2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS) | 978-1-6654-3495-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/MODELS50736.2021.00024

Towards Control Flow Analysis of Declarative Graph Transformations with Symbolic Execution

Florian Ege
Institute of Software Engineering and Programming Languages
Ulm University
florian.ege@uni-ulm.de

Matthias Tichy
Institute of Software Engineering and Programming Languages
Ulm University
matthias.tichy@uni-ulm.de

***Abstract*—The declarative graph transformation language Henshin transforms instance models represented as graphs by applying a series of basic steps that match and replace structural patterns on parts of models. These simple transformation rules are then combined into control flow constructs similar to those of imperative programming languages to create more complex transformations. However, defects in the structure of control flow or in transformation rules might misschedule the application of operations, resulting in basic steps to be inapplicable or produce incorrect output. Understanding and fixing these bugs is complicated by the fact that pattern matching in rules is non-deterministic. Moreover, some control flow structures employ a nondeterministic choice of alternatives. This makes it challenging for developers to keep track of all the possible execution paths and interactions between them.**

For conventional programming languages, techniques have been developed to execute a program symbolically. By abstracting over the concrete values of variables in any actual run, generalized knowledge is gained about the possible behavior of the program. This can be useful in understanding problems and fixing bugs. In this paper, we present an approach to symbolically execute graph transformations for a subset of Henshin, using symbolic path constraints based on the cardinalities of graph pattern occurrences in the model.

***Index Terms*—model driven software engineering, declarative graph transformations, control flow analysis, symbolic execution**

I. Introduction

In model-driven software engineering (MDSE), developers represent complex systems by abstract instance models that adhere to structural constraints defined in metamodels. Application logic and dynamic behavior can be expressed through model transformations. Concrete implementations of MDSE follow diverse programming paradigms and provide a variety of textual or graphical representations for syntax (see [1]).

In this paper, we focus on declarative, endogenous graph transformations as implemented by Henshin [2]. Here, the abstract syntax of instance models and transformation rules is represented graphically. Henshin transformations consist of a hierarchy of compositional units that define the control flow through subunits. At the lowest level, transformation rules form the basic steps that drive the system through changing the model's structure. A rule consists of a graph pattern of model elements that is matched with a part of the model, and some related modifications to be performed on that part. These operations, such as matching and preserving elements, modifying their attributes, deleting or creating elements, then produce the output model.

A transformation rule can be viewed as a single atomic step that allows developers to reason about the structure of models and transformations at a high level of abstraction. However, the pattern matching phase introduces nondeterminism in the execution of rules. When multiple matches are found, one is chosen at random. Moreover, nondeterminism also occurs at the higher level of control flow through units. The semantics of some branching structures consist of random selection of applicable alternatives. Due to this pervasive nondeterminism, a complex transformation can, given the same input model, proceed along many possible execution paths. Since confluence is not guaranteed, it can also result in different output models. Although this declarative style relieves developers from having to deal with the more low-level aspects of control flow and order of operations, it presents special challenges for debugging a complex transformation. Defects can be located in the structure of control flow units, or in rules. They can manifest themselves in three broad categories: the complex transformation (1) does not terminate, e.g., by entering an infinite loop or recursion, (2) fails because a rule that must be executed is not applicable, causing the execution to abort, or (3) terminates but produces an incorrect output model.

Unfortunately, nondeterminism makes it more difficult to locate bugs in the control flow structure or individual rules, since (even with fixed input) each execution could unfold along a uniquely different path, with potentially diverse behavior in terms of possible errors or the eventually produced output. To address these challenges, it is necessary to abstract over the many concrete ways a transformation can be executed and to reason about its possible behavior as a whole. For conventional programming languages, techniques such as abstract interpretation or symbolic execution (see [3]) have been developed. These approaches are based on static analysis of programs with the objective of collecting abstract information about possible states or path constraints for control flow, without actually executing the program on any concrete input (see [4]–[6]). For this purpose, concrete values, e.g., for numeric variables are expressed in abstract domains, like intervals of possible values, or even just the sign or parity. These symbolic values

978-1-6654-3495-9/21/$31.00 ©2021 IEEE
DOI 10.1109/MODELS50736.2021.00024

are then propagated through the control flow graph [7], for which the domination and post-domination relations between nodes have been introduced [8] to deal with questions such as the reachability of points in a program.

However, the static analysis of computationally universal languages is subject to certain fundamental limitations. Rice's theorem [9] states that all nontrivial semantic properties of a program's behavior are undecidable, as they can ultimately be reduced to the halting problem. These include questions such as the reachability of particular points in the program, what state the program has at a certain point, which states are possible at all, etc. Since all these questions are generally undecidable, we have to settle for approximations. In the field of static analysis, the concepts of soundness, precision, and feasibility were introduced to reason about tradeoffs in the design of analysis methods (see [10]). For example, if the goal is to determine that a program is free of a particular kind of problem, *soundness* means that all occurrences are surely detected (i.e., there are no false negative answers). In contrast, *precision* is the ratio of the true negative results to all negatively reported ones. However, the undecidability of "interesting" properties means that the soundness of an analysis necessarily implies that it is imprecise. In practice, a reasonable balance must be struck between the questions that can be answered soundly and the drawbacks in terms of imprecision or cost in terms of feasibility that are involved.

Our objective in this paper is to explore how symbolic execution techniques can be adapted to perform static analysis of declarative graph transformations. We restrict ourselves to a subset of the Henshin language, specifically all the control flow units with their strict and non-strict variants, but only simple rules which preserve, delete or create graph nodes. Hence, our analysis only considers the graph structure of the model, but not object attributes, positive/negative application conditions, or conditions on attributes.

In the context of our analysis we would like to answer questions relevant to understanding the principle behavior of a transformation or to debugging faulty ones, such as:

- Can we infer that a transformation terminates, for all, some or no input models?
- Is a particular point in the control flow graph reachable or unreachable, and if so, which structural constraints must hold for a model in either case?
- What structural constraints can be derived for a model at a particular point in the control flow graph?

With respect to these goals, the main contributions of this paper are:

- We present a scheme to translate complex Henshin transformations which use a variety of control units into a simple unified control flow graph. This is the central data structure on which we then perform our static analysis.
- We introduce an algebra of counted graph pattern expressions as an abstract domain to represent and propagate path constraints through the control flow graph. Knowing which subgraph structures are present in the model, and how often, allows us to draw conclusions about the applicability of rules and thus answer questions about possible control flow.

This paper is structured as follows: In Section II, we present the data model and algorithms used in our approach to symbolically execute graph transformations and derive knowledge about their control flow. Section III provides some background about the limitations of our approach and the rationale for simplifications in our design. In Section IV, we discuss related work regarding static analysis techniques and symbolic execution in the domain of model transformations. We conclude with a summary of this paper and an outlook on future work in Section V.

II. Symbolic Execution

In this section, we present our approach for the symbolic execution of declarative graph transformations. First, we explain how we construct the control flow graph corresponding to a Henshin transformation. Then, we introduce counted subgraph patterns as the abstract domain from which we form path constraints, and the algebra to manipulate them. Finally, we bring these data structures and formalisms together and present the algorithm to symbolically execute a transformation and collect abstract information about intermediate states of the model under transformation.

A. Control Flow Graph

The *control flow graph (CFG)* is the central data structure underlying the symbolic execution of a transformation. In addition to basic transformation rules, Henshin provides several higher-level units as control structures for complex transformations. These units can be nested hierarchically and allow conditional or nondeterministic branches, loops and recursion. In a first step, we generate the CFG from the nested structure of units and rules. In doing so, we "flatten" the various control flow structures and express the transformation as a single CFG using only four types of nodes:

- *Rule nodes,* (boxes labeled with the rule's name) which represent the basic transformation rules.
- *Conditional branching nodes,* (diamonds labeled with the rule to be matched as condition) corresponding to a binary conditional branch with an outgoing edge labeled *true* for the consequent branch, and one labeled *false* for the alternative.
- *Nondeterministic branching nodes,* (empty diamonds) that allow nondeterministic branching between any number of alternatives.
- *Recursive unit nodes* (rounded boxes labeled with the unit's name). When translating a unit to a CFG part, each further recursive occurrence of the same unit is abstracted with a recursive unit node. This is necessary to avoid an infinite expansion of the CFG when translating a structure that contains itself.

Fig. 1 shows units and their translation into corresponding CFG parts. Beginning with the topmost unit, the CFG of the transformation is expanded. The dashed boxes are recursively

substituted with the translation of subunits. The individual CFG parts are then linked together: all edges to an exit node are redirected to the node pointed to by the entry node of the following part. Entry and exit nodes are then removed, resulting in one large connected CFG. Both nodes and edges in the CFG are later annotated with information collected during the symbolic execution of the transformation.

B. Graph Patterns and Cardinalities

As a basic building block from which we later construct expressions of our algebra for rule conditions and path constraints, we introduce *graph patterns.* These are all distinct, connected subgraph patterns with typed nodes and relations, extracted from the graphs in the left-hand (LHS) and right-hand sides (RHS) of rules. We associate with each of these graph patterns its respective *cardinality*, i.e., how many times this pattern occurs in the rule's graph. Cardinalities are specified as intervals, even though at this stage they are a single scalar value. Later, when we perform algebraic operations with graph pattern expressions, cardinalities are generalized to ranges of values. For consistency, all cardinalities are therefore treated as intervals of possible values.

Fig. 2 shows a rule matching a simple pattern of one node of type `A` that is related to two nodes of type `B`. From this graph, the patterns on the right are extracted:

- A single node pattern $P_A^{[1,1]}$ for the single `A`-node `a`, thus with cardinality 1.
- A single node pattern $P_B^{[2,2]}$ with a cardinality of 2 as there are two instances of `B`, `b1` and `b2`.
- A pattern $P_{AB}^{[2,2]}$ for the `A`-node related via the `b`-reference to just one `B`-node. The cardinality is 2, as this pattern occurs twice, once for `a` and `b1` and then for `a` and `b2`.
- Finally, a pattern $P_{ABB}^{[1,1]}$ that represents the entire graph, therefore with a cardinality of 1.

Note that the subscript labels are just informal names to tell the patterns apart. Formally, since they are patterns matching on subgraphs of the LHS- or RHS-graph of the rule, the full types of nodes and relations between them is part of the pattern. These graph patterns, together with their cardinalities, form the basic factors in terms expressing rule conditions and path constraints. As a notation, we combine such patterns in logical conjunctions, like

$$P_A^{[1,1]} \cdot P_B^{[2,2]} \cdot ...$$

meaning there is exactly 1 `A`-node *and* 2 `B`-nodes, etc., or in disjunctions, like

$$P_X^{[1,3]} + P_X^{[5,\infty)} + ...$$

meaning, the cardinality of `X`-nodes is either between 1 and 3, *or* is greater than 5.

C. Algebraic Operations on Graph Pattern Expressions

Following, we discuss how graph pattern expressions can be manipulated in ways similar to Boolean algebra.

1) Dropping Graph Patterns: Any pattern with a cardinality of $[0, \infty)$ can be dropped from both disjunctions and conjunctions. It provides no information at all, since any cardinality must naturally be contained in this range. Moreover, negating such a graph pattern factor is not meaningful, as the complementary range of cardinalities would be the empty interval, and a cardinality must always lie between 0 and ∞.

2) Merging equal Patterns: Equal patterns merge in disjunctions and conjunctions:

$$P_G^{[a,b]} + P_G^{[a,b]} = P_G^{[a,b]} \quad \text{and} \quad P_G^{[a,b]} \cdot P_G^{[a,b]} = P_G^{[a,b]}$$

3) Combining consecutive Cardinality Ranges in Disjunctions: Patterns for the same graph G that form consecutive (or, more general, overlapping) cardinality intervals can be combined in one encompassing interval:

$$P_G^{[a,k]} + P_G^{[k+1,b]} = P_G^{[a,b]}$$
$$P_G^{[a,v]} + P_G^{[u,b]} = P_G^{[a,b]} \text{ where } u \leqslant v$$

4) Combining Graph Patterns in Conjunctions: When patterns for the same graph G are conjoined, there are two cases. First, the cardinality intervals have a non-empty intersection. In this case, they combine to a pattern with the intersection interval:

$$P_G^{[a,b]} \cdot P_G^{[u,v]} = P_G^{[\max(a,u),\min(b,v)]}$$

Second, they can be disjoint. This constitutes a contradiction (e.g., G is supposed to be present at least 5 times, and simultaneously at most 3 times). A contradictory term in a path constraint is an indicator that a rule which matches against this pattern might fail, although other disjoint terms that are satisfiable might still enable a successful match. If a path constraint consists only of contradictory terms, the transformation cannot be guaranteed to progress past this point.

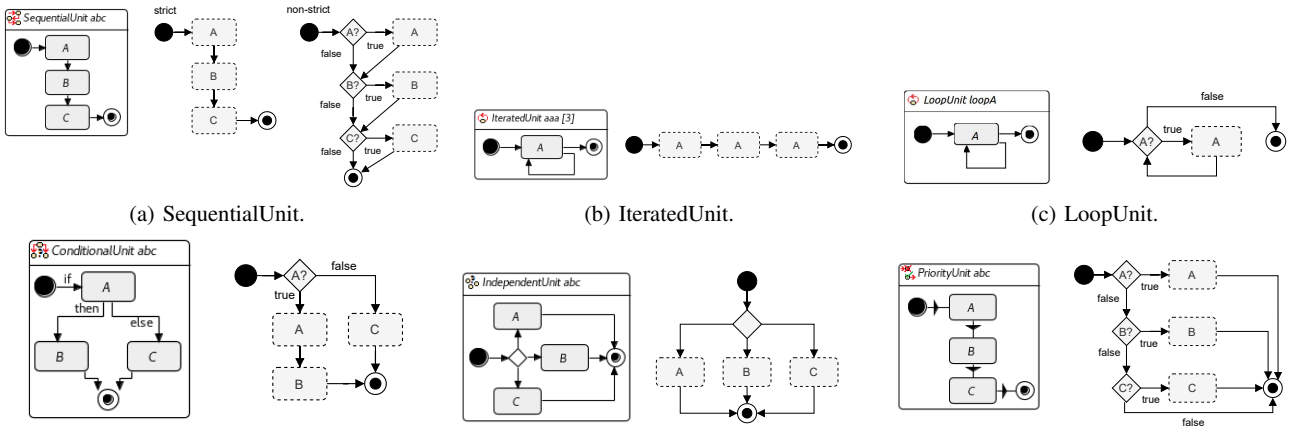
5) Negation: Negation of graph pattern expressions only happens at conditional branching nodes. Here, the graph patterns forming the node precondition (see Sec. II-D) is propagated unchanged along the consequent branch and negated along the alternative branch, where we infer that the LHS of the condition rule R did not match. The only thing we can soundly assume in this case is that the pattern representing the entire LHS-graph (which we denote as $P_{R:left}^{[1,1]}$) did not match. We can't make that assertion for its smaller subgraphs.

The logical negation of the statement that a pattern occurs (at least) once is therefore that it doesn't occur at all:

$$\overline{P_{R:left}^{[1,1]}} = P_{R:left}^{[0,0]}$$

D. Pre- and Postconditions for the Application of Rules

Rule nodes in the CFG have associated graph patterns expressing *preconditions* (structural constraints arising from the LHS) which the model must satisfy for the rule to be applicable, and *postconditions* (from the RHS) that hold for the state of the model after the rule has been successfully applied. Consider the rule shown in Fig. 3. For this rule to be applicable to the model, the elements that are part of the LHS-graph, which are thus to be matched and either preserved or later deleted, must exist in the first place. Here, the precondition is the graph pattern $P_{subst...:left}^{[1,1]} = P_{ABBX}^{[1,1]}$ consisting of the



(a) SequentialUnit. (b) IteratedUnit. (c) LoopUnit.

(d) ConditionalUnit. (e) IndependentUnit. (f) PriorityUnit.

Fig. 1: Translation of Henshin units to parts of the control flow graph.

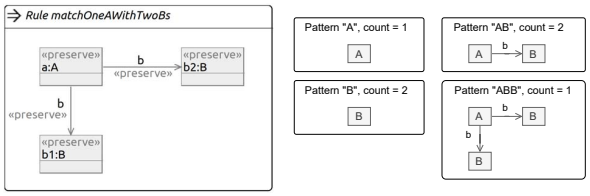


Fig. 2: A simple rule and the corresponding graph patterns and cardinalities.

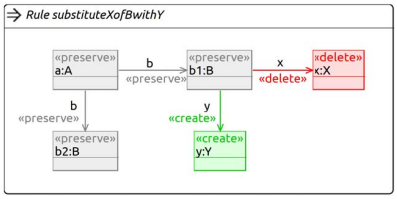


Fig. 3: A rule with preconditions and postconditions.

nodes `a`, `b1`, `b2`, and `x`. Assume, that the incoming path constraint to the rule node consists only of one conjunction, containing these graph patterns (some subgraph patterns have been omitted):

$$\ldots \cdot P_{ABBX}^{[1,1]} \cdot P_{X}^{[5,8]} \cdot P_{BX}^{[5,8]} \cdot P_{Y}^{[2,\infty)} \cdot P_{BY}^{[2,\infty)} \cdot \ldots$$

We then see that the rule's precondition is satisfied. The required elements to be preserved or deleted are present. After executing the rule on a model, which satisfies the above precondition at this point in the control flow, the following holds as postcondition:

$$\ldots \cdot P_{ABBX}^{[\mathbf{0},0]} \cdot P_{X}^{[\mathbf{4},7]} \cdot P_{BX}^{[\mathbf{4},7]} \cdot P_{Y}^{[\mathbf{3},\infty)} \cdot P_{BY}^{[\mathbf{3},\infty)} \cdot \ldots$$

Note that the cardinality ranges for patterns involving `X` and `Y` nodes have been updated to reflect the knowledge that there is now one instance of `X` less than before and, conversely, one additional `Y`.

E. Path Constraints and their Propagation

The information gathered during symbolic execution of a graph transformation is expressed as *path constraints (PC)*, associated with the edges in the CFG. We consider counted topological graph patterns (see Sec. II-B) as abstract domain. A path constraint on an edge in the CFG consists of an expression that is a disjunction of conjunctions of graph patterns. Each conjunction asserts that a combination of particular graph patterns is present (or not present) according to the respective cardinalities of possible occurrences in the instance model at this point in the control flow. In the following sections, we describe how path constraints are propagated through the CFG for the different types of CFG nodes. For brevity, we denote conjunctions of graph patterns with parenthesized single letters like $(A), (B), (R), \ldots$, and omit the cardinalities.

1) Propagation across rule nodes: A rule node can have any number of incoming edges, but has always exactly one outgoing edge. All conjunctions are collected across all incoming edges. Then, the rule's precondition is checked against all these conjunctions. All those without contradictions are then conjoined with the precondition, according to the scheme described in Sec. II-D, and disjoined together. This expression is simplified using the rules of the algebra in Sec. II-C, and then assigned to the outgoing edge (see Fig. 4a).

2) Propagation across nondeterministic choice nodes: A choice node can have any number of incoming and outgoing edges. PCs are propagated through a choice node by forming a disjunction of all the incoming PCs, simplifying it, and assigning it to all outgoing edges (see Fig. 4b).

3) Propagation across conditional branching nodes: A conditional node can have arbitrarily many incoming edges. It has one outgoing edge for the consequent branch and one for the alternative. To derive the PC for the consequent branch, the precondition of the rule R_{left} is conjoined individually with all conjunctions of all incoming edges. All terms without contradictions then form a disjunction which (after algebraic simplification) becomes the PC for the consequent branch. The

PC for the alternative is formed analogously, except that R_{left} is negated, as described in Sec. II-C5 (see Fig. 4c).

4) Propagation across recursive unit nodes: These nodes, which abstract recursive occurrences of a unit within its own CFG expansion, pose special challenges for the handling of PCs. Due to this self-referential nature of recursive nodes, we have chosen to assign empty PCs to the outgoing edge as a simplification for now. One level up, where the unit is actually expanded, PCs are propagated to outgoing edges via branches, unless they contain the recursive node in a tail-recursive position (see Fig. 4d). Nodes of this type can therefore lead to the erasure of incoming PC information on paths leading through them.

F. Analysis

In this section, we describe the algorithm for analyzing a transformation w.r.t. the questions in Sec. I.

First, we assign each CFG node the label *"unknown"*. We then perform the PC propagation as described above in Sec. II-E: Beginning at the entry node of the CFG, all nodes are visited in a breadth-first order, propagating the PCs in a flood-like way. Since there can be cycles in the CFG, e.g., introduced through LoopUnits, we need to set an upper limit on the number of times we visit each node. In the simplest case, the propagation for a node should stop to be reevaluated when all incoming edges have been assigned PCs for the first time. When we derive the PCs, we note the occurrence of conjunction terms with contradictions. If the path constraints on all edges pointing to a rule node are annotated with a graph pattern expression containing the precondition, we can conclude that the rule should be applicable, and the control flow can progress further. We thus label the node as *"reachable"*. If none of the edges satisfies the precondition, the node is labeled *"unreachable"*, and the following parts of the CFG dominated by this node are also assumed to be unreachable. In the absence of path constraints, or when mixed information is available, no assertions about reachability can be made and the status remains *"unknown"*.

Similarly, we examine the other types of CFG nodes. If all conjunctions are contradictory, the constraints in the path to reach this node are unsatisfiable. We hence assign the node the label *"unreachable"*. Conversely, if all conjunctions are free of contradictions, the node is considered *"reachable"*. Should both kinds of terms be present, the node is also *"reachable"*, as there exists a satisfiable path, while the unsatisfiable alternatives hint at problems in other paths. If there are no terms at all, the status remains *"unknown"*.

To infer statements about the reachability of nodes in the CFG and specifically the termination of the transformation as a whole, we perform the above analysis for all nodes for which information to compute the PC propagation is available. We then try to assign a reachability status to nodes which still are labeled as *"unknown"*. If all paths to a node are dominated by unreachable nodes, then that node can be considered unreachable as well. Conversely, if all predecessors are reachable but the node has empty incoming PCs, its reachability remains unknown. If information about reachability can be propagated all the way through to the exit node of the CFG, a statement about the termination of the entire transformation can be made.

III. Limitations

In this section, we discuss some limitations of our approach.

To cope with theoretical limits of static analysis methods and reduce the complexity of our path constraint algebra, we settled on some simplifications. First, we restricted the analyzed transformation language to a subset that excludes in particular negative application conditions, i.e., nodes with the *forbid* action type. This restriction leads to path constraint expressions that are much more compact, as otherwise for all nodes in each graph pattern all possible combinations with potentially related nodes even beyond this subgraph would have to be considered. Also, the abstract domain of graph patterns lacks a concept of identity. Although the cardinalities express invariants, such as that smaller graph structures must be present at least as often as larger ones containing them, knowledge about which instance of a smaller pattern is contained in which instance of a larger pattern is not available. This is due to the fact that nondeterministic rules might change patterns locally without matching a larger context that would be needed to propagate that information.

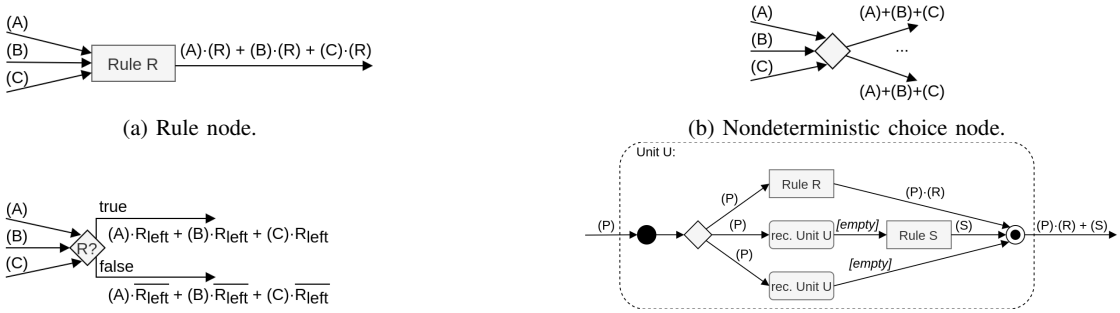
Instead of just dealing with the question whether or not a node supposed to be positively matched is actually present in the model, there would now be the additional category of nodes that are required to be absent but might exist, or not. For once, this would lead to path constraints growing exponentially depending on the number of relations in the meta model as compared to be bounded by the size of graph patterns in rules.

Moreover, mingling information about graph patterns known to be there with simultaneous information about structures that are excluded from being present somewhere in the model, would introduce many potential contradicting terms in path constraints, as it is possible that a pattern could be positively or negatively matched, or both, depending on which part of the model is selected. A possible solution to this challenge could consist in splitting our approach into separate analysis methods, that consider only positive or negative matches, respectively.

IV. Related Work

Following, we discuss related work in the domain of static analysis and symbolic execution of model transformation languages.

Rabbi et al. [11] present an algorithm for checking whether a rule in an endogenous graph transformation preserves conformance to the metamodel. For this, predicates in the form of graph constraints are derived from the metamodel, and each rule by itself is then examined to see which patterns in the graph are matched, and whether changes made by the rule could violate the constraints. The graph patterns we use in our approach for path constraints and pre- and postconditions of rules can also be used to infer the existence or absence of certain patterns in the model at intermediate points in the



(a) Rule node.

(b) Nondeterministic choice node.

(c) Conditional node with consequent and alternative branches.

(d) Recursively nested unit node.

Fig. 4: Propagation of path constraints across control flow graph nodes.

transformation or at its end, yet depending on the presence and form of recursion, this information may not be available at all places of interest. On the other hand, our counted patterns can express knowledge about the sequence of modifications made to the model across multiple rules. This information can be used, e.g., to check constraints on the size of collections or aggregations.

Critical pair analysis (see [12], [13]) checks all rules of a transformation pairwise to discover conflicts and dependencies between any two rules. This does however not take into account the place of the rules in the control flow, so it could detect conflicts or dependencies between two rules that would not interact in this way in reality, e.g., by being on different branches. We avoid this issue by propagating path constraints through the actual control flow. Moreover, PC propagation can uncover conflicts that arise from the interaction of more than just two rules. Nevertheless, due to the limitations of our approach, our analysis might miss a conflict caused by an actual critical pair due to insufficient path constraint information. Also, attribution of a PC contradiction at a particular point to one or more previous rules might not always be straightforward.

Gryphon [14] supports the symbolic verification of graph transformations within certain bounds on the number of model elements by encoding the transformation as Boolean circuits and then using model checkers for hardware systems to verify the reachability of certain states. The restriction to a finite-size model allows for rigorous analysis, where our approach in contrast considers the model entirely in a symbolic way. Thus, while we are therefore not limited to a maximum model size, we pay for generality with compromises regarding soundness.

Another way to enable rigorous and sound analysis of transformations is to constrain the expressive power of the transformation language, as in the case of DSLTrans, where the language's properties of being terminating and confluent allow to execute a transformation symbolically and achieve a full coverage with path constraints, which can then be used to prove structural contracts on input and output models or derive counterexamples (see [15]). Our approach targets Henshin, which as a computationally universal language with nondeterminism is in general neither terminating nor confluent. Hence, undecidability forces us to resort to approximations for some questions that can be answered definitively for a less powerful language, such as reachability or what the output looks like.

Kulcsár proposes "compasses" [16] as abstract domain for graph states in a double pushout graph rewriting. A "compass" is a collection of positive or negative graph patterns that might occur or cannot occur, respectively, in the graph. Using abstract interpretation, properties of the transformation can be verified by updating compasses along the way. Our approach has a more detailed view of occurences, as we consider cardinality intervals instead of mere existence/non-existence of patterns, however it is tailored specifically on symbolic execution of graph transformations on a simplified CFG structure.

V. Conclusion and Future Work

We now summarize our contributions and conclude with an outlook on planned future work.

We introduced data structures to represent the control flow of a complex graph transformation and the associated path constraints. Furthermore, we developed an algebraic formalism for counted subgraph pattern expressions to infer approximate information about which parts of the CFG are reachable and which structural constraints apply to model instances at various points. In future work, we would like to explore how this approach can be extended from the mere topological graph patterns considered so far to deal with more expressive features of Henshin, such as attributes of nodes, positive or negative application conditions, or logical conditions in rules. Also, we will look to improve the treatment of loops and recursion to extract more path constraint information. We also plan to implement a tool to perform these analyses and gather experience on the effectiveness of our approach, as well as evaluate it in terms of performance metrics such as time and space costs as a function of the complexity of metamodels, the size and number of graph patterns in rules and properties of the CFG. We are also interested in how path constraints can be used to generate concrete instance models as witnesses or counterexamples to results as, e.g., knowledge about the reachability of points in the control flow.

REFERENCES

[1] M. Biehl, "Literature study on model transformations," *Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK*, vol. 291, 2010.

[2] D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, and M. Tichy, "Henshin: A usability-focused framework for EMF model transformation development," in *Graph Transformation - 10th International Conference, ICGT 2017, Held as Part of STAF 2017, Marburg, Germany, July 18-19, 2017, Proceedings*.

[3] R. Amadini, G. Gange, P. Schachte, H. Søndergaard, and P. J. Stuckey, "Abstract interpretation, symbolic execution and constraints," in *Recent Developments in the Design and Implementation of Programming Languages, Gabbrielli's Festschrift, November 27, 2020, Bologna, Italy*, ser. OASIcs, F. S. de Boer and J. Mauro, Eds., vol. 86. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 7:1–7:19. [Online]. Available: https://doi.org/10.4230/OASIcs.Gabbrielli.7

[4] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976. [Online]. Available: https://doi.org/10.1145/360248.360252

[5] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, 2018. [Online]. Available: https://doi.org/10.1145/3182657

[6] A. Arusoaie, D. Lucanu, and V. Rusu, "A generic framework for symbolic execution," in *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, ser. Lecture Notes in Computer Science, M. Erwig, R. F. Paige, and E. V. Wyk, Eds., vol. 8225. Springer, 2013, pp. 281–301. [Online]. Available: https://doi.org/10.1007/978-3-319-02654-1_16

[7] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[8] E. S. Lowry and C. W. Medlock, "Object code optimization," *Commun. ACM*, vol. 12, no. 1, pp. 13–22, 1969. [Online]. Available: https://doi.org/10.1145/362835.362838

[9] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[10] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1165–1178, 2008. [Online]. Available: https://doi.org/10.1109/TCAD.2008.923410

[11] F. Rabbi, L. M. Kristensen, and Y. Lamo, "Static analysis of conformance preserving model transformation rules," in *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018*, S. Hammoudi, L. F. Pires, and B. Selic, Eds. SciTePress, 2018, pp. 152–162. [Online]. Available: https://doi.org/10.5220/0006602601520162

[12] L. Lambers, K. Born, F. Orejas, D. Strüber, and G. Taentzer, "Initial conflicts and dependencies: Critical pairs revisited," in *Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig*, ser. Lecture Notes in Computer Science, R. Heckel and G. Taentzer, Eds., vol. 10800. Springer, 2018, pp. 105–123. [Online]. Available: https://doi.org/10.1007/978-3-319-75396-6_6

[13] L. Lambers, D. Strüber, G. Taentzer, K. Born, and J. Huebert, "Multi-granular conflict and dependency analysis in software engineering based on graph transformation," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 716–727. [Online]. Available: https://doi.org/10.1145/3180155.3180258

[14] S. Gabmeyer and M. Seidl, "Lightweight symbolic verification of graph transformation systems with off-the-shelf hardware model checkers," in *Tests and Proofs - 10th International Conference, TAP@STAF 2016, Vienna, Austria, July 5-7, 2016, Proceedings*, ser. Lecture Notes in Computer Science, B. K. Aichernig and C. A. Furia, Eds., vol. 9762. Springer, 2016, pp. 94–111. [Online]. Available: https://doi.org/10.1007/978-3-319-41135-4_6

[15] B. J. Oakes, "A symbolic execution-based approach to model transformation verification using structural contracts," Ph.D. dissertation, McGill University, 2018.

[16] G. Kulcsár, "A compass to controlled graph rewriting," Ph.D. dissertation, Darmstadt University of Technology, Germany, 2019. [Online]. Available: http://tuprints.ulb.tu-darmstadt.de/9304/