Detecting and Preventing Power Outages in a Smart Grid using eMoflon

Sven Peldszus, Jens Bürger, Daniel Strüber
{speldszus,buerger,strueber}@uni-koblenz.de

University of Koblenz and Landau

Abstract

We present a solution to the Outage System Case of the Transformation Tool Contest 2017, based on the bidirectional model transformation language *eMoflon*. The case comprises two tasks, in which the goal is to produce custom model views on a set of input models from a smart-grid system, eventually allowing power outages to be detected and prevented. To facilitate understandability, our solution uses eMoflon's declarative transformation rules. Moreover, to address performance, we use eMoflon's incremental execution mode, as well as a general-purpose-language preprocessing step contributing to the identification of elements participating in the considered views.

1 Introduction

Model-driven engineering emphasizes the use of models to facilitate the development of software systems. As the involved systems grow in size and complexity, maintaining a single model to represent the overall system becomes infeasible. Complex scenarios usually involve a *multitude* of models to represent distinct concerns of the system and its subsystems, working together in some well-defined way.

When working with such a multitude of models, two issues may arise: First, the information relevant for specific tasks may be scattered over multiple models. To represent this relevant information in an easily digestible way, custom *model views* on a set of models are required. Second, in the case of frequently changing models, it is necessary to update these views after each modification. However, computing all views from scratch each time an underlying model changes can be prohibitively expensive. A solution for this issue are *incremental views* that reuse results from earlier view computations. Since the creation of incremental views is essentially a model transformation problem, it is promising to address it with the available transformation tools.

In this context, the *Outage System Case* [Hin17] of the 2017 Transformation Tool Contest provides an interesting benchmark scenario, based on a smart-grid setting. To capture information about a power supply system, the benchmark features three kinds of models: The *Common Information Model* (CIM) describes physical components, measurement data, control and protection elements. The *Companion Specification for Energy Metering* (COSEM) model supports data exchange for meter reading, tariff and load control in electricity metering. The *Substation Configuration Description Language and Logical Node* (SCL/LN) model fosters interoperability of intelligent electronic devices in substation automation systems.

The Outage System Case comprises two tasks that each involve an incremental view on this set of models: (1) an *outage detection* task to identify power outages based on a lack of responsiveness by a smart meter, and

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

(2) an *outage prevention* task based on disturbances that can be observed by comparing the current phasor data to historical ones. Technically, each of the views required for these tasks can be expressed as a model, based on a view-specific meta-model (see Fig. 1a and Fig. 3 in [Hin17]).

In this paper, we present a solution for the Outage System Case based on *eMoflon* [ALPS11], a bidirectional model transformation language and tool based on the paradigm of triple graph grammars (TGGs, [Sch95]). The main use-case of TGGs is the synchronization of two models, called left-hand side and right-hand side model (LHS, RHS). The user specifies a set of *rules*, which map LHS metamodel classes to RHS metamodel classes by using a dedicated *correspondence model*. From these correspondence rules, one can automatically generate transformation code for realizing model synchronization in two directions: the forward direction for propagating changes of the LHS to the RHS model, and the backward direction for propagating changes of the RHS to the LHS model. For both direction, an incremental execution mode is supported.

In our application of eMoflon to the Outage System Case, we interpret the set of input models as the LHS, and the output view as the RHS. Since the views in our scenario are not editable, we only need the forward direction in order to produce and update the required views. We provide the code for our solution at `https://github.com/SvenPeldszus/rgse.ttc17.emoflon.tgg`; the SHARE image URL is found there as well.

The rest of this paper is structured as follows: In Sect. 2, we provide an overview of our solution. In Sect. 3, we show details, including example rules. We evaluate our solution in Sect. 4, and conclude in Sect 5.

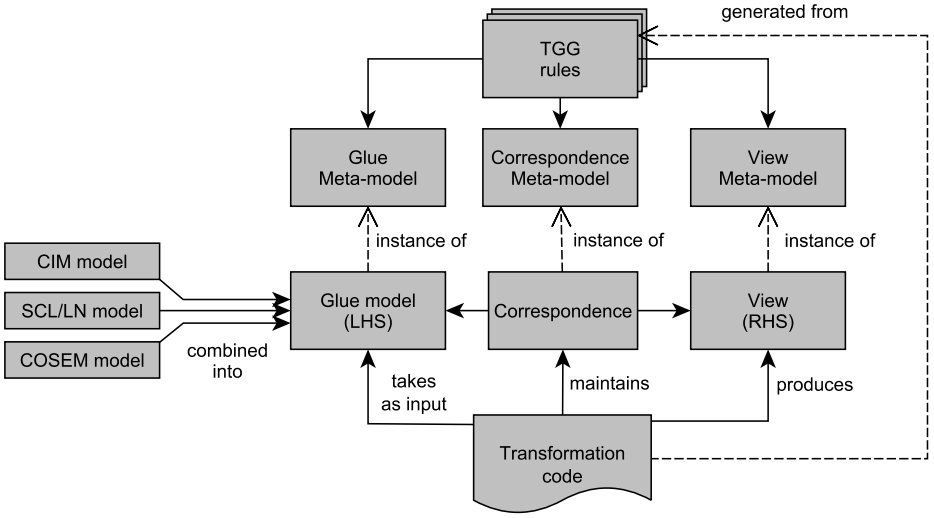

Figure 1: Overview of our view computation.

2 Overview

Fig. 1 depicts an overview of the view computation. In eMoflon, the LHS has to be an instance of exactly one meta-model. To accomplish this, we introduce a *glue meta-model* which is basically a root class with containment references to the root nodes of the three given meta-models. Based on this meta-model, the three input models can be combined into one model that we call *glue model*, instantiating the glue meta-model.

A set of TGG rules describes the TGG used to construct the model view. These rules are composed of graph patterns, where the contained nodes and edges refer to elements of the involved meta-models, specifically, the glue, correspondence and view meta-models. Consequently, each TGG rule includes a pattern of glue-model elements which is matched against the input glue model. Whenever the glue-model pattern of a rule can be fully matched, the respective model elements of the view and the correspondence model are created. The transformation is realized by running transformation code generated by eMoflon from the TGG rules.

Fig. 2 shows the overall process of our solution, including the support for incremental updates. As input, we use the base models as provided by the TTC case resources. Firstly, a preprocessing step is carried out, performing the unification of the input models as well as certain convenience optimizations we will explain later.

The forward transformation generated by eMoflon is applied to the preprocessed model, producing as output the view. Delta models (leading to updated models such as *Model'*) are then successively processed. After the preprocessing, the TGG application synchronizes the view by only considering the changed parts of the input model, and applying the changes as specified in the correspondence and RHS parts of the rules to produce the updated view models.

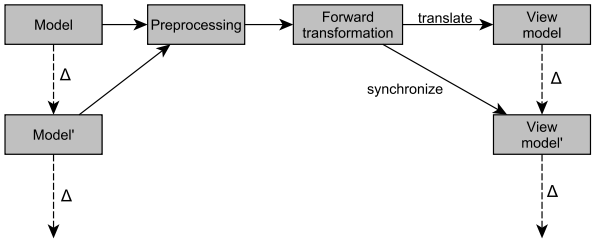

Figure 2: Overview of our incremental transformations.

3 Details

In this section, we present the details of our solution. For each task, we first explain the preprocessing step, followed by an explanation of the transformation.

3.1 Task 1: Outage Detection

Preprocessing. The preprocessing step has two goals: First, it unifies the input models into the glue model used as input for the transformation. Second, it enriches this model with auxiliary references and model elements to support a more compact specification and a more efficient matching.

To unify the input models, we put them all inside the same container element, a dedicated element called `Root`. A `Root` holds a CIM model, a SCL/LN model, and a COSEM model. In addition, it has some references to enable a more convenient retrieval of involved elements: in particular, all `MeterAsset`s of the CIM model are stored in a reference called `assets`. Finally, it holds a set of `MeterAssetsAndPhysicalDevicesPair`s to identify pairs of `PhysicalDevice`s and `MeterAsset`s with the same ID. In a purely declarative specification, finding these pairs can be highly inefficient, since the underlying implementation may consider *all* pairs of `PhysicalDevice`s and `MeterAsset`s, filtering them to keep only those with the same ID. In contrast, we compute these pairs in a Java method of 15 lines of code, where we consider each `PhysicalDevice` and `MeterAsset` element only once, while maintaining a certain hash map. Doing so, we can identify the required pairs in linear time.

Transformation. The view produced in task 1 includes three distinct classes: `EnergyConsumer`, `Location`, and `PositionPoint`. Our transformation includes a TGG rule for producing each of them, amounting to three basis TGG rules in total. These basis rules are further extended by *rule refinements* [ASLS14] for dealing with special cases, such as the *if-then-else* in the view specification.

To understand eMoflon rules, it is important to consider that they declaratively specify *how the considered models should be related* in the end. They do *not* specify the process of how these relationships are established—instead, this process is realized in the generated forward and backward transformation code. Specifically, the upper half of Fig. 3 illustrates the `EnergyConsumer` rule. The basis rule represents lines 2 to 4 of the ModelJoin source of Task 1. The rule includes green nodes and edges for specifying newly created elements, and black nodes and edges for specifying existing ones. LHS model elements are shown with a yellow background, and RHS elements are shown with a red background. Elements of the correspondence model are indicated as diamonds. On the bottom, the rule includes two attribute conditions, specifying equality between pairs of values. Lax equality (`laxEq`) is used to compare a boolean to an integer, based on an underlying Java implementation. Note that the shown graphical representation of rules is a read-only representation generated by eMoflon; the actual rule creation and development was performed in eMoflon's textual syntax.

In a nutshell, this rule specifies the correspondence between a `MeterAsset-PhysicalDevice` pair and an `EnergyConsumer`. The forward transformation generated from the rule establishes that for each such pair in the input model, a corresponding `EnergyConsumer` will be generated in the output model. The attribute conditions ensure that the attributes of the newly created `EnergyConsumer` obtains the values according to the specification. Note that `MeterAssetPhysicalDevicePair` was not contained in the original model; as discussed above, we added in a preprocessing step to enable a more compact specification and improved performance during matching.

The refinement `EnergyConsumerWithID` shown in the lower half of Fig. 3 deals with the following fact: For the `ServiceDeliveryPoint` of the considered `MeterAsset`, an existing `EnergyConsumer` may or may not exist. If it exists, it needs to be treated according to the view specification. We implement this distinction using rule refinement. The underlying semantics is to "glue together" the refinement with all of its super-rules, based on

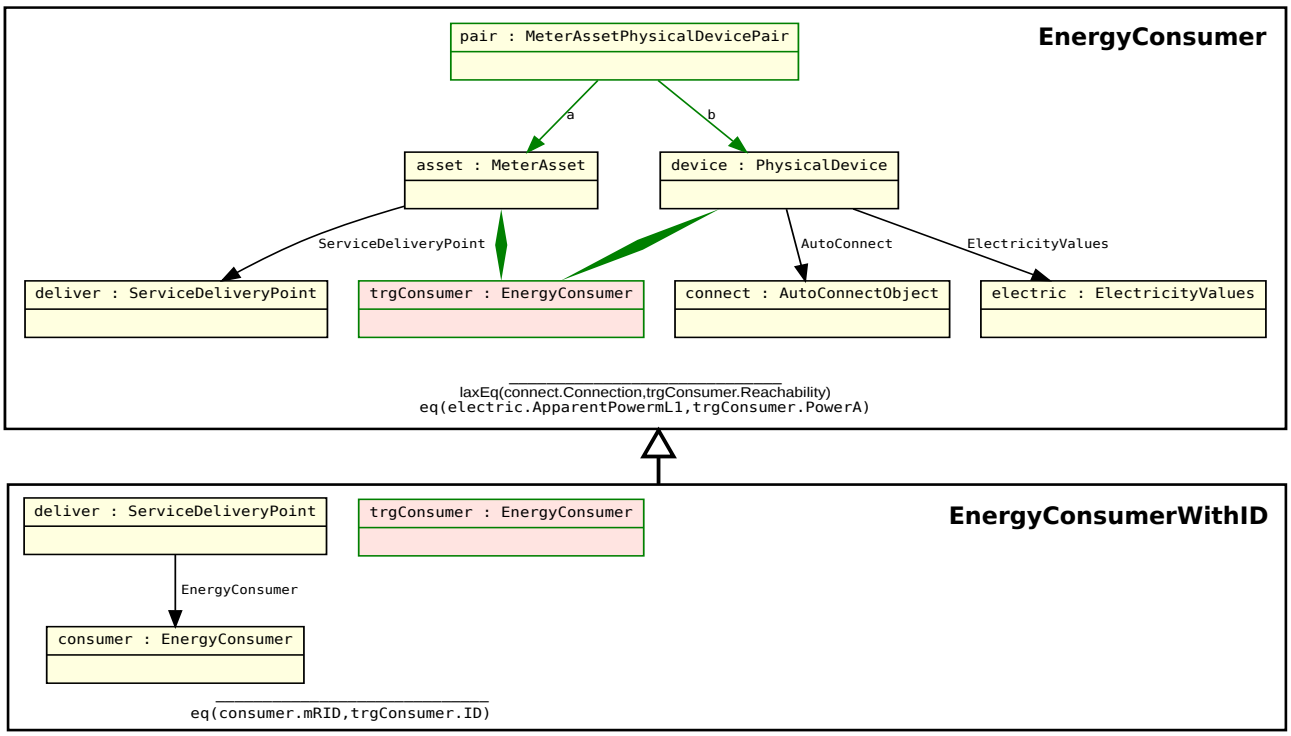

Figure 3: Example of a TGG rule and its refinement to compute the view in Task 1.

name equalities of the included elements. In the example, this allows us to apply the additional attribute condition to the target model, enabling us to propagate the LHS `EnergyConsumer`'s `mRID` to the RHS `EnergyConsumer`'s `ID` field, in case the former exists.

3.2 Task 2: Outage Prevention

Preprocessing. The preprocessing for task 2 follows the same basic ideas as in task 1. We unify the input models, populate certain convenience references, and add auxiliary model elements to facilitate a compact specification and optimized matching. The custom parts of interest in the glue meta-model is a reference `mmxus` keeping track of the `MMXU` instances in the `SCL/LN` model, and a `MeterAssetMMXUPair` class maintaining pairs of `MeterAsset`s and `MMXU`s with the same ID. The purpose of these parts is the same as in task 1.

Transformation. Similar to task 1, we organize the overall TGG rule specification into rules for specific classes from the view meta-model, and rule refinements for dealing with special cases thereof. This time, more different elements types are involved, and we have fewer special cases per element type: We distinguish 19 rules in total, only 3 of them having refinements.

Fig. 4 illustrates two of these rules. The `Location` rule specifies the correspondence of `Location` elements in the LHS and RHS models, which are related to an existing `MeterAsset` and `PMUVoltageMeter`, respectively. To explain why we need to specify the related `MeterAssetMMXUPair` as well, we must dive a bit deeper into how the matching process works: In essence, matching works in two steps; first the LHS and then the RHS is matched. In TGGs, it is assumed that for each LHS match, we can identify a corresponding RHS match. However, in our particular scenario, the issue is that we do not transform all of the `MeterAssets`, but only some of them—those with a corresponding `MMXU`. Therefore, specifying the `MeterAssetMMXUPair` (which, again, is produced during preprocessing) saves us from matching `MeterAsset`s with no RHS counterpart, which would lead to an error.

Based on the correspondence manifested in the `Location` rule, the `Position` rule specifies the correspondence of `PositionPoint` classes in the LHS and RHS. Again, we use attribute conditions to specify the equality of the involved attributes, which will be used during the forward transformation to propagate the values from the LHS element to the RHS counterpart. In the overall forward transformation process, all `Location`s will be transformed first, which enables all `PositionPoint`s to be transformed subsequently.

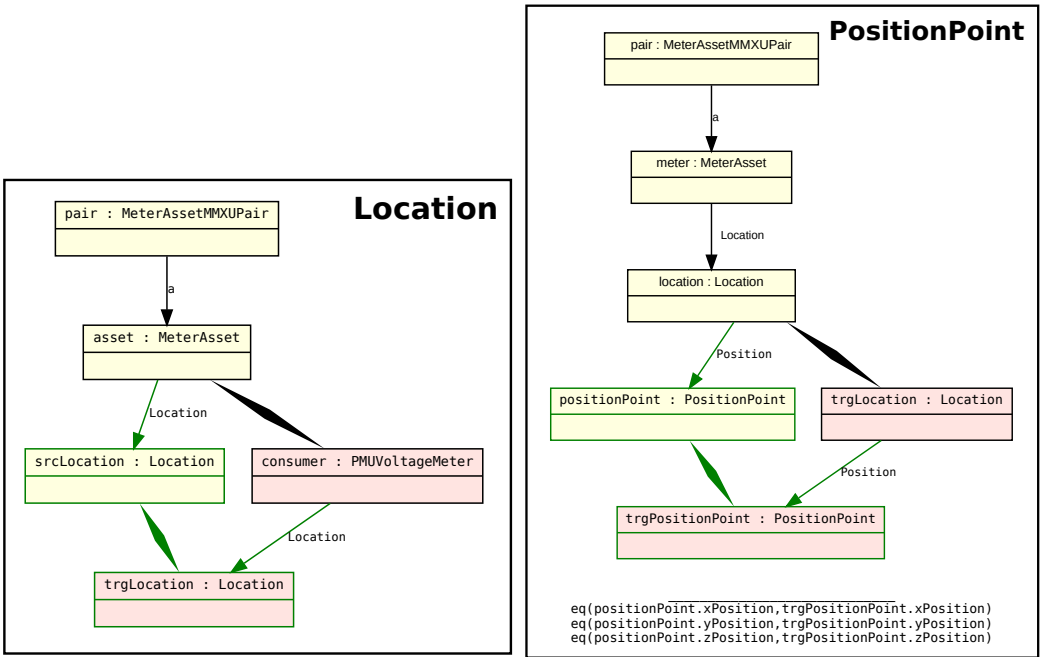

Figure 4: Two related TGG rules for producing the view in Task 2.

4 Evaluation

In this section, we evaluate two aspects of our solution: conciseness and performance.

According to the specification, conciseness is to be evaluated in terms of the number of Lines of Code (LoC). The measurement shall include "the model views and glue code to actually run the benchmark", while the code for converting the change sequences can be ignored. We present the results in Table 1a. Since our solution comprises declarative rule specification as well as some imperative orchestration code, we list both separately, amounting to 1312 LoC for rules and 364 for code. The glue code for running the benchmark comprises 119 lines.

For the performance evaluation, we show the execution times for the solution when applied to the input change sequences. All experiments have been performed on an Ubuntu 16.04 LTS PC with an Intel i5-6200U and 8 GB of RAM of which our implementation was allowed to allocate upto 6 GB for the experiments.

Part	**LoC**
Rules	1312
Orchestration code	364
Benchmark code	119
Total	1795

(a) Conciseness.

Input	**Mode**	**Time[ms]** det/prev
seq1.out000	batch	847.9/1139.2
seq1.delta001	inc	379.7/497.8
seq1.delta002	inc	387.9/598.2
seq1.delta003	inc	321.5/308.2
seq1.delta004	inc	246.1/477.5
seq1.delta005	inc	210.8/350.7

(b) Performance (1)

Input	**Mode**	**Time[ms]** det/prev
seq2.out000	batch	825.9/975.0
seq2.delta001	inc	350.2/504.9
seq2.delta002	inc	483.3/477.1
seq2.delta003	inc	281.4/312.2
seq2.delta004	inc	256.7/329.1
seq2.delta005	inc	217.1/235.5

(c) Performance (2)

Table 1: Results.

5 Conclusion

We presented a solution to the TTC 2017 Outage System Case based on eMoflon. Based on a simple preprocessing step in Java, we facilitated a compact declarative specification as well as an efficient execution of our solution. We believe that the overall solution combines "the best of both worlds", by providing a mostly declarative specification, while using imperative code for performance-critical parts of the rule application process. To support the incremental setting, our solution inherits the benefits of eMoflon's incremental execution mode.

References

[ALPS11] Anthony Anjorin, Marius Lauder, Sven Patzina, and Andy Schürr. eMoflon: Leveraging EMF and Professional CASE Tools. *Informatik*, 192:281, 2011.

[ASLS14] Anthony Anjorin, Karsten Saller, Malte Lochau, and Andy Schürr. Modularizing triple graph grammars using rule refinement. In *International Conference on Fundamental Approaches to Software Engineering*, pages 340–354. Springer, 2014.

[Hin17] Georg Hinkel. The Outage System Case for Incremental Model Views. *10th Transformation Tool Contest (TTC 2017)*, 2017.

[Sch95] Andy Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1995.