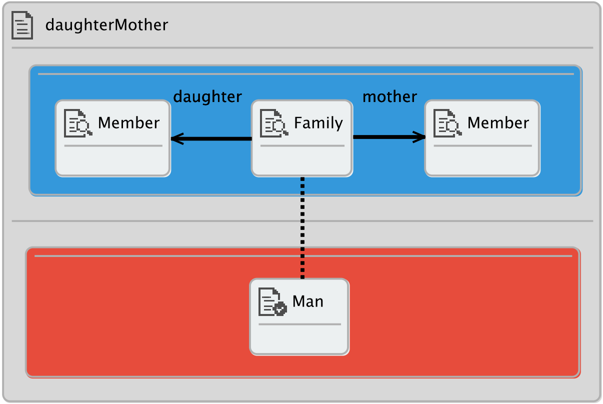
- If blue graph is in input model, then red graph is in output model
- Objective: Prove for all input models/transformation executions
- *A family with a father, mother, son, daughter should always produce two males and two females in the target community*

Contracts



- Reasoning about attributes of elements
- *Is the full name of the produced Person correctly created from the last name of the Family and the first name of the Member?*

Contracts



- A contract that will not hold
- *A family with a mother and a daughter will always produce a community with a man*

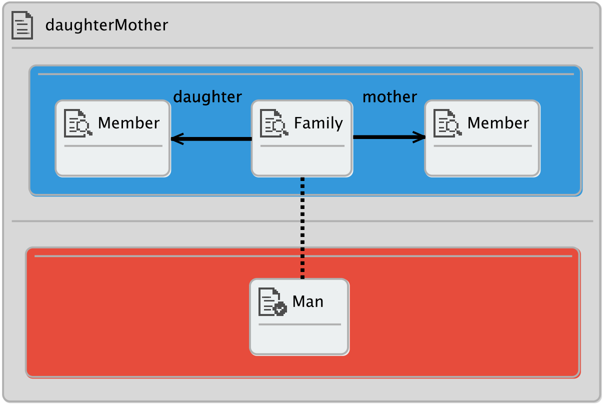
Contract Proving - Part One

- SyVOLT contract proving tool
- All possible executions of the transformation are symbolically constructed
  - Built as sets of rules called path conditions
    - No rules execute, only rule 1 executes, rule 1 and rule 2 both execute
    - Rule dependencies/combinations resolved
  - Final set of path conditions represents all possible transformation executions
- A contract holds for a transformation if it holds for all generated path conditions

---

L. Lúcio, B. Oakes, and H. Vangheluwe. A technique for symbolically verifying properties of graph-based model transformations. Tech. Report SOCS-TR-2014.1, McGill U, 2014.

Levi Lucio et al. SyVOLT: Full Model Transformation Verification Using Contracts



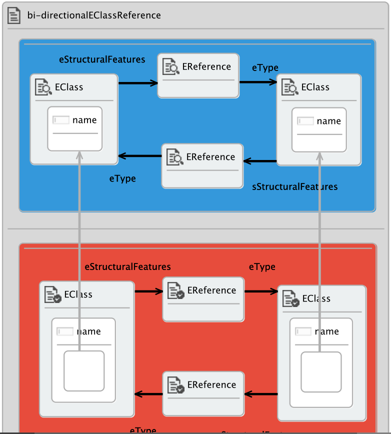
- *A family with a mother and a daughter will always produce a community with a man*
- Fails on path condition: 'HEmpty_HRoot_HMotherRule_HDaughterRule'

Experiments Conducted

- Applicability of the Technique
- Time and Memory Characteristics
- Reducing Contract Proving Time
- Higher-Order Transformation

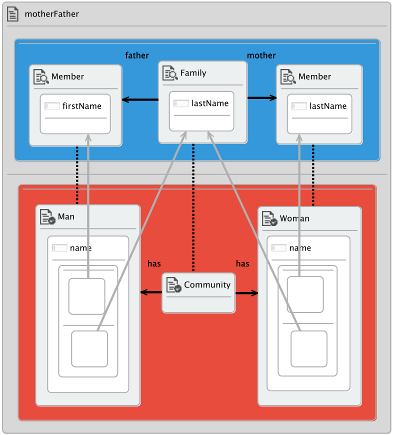
Applicability of the Technique - Part One

- Applied to multiple transformations from ATL zoo
  - Ranging in size from 5-15 ATL rules
  - Example below:
  - Ecore Copier transformation - 11 ATL rules, 24 DSLTrans rules
  - Copies Ecore elements in input model to output model

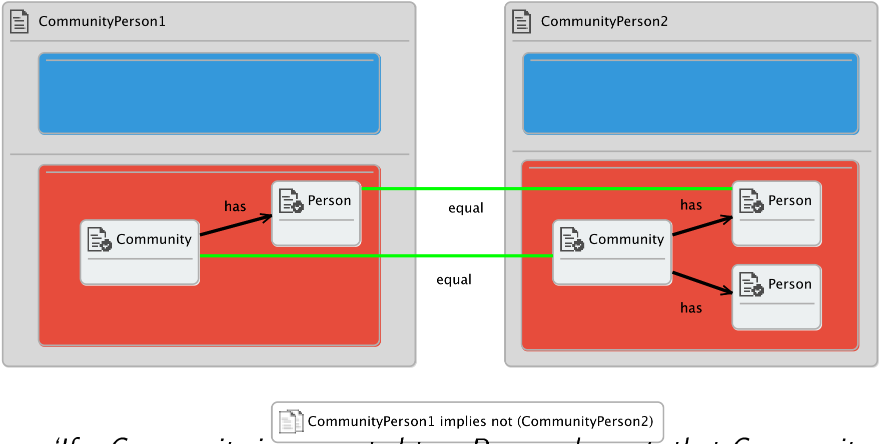


Applicability of the Technique - Part Two

- Technique works with attributes on elements
  - Proving names of people correctly created



Applicability of the Technique - Part Three



- *'If a Community is connected to a Person element, that Community is connected to one and only one Person element'*
- Selim, Gehan. Formal Verification of Graph-Based Model Transformations. PhD Diss. Queen's University, 2015.

Time and Memory Characteristics

	ATL/ DSLTrans Rules	Path Conds. Gen.	Time (s)	Contracts Proved	Time (s)	Memory (MB)
Families-to-Person	5 / 9	52	1.54	4	31.45	45
ER-Copier	5 / 9	70	0.48	1	1.70	43
Ecore-Copier	11 / 24	57890	2894.44	1	1401.45	7800

- Feasible
  - Time - Ranging from 0.5 seconds to 48 minutes (on laptop)
  - Memory - 43 to 7800 MB RAM/disk usage
  - (Both measures have been improved in newer tool versions)

Reducing Contract Proving Time

	ATL/ DSLTrans Rules	Path Conds. Gen.	Time (s)	Contracts Proved	Time (s)	Memory (MB)
Sliced Transformation (Contract 1)	15 / 13	73	3.50	1	9.11	72
Sliced Transformation (Contract 2)	15 / 17	28	0.95	1	0.46	71

- Examined ATL transformation which is transformed into 63 DSLTrans rules
- To make feasible, need to slice transformation based on contract
- Procedure:
  - Find rules that create contract elements
  - Recursively create rule dependency tree
- Manually performed - slicing has since been automated

Higher-Order Transformation

- Question: Is a transformation produced by a HOT equivalent to a hand-built one?

	ATL/ DSLTrans Rules	Path Conds. Gen.	Time (s)	Contracts Proved	Time (s)	Memory (MB)
Industrial (from [18])	5 / 7	3	0.07	9	0.16	43
Industrial (from HOT)	5 / 9	3	0.17	9	0.26	48

- Note that number of rules/transformation shape not optimized
- But HOT produces roughly equivalent result

G. M. Selim, L. Lúcio, J. R. Cordy, J. Dingel, and B. J. Oakes. Specification and verification of graph-based model transformation properties. In Graph Transformation, pages 113–129. Springer, 2014.

Conclusion

- Developed higher-order transformation to transform ATL into DSLTrans
- Can verify visual contracts on DSLTrans transformations in feasible time
- Contracts verified on all transformation executions
- Future work
  - Integrate HOT into SyVOLT tool
  - Investigate contract-based transformation development
- Thank you for your time!

Fully Verifying Transformation Contracts for Declarative ATL

Bentley James Oakes, Javier Troya, Levi Lúcio, and Manuel Wimmer
McGill University, Canada
Vienna University of Technology, Austria

`http://msdl.cs.mcgill.ca/people/levi/files/MODELS2015`

Multiplicity Contract

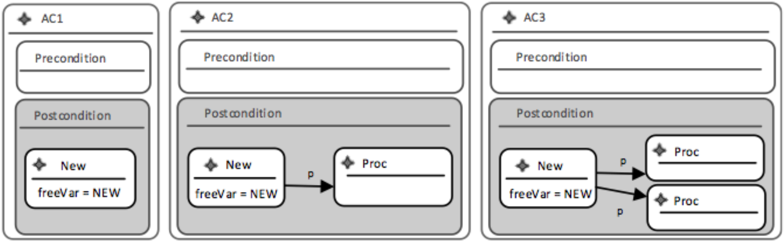


Figure C.1: *AtomicContracts AC1, AC2,* and *AC3* that are used to express *MM1* (Table 6.4) as $AC1 \Longrightarrow_{tc} (AC2 \wedge_{tc} \neg_{tc} AC3)$.

"Multiplicity Invariants ensure that the transformation does not produce an output that violates the multiplicities in the Kiltera metamodel"

Syntactic Invariant

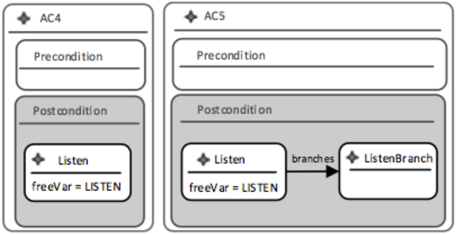


Figure C.5: *AtomicContracts AC4* and *AC5* that are used to express *MM5* (Table 6.4) as $AC4 \Longrightarrow_{tc} AC5$.

"Syntactic Invariants ensure that the generated Kiltera output model is well-formed with respect to Kiltera's syntax."

Pattern Contracts

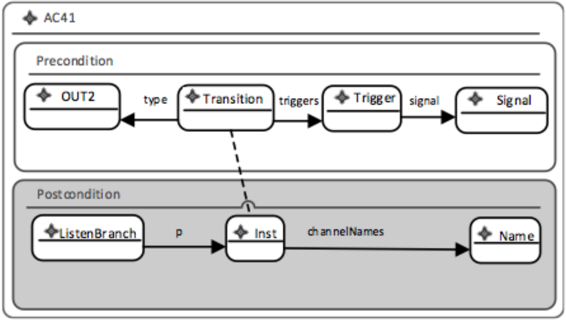


Figure C.16: *AtomicContract AC41* that is used to express *PP2* (Table 6.4).

"Pattern contracts require that if a certain pattern of elements exists in the input model, then a corresponding pattern of elements exists in the output model"