A Look-Ahead Strategy for Rule-Based Model Transformations

Lars Fritsche*, Erhan Leblebici*, Anthony Anjorin† and Andy Schürr*
*TU Darmstadt, Germany
Email: firstname.lastname@es.tu-darmstadt.de
†Paderborn University, Germany
Email: anthony.anjorin@upb.de

***Abstract*—Model-Driven Engineering (MDE) promises an increase in the maintainability and extensibility of modern software systems. Furthermore, it encourages the usage of multiple models of the same system to simplify the involvement of different stakeholders, tools, and domains. The concurrent engineering of such multiple models requires, however, support for consistency management including consistency restoration and consistency checking of and between existing models. Bidirectional transformation (bx) approaches are able to address these challenges with various paradigms that come along with different dis-/advantages. Rule-based techniques, as one of these paradigms, provide a declarative approach to implementing such consistency management solutions. However, rule-based approaches rely on finding an appropriate sequence of rule applications, e.g., such that the transformation process does not lead to a dead-end. In case of a wrong choice of rule applications, backtracking (of arbitrary depth) is in general not a satisfactory solution as it has exponential runtime in the number of rule applications. We, therefore, propose a novel look-ahead strategy for rule-based bx approaches that governs the rule application process by generating additional application conditions based on the current and possible future state of relevant edges in the model. We demonstrate and evaluate our approach on a compact yet non-trivial scenario that requires careful decision making among rule applications.**

***Index Terms*—Model-driven development**

I. Introduction

Model-Driven Engineering (MDE) has become an important technique to cope with the increasing complexity of modern software systems. Its consequent use is said to improve both the quality of developed software as well as the effectiveness of the development process. However, as a software system evolves and becomes more complex, so do the underlying models that represent different views onto the system. While each view has its own characteristics and priorities, they normally share information that may be altered and changed such that it no longer complies with other related models. To cope with these changes and keep those related models consistent to each other is of particular interest for an integrated development process. Hence, tools are needed that analyse models for compliance with their correlated counterparts and, in case consistency has been violated, are able to transform the models into an updated consistent version without incurring unintended loss of information. As different models are maintained by different stakeholders, model transformations often have to be ”bidirectional”, i.e., executable in either direction. In general, a pair of transformations in reverse directions (usually referred to as forward and backward transformations) as well as consistency checking between two given models are needed to tackle consistency challenges in an MDE landscape. These tasks are addressed by bidirectional model transformation (bx) approaches. Bx implementations can be categorized into three paradigms, namely, bidirectional programming languages (e.g., BiGUL [1]), constraint-based approaches (e.g., JTL [2]) and rule-based approaches (e.g., TGG [3]). Bidirectional programming languages are powerful with respect to expressiveness and provide more fine-grained control over the transformation process. However, they usually entangle high-level transformation goals with the low-level (e.g., computation-focused) instructions of the underlying programming language. In contrast, constraint-based approaches are based on relationship specifications that are then checked and enforced with constraint solving techniques which in general yields good results but suffers from performance issues due to large search spaces as compared to the other two techniques [4]. Rule-based approaches, finally, offer a compromise as their performance is better than constraint-based approaches, and transformations are implemented in a more declarative way as compared to bx programming languages. In general, they rely on finding an appropriate sequence of rule applications to perform a transformation. This, however, can lead to dead-ends and undesired transformation results due to wrong choices when applying the rules. A naive but expensive solution is to use backtracking to find the valid sequence or to correct a ”wrong” rule application. It is, however, in general too expensive and scales poorly to use such brute-force methods. Thus, the amount of available choices of rule applications must be limited by applying advanced static analysis techniques.

In this paper we will focus on Triple Graph Grammars (TGGs), which is a declarative, rule-based bx-approach to define consistency between two correlated models. However, as a rule-based technique, TGGs also have to avoid dead-ends. To cope with this, we propose in this paper a look-ahead strategy for rule-based model transformations on the example of a TGG specification. It was inspired by look-ahead techniques of string parsing approaches which were early adapted to improve graph grammar parsers. Furthermore, we conduct experiments with our look-ahead strategy and evaluate its added-value with respect to reliability of model transformations, i.e., if dead-ends are reliably avoided. Last but not least, the performance of our approach is compared to an existing look-ahead strategy based on so-called ”filter NACs” [5] with rather limited search space reduction capabilities and a naive TGG implementation without any look-ahead at all.

The rest of the paper is organized as follows: In Chapter II we will introduce our running example and TGGs. Chapter III introduces a novel look-ahead strategy that brings a minor overhead for forward (and analogously backward) transformations for the sake of more reliable transformations. In case of more rule application-intensive tasks such as consistency checking, the strategy even yields a substantial performance gain by reducing the search space of rule applications. In Chapter IV we present the results of our approach with respect to success rates and performance. Subsequently, in Chapter V we will discuss related work. Finally, in the last chapter we will summarize the results and give an overview over future work.

II. Running Example and Fundamentals

Our running example is depicted in Figure 1. It is inspired by an example from the example zoo provided by [6] and is concerned with the consistency between class diagrams (metamodels) in the Eclipse Modeling Framework (EMF) and a custom documentation structure. On the left side, an EMF class diagram model consisting of two *EPackages* and four *EClasses* is depicted while on the right side, we have two *Folders* and four *Doc*-files. The blue dotted connections between the EMF and documentation model represent traceability relationships between related elements in different models. In detail, the example consists of an *EPackage* which contains two interfaces, namely `Serializable` and `Observable` that are of type *EClass* (note the attribute values interfaces = true). Furthermore, the *EPackage* contains additionally a sub-*EPackage* with the two *EClasses* `Person` and `Employee` which are concrete classes. There exist multiple inheritance relations from `Person` to `Serializable` and `Observable` where `Employee` inherits from `Person`. The point of interest in this example is the definition of multi-inheritance which may be defined in two ways, namely as generalization or as realization. EMF does not differ between generalization and realization and represents both relations uniformly. To still be able to differ between both, we have to take a look at the target of such an edge to see if it is defined as an interface or a concrete class. The documentation model on the right side represents the same information as the left side with the minor difference that there are two explicit types of inheritance links, namely *realizes* and *generalizes*. Both links are visualized in conformance to UML syntax. Generally speaking, it is highly beneficial to be able to automatically extract a meaningful documentation from an existing project and furthermore, to keep it consistent. It might also make sense to first define a documentation structure and to extract an EMF skeleton that conforms to the documentation so that developers may work more purposively. Last but not least, for given projects and documentations in ongoing software projects it is often not clear which parts are still consistent to each other and which are not. To discover (in-)consistencies and return a valid mapping, that covers as many elements of both sides as possible, is sometimes crucial when the alternative would be to start the documentation from scratch. These scenarios may be handled by bx approaches in general; however, we will focus on Triple Graph Grammars (TGGs) as our tool of choice which is a rule-based bx technique.

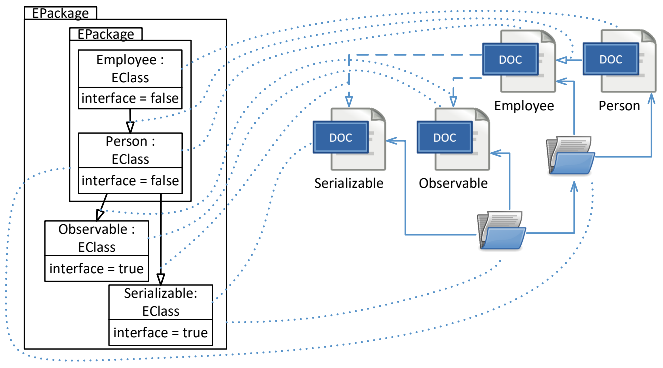
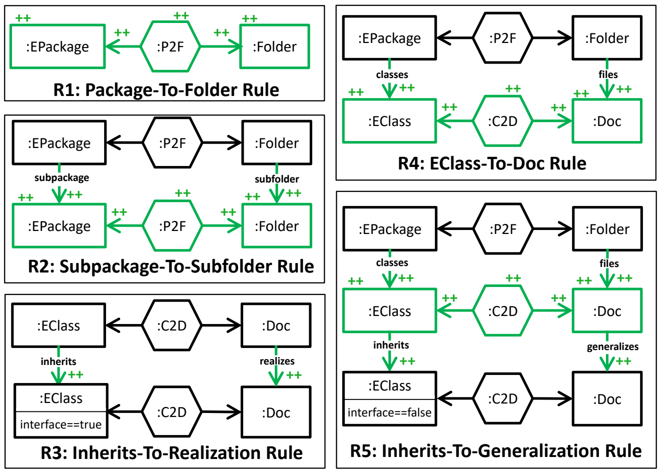


Fig. 1. Running example

TGGs [3] are a declarative, rule-based bidirectional transformation approach which was initially proposed by Schürr. While ”meta-models” define the structure of models in a MDE context, a TGG specification defines consistency between instances of two meta-models. A TGG consists of a finite set of transformation rules that define how consistent pairs of both source and target model co-evolve. Consistency is defined between structural features of the source and target meta-model via a third one which is called the correspondence meta-model. Each rule defines a pattern where elements are said to be in source, correspondence or target domain if they belong to the corresponding meta-model.

Figure 2 depicts the TGG rules of our running example. We focus on an excerpt including structural features of and between *EPackage* and *EClass*. The target side consists of *Folders* and *Doc*(-Files) which ideally should document the structure of Ecore packages and classes with their inheritance relations. Each TGG rule consists of two kinds of structural features, namely nodes and edges. Every node and edge has a state which determines if it is a context or created element. The first state is depicted as black nodes and edges of a pattern, in the following referred to as context elements, that are taken as pre-requisite for each rule application. Green nodes and edges, additionally



R1: Package-To-Folder Rule
R2: Subpackage-To-Subfolder Rule
R3: Inherits-To-Realization Rule
R4: EClass-To-Doc Rule
R5: Inherits-To-Generalization Rule

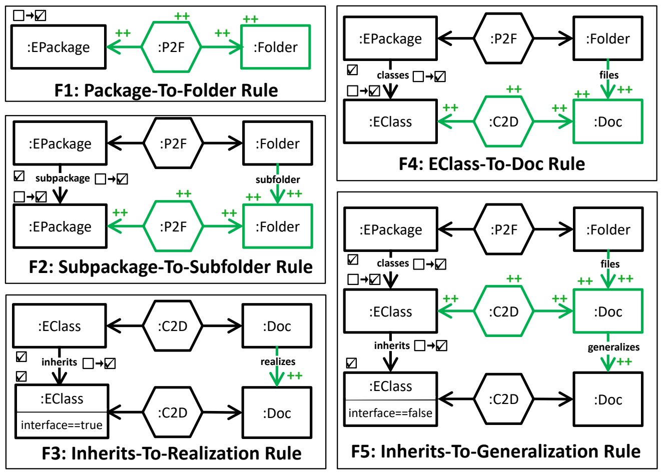
Fig. 2. EMF To Doc Example: TGG Rules

marked by '++', define which structural features are to be created in conformance to the pattern in case that the pre-requisite of the pattern is fulfilled. A major goal of defining patterns is to find matches, where a match is defined as a valid mapping of pattern elements to model elements. Thus, a match defines an occurrence of the pattern in a model.

In the following, we will explain each of the five rules in detail: 1) The first rule R1 is axiomatic, due to the lack of context elements. It does not define any pre-requisites but creates an instance of type *EPackage* and a corresponding instance of type *Folder*. This correspondence is expressed as an instance of type *E2F* which is visualized by a hexagon and connects both of *EPackage* with *Folder*. 2) Given an existing correspondence between elements as defined in R1, R2 creates a sub-*EPackage* with a corresponding sub-*Folder* under the pre-requisite that a parent *EPackage* and *Folder* have been found previously. 3) The other rules can be read analogously, where R3 creates an *inherits* link on the source domain and a corresponding *realizes* link on the target domain if the target *EClass* of the *inherits* link is an interface. 4) R4 creates an *EClass* and a corresponding *Doc* given already processed *EPackage* and *Folder* instances. 5) Last but not least, R5 defines that an *EClass* and an *inherits* link on source side are created together with a corresponding *Doc* and a *generalizes* link on target side, under the condition that the context *EClass* node is not an interface.

Note that the distinction between R3 and R5 is made due to the EMF specification of *EClass* that internally describes whether it is an interface or a real class. In Java a class may generalise only one other class which prohibits multiple inheritance using the extends directive. It is, however, possible to mimic multiple inheritance using interfaces and the implements directive which is extensively used by EMF. For this reason R3, in combination with R1, is able to define correspondence of multiple *inherits* links originating from one *EClass*. In contrast, R5 also defines correspondence for an *inherits* link but together with the *EClass* from which it origins. In summary, we demand that generalization can only be translated once together with the originating node. Furthermore, we allow multiple realizations as long as the targeted *EClass* is an interface.

A TGG is a consistency specification for which operations such as forward and backward transformations or even consistency checks can be derived. This means that in order to perform those operations, we have to operationalize the TGG rules first. In the following we will explain how to operationalize TGG rules to perform forward transformations; backward transformations are handled analogously. For a forward transformation, the source side is given as an input model and the target side is to be derived. Hence, in contrast to model generation, we do not create new elements on the source side, but rather mark those elements that have already been translated. This implies that elements on the source side of a TGG rule are all context elements for a forward operationalization. Figure 3 depicts all "forward rules" for



F1: Package-To-Folder Rule
F2: Subpackage-To-Subfolder Rule
F3: Inherits-To-Realization Rule
F4: EClass-To-Doc Rule
F5: Inherits-To-Generalization Rule

Fig. 3. EMF To Doc Example: Forward Rules

our given TGG. Note that all former created nodes on source side have been converted to context elements. The difference between those elements that are context in the plain TGG rule and those we converted is depicted as marking signs on each node and edge on source side. ☑ indicates that this element has to have a marking which in turn means that it must be already translated. ☐ → ☑ indicates that this rule marks an unmarked element to be translated if it is applied. These forward rules are able to transform a given source model into a corresponding target model in accordance with the original TGG rules.

With the given forward rules this process is not deterministic as is demonstrated in Figure 4. It depicts a source model with two *EPackages*, namely `root` and `sub` where `sub` is a sub-package of `root`. Given this model and our forward rules, there are two rule application sequences that are

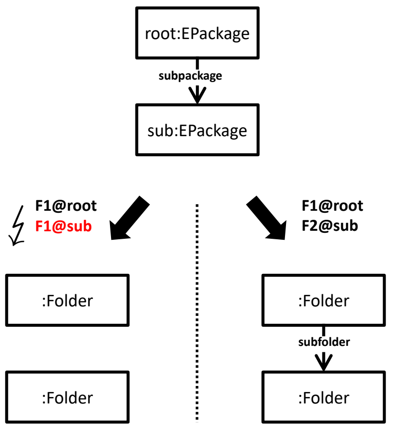


Fig. 4. The upper model may be translated with two rule application sequences. The left sequence ends in a state where the *subpackage* edge can not be translated with any forward rule. The right sequence translates all elements.

possible and are depicted at the bottom of the Figure. The sequence of the left is an application of the axiomatic F1 rule on both *EPackages*; however, this results in a state where the *subpackage* edge can no longer be translated with any forward rule. The reason for this is that there are no forward rules that will enable us to translate the edge after we applied F1 to `sub`. The second sequence on the right depicts a valid rule application sequence that does indeed translate all elements. It translates both *EPackages* plus the *subpackage* link between them if we apply F1 to `root` and F2 to `sub` and the link. The choice of which rule application sequence to use is not trivial, but may be solved through backtracking by revoking former application, if a dead-end has been detected. However, backtracking can result in exponential runtime in model size which quickly becomes infeasible [5]. A standard approach (inspired from string parsing) is to use a look-ahead to reduce the cases in which backtracking is required.

Last but not least, we need to operationalize the rules such that for a given source and target model we find correspondences between both which is often referred to as consistency check. In the context of TGGs this means that we have to construct the correspondence model given both source and target models. In general, this means that we have to collect matches of the source and target sides of each TGG rule, respectively. Given these two sets, we have to find correspondences if possible and in an optimal way. This means that a pairing has to be determined between matches coming from the source side of a rule to those of the target side. The challenge is the complexity of this task regarding possible pairing candidates and that certain pairs of matches contradict each other. This problem can be solved efficiently with constraint solving techniques as was proposed by Leblebici et al. [7] as long as the search space of possible pairing candidates remains moderate.

III. Look-Ahead Strategy

For forward and backward transformations, the challenge is how to find a sequence of rule applications that does not lead to a dead-end where certain elements remain untranslatable. To cope with this, advanced static analysis techniques [5], [8] can be used to filter sequences such that ideally all applicable rules avoid dead-ends by construction. Let us assume that the TGG rules of Figure 2 have been operationalized for forward transformations as explained in the previous chapter. Furthermore, we assume that our input model is equivalent to the one presented in Figure 4. It consists of two instances of *EPackage* where `sub` is a *subpackage* of `root`. In the left case we translate both *EPackages* using F1; however, this lets the *subpackage* edge remain untranslated as there is no forward rule that translates a single *subpackage* edge. The right case represents a viable sequence of rule applications. The `root` element still gets translated using F1 but `sub` and *subpackage* are both translated using F2, leaving no element or edge untranslated. Summarizing, the conflict arises from a local choice of applicable rules where one choice leads to a dead-end.

To prevent this, Herman et al. [5] proposed a static analysis with the goal to guide the transformation process towards a control flow that avoids untranslated edges. The analysis consists of three steps which are explained in the following: 1) First, we analyse those situations where a node gets translated taking into account the possibility of other rules being able to translate all adjacent edges. 2) For every detected node and for every possible edge, the analysis tries to find a forward rule which might be applied in the future to translate this edge. In general this means that a rule has to be found where the current to-be-translated element is a context element and the problematic edge is translated. If we find such a rule the analysis terminates. 3) If such a ”saving” rule can not be found then the current rule is extended by a negative application condition (NAC) that forbids the existence of such an edge for the given node.

These NACs are called ”filter NACs” as their intention is to filter those rule applications that would otherwise certainly lead the model transformation process into a dead-end. An example of this is given in Figure 5. It depicts the F1 rule which is extended by a NAC that prohibits the existence of an incoming *subpackage* edge for the `p1` element, such an edge would always remain untranslated in case that this rule is applied. This solves the previously encountered problem as for now there will never be an F1 match that translates `sub` leaving the only translating forward rule to be the one that does not lead to a dead-end.

Now, if we extend our example model as depicted in Figure 6, we will very likely encounter a new dead-end case. The example consists of the `sub` *EPackage* plus two *EClasses* in `sub`, namely `Employee` and `Person`.

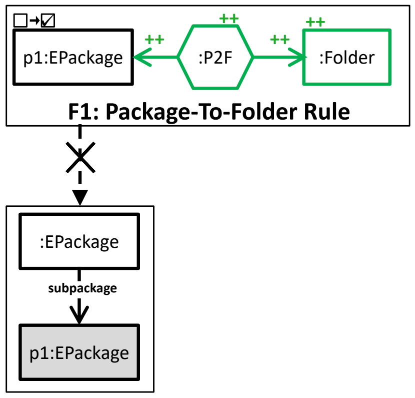


Fig. 5. Package-To-Folder - NAC

Additionally, Employee inherits from Person. Translating this model requires translating the *EPackage* first which is handled the same way as before and may be disregarded, which leads us to the translation of both *EClasses*. Now, we have to use F3, F4 and F5 to translate the remaining elements and the *inherits* edge between them but, as in the previous example, we may end up in a dead-end depending on which forward rules we apply. A valid rule sequence would be to apply F4 to Person following the application of F5 to Employee and the remaining *inherits* edge. But, starting with F4 applied to Employee would lead to an untranslated *inherits* edge as there would never again be any rule able to process the edge. The reason why the previous analysis does not hold in this case is that the F3 rule is theoretically able to translate an *inherits* edge wherever it originates from. Thus, there won't be any NAC generated for the F4 rule that would guide the translation process safely.

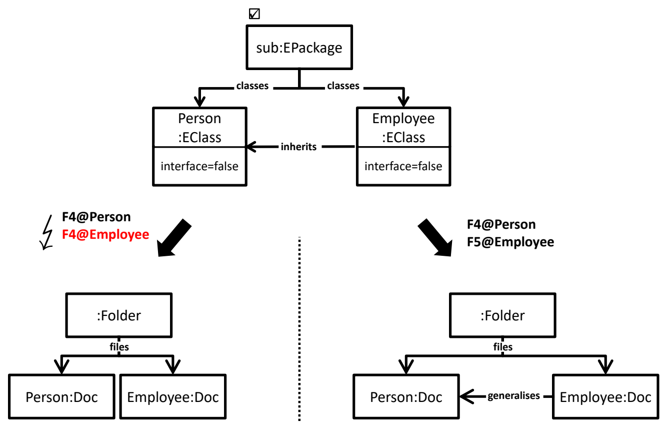


Fig. 6. The upper model may be translated with two rule application sequences. The left sequence ends in a state where the *inherits* edge can not be translated with any forward rule. The right sequence translates all elements.

To rule out these cases we propose a more sophisticated look-ahead strategy which also takes into account that context and attribute constraints might prevent a false "saving" rule from later application, thus, leaving elements untranslated. The steps are described in the following: 1) Similar to the previous analysis we extend the given rules such that for every created element we search for existing edges that are not translated by this rule. 2) If such an edge does not exist, the rule may be applied directly. 3) However, in case there are such potential edges, the analysis demands that there also exists a valid match of the source[1] side of a saving rule. Hence, we demand that after applying this rule, the edge is still translatable in the future. This concept of demanding the existence of a certain pattern C, that extends a match of another pattern P, is called positive application condition (PAC) which intuitively represents an "if-then" relationship.

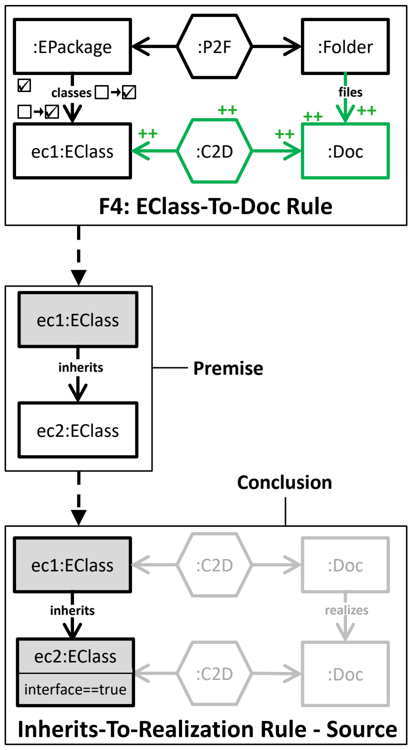


Fig. 7. EClass-To-Doc - PAC

Figure 7 depicts the approach applied to F4. Analogously to the former analysis, we inspect ec1 as a node whose translation might lead to an untranslated *inherits* edge. In contrast to the former approach, the possibility that the edge might be translated by F3 no longer suffices. We, thus, demand that in case the *inherits* with ec2 as target exists, there must also exist a concrete match of the source side of the F3 rule. This ensures that the edge can be translated by F3 in the future, which means that we have a real look-ahead for specific edges to prevent them from not being translatable later on.

[1]target for backward translation

In case there are no such saving rules, this approach is equivalent to the former static analysis because the Premise may be true but the Conclusion will always be false. This means that the resulting Premise and Conclusion are reduced to a (filter) NAC forbidding the existence of a certain edge as it would remain untranslatable. Hence, if no saving rules are found, the PAC-based look-ahead strategy falls back to filter NACs. This means, however, that filter NACs are a subset of our new PAC-based look-ahead strategy.

The PAC-based look-ahead strategy can also be used to improve the performance of consistency check. As was explained in the previous chapter, consistency checking relies on the discovery of matches of the source and target side of each rule. By applying our strategy on both source and target side, we can filter matches analogously to the forward transformation case. Hence, by decreasing the number of possible matches, we decrease the search space of possible pairings. This, finally, increases the efficiency as the number of possible candidates increases quadratically in the number of matches. The experimental results that support this conclusion are presented in the next chapter.

The implementation of this strategy appears computationally more expensive at a first glance as for every rule which is applied, we have to find also occurrences for those patterns that have been added by the analysis. However, given a closer look specifically at the generated PAC, the matches for all saving rules must have already been collected before, which is in general the bottleneck of rule-based bx tools. Thus, before applying a certain match, we only have to check for the existence of these pre-calculated matches regarding possible saving rules.

There are cases when this analysis does not suffice and the translation process still ends up in a dead-end. Any scenario which requires an analysis of a sequence of future rule applications of length greater than one to avoid a dead-end is not considered here. Handling these scenarios requires a look-ahead greater than one; however, this requires the generation of nested application conditions. For a look-ahead of k, a nesting level of k+1 is needed, which is not covered by our implementation that only supports simple application conditions with a Premise and a Conclusion pattern for a look-ahead of one. Nonetheless, the extension to nested application conditions can be handled analogously by generating PACs for every Conclusion.

Another scenario is connected to our assumption that dead-ends in a transformation process correspond to single untranslatable edges. This means that we always identify such cases when there is a possibly untranslatable edge; however, the current implementation may underestimate the rule application sequence needed to translate all adjacent edges. It does not incorporate that there are dependencies between Conclusion patterns (saving rules) where the translation of one remaining edge contradicts the translation of another, yet, untranslated edge.

IV. Experimental Results

In the preceding chapter we presented both filter NACs and our novel PAC-based look-ahead strategy. Both approaches were shown on a TGG specification based on our running example. In the following, we will evaluate the new approach by comparing it to filter NACs and an implementation that does not use any context analysis techniques. We, therefore, state three research questions that are investigated with our experiments: ***RQ1:*** For each approach, how does consistency check scale with increasing model size? ***RQ2:*** Does the new approach introduce an advantage to model transformations regarding success rates? ***RQ3:*** How is the performance of model transformations comparing all three approaches and does the new analysis technique introduce any noticeable overhead?

As our tool of choice we use eMoflon [9] a graph transformation tool that supports TGGs. In the current version it is based on an incremental pattern matcher [10] and an ILP solver [11]. The experiments are executed on a machine with an Intel Core i5-3550, 16GB memory and Ubuntu 16.04.

The models used for forward transformation and consistency check are generated in conformance to the previously introduced TGG specification. Thus, by applying the original TGG rules directly, we are able to simultaneously build up all three domains. Consequently, we can create models of arbitrary size from scratch that are consistent with respect to our TGG specification by applying rules this way. Hence, we define the model complexity as the number of rule applications that were used to generate a model. Note, however, that this model size is not equal to the number of nodes since R3 does not create a node but rather an edge between two existing ones; thus, the model size is an indicator for the complexity that results not only from nodes but also from inheritance relationships. For both tested scenarios we generate 20 variations for each examined model size and repeat execution of each variation five times. The resulting times are the average values over the median times of each variation.

For consistency check, we will measure the time to highlight performance differences between a naive approach, filter NACs and our PAC-based look-ahead strategy. Since consistency check is handled as a constraint solving problem, we do not have to measure the success rate because even if irrelevant matches are processed, the optimal solution of the problem does not change.

RQ1: Figure 8 depicts the measured times for consistency check, including the search for rule occurrences on both source and target model and the constraint solving to find an

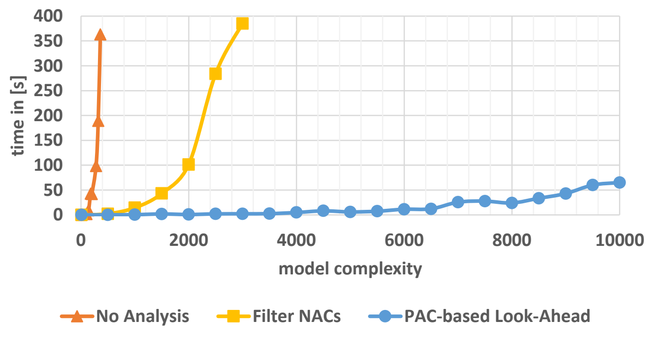


Fig. 8. Consistency Check - Measured times

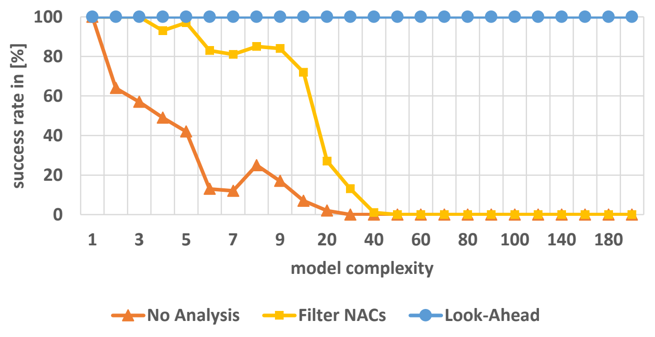


Fig. 9. Forward transformation - Success rates

optimal mapping. The biggest model size[2] using the naive approach is 360 with a runtime of ~6 minutes. Using filter NACs, consistency check is able to process a maximal model size of 3000 in slightly more than 6 minutes. Finally, the PAC-based look-ahead enables consistency check to process a maximal model size of 10000 within approximately one minute.

The results show a significant performance gain of the PAC-based look-ahead and filter NACs compared to an implementation without analysis. The naive approach tends to be impractical for even moderate model sizes as the runtime increases too steeply. The filter NACs perform better in comparison to the naive approach; however, for models bigger than 2000 the runtime increases substantially. Finally, the runtime of PAC-based look-ahead rises very slightly and stays close to one minute for model sizes of 10000. These measurements show that both filter NACs and our PAC-based look-ahead are able to decrease the complexity of the consistency check task. Regarding the PAC-based look-ahead, we can even further increase the performance and process a multiple of the maximal model size of filter NACs.

Regarding forward transformation, we are interested in two measurements. First, the success rate of the transformation where an execution is considered as failed if not all elements were translated[3]. Second, pattern matching is usually the bottleneck of modern rule-based tools and both filter NACs and our PAC-based look-ahead strategy introduced new patterns that have to be found in the input model. Thus, we are interested in how expensive the new approach is with respect to the runtime performance.

RQ2: Figure 9 depicts the success rate of each experiment. All three plots start at a success rate of 100 percent for a model size of one as this is equivalent to a model with one single package and dead-ends are not to be expected. However, after adding a sub-*EPackage* or a new *EClass* the success rate for the naive approach drops to 60 percent and it almost reaches 0 percent for model sizes bigger than 30. In comparison to the former plot, filter NACs tend to perform much better as long as we transform solely *EPackages*, since filter NACs can not cope with dead-ends that occur during translation of an *EClass* hierarchy. The success rate drops from almost 100 percent to 80 for a model size of 6 and rapidly falls down to almost 0 percent for model sizes 10 to 40. In contrast, the PAC-based look-ahead strategy is always able to translate all elements such that no nodes or edges stay untranslated.

The success rates of filter NACs and an implementation without analysis drop rapidly until they reach ~0 percent at a model size of 40. In contrast, the PAC-based look-ahead strategy remains constantly at 100 percent. This means that the transformation does not end up in a dead-end and that all elements were translated.

RQ3: Given those results, we now want to take a look at the runtime performance which is depicted in Figure 10. As can be seen, all three plots are not easy to keep apart as they lie very closely. The maximum times for a maximal model size of 4500 are 89 seconds for the naive approach, 97 seconds for filter NACs, and 97 seconds for the PAC-based look-ahead strategy.

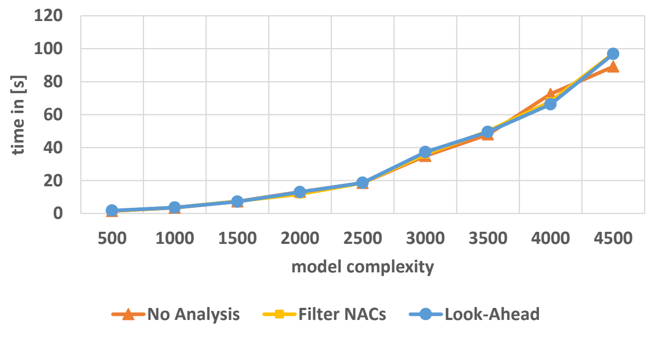


Fig. 10. Forward transformation - Measured times

The measurements indicate that all three approaches have a very similar performance. The naive implementation

[2]Equivalent to the number of rule applications used to generate the model
[3]Note that each input model is by construction fully translatable

using no analysis tends to be slightly faster; however, the performance gain comes at the price of a very low success rate. Comparing filter NACs with the PAC-based look-ahead, there is no considerable difference of performance. Note, however, that the naive approach and filter NACs leave certain elements untranslated. This means that the runtime may be affected such that it appears slightly faster than it would if all elements were processed. Nonetheless, since the runtime of our PAC-based look-ahead strategy does not differ noticeable from the other approaches, we conclude that the introduced overhead is not relevant and may be disregarded.

Threads to Validity. *External validity* is of major interest but it requires investigation of further non-trivial case studies. We argue, however, that the presented example represents a significant challenge that emerges frequently with increasing case complexity. Furthermore, the results are based on an incremental pattern matching framework and a constraint solver. It remains to be investigated if the performance changes considerably with the use of other frameworks and solvers. *Internal validity* has to be investigated further as the generation of test cases was not deterministic. This means that, despite of the number of repetitions and variations for each model size, the results have a certain variance of around +/- 15 percent.

V. Related Work

In this paper we proposed a PAC-based look-ahead strategy for rule-based bx approaches. This strategy was inspired by predictive string parsing techniques which use a variable look-ahead to resolve issues that arise while parsing. Early approaches that tried to adapt string parsing techniques to graph grammars were inspired by the Early-Style-Parser [12] which is a top-down-parser that uses dynamic programming to determine rule application entry points. As such it is related to our TGG approach; however these parsers were restricted to context-free grammars and suffered from a high computational complexity [13], [14]. Nonetheless, recent work also shows that predictive parsing yields promising results for graph grammar parsing, by substantially reducing the runtime complexity for specific scenarios [15]. To our knowledge, we are the first to elevate the idea of using a look-ahead for rule-based bx approaches to reduce the space of conflicting rule applications.

Our implementation is based on eMoflon as our tool of choice; however, there are other TGG tools such as TGG Interpreter [16] or MoTE [17]. MoTE is restricted to conflict-free TGG specifications, implying that for any translatable element there is only one applicable rule at any time. Using this restriction, MoTE does not need backtracking or a look-ahead to resolve issues as they forbid ambiguous specifications in general [18]. TGG Interpreter is not limited to conflict-free TGG specifications; however, the tool is not able to use any backtracking or look-ahead to avoid conflicts. Hence, it does not guarantee that its results are correct and that all elements were successfully translated [18].

Our implementation of filter NACs is based on the work of Klar et al. [8] and Hermann et al. [5]. Both papers analysed TGGs with negative application conditions and introduced a strategy to avoid several rule application sequences that would leave elements untranslated. Klar et al. focussed on the identification of dangling edges, i.e., edges that may remain untranslated if a node is translated using a specific rule. If such an edge is detected, the analysis reacts as was explained in Chapter VI and tries to generate a NAC that forbids the application of the rule in this particular case. In contrast, Hermann et al. used the critical pair analysis [19] to argue on how to avoid several critical pairs of rules that conflict with each other; however, both yield similar results using NACs and can not cope with scenarios in which it is necessary to use PACs.

VI. Conclusion and Future Work

In this paper we introduced a novel PAC-based look-ahead strategy for rule-based bx techniques. Although the approach was explained and exemplary implemented for a TGG specification, it is not limited to TGGs. In fact, it offers a strategy for rule-based approaches in general to govern the transformation process in order to avoid dead-ends or even enhance the performance. To evaluate our approach, we chose a compact excerpt of the Eclipse Modeling Framework together with a corresponding documentation structure as our running example. We conducted experiments using a naive TGG implementation, an existing statical analysis called filter NACs and our PAC-based look-ahead strategy. In the case of forward transformations, we showed that the new approach is able to avoid dead-ends that both other approaches encounter. Furthermore, the results indicate that the costs of the new strategy do not raise significantly regarding the performance. For consistency check we showed that the new approach is able to greatly increase the performance with respect to runtime and maximal model size as it reduces the overall number of possible rule applications in the search space.

For future work, it remains open to investigate more challenging scenarios, e.g., more complex models and consistency specifications with intensive look-ahead requirements. Another important topic is the extensibility of this approach. Examples for such extensions might be, a look-ahead factor of more than one, or a generalisation of this approach to also handle conflicts that do not only arise from adjacent edges of nodes but also from nodes themselves. Last but not least, the evaluated implementations are based on an external incremental pattern matcher framework and an ILP solver. For future work, we would like to investigate how the performance changes with respect to other frameworks.

References

[1] H.-S. Ko, T. Zan, and Z. Hu, "Bigul: A formally verified core language for putback-based bidirectional programming," in *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial*

Evaluation and Program Manipulation, ser. PEPM '16. New York, NY, USA: ACM, 2016, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/2847538.2847544

[2] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, *JTL: A Bidirectional and Change Propagating Transformation Language*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 183–202. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19440-5_11

[3] A. Schürr, *Specification of graph translators with triple graph grammars*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 151–163. [Online]. Available: http://dx.doi.org/10.1007/3-540-59071-4_45

[4] A. Anjorin, Z. Diskin, F. Jouault, H. Ko, E. Leblebici, and B. Westfechtel, "Benchmarx reloaded: A practical benchmark framework for bidirectional transformations," in *Proceedings of the 6th International Workshop on Bidirectional Transformations co-located with The European Joint Conferences on Theory and Practice of Software, BX@ETAPS 2017, Uppsala, Sweden, April 29, 2017.*, 2017, pp. 15–30. [Online]. Available: http://ceur-ws.org/Vol-1827/paper6.pdf

[5] F. Hermann, H. Ehrig, U. Golas, and F. Orejas, "Efficient analysis and execution of correct and complete model transformations based on triple graph grammars," in *Proceedings of the First International Workshop on Model-Driven Interoperability*, ser. MDI '10. New York, NY, USA: ACM, 2010, pp. 22–31. [Online]. Available: http://doi.acm.org/10.1145/1866272.1866277

[6] "Bidirectional transformations - ecore2html example," http://bx-community.wikidot.com/examples:ecore2html, 2017, accessed: 2017-06-29.

[7] E. Leblebici, A. Anjorin, and A. Schürr, *Inter-model Consistency Checking Using Triple Graph Grammars and Linear Optimization Techniques*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 191–207. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-54494-5_11

[8] F. Klar, M. Lauder, A. Königs, and A. Schürr, *Extended Triple Graph Grammars with Efficient and Compatible Graph Translators*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 141–174. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17322-6_8

[9] E. Leblebici, A. Anjorin, and A. Schürr, *Developing eMoflon with eMoflon*. Cham: Springer International Publishing, 2014, pp. 138–145. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08789-4_10

[10] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi, "Road to a reactive and incremental model transformation platform: three generations of the viatra framework," *Software & Systems Modeling*, vol. 15, no. 3, pp. 609–629, 2016.

[11] "Gurobi solver," http://www.gurobi.com, 2017, accessed: 2017-06-29.

[12] J. Earley, "An efficient context-free parsing algorithm," *Commun. ACM*, vol. 13, no. 2, pp. 94–102, Feb. 1970. [Online]. Available: http://doi.acm.org/10.1145/362007.362035

[13] H. Bunke and B. Haller, *A parser for context free plex grammars*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 136–150. [Online]. Available: http://dx.doi.org/10.1007/3-540-52292-1_10

[14] G. Costagliola, M. Tomita, and S.-K. Chang, "A generalized parser for 2-d languages," in *Proceedings 1991 IEEE Workshop on Visual Languages*, Oct 1991, pp. 98–104.

[15] F. Drewes, B. Hoffmann, and M. Minas, *Predictive Top-Down Parsing for Hyperedge Replacement Grammars*. Cham: Springer International Publishing, 2015, pp. 19–34. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21145-9_2

[16] J. Greenyer and J. Rieke, "Applying advanced tgg concepts for a complex transformation of sequence diagram specifications to timed game automata," in *Proceedings of the 4th International Conference on Applications of Graph Transformations with Industrial Relevance*, ser. AGTIVE'11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 222–237. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34176-2_19

[17] D. Blouin, A. Plantec, P. Dissaux, F. Singhoff, and J.-P. Diguet, *Synchronization of Models of Rich Languages with Triple Graph Grammars: An Experience Report*. Cham: Springer International Publishing, 2014, pp. 106–121. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08789-4_8

[18] E. Leblebici, A. Anjorin, and y. v. Andy Schürr and Stephan Hildebrandt and Jan Rieke and Joel Greenyer, journal=ECEASST, "A comparison of incremental triple graph grammar tools."

[19] D. Plump, "Essentials of term graph rewriting," *Electronic Notes in Theoretical Computer Science*, vol. 51, pp. 277 – 289, 2002, gETGRATS Closing Workshop. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S157106610480210X