Debugging of Model Transformations and Contracts in SyVOLT

Bentley James Oakes
McGill University, Montreal, Canada
bentley.oakes@mail.mcgill.ca

Clark Verbrugge
McGill University, Montreal, Canada
clump@cs.mcgill.ca

Levi Lucio
fortiss GmbH, München, Germany
lucio@fortiss.org

Hans Vangheluwe
McGill University, Montreal, Canada
University of Antwerp, Antwerp, Belgium
Flanders Make, Belgium
Hans.Vangheluwe@uantwerpen.be

ABSTRACT

The *SyVOLT* tool verifies DSLTrans transformations by generating a state-space for the transformation's execution, and then proving structural contracts on that state-space. As with any verification activity, it is non-trivial to ensure that these contracts are error-free and correspond to the user's intention.

SyVOLT detects and localizes errors in the input artefacts for the verification activity to provide the user with assistance in debugging the transformation and/or the contracts. This experience report details the techniques built into the *analysis*, *monitoring*, and *reporting* stages of the tool. These techniques include detection of invalid rules and contracts, a form of reachability analysis during state-space generation, and assisting the user in understanding why a contract fails to be satisfied.

CCS CONCEPTS

• **Computing methodologies** → **Model verification and validation**; • **Software and its engineering** → *Domain specific languages*; *Automated static analysis*;

ACM Reference Format:
Bentley James Oakes, Clark Verbrugge, Levi Lucio, and Hans Vangheluwe. 2018. Debugging of Model Transformations and Contracts in SyVOLT. In *Proceedings of Debugging in Model-Driven Engineering (MDEbug).* ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

As the adoption of model-driven engineering increases in both academia and industry, the verification of model-to-model or model-to-text transformations becomes a critical part of software development. Our verification research [9, 10] focuses on the proving of *structural contracts*, which represent patterns on the input and output models of a transformation written in the DSLTrans model transformation language. If a contract is said to hold by our technique, then whenever a transformation's input model contains the pattern specified in the pre-condition of the contract, the output model produced by that transformation will contain the pattern specified in the contract's post-condition (including traceability constraints).

The SyVOLT tool [8] verifies a set of contracts on a transformation by generating a symbolic state-space for the transformation through the combination of rule applications. The contracts are then proved over the state-space, yielding combinations of rules where the contract is satisfied, and combinations where it is not. The SyVOLT tool and its input artefacts are presented in Section 2 to provide background information.

However, the result of the verification process only indicates that the contracts *are* or *are not* satisfied by the transformation by providing counter-examples. It is still a significant effort to examine these counter-examples and localize the fault in the transformation and/or the contracts. As well, once the fault is localized, it may be non-trivial to modify the artefacts to obtain the desired contract result. One particular problem, partially addressed in this paper by the contract result analysis, is the situation where a contract is expected to fail, but instead is reported as *satisfied*.

The current paper addresses the difficulty in understanding the interaction of contracts with the transformation under study. As our contribution, we present three stages of fault detection and localization within the SyVOLT tool, at the appropriate level of abstraction. The first stage is *analysis* (Section 3), where transformation and contracts are examined to record dependency information, and ensure that transformations and contracts are valid before state-space generation begins. The second stage is the *monitoring* of state space generation (Section 4), where rules are examined to ensure they symbolically execute. Finally, the *reporting* stage (Section 5) offers an explanation for why contracts have failed to be satisfied.

2 BACKGROUND

Our verification research concerns the creation of a symbolic execution state-space for transformations, and verification of structural contracts over that representation of the state-space. These operations are performed by the SyVOLT contract verifier[1], which has been employed in transformation verification research [1].

2.1 DSLTrans Transformation Language

Our symbolic execution technique requires that the transformation language being verified has reduced expressiveness, as in work by Varró *et al.* [18]. Our verification research operates on the model transformation language DSLTrans [2], which has the properties of *termination* and *confluence*. A full treatment of the semantics of DSLTrans is found in the thesis of Oakes [10].

The essential unit of a DSLTrans transformation is the *rule*. Rules contain matching and production elements and links in the *MatchModel* and the *ApplyModel*. In Figure 1 on the following page, the

[1] Available at https://github.com/levilucio/SyVOLT.

MDEbug, October 2018, Copenhagen, Denmark
2018.

MatchModel is the white box in the top half of the rule, while the *ApplyModel* is the yellow box in the bottom of the rule. Rule application begins by matching elements and links found in the *MatchModel* (as well as connected elements in the *ApplyModel*) onto the input model of the rule. If this can be accomplished, then the elements and links in the *ApplyModel* (not connected to the *MatchModel*) are created in the output model.

As well, rules specify how elements of the output model were created from specific elements of the input model during rule application. This traceability information for the transformation is stored as *traceability links*, which are built between a newly generated element in the output model, and the elements of the input model that originated it.

The *backward link* construct in DSLTrans rules is used to match over these traceability links. Figure 1 depicts a rule containing two backward links. The backward links are denoted as dotted lines, connecting the *Country* and *Community* elements and the *Child* and *Woman* elements. These links enforce a dependency that the *Community* element must have been produced by a rule that matched on the *Country* element in an earlier layer of the transformation.

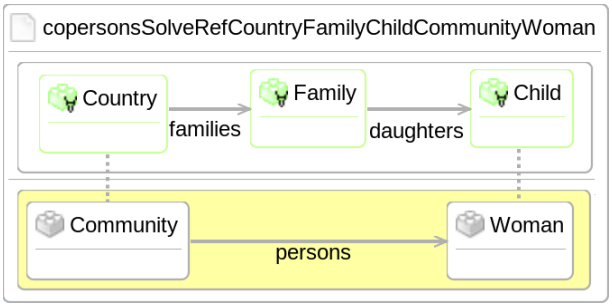


Figure 1: A DSLTrans rule with backward links.

Rules are placed into *layers* in the transformation. The semantics of DSLTrans specify that all rules in a layer fully apply before the next layer is executed. Note that rules in a layer cannot match on the output produced by another rule in the same layer.

2.2 Path Conditions

Our contract verification technique builds structures called *path conditions*, which represent the symbolic application of rules in the transformation by explicitly including the input and output elements and links matched over or created by those rules.

An example path condition is shown in Figure 2, which represents the application of four rules in the transformation. These rules are bounded in dashed boxes and contain elements and links which are typed from the input and output meta-models for the transformation.

The white top-half of the path condition is the *input graph*, while the grey bottom-half is the *output graph*. Lines between the input graph and output graph are *traceability links*, which record how the output elements were created during the execution of the transformation. This path condition can therefore be read as: "If only these four rules apply during a transformation execution, then (at least) these upper elements were present in the input model, and these lower elements are present in the output model."

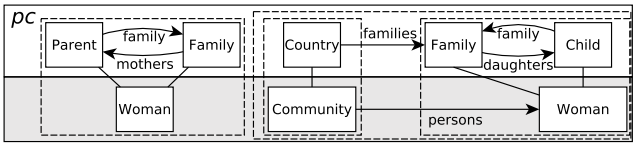


Figure 2: A path condition which represents the application of four transformation rules.

2.3 Contracts

The structural contracts used in our verification research represent patterns on the input and output models of a transformation.

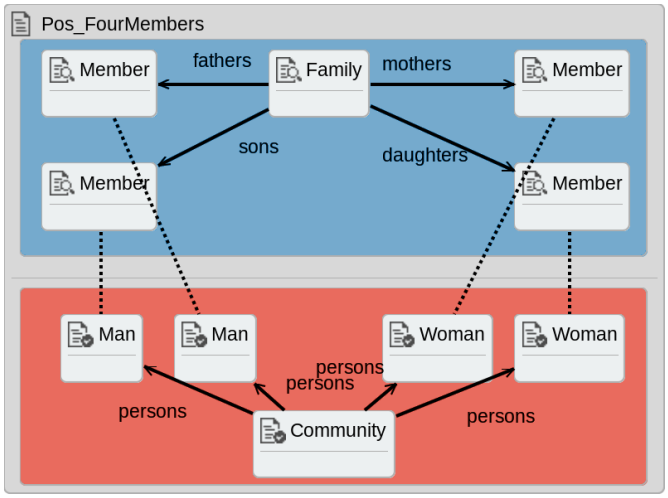


Figure 3: *Pos_FourMembers* contract, representing the informal statement "A *Family* with a *father, mother, son* and *daughter* should always produce two *Man* and two *Woman* elements connected to a *Community*"

For example, the *input graph* of the contract in the upper-half of Figure 3 contains a *Family* element and four connected *Member* elements. The *output graph* of the contract in the lower-half of the figure contains a *Community* element, two *Man* elements, and two *Woman* elements. The dashed lines represent the restriction that the output elements must have been produced by a rule that matched over the connected input element.

2.3.1 Contract Verification. Once the set of path conditions has been created for a transformation, then the pre-condition and post-condition for each contract are matched over these path conditions. In Figure 3, the pre-condition of the contract is composed of the elements in the upper half of the contract. The post-condition of this contract is all elements and links as shown in Figure 3.

If a contract is said to be satisfied by our technique, then whenever a transformation's input model contains the pattern specified in the pre-condition of the contract, the output model produced

by that transformation will contain the pattern specified in the contract's post-condition (including traceability constraints). Otherwise, if the post-condition does not hold on a path condition, then we have found a counter-example to the contract, and those rule combinations represented by the path condition do not satisfy the contract.

The present paper focuses on how to assist the user in understanding the result of contract satisfaction through a number of debugging techniques.

3 DEBUGGING STAGE 1: ANALYSIS

The main operation of the SyVOLT tool is the generation of the state-space of rule execution. However, first an *analysis* stage occurs, where the transformation and contract are examined to detect dependencies between the contract and rules in the transformation, as well as inter-rule dependencies.

This dependency analysis has three primary objectives in a debugging context. The first objective is a sanity check to ensure that all rules can symbolically execute and all contract elements can be produced, such that the user is alerted to an invalid transformation or a contract which cannot be verified. The second objective is to record which elements of the contract and transformation depend on which rules in the transformation, as this information will be used when presenting the results of verification (Section 5).

Finally, the last objective is to support a *slicing* optimization [10, 11], which restricts the symbolic execution of the transformation to only those rules needed to verify a contract. This optimization therefore improves the debugging process for the user by winnowing away all rules that can not change the satisfaction of a contract, and allowing the user to concentrate on the essential rules.

3.1 Procedure

The analysis process begins by examining all rules in the transformation to determine if any rule produces the elements and links in a particular contract. As well, we ensure that all backward links between input and output elements (denoting dependencies) in the transformation's rules can be satisfied by earlier rules.

This element-matching allows our analysis to check if there exists an element or link in a contract or a rule that cannot be satisfied in the transformation. In this case, even before the contract prover executes we know that there is an error with the contract or the transformation such that the validation process will fail. When this occurs, the user is alerted to the precise missing element or link in the rule or contract so that it may be fixed.

3.2 Example

Consider an intentional error in the *FourMembers* contract presented in Figure 3. Instead of having *Woman* elements in the *Output* graph (lower half) of this contract, assume that these are *Female* elements in the erroneous version. This represents the contract builder unintentionally using an old meta-model or making a typo.

When the contract prover is determining which rules the contract depends on, the output in Figure 4 will be produced. In the section marked *a)*, the analysis indicates that the link from the *Community* element to the *Female* element cannot be found in the transformation. This indicates that no path condition will exist

```
a) Error: Elements in contract ErrFourMembers link are missing:
    Community - directLink_T - Female
b) Error: Backward link might be missing in contract ErrFourMembers:
    Member - trace - Female
c) Looking for meta-model element: `Member'
Rule Daughter2Woman contains meta-model element:
     `Member' as `Child'
Rule Father2Man contains meta-model element:
    `Member' as `Parent'
...
d) Looking for meta-model element: `Female'
Error: Meta-model element `Female' not found in any rule!
```

Figure 4: Dependency analysis presented for the erroneous *ErrFourMembers* contract.

where the contract's post-condition can be satisfied, and thus the contract will never hold.

The output marked *b)* states that there is also a backward link in the contract which cannot be satisfied by the transformation. That is, there is no rule that matches over a *Member* element and produces a *Female* element. To assist the user, the rules where these elements are presented are indicated in sections *c)* and *d)*.

The *c)* section displays the transformation rules that contain a *Member* element. Note that these rules may contain the element as a sub-class instead. For example, the *Father2Man* rule contains a *Parent* element which is a sub-class of the *Member* element. This allows the user to focus on the rules which are likely candidates for errors.

Finally, the section marked *d)* informs the user of the error that no rule contains a *Female* element. All elements of the contract must be present in the transformation for successful verification, otherwise the user must modify the contract or transformation such that the element exists.

3.3 Fixing Input Errors

During our verification research, we noticed that our case studies suffered from typos/inconsistencies, which meant that a number of contracts could never be satisfied. For example, a contract for the *mbeddr* case study (presented in the thesis of Oakes [10]) contained *ReferenceExpression* and *GlobalRefExpr* elements instead of the correct *ReferenceExpr* and *GlobalVarRef* elements.

In this section, we wish to emphasize how crucial it is to detect or prevent these simple errors as early as possible in the verification process. These errors can potentially consume a large portion of debugging time, while not addressing the "interesting" reasons for verification failure.

The source of these errors was the manual nature of the Eclipse-based editing environment used to initially create the transformations and contracts [8]. In the current version of the visual editor, the user is able to type in any name for rule elements without any validation against the transformation meta-models.

More recently, we have developed an editor plugin[2] for the Meta Programming System (MPS) editor [3]. MPS is a *language workbench* for the rapid creation and use of *domain-specific languages* (DSLs), which can ease the development of software [17].

[2]The plugin is available from within the MPS plugin repository under the name *DSLTrans*, or at the repository site https://github.com/mbeddr/language_verification.
[3]https://www.jetbrains.com/MPS

As part of our verification research, we created the language constructs necessary to build DSLTrans transformations and contracts. The explicit modelling of languages in MPS allows for desirable features during modelling activities, such as auto-completion and type checking in the *projectional editor* [5].

For example, when creating DSLTrans rules or contracts, elements and their attributes in the rule must be typed by the input or output meta-models of the transformation. We (and others [19]) note that this deep integration of languages allows for quick and accurate creation of code/transformations, with fewer errors than a manual method.

Figure 5: Auto-complete during rule construction in the Meta-Programming System (MPS) editor.

Figure 5 demonstrates the creation of the *Father2Man* rule in the textual editor, where the rule's *MatchModel* includes a *Parent* element and a *Family* element. Note that the user is in the process of adding an attribute to the *Family*, and the auto-complete pop-up indicates that the *Family* concept includes the *name* and *lastName* as valid attributes, as is specified in the input meta-model. Thus, all elements and attributes in the transformation and the contracts are typed according to the input and output languages, preventing typos or inconsistencies before they can affect the verification process.

4 DEBUGGING STAGE 2: MONITORING

The second stage of fault detection and localization in the SyVOLT is performed during the path condition generation process itself. As the state-space for rule execution is constructed, the user is informed if a rule has not been symbolically executed[4].

A rule not symbolically executing may indicate one of three issues with the transformation which should be reported to the user to investigate. The first case is where the required dependencies for the rule were not present in the transformation. This may occur if a *backward link* specifies a dependency not satisfied by any other rules. The second case is where the user has not indicated that a rule should symbolically execute enough times, as specified by a dependency analysis [10].

Finally, the third case is that there may be an undesired interaction between the transformation and its meta-model that has caused our *pruning* technique to remove all path conditions containing that particular rule. This technique removes path conditions if they violate the output meta-model with respect to containment links. For example, a path condition containing *Female* elements in the output model not properly contained by a *Community* element would be removed. Careful use of this technique is required, as it may remove counter-examples to contracts [10].

The current implementation for the rule reachability analysis involves checking all path conditions periodically to determine which rules they abstract. If a rule appears in no path condition, then it either did not execute in that layer, or it has been removed by the pruning technique. An error is raised informing the user which rule did not execute, as well as the dependency information generated in the first stage of analysis (Section 3). This (rudimentary) information allows the user to debug and understand the execution of the rule, before further layers in the transformation are symbolically executed.

5 DEBUGGING STAGE 3: REPORTING

The last stage of fault detection and localization is the reporting of results to the user at the end of contract validation. Our aim is to assist the user in understanding why a particular contract has been or has not been satisfied by presenting the rules and rule elements that the contract requires. This allows the user to ensure that their intention for the contract matches the verification result, and if not, where modifications should be made.

In the contract verification step of the SyVOLT tool, we collect two sets of path conditions for each contract. The first set (*success*) is where the contract is satisfied, and the second set (*fail*) is where the contract is not satisfied. The tool examines these sets and provides the following information:

- $rules_{success}$ - Rules found in all successful path conditions
- $rules_{fail}$ - Rules found in all failed path conditions, and not present in $rules_{success}$
- The precise elements and links that the contract requires from the set of rules $rules_{success}$
- The contract is also tested against a failed path condition
  - The elements and links that could not be found on this path condition are reported
  - The user can use this information and a visualization of the path condition to better understand the failure of the contract, as in Figure 8 on the facing page

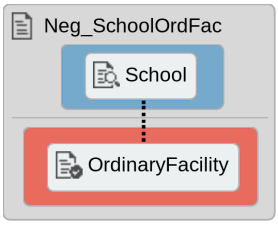
This comprehensive information allows the user to precisely determine how the contract is succeeding or failing, and have assurance that the result is as intended.

5.1 Example

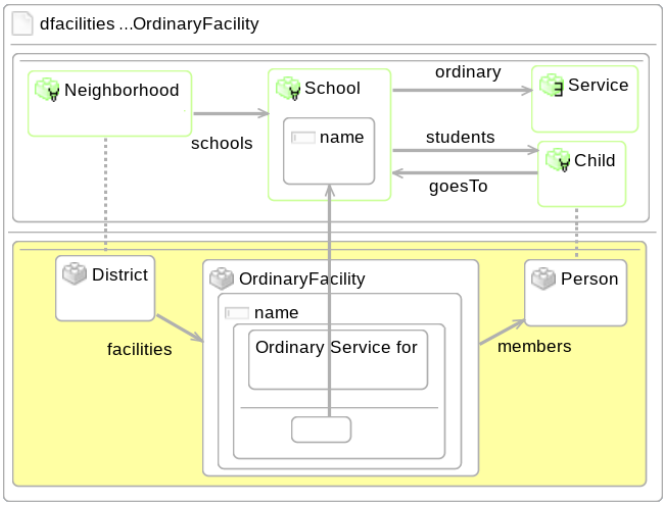
We will present the *Neg_SchoolOrdFac* contract in Figure 6a as an example for the information reported when a contract has counter-example path conditions.

Note that for this contract we expect counter-examples to be found. This contract represents the statement 'Every *School* in the input model will produce an *Ordinary Facility*'. The rule *dfacilities-...OrdinaryFacility* which performs this production is displayed in Figure 6b. However, the transformation also contains the *dfacilities-...SpecialFacility* rule, which matches over a *School* element with

[4]Note that the work of Selim [12] defined this check as a series of *rule reachability* contracts.



(a) An example contract for analysing success/failure results.



(b) Rule needed for contract success.

Figure 6: Contract and rule examples for result analysis.

a *special Service* to produce a *Special Facility* element. Thus, the contract is expected to fail on path conditions that include the *dfacilities...SpecialFacility* rule.

For the *Neg_SchoolOrdFac* contract, as seen in Figure 7 on the line marked *a)*, the contract prover indicates that there are six path conditions where this contract holds, and three path conditions where it does not.

```
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
c) Contract requires elements from successful rules of type:
    School
    OrdinaryFacility
d) Examining failed path condition:
    Daughter2Woman-Neigh2District-dfacilities...SpecialFacilityPerson
e) Elements of contract that fail on this path condition:
    Could not find links of type:
        OrdinaryFacility - trace_link - School
```

Figure 7: Result analysis for the *Neg_SchoolOrdFac* contract.

The contract result analysis indicates in the section marked *b)* that the *dfacilities...OrdinaryFacility* rule is common to all the successful path conditions. This is correct, as this rule must symbolically execute for the contract's pre- and post- condition to match. As well, the *dfacilities...SpecialFacility* rule is common to all of the failed path conditions. This is sensible as this rule produces path conditions where the *School* element in the contract pre-condition is found, but the *OrdinaryFacility* element is not. Note that the analysis does not report rules that are both in the successful and failed path conditions to assist in narrowing down causes of success or failure.

The last two analyses precisely identify *why* the contract fails on the path conditions. Based on the dependency analysis performed in Section 3, the contract prover has a record of which elements and links the contract depends on for each rule. The analysis marked *c)* specifies that the contract depends on the *School* and *OrdinaryFacility* elements produced by the *dfacilities...OrdinaryFacility* rule.

On the line marked *d)*, a path condition is selected from the failed set of path conditions to examine further. A matching is performed with extra debugging information which indicates on the line marked *e)* that the *School - OrdinaryFacility* traceability link could not be found in the path condition.

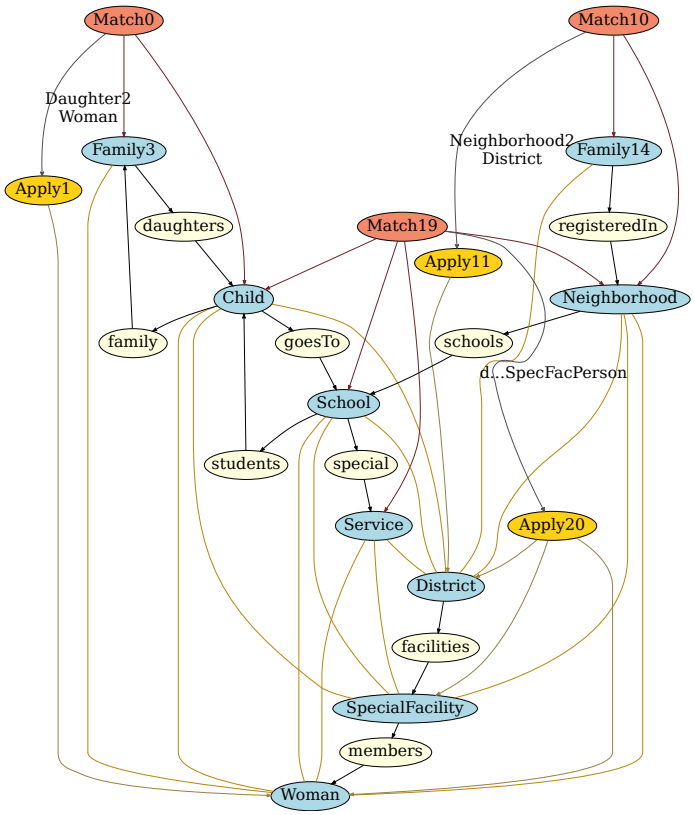


Figure 8: An unintuitive visualization of a path condition that fails to satisfy the *SchoolOrdFac* contract.

5.1.1 Visualization. Figure 8 shows a visualization for the failed path condition mentioned in part *d)* from Figure 7. This information is presented for the user to understand how the contract fails to match on the path condition. However, it is difficult to comprehend

as it presents a technical, graph-based representation. In comparison, Figure 2 on page 2 shows a hand-made (but more natural) visualization for a different path condition.

Another critique of this visualization is the inability to see the structure which would allow the contract to be satisfied. For example, the *Neg_SchoolOrdFac* requires a *School* element connected to a *Service* element through an *ordinary* link, not the *special* link shown in a pale yellow oval in the middle of Figure 8.

In future work, we intend to improve our visualization techniques by creating MPS plugins for visual viewers and editors of transformations, contracts, and path conditions.

6 RELATED WORK

There is significant previous work in the general field of model transformation verification [6, 12]. Our present work focuses on how to relate the result of contract verification to the transformation.

The work of Burgueneo *et al.* [3] compares the elements in *Tracts* to the rules in the transformation. These tracts define a set of constraints on the source and target meta-models, a set of source-target constraints, and a tract test suite. The intention is to suggest to the user which rules are causing constraints to fail by matching the classes and associations in the rules and constraints.

Cuadrado *et al.* [4] discuss a technique for statically analyzing ATL transformations. Their technique uses type inference and other analyses to determine potential problems in the transformation. From this, OCL constraints are generated and used to produce "witness" input models which precisely target the error.

The thesis of Amstel [14] tackles the issue of evaluating the *quality* of a model transformation. User studies are performed to rate transformations based on qualities such as re-usability, completeness, and conciseness. These qualitative facets of the transformation are then correlated with quantitative metrics, such as the number of rules or variables in the transformation, to provide a guide for the transformation designer to build intuitive transformations.

7 CONCLUSION

In this paper, we have discussed three analyses implemented in the SyVOLT tool to detect and report errors in the input transformation and contract to the user so that these errors can be fixed and successful verification can be achieved.

The checks and analyses detailed in this paper arose organically out of the development of the contract verification tool, and have been quite successful in detecting errors as well as giving insight into the behaviour of the contracts and transformations [10]. In the future, we aim to improve these techniques to allow for more precise detection, visualization, and correction of errors.

An especially crucial feature which is lacking in SyVOLT is the visualization of transformation execution, both before and after the application of rules as well as the actual matching and rewriting steps of each rule [7, 15]. Implementing these techniques would assist with two key questions we encountered during the development of SyVOLT: *why is this rule/contract not matching, when it should?* and *why is this rule/contract matching, when it shouldn't?*

We note that these debugging questions are relatively simple, and yet the igraph and T-Core libraries used in the SyVOLT tool lack sufficiently powerful debugging mechanisms, hindering development. For future work, we are interested in developing and promoting reusable libraries for graph-matching visualization and debugging in concert with the ModelVerse [16] and AToMPM [13] projects.

Developing a systematic approach to developing contracts is also a goal of our research, requiring user experiments to determine how users build contracts and what debugging information they require. As well, an important task is to transfer our contract verification and debugging techniques from the DSLTrans model transformation language to others with more expressiveness.

REFERENCES

[1] Banafsheh Azizi, Bahman Zamani, and Shekoufeh Kolahdouz-Rahimi. 2017. Contract verification of ETL transformations. In *International Conference on Computer and Knowledge Engineering*. IEEE, 154–160.

[2] Bruno Barroca, Levi Lúcio, Vasco Amaral, Roberto Félix, and Vasco Sousa. 2011. DSLTrans: A Turing Incomplete Transformation Language. In *International Conference on Software Language Engineering*. Springer Berlin Heidelberg, 296–305.

[3] Loli Burgueno, Javier Troya, Manuel Wimmer, and Antonio Vallecillo. 2015. Static Fault Localization in Model Transformations. *IEEE Transactions on Software Engineering* 41, 5 (2015), 490–506.

[4] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2017. Static analysis of model transformations. *IEEE Transactions on Software Engineering* 43, 9 (2017), 868–897.

[5] Martin Fowler. 2008. Projectional Editing. https://martinfowler.com/bliki/ProjectionalEditing.html.

[6] Esther Guerra, Juan De Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland. Schwinger. 2013. Automated Verification of Model Transformations based on Visual Contracts. *Automated Software Engineering* 20, 1 (2013), 5–46.

[7] Maris Jukšs, Clark Verbrugge, and Hans Vangheluwe. 2017. Transformations Debugging Transformations. In *MDEbug: Debugging in Model-Driven Engineering at International Conference on Model Driven Engineering Languages and Systems.*

[8] Levi Lúcio, Bentley Oakes, Cláudio Gomes, Gehan Selim, Juergen Dingel, James Cordy, and Hans Vangheluwe. 2015. SyVOLT: Full Model Transformation Verification Using Contracts.. In *International Conference on Model Driven Engineering Languages and Systems.* 24–27.

[9] Levi Lúcio, Bentley Oakes, and Hans Vangheluwe. 2014. *A technique for symbolically verifying properties of graph-based model transformations.* Technical Report SOCS-TR-2014.1. McGill University.

[10] Bentley Oakes. 2018. *A Symbolic Execution-Based Approach to Model Transformation Verification Using Structural Contracts.* Ph.D. Dissertation. McGill University.

[11] Bentley Oakes, Javier Troya, Levi Lúcio, and Manuel Wimmer. 2016. Full contract verification for ATL using symbolic execution. *Software and Systems Modeling* (2016), 1–35.

[12] Gehan Selim. 2015. *Formal Verification of Graph-Based Model Transformations.* Ph.D. Dissertation. Queen's University.

[13] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Huseyin Ergin. 2013. AToMPM: A web-based modeling environment. In *International Conference on Model Driven Engineering Languages and Systems.* 21–25.

[14] Marinus Franciscus van Amstel. 2012. *Assessing and improving the quality of model transformations.* Ph.D. Dissertation. Technische Universiteit Eindhoven.

[15] Simon Van Mierlo. 2015. Explicitly Modelling Model Debugging Environments.. In *ACM Student Research Competition at the International Conference on Model Driven Engineering Languages and Systems.* 24–29.

[16] Simon Van Mierlo, Bruno Barroca, Hans Vangheluwe, Eugene Syriani, and Thomas Kühne. 2014. Multi-level modelling in the Modelverse.. In *International Conference on Model Driven Engineering Languages and Systems.* 83–92.

[17] Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, and Hans Vangheluwe. 2017. Domain-specific Modelling For Human–computer Interaction. In *The Handbook of Formal Methods in Human-Computer Interaction.* Springer International Publishing, 435–463.

[18] Daniel Varró, Szilvia Varró-Gyapai, Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. Termination Analysis Of Model Transformations By Petri Nets. In *International Conference on Graph Transformation* (2006), Vol. 4178. Springer, 260–274.

[19] Markus Voelter, Arie van Deursen, Bernd Kolb, and Stephan Eberle. 2015. Using C language extensions for developing embedded software: A case study. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 655–674.