A Symbolic Execution-Based Approach to Model Transformation Verification using Structural Contracts

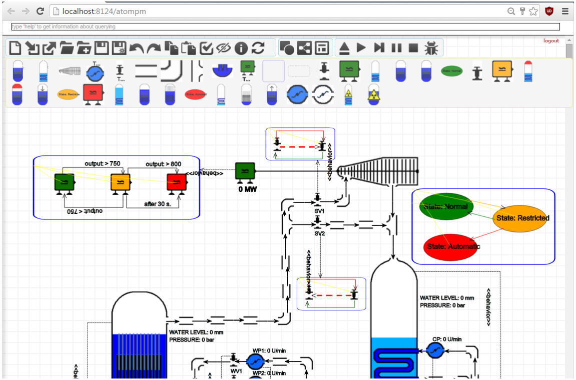
Bentley James Oakes

McGill University

bentley.oakes@mail.mcgill.ca

September 4, 2018

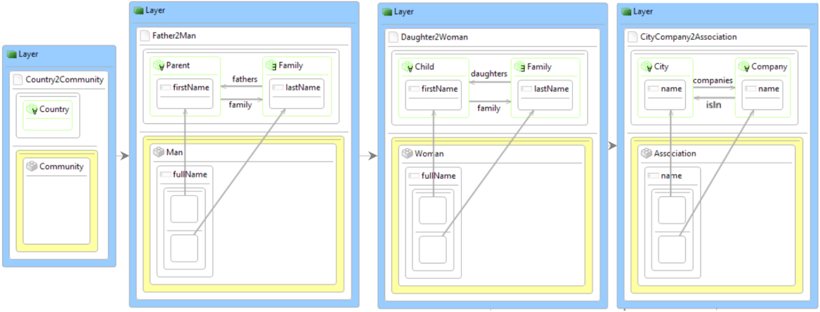
- Model-driven engineering is crucial for creating and understanding complex systems
- Goal: Build a model of a system in the most appropriate language(s) for multiple stakeholders



- Example: A model for simulation/code synthesis/safety analysis of a nuclear reactor (Van Mierlo 2017)
    - Concepts include tanks, pipes, water level, pressure
    - Human-readable, activities possible through model transformation

Model Transformations

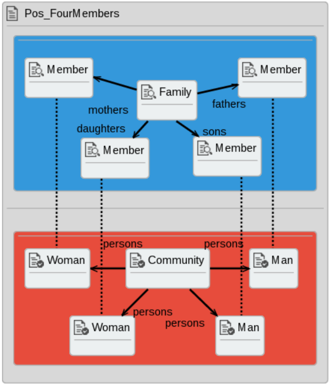
- Problem: Want to have a structured and understandable way to modify/translate/simulate models
- Solution: Use *model transformations*, which are composed of *rules* to manipulate model elements



- Rules are executed in a particular schedule, and *match* elements in the input model to *produce* elements in the output model

Issue: Difficult to understand interactions of rules from examining a transformation

- Problem: Want to understand how an input model to the transformation relates to the corresponding output model
- Solution: Verify pre-/post-condition patterns (*structural contracts*) which guarantee relations and traceability between elements if the contract is *satisfied*



"A *Family* with a *father*, *mother*, *son* and *daughter* should always produce two *Man* and two *Woman* elements connected to a *Community*."

- Result: Combinations of rules where the contract is *satisfied*, and combinations where the contract is *not satisfied*
- Allows the user to better understand the behaviour of the transformation

Research Questions

Five research questions answered by the thesis and this presentation:

Formalization of DSLTrans transformation language

- RQ1) How can the DSLTrans language be precisely formalized?

Transformation verification using contracts

- RQ2) How can the infinite execution possibilities of a DSLTrans transformation be represented in an explicit finite set?
- RQ3)
  - a) How can contracts be proved to be satisfied or non-satisfied on these representations?
  - b) When not satisfied, how do the counter-examples relate to the transformation?

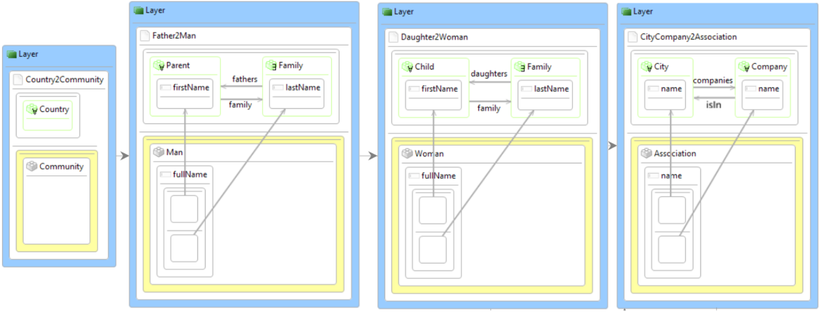
Development of the SyVOLT proving tool

- RQ4) What is the design and work-flow of a contract verification tool?
- RQ5) What are techniques for improving the scalability of the verification tool?

Outline

DSLTrans Overview

- DSLTrans transformation language was conceived by Barroca *et al.* (2011), so not a contribution of the thesis
- Intent was to create a language of limited expressiveness, so that by construction each transformation *terminates* and is *confluent* (rules cannot conflict)

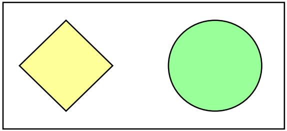


- Rules in DSLTrans are scheduled in *layers*. Rules are fully applied in one layer before moving on
    - Layered nature crucial for understandability and analysability
- Rules provide *traceability links*, to record how elements were built

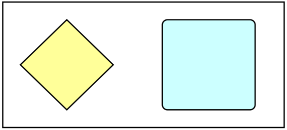
- Problem: Not all constructs of DSLTrans were formally specified
  - Behaviour defined by implementation
- Solution: Describe semantics of DSLTrans in a widely-used model transformation formalization
- Double-pushout approach answers RQ1) *How can the DSLTrans language be precisely formalized?*
- Reasons for selecting the double-pushout approach:
  - Provides elegant (and most appropriate) explanation
  - Aligns semantics with other transformation languages
  - Provides rigour for DSLTrans usability and analysability

- The double-pushout approach divides the application of a rule into the *matching* and *rewriting* stages
- Consider a rule which:
  - Matches on a model with a diamond and a circle
  - Replaces the circle with a rounded square

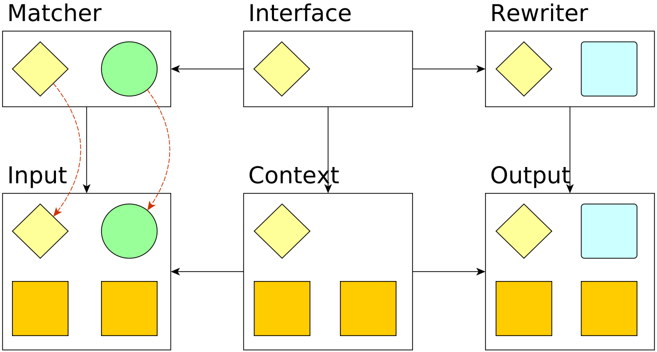
Pre-condition/Matcher / LHS



Post-condition/Rewriter / RHS

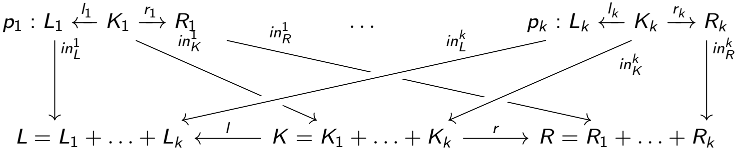


Double-Pushout Approach



- Arrows are *morphisms* between components, providing mappings of nodes and edges
- Element creation is performed through matching and the union operator, termed *push-outs*

- Double-pushout approach allows the creation of elegant *mass productions*
- Technique: Combine the matchers and rewriters of multiple rules



- Allows for all rules in a layer to be applied at once, ensuring they cannot interfere with each other

Formalization Contributions

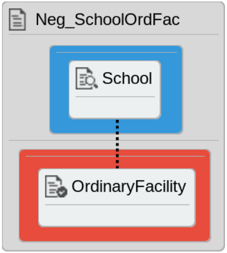
- Precise semantics provided for all DSLTrans constructs in the double-pushout approach
- Examination of termination and confluence properties

Outline

Contract Proving Results

- Verification results indicate which combinations of rules (*path conditions*) *satisfy* or *do not satisfy* each contract
- Allows the user to better understand transformation behaviour

Contract:



"A *School* will always produce an *OrdinaryFacility*"

- Reality: Contract should fail, as a rule exists in the transformation such that *SpecialFacilities* can also be produced

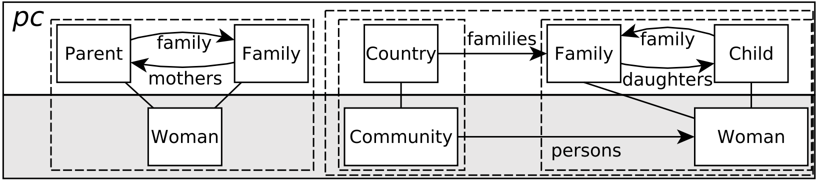
Verification results:

```
a) Name: Neg_SchoolOrdFac
    Num Succeeded Path Conditions: 6
    Num Failed Path Conditions: 3

b) Explaining contract result:
    Good rules: (Rules in success set and not failure set)
        dfacilities...OrdinaryFacilityPerson
    Bad rules: (Rules common to all in failure set)
        dfacilities...SpecialFacilityPerson
```

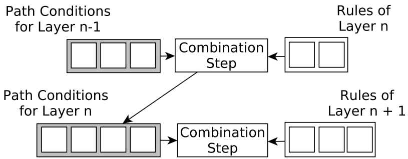
- Answers RQ3 b) When not satisfied, how do the counter-examples relate to the transformation?

- Contract proof is performed on *path conditions*, which represent valid combinations of rule applications
  - Record element presence when rules apply
- First step is to build the path conditions, then the second step is to match the contract onto the path conditions

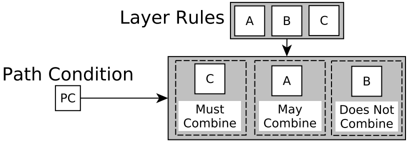


- Example: This path condition represents the application of four rules, where each rule has applied at least once
- Through a formalized *abstraction relation*, this path condition represents an infinite set of transformation executions
  - Abstracts over rule application multiplicity and element overlap
- A set of path conditions answers RQ2: *How can the infinite execution possibilities of a DSLTrans transformation be represented in an explicit finite set?*

- Path conditions are produced through a *symbolic execution* of the transformation's rules, layer-by-layer

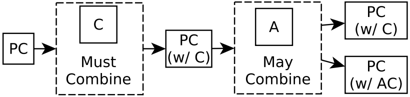


Rules are classified depending on combination with the path condition



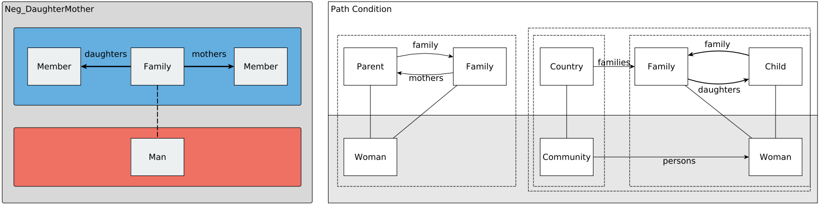
Rule Classification

Combining the rules and path condition creates new path conditions



- Final set of path conditions represents all valid transformation executions

- After path condition generation, the elements in the contracts are matched against the path conditions to determine contract satisfaction
- Answers RQ3: a) How can contracts be proven to be sat
- Issue: In path conditions, unknown whether rule elements overlap or not

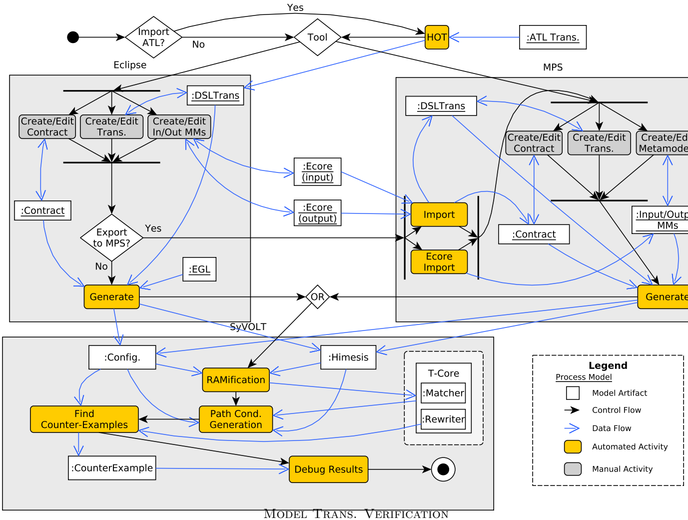


- Major contribution is the "split" morphism for matching contracts and path conditions
  - Allows one contract element to match over multiple path condition elements

Contract Proving Contributions

- Detailed procedure for building the set of path conditions
- Technique for matching contracts onto path conditions to determine counter-examples
- Precise definition for validity of proof result
  - Results of proof on path conditions is related to proof on abstracted transformation executions

- SyVOLT is a tool for contract verification of DSLTrans transformations
- Major contribution of thesis is on efficiency and applicability to industrial-sized transformations
- FTG-PM presented answers RQ4) What is the design and work-flow of a contract verification tool?



Efficiency Techniques

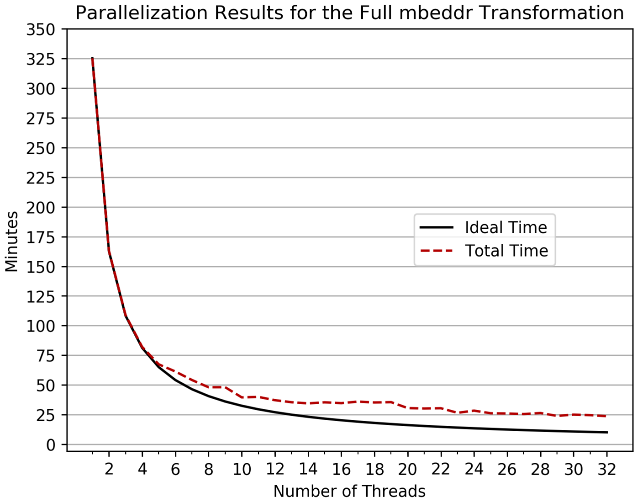
- Thesis presents five case studies ranging from 7 rules to 46 rules
- To improve the scalability of the tool, three efficiency techniques are presented:

- Parallelization
- Slicing
- Pruning

Answers RQ5) What are techniques for improving the scalability of the verification tool?

Parallelization

- Idea: Employ multiple threads to speed up tool
- Technique: Divide path condition generation and contract proving amongst multiple threads
- Results on the 32-core supercomputer for the largest case study of 46 rules:

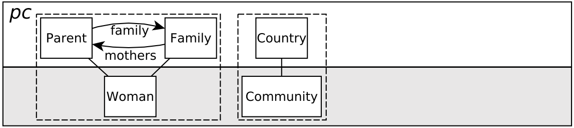


- Idea: Instead of symbolically executing all rules, select the rules relevant to a contract
- Technique: Match elements from contract onto rules to find dependencies
- Speed-up: 3.8x - 72.6x
- Trade-off between generating all path conditions, and smaller set for one contract

Contract Name	**Rules**	**Path Conds.**	**Total Time (s)**
Full	19	4916	59.18
CityCompany	8	43	0.30
CountryCity	6	10	0.13
SchoolOrdFac	5	17	0.17
DaughterMother	9	64	0.43
AssocCity	9	64	0.35
ChildSchool	5	17	0.17
FourMembers	9	64	0.43
MotherFather	9	64	0.45
ParentCompany	5	18	0.16
TownHallComm	11	184	0.86

Pruning

- Idea: Remove invalid path conditions to decrease state space
- Technique: Check if path conditions violate meta-model containment restrictions
  - Example: A meta-model requires that all *Woman* elements are contained in a *Community* through a *persons* link
  - If a *Woman* element exists, but is not connected by a *persons* link, then that path condition is invalid as it doesn't represent a valid output model



- Speed-up: 0.9x - 14.2x
- Warning: Pruning can change contract satisfaction results
  - Counter-examples to a contract can be pruned away

SyVOLT Contributions

- Development of an efficient and scalable contract verification tool
- Detailed presentation of core algorithms and efficiency techniques, their complexity, and advantages/disadvantages
- Examination of multiple case studies (toy to industrial) with results of contract proof

Outline

Conclusion

Thesis provides end-to-end approach to contract verification

- Semantics provided for DSLTrans enables precise usage and verification for all structures
- Contract verification approach efficiently determines counter-examples to contracts, offering insight into transformation behaviour
- Algorithmic design and implementation of a contract prover, which is scalable to industrial-sized transformations

Future Work

- Investigate how symbolic execution and contract proving approach can be transferred to other transformation languages
- Explore assisting the user in systematically creating contracts to verify transformations
  - Promote "contract-based design" of model transformations, with continuous verification

Thank you for your time and attention

B. Oakes, C. Verbrugge, L. Lúcio,and H. Vangheluwe. Debugging of Model Transformations and Contracts in SyVOLT. Submitted to the Debugging in Model-Driven Engineering (MDEbug 2018) workshop.

B. Oakes, L. Lúcio, C. Gomes, and H. Vangheluwe. Expressive Symbolic-Execution Contract Proving for the DSLTrans Transformation Language. Technical Report SOCS-TR-2017.1, McGill University, 2017.

B. Oakes, J. Troya, L. Lúcio, and M. Wimmer. Full Contract Verification for ATL using Symbolic Execution. *Software and Systems Modeling*, pages 1–35, 2016.

B. Oakes, J. Troya, L. Lúcio, and M. Wimmer. Fully Verifying Transformation Contracts for Declarative ATL. In *International Conference on Model Driven Engineering Languages and Systems*, pages 256–265, 2015.

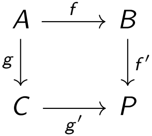
L. Lúcio, **B. Oakes**, C. Gomes, G. Selim, J. Dingel, J. Cordy, and H. Vangheluwe. *SyVOLT: Full Model Transformation Verification using Contracts*. In *International Conference on Model Driven Engineering Languages and Systems*, pages 24–27, 2015.

G. Selim, J. Cordy, J. Dingel, L. Lúcio, and **B. Oakes**. *Finding and Fixing Bugs in Model Transformations with Formal Verification: An Experience Report*. In *Proceedings of Analysis of Model Transformations workshop at Model Driven Engineering Languages and Systems*, pages 26–35, 2015.

L. Lúcio, **B. Oakes**, and H. Vangheluwe. *A Technique for Symbolically Verifying Properties of Graph-based Model Transformations*. Technical Report SOCS-TR-2014.1, McGill University, 2014.

G. Selim, J. Cordy, J. Dingel, L. Lúcio, and **B. Oakes**. *Specification and Verification of Graph-Based Model Transformation Properties*. In *Proceedings of International Conference on Graph Transformation*, pages 113–129, 2014.

Pushout Explanation

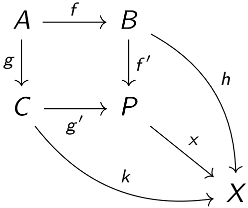


P is a pushout over the morphisms $f : A \to B$ and $g : A \to C$ defined by:

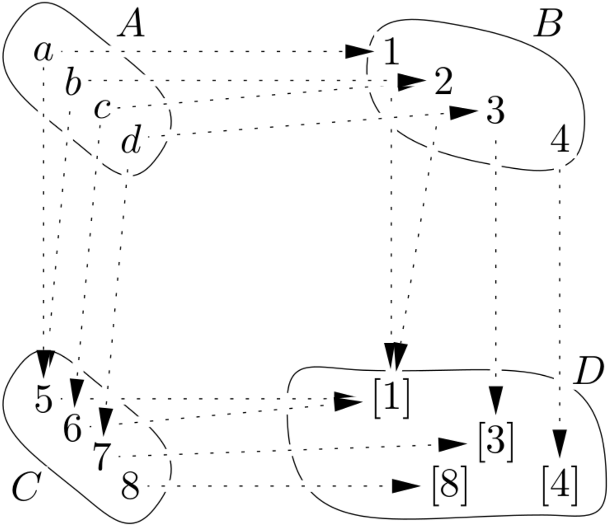
- a pushout object P
- the morphisms $f' : B \to P$ and $g' : C \to P$ with $f' \circ g = g' \circ f$

Definition

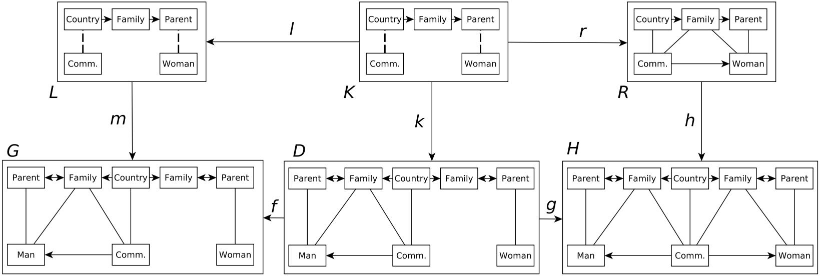
Universal Property
For all objects X, morphisms $h : B \to X$, $k : C \to X$ with $k \circ g = h \circ f$:
there is a unique morphism $x : P \to X$ with $x \circ g' = h$ and $x \circ f' = k$.

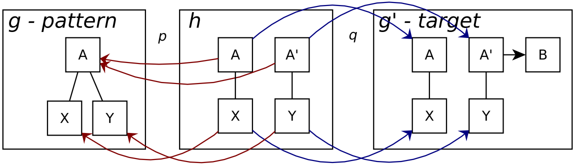


Pushout Example



Double-Pushout Approach Example

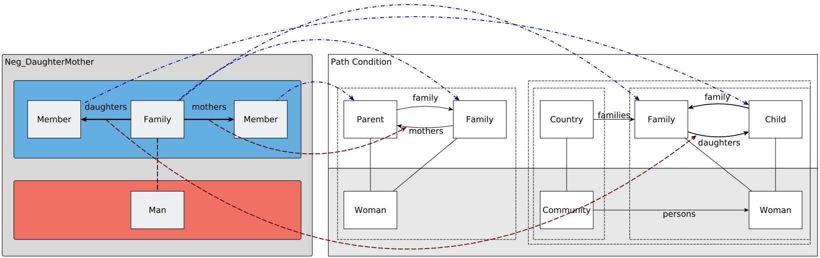




Issue: Need one pattern node to be matched to two target nodes

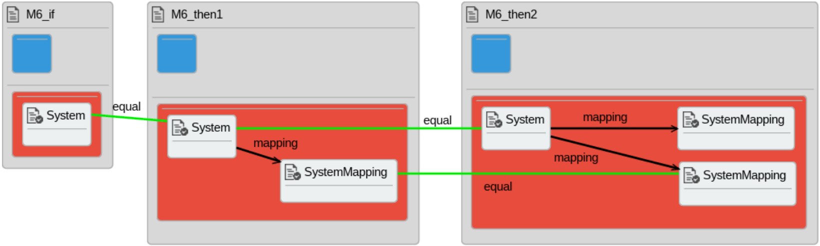
Requirements:

- Pattern nodes are totally matched
- One pattern node to multiple target nodes
  - Attributes must be matched
- Edges must be one-to-one



- The issue is that we are representing rule application, but not all possible ways rules can match over the same elements
- This is prohibitively expensive to calculate explicitly
- This structural information is discarded in the split morphism, allowing the contract to "split" over the path condition, and consider the *Family* elements to be unified

- Contracts can only express very basic structural information
  - Constructs available: *And*, *Or*, *If-Then*, *Not*, *pivots*
  - Can't represent universal operators ("for all element A's, B's must be attached"), or temporal properties
- Multiplicity is unintuitive



If a *System* element exists in the output model, then that *System* element should be connected to a *SystemMapping* element, and not (always) connected to two *SystemMapping* elements

- Language not placed in formal logic (first-order, second-order) framework
- Future work: Replace contract language with another, rather than improve