securityandtrust.lu

Triple Graph Grammars in the Large for Translating Satellite Procedures – Extended Version

Frank Hermann Université du Luxembourg / SnT, Luxembourg
Susann Gottmann Université du Luxembourg / SnT, Luxembourg
Nico Nachtigall Université du Luxembourg / SnT, Luxembourg
Hartmut Ehrig TU Berlin, Germany
Benjamin Braatz Université du Luxembourg / SnT, Luxembourg
Gianluigi Morelli SES, Luxembourg
Alain Pierre SES, Luxembourg
Thomas Engel Université du Luxembourg / SnT, Luxembourg
Claudia Ermel TU Berlin, Germany

Wednesday 23rd April, 2014

ISBN: 978-2-87971-128-7

Report No.: TR-SnT-2014-7

Triple Graph Grammars in the Large for Translating Satellite Procedures – Extended Version

Frank Hermann[1], Susann Gottmann[1], Nico Nachtigall[1], Hartmut Ehrig[2], Benjamin Braatz[1], Gianluigi Morelli[3], Alain Pierre[3], Thomas Engel[1], and Claudia Ermel[2]

[1] Interdisciplinary Centre for Security, Reliability and Trust, Université du Luxembourg, Luxembourg
firstname.lastname@uni.lu
[2] Technische Universität Berlin, Germany
firstname.lastname@tu-berlin.de

Abstract. Software translation is a challenging task. Several requirements are important – including automation of the execution, maintainability of the translation patterns, and, most importantly, reliability concerning the correctness of the translation.
Triple graph grammars (TGGs) have shown to be an intuitive, well-defined technique for model translation. In this paper, we leverage TGGs for industry scale software translations. The approach is implemented using the Eclipse-based graph transformation tool Henshin and has been successfully applied in a large industrial project with the satellite operator SES on the translation of satellite control procedures. We evaluate the approach regarding requirements from the project and performance on a complete set of procedures of one satellite.

Keywords: model transformation, software translation, refactoring, triple graph grammars, Eclipse Modeling Framework (EMF)

1 Introduction

Migration of software systems is an important but complex task, especially for enterprises that are highly dependent on the reliability of their running systems. The general problem is to translate the source code of a software that is currently in use into corresponding source code that shall run on the new system. Up to now, this problem was addressed based on manually written converters, parser generators, compiler-compilers or meta-programming environments using term rewriting or similar techniques. Model transformation based on triple graph grammars (TGGs) is a general, intuitive and formally well-defined technique for the translation of models [28,29,15]. While previous concepts and case studies were focused mainly on visual models of software and systems, this paper shows that model transformation based on TGGs provides a powerful technique for software translation as well. Since software systems are on average much larger

than visual models, we provide a general technique for efficiency improvement and show its applicability within a large scale industrial project.

The general idea of TGGs is to specify a language of integrated models. Such an integrated model consists of a model of the source domain, a model of the target domain, and explicit correspondence structures in the middle component. The source and target models in the present scenario are abstract syntax trees of source code. The operational rules for executing the translation are generated from the specified TGG and executed via the graph transformation tool Henshin [7]. TGGs are equivalent to a restricted class of plain graph transformation systems [8,15]. This restriction ensures the existence of the explicit correspondence structures and formal properties concerning correctness and completeness [17]. In this paper, we use rather simple and intuitive but non-trivial translation patterns. The full translation contains several more complex ones, e.g., for the reordering and regrouping of blocks. Translation strategies that are solely based on finding and replacing words (like e.g. Awk[3]) will fail due to the highly context-sensitive structural dependencies in the source code.

Within the research project *PIL2SPELL* with the industrial partner SES (Société Européenne des Satellites), we developed the general approach for software translation in this paper. SES is operating a fleet of 56 satellites manufactured by different vendors that often use their own proprietary programming language for automated operational satellite procedures. In order to reduce the high complexity and efforts during operation caused by this heterogeneity, SES developed the open source satellite language SPELL [30] (Satellite Procedure Execution Language & Library), which is nowadays used by more and more operators and may become a standard in this domain. The main aim of the project was to provide a fully automated translation of existing satellite control procedures written in PIL (Procedure Intermediate Language) of the satellite manufacturer ASTRIUM into satellite control procedures in SPELL.[4] Since the PIL procedures are already validated, the translation has to ensure a very high level of reliability in terms of fidelity, precision and correctness in order to minimise the efforts for revalidation. In our first contribution of this paper we propose and validate the use of TGGs for software translation in the *PIL2SPELL* project. Since the *PIL2SPELL* project is an industrial application of rather large size (more than 200 translation rules were specified), a technique was needed to improve the efficiency of the TGG rewriting method and tool. Hence, the second contribution of this paper is a general approach for improving efficiency of graph transformation systems applied to leverage TGGs for software translations in industry and we evaluate the implementation in Henshin [7]. This technical report is an extended version of the corresponding conference article [19] and provides full technical details on the formal constructions and full proofs.

Sec. 2 introduces our running example, Sec. 3 presents the general concept and Sec. 4 describes the applied TGG techniques. Thereafter, Sec. 5 presents results for improving the efficiency and scalability, and Sec. 6 evaluates the

[3] Awk Community: `http://awk.info/`

[4] In [18], we present a short overview of the *PIL2SPELL* project.

```
1  SELECT
2    CASE ($BATT = "HIGH")
3      CHECKTM(TEMP_C1)
4      CHECKTM(VOLT_D2 = 4)
5    ENDCASE
6    CASE ($BATT = "LOW")
7      SEND SWITCH_B1_B2
8        CHECKTM(VOLT3 = 5)
9      ENDSEND
10   ENDCASE
11 ENDSELECT
```

```
1 if (BATT == 'HIGH'):
2   GetTM('T TEMP_C1')
3   Verify([['T VOLT_D2', eq, 4]])
4 elif (BATT == 'LOW'):
5   Send(command = 'C SWITCH_B1_B2',
6        verify  = [['T VOLT3', eq, 5]])
7 #ENDIF
```

Fig. 1. Procedure written in *PIL* (left) and translated procedure in *SPELL* (right)

approach. Sec. 7 discusses related work and Sec. 8 provides a conclusion and discusses aspects of future work.

2 Case Study *PIL2SPELL*

We illustrate the methodology for software translation on some details of the project *PIL2SPELL*. Fig. 1 presents a simplified *PIL* procedure for battery maintenance and its translation in *SPELL*. Structures of the form `SELECT-CASE-ENDSELECT` are translated into structures of the form `if-elif-#ENDIF`. `SEND` instructions (lines 7-9) for sending telecommands to the satellite are mapped to corresponding `Send` statements with the same command-id as argument prefixed with a `C` (lines 5-6). Instructions for checking telemetry values (*PIL* instruction `CHECKTM`) are handled in three ways:

1. `CHECKTM(X)` (line 3): parameter checks without condition are used to retrieve and display a telemetry value from the satellite. They are translated into `GetTM` statements, where prefix `T` is added to the parameter (line 2).
2. `CHECKTM(X = Y)` (line 4): parameter checks with additional condition are used to verify telemetry values and are mapped to `Verify` statements with a corresponding condition (line 3).
3. `CHECKTM(X = Y)` (line 8): parameter checks within a `SEND` instruction are translated into a `verify` argument of the corresponding `Send` statement (line 6). △

Note that the translation is context-sensitive as it treats e.g. a `CHECKTM` instruction inside a `SEND` instruction differently from a not nested `CHECKTM` instruction. Moreover, PIL and SPELL use different concepts for calling subroutines. In order to respect the execution semantics, block structures of the form `STAGE..ENDSTAGE` in PIL have to be translated into two SPELL structures. The first one is a function call that remains in the main part and the second one is a function definition containing the translated body of the block structure and it is placed at the beginning of the SPELL procedure. This restructuring and reordering of information motivates to perform a separation of concerns by splitting the translation into parsing, translation and serialisation instead of using an integrated approach, where some of the phases are merged.

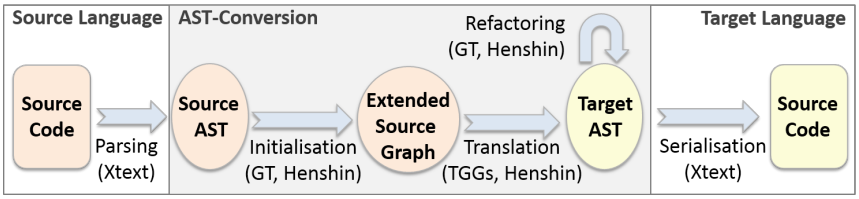

Fig. 2. Concept for software translation

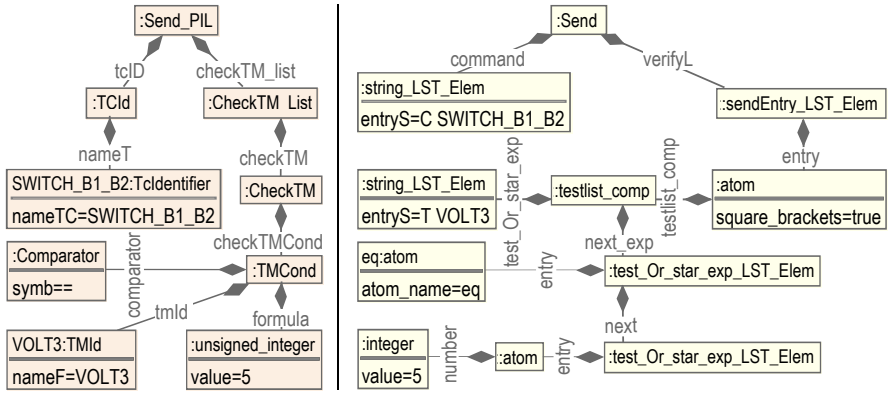

Fig. 3. Fragment of source AST (left) and target AST (right)

3 Concept for Software Translation

The general concept for software translation in Fig. 2 consists of the phases *parsing*, *AST conversion* (main phase), and *serialisation*. It is executed using the Eclipse Modeling Framework (EMF) tools *Xtext* [6] and *Henshin* [7]. Xtext supports the syntax specification of textual domain specific languages (DSLs), in particular of programming languages. Based on the EBNF (Extended Backus-Naur Form) grammar specification of a DSL and an additional formatting configuration, the Xtext framework generates the corresponding parser and serialiser. The parser checks that the input source code is well-formed and the serialiser ensures that the generated output source code is well-defined. The Xtext serialiser enables us to check and ensure that the output conforms to the given EBNF for the target language and that additional AST-specific formatting guidelines are respected. SES explicitly required the conformance to the SPELL EBNF and to SES formatting guidelines (e.g. alignment of list entries and semantic indentation), which goes beyond the power of generic template specification. Henshin is an Eclipse plugin supporting the visual specification and execution of EMF transformation systems, which is used for the main phase (AST conversion).

Example 1 (Parsing & Serialisation). Fig. 3 (left) shows a fragment of the AST obtained by parsing the *PIL* source code example in Fig. 1 (left, lines

7-9). Root node : `Send_PIL` represents the `SEND` – `ENDSEND` structure (lines 7-9) with telecommand-id (`SWITCH_B1_B2`, left branch) and telemetry parameter check (`CHECKTM`, right branch). Fig. 3 (right) shows the obtained SPELL AST fragment after translation. The serialisation of the SPELL AST yields the corresponding source code in Fig. 1 (right, lines 5-6). Root node : `Send` represents the `Send` statement with telecommand-id (`C SWITCH_B1_B2`) in the left branch and telemetry parameter verification argument (`verify`) in the right branch. △

The AST-conversion consists of three phases (see Fig. 2). The first and third phases (initialisation and refactoring) are general in-place transformations and are performed via plain graph transformation (GT) systems. The second phase (translation) is performed using a triple graph grammar (TGG), which is presented in detail in Sec. 4. Note that TGGs can be fully encoded as plain graph transformations [15]. The initialisation phase is used to extend the given AST of the source language with additional structures that simplify the specification of the translation rules in Phase 2. The refactoring phase refines the resulting AST in order to satisfy certain coding guidelines required in the target domain. These refactorings are specified by compact GT rules that also delete substructures. Employing a TGG for the refactoring phase instead would drastically increase the amount of rules.

To reduce the complexity of the translation rules, the initialisation phase is used to pre-process information and to create additional helper structures that store this information locally in the source AST. In our case study, the initialisation rules are used, e.g., to compute a global numbering for the subcomponents of a satellite procedure that are needed in SPELL. Moreover, we create explicit pointers from complex instructions to their subcomponents (see, e.g. Ex. 2).

As TGGs are non-deleting, the source model is preserved completely during the translation. The translation markers ensure that each element is translated exactly once. At each translation step, a substructure of the given AST is translated and trace links are created. The resulting fragments in the target domain are connected according to the tree structure of the input AST. These properties help to ensure that the resulting output graph has a tree structure and is in fact an AST.

4 Triple Graph Grammars with Henshin

In the following, we briefly review main concepts for model transformation based on TGGs [11]. A triple graph is an integrated model consisting of a source model, a target model and explicit correspondences between them. More precisely, it consists of three graphs G^S, G^C, and G^T, called source, correspondence, and target graphs, respectively, together with two mappings (graph morphisms) $s_G{:}\, G^C \to G^S$ and $t_G{:}\, G^C \to G^T$. The two mappings in G specify a correspondence relation between elements of G^S and elements of G^T.

Triple graphs are related by triple graph morphisms $m : G \to H$ [28,11] consisting of three graph morphisms that preserve the associated correspondences (i.e., left diagrams in Fig. 4 commute). Triple graphs are typed over a

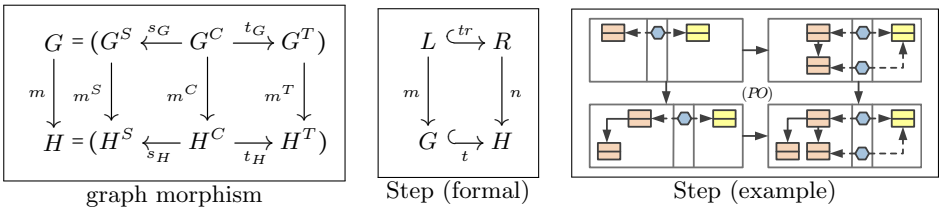

Fig. 4. Triple graph morphism and transformation step

triple type graph TG and attributed according to [11]. For a triple type graph $TG = (TG^S \leftarrow TG^C \rightarrow TG^T)$, we use $\mathcal{L}(TG)$, $\mathcal{L}(TG^S)$, and $\mathcal{L}(TG^T)$ to denote the classes of all graphs typed over TG, TG^S, and TG^T, respectively.

A triple graph grammar $TGG = (TG, S, TR)$ consists of a triple type graph TG, a triple start graph S and a set TR of triple rules, and generates the triple graph language of consistently integrated models $\mathcal{L}(TGG) \subseteq \mathcal{L}(TG)$ with consistent source and target languages $\mathcal{L}(TGG)^S = \{G^S \mid (G^S \leftarrow G^C \rightarrow G^T) \in \mathcal{L}(TGG)\}$ and $\mathcal{L}(TGG)^T = \{G^T \mid (G^S \leftarrow G^C \rightarrow G^T) \in \mathcal{L}(TGG)\}$. TG^C differentiates the possible types of correspondences.

A triple rule specifies how a given consistently integrated model can be extended simultaneously on all three components yielding again a consistently integrated model. It is non-deleting and therefore, can be formalised as an inclusion from triple graph L (left hand side) to triple graph R (right hand side), represented by $tr : L \hookrightarrow R$ with $tr = (tr^S, tr^C, tr^T)$. Applying a triple rule tr means to find a match morphism $m : L \rightarrow G$ and to perform a triple graph transformation step $G \overset{tr,m}{\Longrightarrow} H$ yielding triple graph H defined by the gluing construction[5] in Fig. 4 where the occurrence of L in G is replaced by the occurrence of R in H and glued to the remaining graph elements) [29]. Moreover, triple rules can be extended by application conditions for restricting their application to specific matches [15].

The operational forward translation rules for executing forward model transformations are derived automatically [15] from the TGG. A forward translation rule tr_{FT} and its original triple rule tr differ only on the source component: elements (nodes, edges or attributes) created by tr become elements that are preserved and marked as "translated" by the forward translation rule.

Example 2 (Operational Triple Rules). Fig. 5 shows screenshots (tool Henshin [7]) of some generated forward translation rules of the *TGG* for *PIL2SPELL* in short notation. Left- and right-hand side of a rule are depicted in one triple graph and the elements to be created have the label ⟨++⟩. Translation attributes are indicated by label ⟨`tr`⟩. The depicted rules are typical operational rules of average rule size. Rule (1) translates an existing `Instruction_LST_Elem` node into its corresponding `stmt_LST_Elem` node. Both node types are containers for

[5] Formally, this is a pushout diagram (PO) in the category of triple graphs.

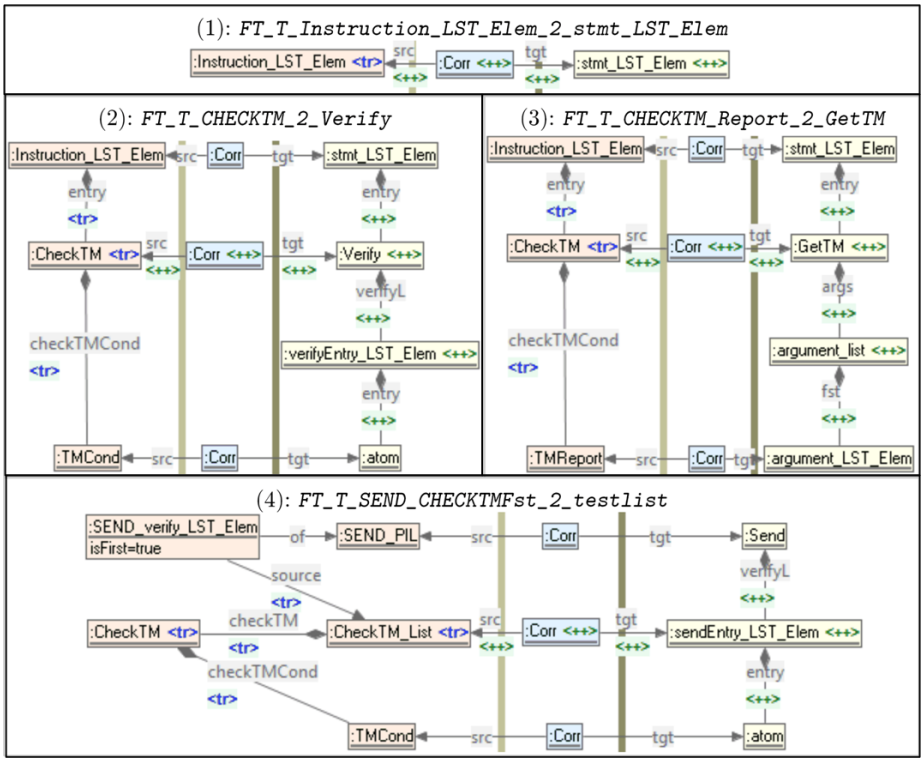

Fig. 5. Forward translation rules (generated by Henshin)

specific instructions and statements. Rules (2) and (3) depend on rule (1) as they use the `stmt_LST_Elem` nodes as context.

Rules (2)-(4) are some of the rules that translate `CHECKTM` instructions. They depend on further rules for the translation of their parameters (`TMCond` or `TMReport`). Depending on the parameter type, the respective SPELL statement is created, i.e., telemetry conditions (`TMCond`) yield a `Verify` statement, telemetry reports (`TMReport` - label without condition) yield a `GetTM` statement and telemetry conditions within a `SEND` instruction become an argument in a `verify` list of the corresponding `Send` statement. This corresponds to items 1–3 in Sec. 2. Rules (2) and (3) translate `CHECKTM` instructions that are not embedded within a specific context while rule (4) translates `CHECKTM` instructions within a `SEND` instruction.

Note that the node type `SEND_verify_LST_Elem` is created in the initialisation phase as helper structure and used to mark exactly those `CheckTM` elements that handle a telemetry condition (`TMCond`). The remaining `CheckTM` elements of a `SEND` instruction are translated to `GetTM` statements outside the scope of the SPELL `Send` statement. △

A *forward translation sequence* $(G^S, G_0 \overset{tr^*_{FT}}{\Longrightarrow} G_n, G^T)$ is given by an input source model G^S, a transformation sequence $G_0 \overset{tr^*_{FT}}{\Longrightarrow} G_n$ obtained by executing the forward translation rules TR_{FT} on $G_0 = (G^S \leftarrow \varnothing \rightarrow \varnothing)$, and the resulting target model G^T obtained as restriction to the target component of triple graph $G_n = (G^S \leftarrow G^C \rightarrow G^T)$. A *model transformation* based on forward translation rules $MT{:}\mathcal{L}(TG^S) \Rrightarrow \mathcal{L}(TG^T)$ consists of all forward translation sequences. Note that a given source model G^S may correspond to different target models G^T. In order to ensure unique results, we presented in [15] how to use the automated conflict analysis engine of AGG for checking functional behaviour of model transformations.

5 Leveraging TGGs for Software Translations in Industry

As described in the previous section, the basic execution algorithm for forward translations based on TGGs does not use any kind of pre-defined order on rules. For medium and large scale projects, the application of rules in a non-deterministic way would result in poor efficiency. In this section, we present a general approach for graph transformation systems, with which we leverage TGGs for larger software translations. This concerns grammars containing more than 200 rules, like the manually specified rules for the *PIL2SPELL* project that were derived from a document of correspondence patterns (small corresponding source code fragments). The approach is orthogonal to the analysis and reduction of conflicts via filter NACs for TGGs [15]. Both approaches can be combined - the second one improves the rules directly while the first provides a structuring technique on them.

The main observation is that the efficiency of the execution can be improved significantly by analysing the potential dependencies. For example, rules (2) and (3) in Fig. 5 can only be applied after rule (1) was applied to translate the node of type `Instruction_LST_Elem`. Our strategy is partly inspired by several existing optimisations in TGG implementations [20] and dependency analysis for graph transformation systems [14]. It generalises the idea of precedence triple graph grammars [25] from node type dependencies towards general rule dependencies and works also for TGGs with attributes. It uses the general formal results on critical pair analysis [9,24] including the case of transformation rules with application conditions. Practically, we use the critical pair analysis engine of the tool AGG [31] for determining the dependencies and conflicts between the rules. Based on the results, we group those rules together that show cyclic dependencies or conflicts. The resulting set of groups of rules shows a partial order that we linearise to a complete order. Finally, we apply this grouping and ordering technique to the set of forward translation rules.

In order to group the rules of a given rule set R, their sequential dependencies and conflicts are represented by a dependency-conflict graph $DCG(R)$ containing the rules as nodes and rule dependencies/conflicts as edges. A pair of rules (r_1, r_2) is in conflict if there exists a critical pair for (r_1, r_2) [9], i.e., there

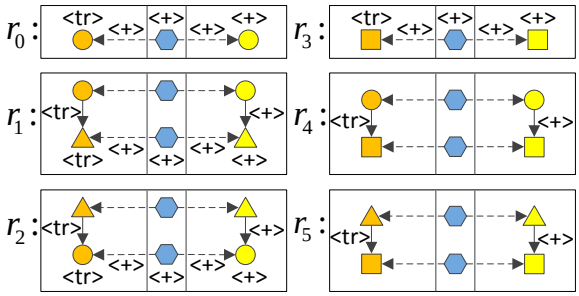

Fig. 6. Forward translation rules (dependency-conflict graph)

are two parallel dependent transformation steps $t_1 = G_0 \overset{r_1}{\Longrightarrow} G_1$, $t_2 = G_0 \overset{r_2}{\Longrightarrow} G_2$. A pair of rules (r_1, r_2) is sequentially dependent if there is a transformation sequence $t = (t_1; t_2) = G_0 \overset{r_1}{\Longrightarrow} G_1 \overset{r_2}{\Longrightarrow} G_2$, where t_2 sequentially depends on t_1 (produce-use or forbid-create dependency). Note that the order is relevant for sequential dependencies. Both concepts can be analysed statically using the tool AGG [31]. The graph $DCG(R)$ may contain cycles. These cycles are used to define non-overlapping clusters of rules leading to the acyclic dependency-conflict cluster graph $CLG_{DC}(R)$. By $N(G)$ we denote the set of nodes of a graph G.

Definition 1 (Dependency-Conflict Cluster Graph). *Let R be a set of rules, then we define:*

- dependency-conflict graph $DCG(R)$ *with nodes* $N(DCG(R)) = R$ *and edges* $E_{DCG} = \{(r \to r') \mid (r, r')$ *is a sequentially dependent pair* $\} \cup \{(r \to r'), (r' \to r) \mid \exists$ *a critical pair for* $(r, r')\}$,
- *for* $r \in R$ *the* dependency-conflict cluster $[r]_{DC} = \{r\} \cup \{r' \in R \mid \exists$ *a path* $(r \to \ldots r' \ldots \to r)$ *in* $DCG(R)\}$,
- dependency-conflict cluster graph $CLG_{DC}(R)$ *with nodes* $N(CLG_{DC}(R)) = \{c \mid c = [r]_{DC} \wedge r \in R\}$ *and edges* $E = \{(c \to c') \mid \exists\ r \in c, r' \in c' : (r \to r')$ *in* $DCG(R)\}$. △

Example 3 (Translation rules for dependency analysis). We use the set of translation rules in Fig. 6 to illustrate the construction of the dependency-conflict cluster graph. For simplicity, the rules translate abstract geometrical forms (circles, triangles and rectangles) and their interconnections (the forms are connected to sequences of alternating circles and triangles beginning with a circle as root element where each circle and triangle may be connected with a rectangle). Nodes and edges marked with $<\texttt{tr}>$ are translated by creating those nodes and edges marked with $<+>$ when applying a rule. Rule r_0 translates the circle root element and rules r_1 and r_2 the succeeding triangles and circles. The remaining rules translate rectangles (rule r_3) and their connections to circles (rule r_4) and triangles (rule r_5). The rules can be easily adapted to rules for the translation of abstract syntax trees with circles and triangles being succeeding statements of different types and rectangles being comments of statements. △

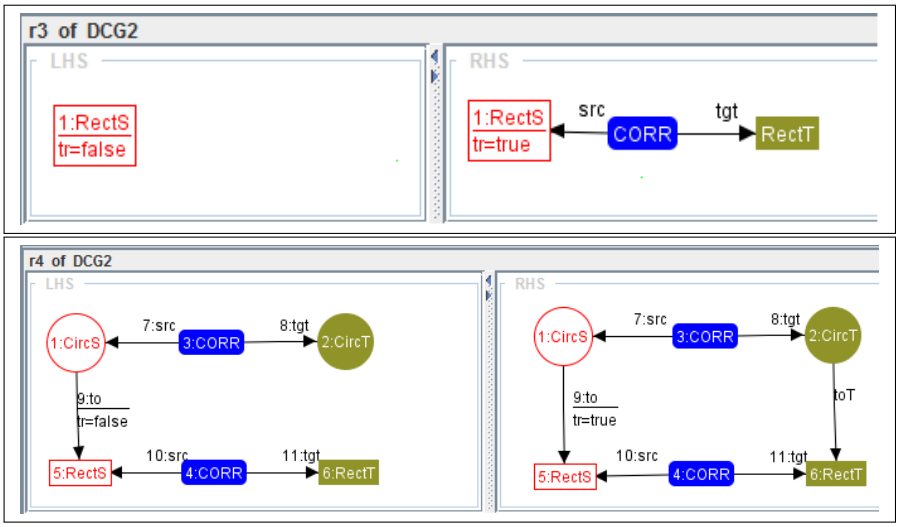

Fig. 7. Forward translation rules r_3 (top) and r_4 (bottom) in AGG

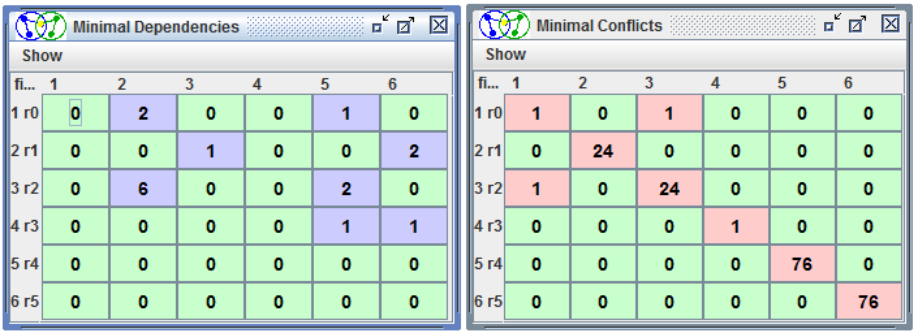

Fig. 8. Dependencies (left) and conflicts (right) of translation rules in AGG

Example 4 (Analysis in AGG and Construction of Conflict-Cluster-Graph). AGG is used to automatically analyse the dependencies and conflicts of the rules. Fig. 7 illustrates rule r_3 and r_4 in their AGG syntax. Each rule contains a left-hand-side that contains all elements that are matched and a right-hand-side that illustrates the updated and created elements during the rule application. Elements marked with < `tr` > have an attribute `tr` with value `false` that is changed to `true` by their rule. This ensures that an element is translated only once.

Fig. 8 depicts the result of the dependency and conflict analysis of all rules of Fig. 6 with AGG. Each blue (dark) box in the left matrix highlights a dependency and each red (dark) box in the right matrix highlights a conflict between two rules. Since rules r_0 and r_2 translate circles, both rules are in conflict. More-

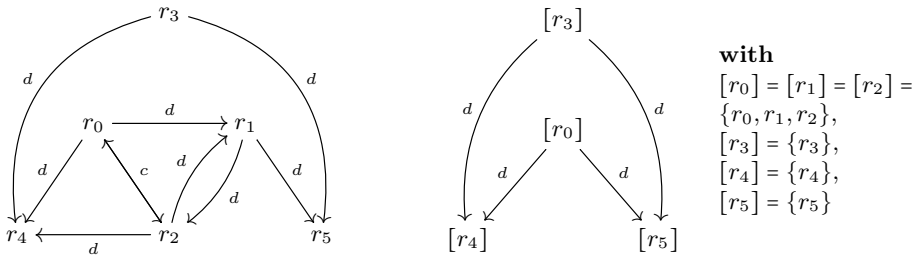

Fig. 9. Dependency-conflict graph (reflexive edges ($r \to r$) are not shown) (left) and cluster graph (right)

over, each rule is in conflict with itself. Rule r_1 translates triangles that are connected to already translated circles whereas rule r_2 translates circles that are connected to already translated triangles, i.e., both rules are (potentially) sequentially dependent on each other. Furthermore, rule r_1 depends on rule r_0. Rule r_4 depends on rules r_0, r_2, r_3, since, it uses the circles and rectangles created by them. Analogously, rule r_5 depends on rules r_1 and r_3. Fig. 9 (right) shows the dependency-conflict-cluster graph derived via the analysis of the dependencies and conflicts of the rules with AGG. △

A DC-Layered Transformation System (DC-LTS) linearises the partial order on clusters of a given $CLG_{DC}(R)$ to a complete order where each cluster becomes a layer and the sequential order of the layers respects the dependencies between the clusters. Formally, a layered transformation system $LTS = (R, S)$ consists of a set of rules R and a sequence $S = (S_i)_{i \in I}$ of subsets of R as layers. Given a graph G, then an execution of LTS is performed by applying each layer consecutively according to the sequence S, where the rules in each layer S_i are applied exhaustively.

Definition 2 (DC-Layered Transformation System). *Let $CLG_{DC}(R)$ be the derived dependency-conflict cluster graph for R, then $LTS = (R, S)$ with $S = (S_i)_{i \in I}$ is a* DC-layered transformation system*, if the following conditions hold*

1. *S is a permutation of the clusters in $N(CLG_{DC}(R))$ (cluster compatibility)*
2. *$\forall$ edges $(a \to b)$ in $CLG_{DC}(R)$: $a = S_k \wedge b = S_l \Rightarrow k < l$ (sequential order)* △

Example 5 (DC-Layered Transformation System). Based on the dependency-conflict cluster graph for the translation rules in Ex. 4, a DC-layered transformation system can be derived by permutations of the clusters with r_0 and r_3 being the two first elements in arbitrary order and clusters r_4 and r_5 being the two last elements in arbitrary order, e.g., permutation $S = (r_3, r_0, r_5, r_4)$ is a valid linearisation of the clusters.

Definition 3 (Independence of Rules). *A pair of rules (p_1, p_2) is called* sequentially independent*, if any two transformation steps $G_0 \overset{p_1}{\Longrightarrow} G_1 \overset{p_2}{\Longrightarrow} G_2$ are sequentially independent.* △

Fact 1 (Characterization of Independent Rules) *A pair of rules* (p_1, p_2) *is sequentially independent if and only if there is no critical pair for* (p_1^{-1}, p_2)*, where* $p_1^{-1} = ((R \xleftarrow{r} K \xrightarrow{l} L), ac')$ *denotes the inverted rule of the rule* $p = ((L \xleftarrow{l} K \xrightarrow{r} R), ac)$ *and* ac' *is obtained by shifting application condition* ac *over rule* p*.* △

Proof. The proof is shown for rules with negative application conditions for Fact 2 in [16] using the completeness result for critical pairs concerning rules with NACs (Thm. 3.7.6 in [24]). According to Thm. 1 in [10] the result for completeness of critical pairs was extended to the case of nested application conditions. This implies that Fact 1 holds. □

Lemma 1 (Existence of Switch Equivalent Layered Sequence). *Let* $s = (G_0 \Rightarrow^* G_n)$ *be a transformation sequence via rules* R *and let* $LTS = (R, S)$ *be a DC-LTS for* R*. Then, there is a transformation sequence* $s' = (G_0 \Rightarrow^* G_n)$ *respecting the order* S *of rule clusters.* △

Proof. By induction over the length of s. Base case: $s = \varnothing$ (empty sequence). Then, $s' = s$ is a sequence respecting S. Inductive step: Assume that $s = (G_0 \Rightarrow^* G_n)$ via R and there is a switch equivalent sequence $\hat{s}$ respecting the order of S. Therefore, $\hat{s} = (G_0 \Rightarrow^* G_n)$, i.e. the last graph G_n in s coincides with that in $\hat{s}$ (up to isomorphism) due to switch equivalence. We show that for $s; s_{n+1} = (G_0 \Rightarrow^* G_n \overset{p_{n+1}}{\Longrightarrow} G_{n+1})$ via R, there is a switch equivalent sequence s' respecting the order of S. Let S_i be the cluster in S with $p_{n+1} \in S_i$. Let $G_{i-1} \overset{p_i}{\Longrightarrow} G_i$ be the last step in $\hat{s}$ via a rule in S_i. Then, for each step $G_{k-1} \overset{p_k}{\Longrightarrow} G_k$ in $\hat{s}$ with $k > i$ we know that (p_k, p_{i+1}) is sequentially independent (note that the order in a rule pair is important for sequential dependency). We can stepwise apply the Local Church-Rosser Theorem for rules with application conditions (Thm. 1 in [12]) for all steps $k > i$ and switch the steps, which shifts the step via p_{n+1} backward. If $n > i$, we start with step n and derive $G_{n-1} \overset{p_{n+1}}{\Longrightarrow} G'_n \overset{p_n}{\Longrightarrow} G_{n+1}$. We continue until rule p_{n+1} is applied right after step i yielding $s' = (G_0 \Rightarrow^* G_{i-1} \overset{p_i}{\Longrightarrow} G_i \overset{p_{n+1}}{\Longrightarrow} G'_{i+1} \Rightarrow^* G_{n+1})$. Now, step $i+1$ is the last step via a rule in S_i such that the sequence of steps via S_i in s' is extended by one more step in comparison to $\hat{s}$. Since $\hat{s}$ respects S and $\hat{s}$ is switch equivalent to s by induction hypothesis, we derive that s' respects S and s' is switch equivalent to $s; s_{n+1}$. □

Lemma 2 (Existence of exhaustive switch equivalent layered sequence). *Let* $s = (G_0 \Rightarrow^* G_n)$ *be a terminated sequence via rules* R *and let* LTS *be a DC-LTS for* R*. Then, there is a terminated sequence* $s' = (G_0 \Rightarrow^* G_n)$ *via* LTS*, which is switch equivalent to* s*.* △

Proof. Let s be a terminated sequence via R. According to Lemma 1, we derive a switch equivalent transformation sequence s' that respects the order of S, such that s' is switch equivalent to s. Let $S = (S_1, \ldots, S_n)$, then we can divide s' in subsequences $s' = (s'_1, \ldots, s'_n)$, where the applied rules in each subsequence s'_i all belong to cluster S_i. We have to show that s' is a sequence via LTS, i.e., each subsequence is exhaustive, i.e. no more rule of S_i can be applied.

Let $s'_i = G_{i,0} \Rightarrow^* G_{i,k}$. Assume that there is a step $G_{i,k} \overset{p_i}{\Longrightarrow} G_{i,k+1}$ via a rule p_i in S_i. If $i = n$, we then know that step $G_{i,k} \overset{p_i}{\Longrightarrow} G_{i,k+1}$ can be performed at the end of s as well. Since s is terminated, this would be a contradiction. Thus, $i \neq n$. We consider the subsequent subsequence s'_{i+j} with $j > 0$ that is not empty. Thus, $s'_{i+j} = G_{i,k} = G_{i+1,0} \overset{p_{i+1}}{\Longrightarrow} G_{i+1,1} \Rightarrow^* G_{i+1,l}$. Thus, we have the two steps $G_{i,k} \overset{p_{i+1}}{\Longrightarrow} G_{i+1,1}$ and $G_{i,k} \overset{p_i}{\Longrightarrow} G_{i,k+1}$. By definition of the dependency conflict clusters, we know that for all rules $p'_{i+1} \in R \setminus S_i$ there is no critical pair (p_i, p'_{i+1}). Therefore, the two steps are parallel independent according to completeness of critical pairs (Thm. 1 in [10]). We can apply the Local Church-Rosser Theorem (Thm. 1 in [12]) and can shift step $G_{i,k} \overset{p_i}{\Longrightarrow} G_{i,k+1}$ forward. This can be repeated for all successive steps and we derive the sequence $s'; s'_{n+1}$ with $s'_{n+1} = G_n \overset{p_i}{\Longrightarrow} G_{n+1}$. This means that $G_n \overset{p_i}{\Longrightarrow} G_{n+1}$ is a step that extends sequence s, which is a contradiction to the precondition that s is terminated. Therefore, the assumption that there is a step $G_{i,k} \overset{p_i}{\Longrightarrow} G_{i,k+1}$ via a rule p_i in S_i is false, which means that s' is exhaustive. □

The construction of a DC-layered transformation system *LTS* for a set of rules R reduces the amount of rules to be checked for applicability at each step. By definition, the execution of a layer in an *LTS* concerns only rules in that layer. Thm. 1 below ensures preservation of the input-output behaviour. All terminated sequences via R (i.e., no more rules are applicable) can be performed via *LTS*.Each rule only depends on rules in a preceding layer and rules in the same layer. The input-output relation IO_{TS} of a transformation system *TS* contains all pairs (G_I, G_O) with a terminated transformation sequence $G_I \Rightarrow^* G_O$ via *TS*.

Theorem 1 (Completeness of DC-LTS). *Let R be a set of rules and LTS be a DC-layered transformation system for R, then: $IO_R = IO_{LTS}$, i.e.*
($\exists$ terminated ($G_0 \Rightarrow^* G_n$) via R) $\Leftrightarrow$ ($\exists$ terminated ($G_0 \Rightarrow^* G_n$) via LTS). △

Proof. **Direction "⇐":** Let $LTS = (R, S)$ If s' is a terminated sequence via *LTS*, then $s = s'$ is also a terminated sequence via R, because the rules in S are also contained in R.
Direction "⇒": This is a direct consequence of Lemma 2. □

A DC-LTS can reduce the effort for backtracking. By Thm. 2 below, functional behaviour of the layers eliminates the need for backtracking of transformation steps that are not in the current layer. A transformation system *TS* has functional behaviour, if IO_{TS} is right unique, i.e. for each input graph, there is at most one output graph up to isomorphism. A layer S_i of an $LTS = (R, S)$ has functional behaviour, if the induced transformation system with rules S_i has functional behaviour, which can be analysed statically with the tool AGG [15,31].

Theorem 2 (Reduction of Backtracking). *Let LTS be a DC-layered transformation system, where each layer has functional behaviour. Then, there is no need to backtrack already completed layers during the computation of a terminated sequence $G_0 \Rightarrow^* G_n$ via LTS. Moreover, LTS has functional behaviour.* △

Proof. Assume we backtrack already completed layers, then we will obtain the same output graphs for these layers due to functional behaviour and thus, we derive the same input graph for the current layer. $LTS = (R, S)$ has functional behaviour, because each layer has functional behaviour and the layers are executed via the fixed sequence S. □

The effect of Thm. 2 is that the effort for checking functional behaviour of the whole system is reduced to the analysis of each layer separately. Note that application conditions for rules are an appropriate method to ensure functional behaviour [15]. Our approach can be combined with the generation of filter NACs [15], which eliminates some types of rule conflicts, but not all.

We improve the performance of a model transformation MT by applying the concept of a DC-LTS to the set of operational rules of MT. By $TRAFOS(MT)$ we denote the set of all model transformation sequences $TRAFOS(MT) = \{s \mid s = (G^S, G_0 \Rightarrow^* G_n, G^T)$ is a model transformation sequence via $MT\}$ for a model transformation MT.

Definition 4 (DC-optimised Model Transformation). *Let $LTS = (TR_{FT}, S)$ be a DC-layered transformation system for the forward translation rules TR_{FT} of a TGG with induced model transformation MT. The* DC-optimised model transformation $MT_{LTS}: \mathcal{L}(TG^S) \Rightarrow \mathcal{L}(TG^T)$ *is obtained from MT by restriction to the LTS-compatible model transformation sequences, i.e., $TRAFOS(MT_{LTS}) = \{s \in TRAFOS(MT) \mid s = (G^S, G'_0 \overset{tr^*_{FT}}{\Longrightarrow} G'_n, G^T)$ and $G'_0 \overset{tr^*_{FT}}{\Longrightarrow} G'_n$ is a transformation sequence via $LTS\}$.* △

By Thm. 3 below, we show that the execution of the DC-LTS does not affect the existing results for TGGs concerning the notion of correctness and completeness (see Def. 5 below according to [15]).

Definition 5 (Correctness and Completeness). *A model transformation MT is* correct*, if for each MT-sequence $(G^S, G_0 \Rightarrow^* G_n, G^T)$ there is a triple graph $G = (G^S \leftarrow G^C \rightarrow G^T) \in \mathcal{L}(TGG)$. It is called* complete*, if for each $G^S \in \mathcal{L}(TGG)^S$, there is an MT-sequence $(G^S, G_0 \Rightarrow^* G_n, G^T)$.* △

Theorem 3 (Correctness and Completeness). *Each DC-optimised model transformation $MT_{LTS}: \mathcal{L}(TG^S) \Rightarrow \mathcal{L}(TG^T)$ is correct and complete.* △

Proof. By Thm. 1 in [15], we know that model transformations MT based on forward translation rules are correct and complete. By Thm. 1, we derive that MT and MT_{LTS} have the same input/output relation and thus, MT_{LTS} is correct and complete. □

6 Evaluation

Fig. 10 shows the evaluation of the efficiency improvement using a standard consumer laptop (CPU: i7-2860QM, RAM: 8GB, Java: 1.7U25, OS: 64-bit version

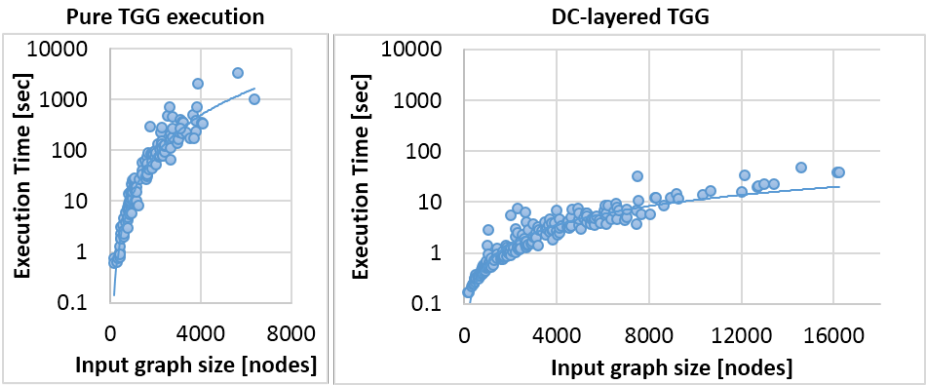

Fig. 10. Measurements for satellite ASTRA 1N (logarithmic scale) using Henshin

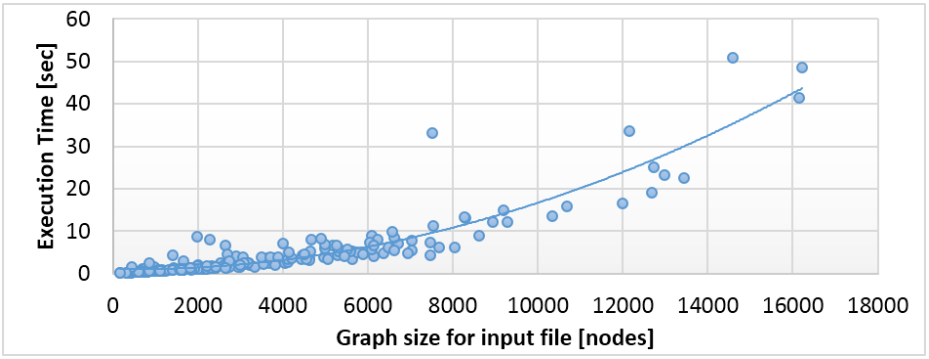

Fig. 11. Measurements for satellite ASTRA 1N for DC-layered TGG (linear scale)

of Windows 7) for translating all control procedures (202 files, 199,853 lines of code (LOC)) that were developed by ASTRIUM for the satellite ASTRA 1N. The construction of the dependency conflict clusters is performed once statically for the TGG and thus, not contained in the execution times. The left chart shows the translation via the TGG without efficiency improvement for the smallest 126 files[6] (<50KB) – file no. 127 reached a timeout of 10 hours. The amount of nodes of an AST graph is on average about 4 times the amount of LOC of the file. The execution of the DC-layered TGG (right chart) is faster (approximately 100 times as fast for graphs with 4,000 nodes) - mainly due to the massively reduced amount of rule match computations at each step. Fig. 11 shows the execution times for the translation via the TGG with efficiency improvement with linear scale. Fig. 10 shows the execution times for translating each input file separately. The effective translation of the full set of files at SES is performed by distributing the files to eight parallel Java threads (four physical cores). This leads to

[6] A file contains the code for one satellite control procedure.

Table 1. Evaluation of requirements

Requirement	**Evaluation**
Syntactical correctness and completeness	Ensured for Phase 2 of the AST conversion by Thm. 3; TGGs simplify the guarantee of a resulting tree structure
Precision/fidelity, minimal efforts for revalidation	TGG rules are obtained from DSL mapping document that was specified by domain experts containing pairs of corresponding source and target code fragments
Complete automation	Yes: no user interaction, no manual editing of output files.
Maintainability	- Visual and intuitive GUI for TGG rules - No complex control structures for execution - Automated check of rule dependencies with AGG [31]
Readability	- The output source code in SPELL is well aligned - Output is compliant with SPELL coding guidelines - All header entries and comments are generated adequately
Efficiency, scalability	- Metamodels of generated Xtext plugins: >140 types - Rules: 484 (TGG: 249, initialisation + refactoring: 235), - Internal XML representation: ~50,000 LOC (lines of code) - Benchmark: ~5:00 min. for satellite Astra 1N (see Fig. 10)
Direct savings	1–2 man years per satellite (estimated by SES, compared to manual conversion and validation)

an additional average speed up factor of three such that the translation for one satellite takes about five minutes. SES appreciated the obtained speed as it is largely above what is needed for practical use.

Table 1 provides an overview of the evaluation of the translator concerning the industrial requirements of SES. The implementation has been delivered to SES and was successfully assessed and validated by SES and the satellite manufacturer ASTRIUM. According to Thm. 3, the translation ensures syntactical correctness and completeness for Phase 2 of the AST conversion via the TGG. TGGs simplify the challenge to ensure that the resulting graph of the model transformation forms an AST. The source model is always preserved and the execution ensures that elements are translated exactly once. This reduces the challenge of checking that the rules translate each path or subtree of the source AST into a path or subtree in the target graph attached to the corresponding parent node. The size of the TGG, the processed input files and the corresponding execution times in Table 1 show that the presented approach is applicable for large scale applications. Currently, the following six satellites are running on the generated control procedures: Astra-1M, Astra-1N, Astra-2E, Astra-2F, Astra-3B, and SES-6. Moreover, SES is validating two further TGG-translators for the satellite control languages of the satellite manufacturers THALES and BOEING.

7 Related Work

Other solutions for software translation include manually writing a converter, using a compiler-compiler or meta-programming based on term rewriting or similar techniques. In fact, a fully manual rewrite in the target language, using the source language artefact only as a reference, is also feasible in some situations and even has been the preferred approach for the mission-critical satellite control procedures at SES, before the approach presented in this paper has been taken into account.

Converters that are manually written in general-purpose programming languages are prone to errors and hard to maintain in the case of language changes and new requirements that emerge during their lifetime. Maintainability requires at least an extensive documentation of the converter source code, but even then the converter can only be inspected by domain experts that have sufficient knowledge of the programming language of the converter. In contrast to that, the (visual) translation rules of our approach use the abstract syntax concepts of source and target language and are, hence, open to be inspected and discussed by domain experts without specific programming background.

Compiler-compilers or parser generators, such as ANTLR [27], can be used to generate a parser based on the grammar of a source language. Then, the generation of the target language has to be programmed either in annotation to the source grammar or by traversing the generated abstract syntax tree. In both cases, only the source language can be specified in an adequate way by its grammar, while the target language is implicit in the manually written code.

Source transformation systems based on term rewriting include the DMS system [2], TXL [4], the Rascal language [22] and the Spoofax language workbench [21] with the Stratego/XT engine [3]. Using these systems is quite similar to our approach, which can be seen, e. g., in the Extract-Analyze-Synthesise (EASY) Paradigm for Rascal [23]. Both, the source and the target language, are specified in some form of grammar formalism and the transformation between the languages is given by a set of transformation rules, where all the abovementioned systems use some sorts of rewriting rules, which are specified in a textual syntax.

While these systems aim at providing integrated systems, we are using separate building blocks that are already available in the EMF ecosystem – Xtext for parsing and serialising and Henshin for transformation. Parsers and/or serialisers can also be generated from XML Schema Definition (XSD) files by the core EMF system if the language is an XML dialect. Source and/or target language can also be visual languages implemented by EMF-based tools like the Graphical Modeling Framework (GMF). This provides for a seamless integration of heterogeneous languages. Moreover, the basic language definitions – Xtext grammars, XSD files, GMF projects – and the resulting plugins are reusable for all translation, refactoring and model transformation projects involving the same language.

The textual programming of a specific term rewriting language has quite a steep learning curve [5], while we experienced that the visual specification of pattern-based graph transformation rules on EMF models provides more in-

tuitive access. Our division of the conversion by graph transformation into the three phases – initialisation, forward translation based on triple graph grammars, and refactoring of the result – yields a separation of concerns that additionally helps in keeping the solution comprehensible. Our example from Sec. 4 already shows non-trivial structural differences between the abstract syntax structures of source and target language. In our industrial case study, the visual representation provided a more intuitive access to those structural differences than a textual, tree-oriented representation.

Several performance improvements for TGGs have been proposed for restricted kinds of TGGs using dependency information on nodes only [25,13]. The present paper provides a general technique for arbitrary TGGs and yields a layered transformation system, where functional input/output behaviour avoids the need for backtracking of already executed layers. We use the general notion of rule conflicts and dependencies - in particular, we take into account dependencies on edges, attributes and application conditions. We are confident that the existing approaches can be integrated in the new one by applying them locally to each layer.

Regarding performance of model transformations in general, Mészáros et al. [26] have proposed manual and automatic optimizations based on overlapping of matches. Specifically for Henshin, Tichy et al. [32] have identified several "bad smells", i. e., features of transformation rules that possibly result in poor transformation performance and should be avoided if possible. During the development of the *PIL2SPELL* translation, in addition to our dependency-based strategy, we followed the guidelines from [32].

8 Conclusion

In this article, we presented a formal and fully automated approach to industrial software source code translation. We provided a general concept for efficiency improvement of graph transformation systems (Thms. 1 and 2). In our main result (Thm. 3), we have shown the correctness of the approach. We evaluated the approach within a safety critical industrial application: the translation of satellite control procedures. In particular, we evaluated the industrial requirements, including reliability, efficiency and code readability. Our approach considerably improves the rewriting efficiency of the used triple graph transformation approach while guaranteeing the correctness. As an effective result, six communication satellites are running on the generated procedures.

Regarding the Henshin tool, work is in progress to implement the critical pair analysis directly instead of using AGG. The performance results achieved by our proposed approach shall be further evaluated by making use of recently developed benchmarks [20,1].

In future work, we will employ the rich formal foundation of TGGs and apply them for the synchronisation between source code and possible visualisations of software. We also plan to apply graph transformation techniques for analysing test coverage and generating valid test cases.

Acknowledgments. This project is part of the Efficient Automation of Satellite Operations (EASO) project supported by the European Space Agency (ESA)[7].

Supported by the Fonds National de la Recherche, Luxembourg (3968135, 4895603).

References

1. Anjorin, A., Cunha, A., Giese, H., Hermann, F., Rensink, A., Schürr, A.: Benchmarx. In: Bidirectional Model Transformations 2014. ECEASST, European Association of Software Science and Technology (2014), to appear.
2. Baxter, I., Pidgeon, P., , Mehlich, M.: DMS: Program transformations for practical scalable software evolution. In: Software Engineering (ICSE'04). IEEE Press (2004)
3. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. a language and toolset for program transformation. Science of Computer Programming 72(1-2), 52–70 (2008)
4. Cordy, J.R.: The TXL source transformation language. Science of Computer Programming 61(3), 190–210 (2006)
5. Cordy, J.R.: Excerpts from the TXL cookbook. In: Generative and Transformational Techniques in Software Engineering (GTTSE 2009). LNCS, vol. 6491, pp. 27–91. Springer (2011)
6. The Eclipse Foundation: Xtext – Language Development Framework – Version 2.2.1 (2012), http://www.eclipse.org/Xtext/
7. The Eclipse Foundation: EMF Henshin – Version 0.9.4 (2013), http://www.eclipse.org/modeling/emft/henshin/
8. Ehrig, H., Ehrig, K., Hermann, F.: From model transformation to model integration based on the algebraic approach to triple graph grammars. ECEASST 10, 14 (2008)
9. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
10. Ehrig, H., Habel, A., Lambers, L., Orejas, F., Golas, U.: Local confluence for rules with nested application conditions. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) Proceedings of Intern. Conf. on Graph Transformation (ICGT' 10). LNCS, vol. 6372, pp. 330–345. Springer (2010), http://www.springerlink.com/index/X273147851566804.pdf
11. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information preserving bidirectional model transformations. In: Fundamental Approaches to Software Engineering. LNCS, vol. 4422, pp. 72–86. Springer (2007)
12. Ehrig, H., Habel, A., Lambers, L.: Parallelism and concurrency theorems for rules with nested application conditions. ECEASST 26 (2010)
13. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling 8(1), 21–43 (2009)
14. Hegedüs, Á., Horváth, Á., Varró, D.: Towards guided trajectory exploration of graph transformation systems. ECEASST 40 (2010)
15. Hermann, F., Ehrig, H., Golas, U., Orejas, F.: Efficient analysis and execution of correct and complete model transformations based on triple graph grammars. In: Model Driven Interoperability (MDI 2010). pp. 22–31. ACM (2010)

[7] http://www.esa.int/ESA

16. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y.: Correctness of Model Synchronization Based on Triple Graph Grammars - Extended Version. Tech. Rep. TR 2011-07, TU Berlin, Fak. IV (2011)
17. Hermann, F., Ehrig, H., Orejas, F., Golas, U.: Formal analysis of functional behaviour of model transformations based on triple graph grammars. In: Graph Transformations (ICGT 2010). LNCS, vol. 6372, pp. 155–170. Springer (2010)
18. Hermann, F., Gottmann, S., Nachtigall, N., Braatz, B., Morelli, G., Pierre, A., Engel, T.: On an Automated Translation of Satellite Procedures Using Triple Graph Grammars. In: Theory and Practice of Model Transformations, LNCS, vol. 7909, pp. 50–51. Springer (2013)
19. Hermann, F., Gottmann, S., Nachtigall, N., Ehrig, H., Braatz, B., Engel, T.: Triple Graph Grammars in the Large for Translating Satellite Procedures. In: Proc. of Int. Conf. on Theory and Practice of Model Transformations 2014 (ICMT 2014). LNCS, Springer (2014)
20. Hildebrandt, S., Lambers, L., Giese, H., Rieke, J., Greenyer, J., Schäfer, W., Lauder, M., Anjorin, A., Schürr, A.: A survey of triple graph grammar tools. In: Stevens, P., Terwilliger, J.F. (eds.) Bidirectional Transformations 2013. ECEASST, vol. 57. European Association of Software Science and Technology (2013)
21. Kats, L.C.L., Visser, E.: The Spoofax language workbench. rules for declarative specification of languages and IDEs. In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010) (2010)
22. Klint, P., Vinju, J.J., van der Storm, T.: RASCAL: A domain specific language for source code analysis and manipulation. In: Source Code Analysis and Manipulation. pp. 168–177. IEEE Computer Society (2009)
23. Klint, P., van der Storm, T., Vinju, J.: EASY meta-programming with Rascal. In: Generative and Transformational Techniques in Software Engineering (GTTSE 2009). LNCS, vol. 6491, pp. 222–289. Springer (2011)
24. Lambers, L.: Certifying Rule-Based Models using Graph Transformation. Ph.D. thesis, Technische Universität Berlin (2009)
25. Lauder, M., Anjorin, A., Varró, G., Schürr, A.: Bidirectional model transformation with precedence triple graph grammars. In: Proc. Eur. Conf. on Modelling Foundations and Applications (ECMFA'12), LNCS, vol. 7349, pp. 287–302. Springer (2012)
26. Mészáros, T., Mezei, G., Levendovszky, T., Asztalos, M.: Manual and automated performance optimization of model transformation systems. International Journal on Software Tools for Technology Transfer 12(3-4), 231–243 (July 2010)
27. Parr, T., Fisher, K.: LL(*): the foundation of the ANTLR parser generator. ACM SIGPLAN Notices 46(6), 425–436 (2011)
28. Schürr, A.: Specification of graph translators with triple graph grammars. In: Graph-Theoretic Concepts in Computer Science. LNCS, vol. 903, pp. 151–163. Springer (1994)
29. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Graph Transformations (ICGT 2008). LNCS, vol. 5214, pp. 411–425 (2008)
30. SES Engineering: SPELL - Satellite Procedure Execution Language & Library – Version 2.3.13 (2013), `http://code.google.com/p/spell-sat/`
31. TFS-Group, TU Berlin: AGG (2014), `http://www.tfs.tu-berlin.de/agg`
32. Tichy, M., Krause, C., Liebel, G.: Detecting performance bad smells for Henshin model transformations. In: Baudry, B., Dingel, J., Lucio, L., Vangheluwe, H. (eds.) Proc. of the Second Workshop on the Analysis of Model Transformations (AMT 2013). CEUR Workshop Proceedings, vol. 1077 (2013)