PhD-FSTC-2016-60
The Faculty of Sciences, Technology and Communication

DISSERTATION

Presented on 02/12/2016 in Luxembourg
to obtain the degree of

DOCTEUR DE L´UNIVERSITÉ DU LUXEMBOURG EN INFORMATIQUE

by

Susann Gottmann

Born on 11th November 1982 in Zwenkau (Germany)

Synchronisation of Model Visualisation and Code Generation Based on Model Transformation

Dissertation defense committee:

Dr. Thomas Engel, Dissertation Supervisor
Professor, Université du Luxembourg, Luxembourg

Dr. Frank Hermann
Carmeq GmbH, Germany

Dr. Ulrich Sorger, Chairman
Professor, Université du Luxembourg, Luxembourg

Dr. Claudia Ermel
Technische Universität Berlin, Germany

Dr. Raimondas Sasnauskas
SES Engineering, Luxembourg

Abstract

The development, maintenance and documentation of complex systems is commonly supported by model-driven approaches where system properties are captured by visual models at different layers of abstraction and from different perspectives as proposed by the Object Management Group (OMG) and its model-driven architecture. Generally, a model is a concrete view on the system from a specific perspective in a particular domain. We focus on visual models in the form of diagrams and whose syntax is defined by domain-specific modelling languages (DSLs). Different models may represent different views on a system, i.e., they may be linked to each other by sharing a common set of information. Therefore, models that are expressed in one DSL may be transformed to interlinked models in other DSLs and furthermore, model updates may be synchronised between different domains. Concretely, this thesis presents the transformation and synchronisation of source code (abstract syntax trees, ASTs) written in the Satellite-Procedure & Execution Language (SPELL) to flow charts (code visualisation) and vice versa (code generation) as the result of an industrial case study. The transformation and synchronisation are performed based on existing approaches for model transformations and synchronisations between two domains in the theoretic framework of graph transformation where models are represented by graphs. Furthermore, extensions to existing approaches are presented for treating non-determinism in concurrent model synchronisations. Finally, the existing results for model transformations and synchronisations between two domains are lifted to the more general case of an arbitrary number of domains or models containing views, i.e., a model in one domain may be transformed to models in several domains or to all other views, respectively, and model updates in one domain may be synchronised to several other domains or to all other views, respectively.

Acknowledgements

I would like to thank all those who supported me during my PhD studies. First of all, I would like to thank my supervisor Frank Hermann for his strong guidance, support and constant motivation. He always had an open word for questions and problems and always helped to clarify them. Furthermore, I would like to thank Frank for giving me the opportunity to get in touch with teaching and supervising (master) students. I would like to thank Claudia Ermel for her continous encouragement and help, and for supervising my work. Thanks to Frank and Claudia for introducing me into academic research. It was a pleasure to work with both of them. Also many thanks to my supervisor Thomas Engel for trusting me and giving me the opportunity to perform this PhD studies. I would like to express my gratitude to Raimondas Sasnauskas for supervising my work as well as Ulrich Sorger for being part of my PhD jury. Many thanks also to my colleagues for motivation and scientific help, especially Benjamin Braatz and Nico Nachtigall, who were part of the "graph transformation team" at the SnT. I would like to express my gratitude to SES, especially Gianluigi Morelli, Alain Pierre and Rafael Chinchilla, for introducing me into the field of satellite controlling, for sharing their expertise and supporting me in order to perform the industrial case study. I would like to say thank you to my master students, Marc Paul, Nida Khan and Bruno Pinto Gonçalves, for giving me the opportunity to supervise their master theses. Last but not least, special thanks to my little family: Nico, Emil and Max, for their patience and support.

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

Contents

Introduction & Problem Statement

1

Model-driven engineering (MDE) [SVC06] is commonly used to support the software and systems development process on a higher abstraction level [HWRK11]. It is proposed and supported by the Object Management Group (OMG) [OMG16], which was founded in 1989 as an international consortium of different leading industrial companies (e.g., IBM, Sun, Apple). Up to now, many partners entered the consortium leading to a total number of around 800 members. The OMG pushes forward the development of standards in the area of object-oriented software and systems development to which the MDE belongs to. One development of the OMG consortium is the Unified Modeling Language (UML) [UML16] which became a standard tool in model-driven software and systems engineering [Sel12].

Models form the basis in MDE which are used for the description and the generation of software systems [EEGH15]. Models, especially UML diagrams, form an abstraction of the system which is represented by models. Models can be any kind of representation: visual or textual, but also in any other form [Mah09]. In the current work, we focus on visual models that are used to represent source code, i.e., software. Within the applied industrial project (cf. Sec. 1.1), we focus on flow chart models [FC169] that were extended by specific needs of our industrial partner. The theoretical concepts that we developed in this work are not bound to any specific model type and therefore can be applied to any kind of (visual) model.

Figure 1.1: Short Overview on Model Transformation Approaches [CH06]

Model transformations define how to transform a model into another model. Fig. 1.1 illustrates a distinction of different model transformation

approaches. It is based on [CH06]. Model transformation approaches can be categorised into *model-2-model approaches* and *model-2-text approaches*. In the model-2-model approach, a model that corresponds to a meta-model is transfomred to another model that corresponds to another meta-model. In contrast, in the model-2-text approach, a model that corresponds to a meta-model is translated into strings, i.e., text. In general, the model-2-text approach can be seen as a special case of the model-2-model approach, but due to the fact that current compilers usually work on text files, the differentiation of both cases is usefull [CH06]. In Chap. 7, we will discuss the model transformation approaches of this figure in greater detail and enumerate model transformation tools for each category of model transformation approach.

Let us consider Fig. 1.2 which depicts a scheme of each source and target instance that will be translated by model transformations. In model transformation, a source instance shall be transformed into a target instance. An Instance (M0) is usually a visual model describing a software artifact, e.g., flow chart, UML class diagram, entity relationship diagram, etc., but also a text, e.g., source code, XMI code, etc.. each instance is described by a model (M1), which is again defined by a meta-model. The meta-model itself conforms to a meta-meta-model (M4). The meta-meta-model again conforms to a meta-meta-meta-model. This can be continued endlessly, but in general, the meta-meta-model conforms to itself, i.e., it follows the structure defined by itself.

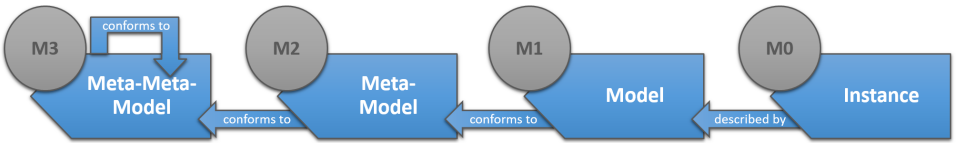


Figure 1.2: Instance, Model and (Meta-)Meta-Models

Fig. 1.3 illustrates the model transformation process: One or more *source models* that conform to a *source meta-model* shall be transformed to one or more *target models* that conform to a *target meta-model*. The *transformation engine* executes the model transformation in following the *transformation specification*. The transformation specification defines ways, i.e., rules, according to which the model transformation will be executed. The transformation specification conforms to a given *transformation language*, i.e., it is no arbitrary specification. The source meta-model, the target meta-model and the transformation language conform to the *meta-meta-model* (cf. (M4) in Fig. 1.2).

Note, the model transformation can be either executed *unidirectional* from a source model to a target model or vice versa, but also *bidirectional*, i.e., from source model to target model and back. The model transformation

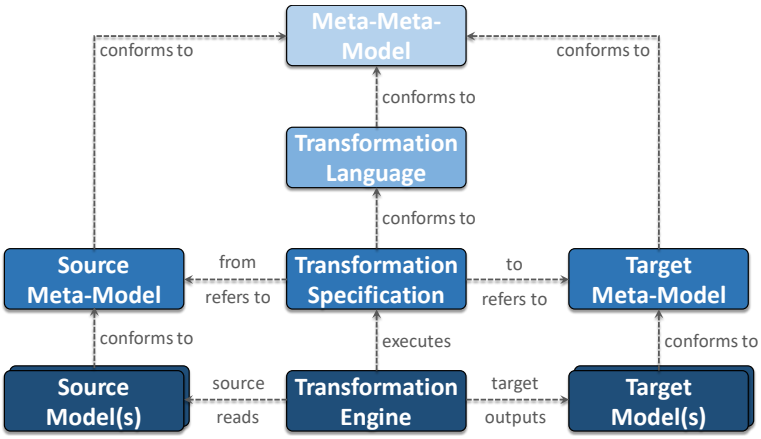


Figure 1.3: Model Transformations

is either executed *horizontal* or *vertical* which is illustrated in Fig. 1.4. A horizontal model transformation means that a model in a domain D1 is transformed into a model in domain D2, i.e., a model that conforms to a certain meta-model is transformed into a model of another meta-model. The transformation is performed on the same abstraction level. In contrast, the vertical model transformation translates a model in one domain to a model in the same domain, but the transformation is performed between different abstraction layers, e.g., from a platform-independent model to a platform-specific model. It is also possible to transform models in the same domain to the same abstraction level, as well as the transformation from one abstraction level in one domain to another abstraction level to another domain (vertical+horizontal).

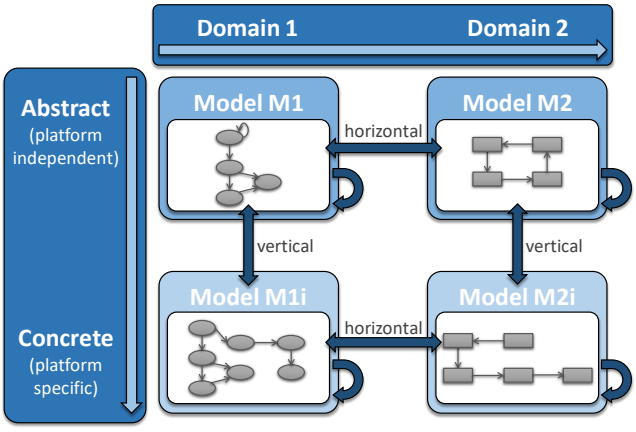


Figure 1.4: Model Transformation Specification

1.1. Background on the Industrial Project

In the framework of the ongoing cooperation between the Center of Security, Reliability and Trust (SnT) and SES which is a world-leading satellite operator [SES16], an automated translation called "PIL2SPELL" from the satellite control language PIL to the satellite control language SPELL was developed based on triple graph grammars [HGN+13, HGN+14a, HGN+14b]. This project was applied successfully in practice with the satellite "A2F" being the first satellite operating in space using this automated translation [PIL12], but more satellites followed already [PIL14].

Due to the success of PIL2SPELL, 38 out of 53 satellites are operated with the satellite control language SPELL, which is a unified and open-source satellite control language based on Python usable for each satellite vendor [SPE15, CNB15]. Therefore, new aims came up which will enhance the work of satellite operators and developers using SPELL: In detail, the idea is to develop and use a visualisation for SPELL source code which is similar to flow charts [FC169] but adapted to the domain of satellite control languages. This visual language which we developed during this project is called "SPELL-Flow". Then, it is desired to develop an automated translation from SPELL to SPELL-Flow (visualisation) in order to provide the visualisation to the satellite controllers. It is also interesting to provide a backward direction, i.e., the code generation out of the visual model which can be used by satellite control procedure devolopers. In that case, a bidirectional approach is desired so that coding in SPELL but also in SPELL-Flow is possible for the developers.

1.2. Research Questions

In this work, we will answer the following resaerch questions formally with regard to the chosen formal modelling technique, namely triple graph grammars which is based on algebraic graph transformation (cf. Chap. 2). In Fig. 1.5, we give a graphical overview of the research questions and we classify the questions to the corresponding research area with regard to model transformations.

In Chap. 3 "Methodology", we want to analyse two questions regarding the unidirectional model transformation from one language into another:

According to which concept is it possible to transform model $\mathcal{L}1$ *to model* $\mathcal{L}2$*? (Q1).*

Based on the results we worked out regarding Q1, we will extend this methodology so that we take the bidirectional translation between two languages into account, i.e., the translation from one language to another and back.

According to which concept is it possible to transform model $\mathcal{L}1$ *to model*

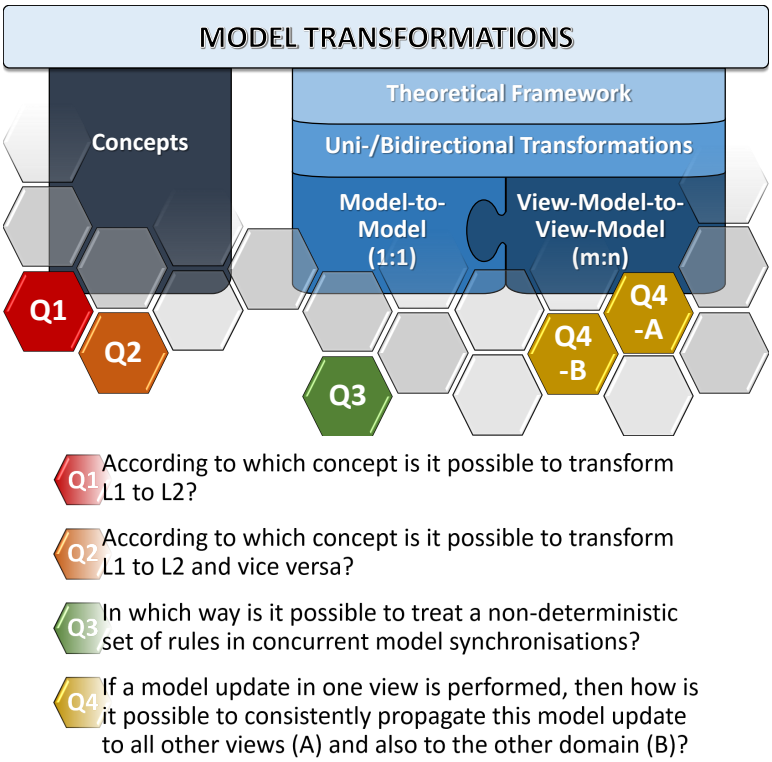


Figure 1.5: Research Questions: Summary & Classification to Research Areas

$\mathcal{L}2$ and vice versa? (Q2).

The next chapter "Model Synchronisation" in Chap. 4 will review the formalisation of the model synchronisation framework based on triple graph grammars which was published in [HEO+15]. During the work, questions referring conflicts in the concurrent model synchronisation process occured which we will discuss in this chapter, especially in the case of a non-deterministic set of triple triple rules. So, we will answer the following research question:

In which way is it possible to treat a non-deterministic set of rules in concurrent model synchronisations? (Q3)

Chap. 5 "Propagation of Model Updates in Multi-View Models" presents a new framework for propagating model updates in multi-view models. A multi-view model is a model, which contains redundant information represented in different view models. This information is connected with each other, i.e., if a manual update is performed, then an update of corresponding information in a different view of the same model might become necessary. The presented derived propagation framework is based on triple graph grammars. Therefore, our approach also includes a solution on the consis-

tent propagation of the model update to the corresponding model in the other domain. We formulate the research question as follows:

If a model update in one view is performed, then how is it possible to consistently propagate this model update to all other views (Q4-A) and also to the other domain? (Q4-B).

1.3. Running Example

The running example of this thesis is derived out of real examples used in the industrial project *SPELL* ⇔ *SPELL-Flow* which is a cooperation with SES. Due to a Non-disclosure agreement (NDA), we are not allowed to show real SPELL code in this thesis, instead we modified real code in order to derive a running example which is closely related to real SPELL code (but without any "realistic" meaning).

Example 1.3.1 (SPELL code - source language)**.** *We use the following SPELL source code, given in Listing 1.1. This example is derived and defamiliarised out of a real SPELL procedure that originally contained about 2000 lines of code (LOC).*

Listing 1.1: Running example: SPELL code

```
1  ##################################################################
2  #
3  # NAME              : RUNNING_EXAMPLE
4  # FILE              : RUNNING_EXAMPLE.py
5  #
6  ##################################################################
7  ARGS['$ARG1']=Var(Type=ABSTIME, Confirm=False)
8  ARGS['$ARG2']=Var(Type=ABSTIME, Confirm=False)
9  ARGS['$ARG3']=Var(Type=ABSTIME, Confirm=False)
10 ARGS['$ARG4']=Var(Type=ABSTIME, Confirm=False)
11 IVARS['$VAR1']=Var(Type=LONG)
12 IVARS['$VAR2']=Var(Type=LONG)
13 IVARS['$VAR3']=Var(Type=LONG)
14 IVARS['$VAR4']=Var(Type=LONG)
15 IVARS['$VAR5']=Var(Type=ABSTIME)
16 IVARS['$VAR6']=Var(Type=ABSTIME)
17 IVARS['$VAR7']=Var(Type=STRING, Default='YES')
18 IVARS['$VAR8']=Var(Type=STRING, Default='NO_MONIT')
19 #
20 ##################################################################
21 #                                                                #
22 #                        Main Procedure                          #
23 #                                                                #
24 ##################################################################
25 #
26 Step('INIT', 'Confirm procedure execution')
```

```
27 if not Prompt('Do you really want to execute this
28                 procedure?', YES_NO):
29     Finish('Execution aborted by user')
30 #ENDIF
31 ###################################################################
32 Step('1', 'DATA PRESENTATION')
33 # Comments and infos about the variables defined above:
34 # ARG1 - ARG4 and VAR1 - VAR8.
35 #
36 ###################################################################
37 Step('2', 'INITIAL VERIFICATIONS')
38 # Check values
39 # - group A
40 GetTM('TTEST13A ⟨|TEST_GROUP_13A|⟩')
41 GetTM('TTEST14A ⟨|TEST_GROUP_14A|⟩')
42 GetTM('TTEST15A ⟨|TEST_GROUP_15A|⟩')
43 # - group B
44 GetTM('TTEST13B ⟨|TEST_GROUP_13B|⟩')
45 GetTM('TTEST14B ⟨|TEST_GROUP_14B|⟩')
46 GetTM('TTEST15B ⟨|TEST_GROUP_15B|⟩')
47 # - Special group
48 GetTM('TTEST12A ⟨|TEST_GROUP_12A|⟩')
49 GetTM('TTEST12B ⟨|TEST_GROUP_12B|⟩')
50 Prompt('WARNING: Some warning.', OK)
51 Pause()
52 Prompt('WARNING: Another warning.', OK)
53 Pause()
54 # Check something else
55 GetTM('TTESTCHECK1 ⟨|AL1|⟩')
56 GetTM('TTESTCHECK2 ⟨|AL2|⟩')
57 GetTM('TTESTCHECK3 ⟨|AL3|⟩')
58 GetTM('TTESTCHECK4 ⟨|AL4|⟩')
59 Prompt('WARNING: Again a warning.', OK)
60 Pause()
61 #
62 ###################################################################
63 Step('3', 'GET SOME INFORMATION')
64 IVARS['$VAR3'], IVARS['$VAR4']=SomeFunctionCall()
65 #
66 ###################################################################
67 Step('4', 'ACTION SELECTION')
68 # This stage allows to select the required action
69 user_Choice=Prompt('Choose an option:', [ 'Option 1',
70                                           'Option 2',
71                                           'Option 3'],
72                    LIST | ALPHA )
73 if (user_Choice == 'Option 1'):
74     # GOTO Step of option 1
75     Goto('5')
```

```
76 elif (user_Choice == 'Option 2'):
77     # GOTO Step of option 2
78     Goto('6')
79 elif (user_Choice == 'Option 3'):
80     # GOTO Step of option 3
81     Goto('7')
82 #ENDIF
83 ##################################################################
84 Step('5', 'MANAGEMENT OF COMPONENT A')
85     Prompt('WARNING: Check something before.', OK)
86     Pause()
87     Prompt('WARNING: Check something else.', OK)
88     Pause()
89     Verify([[ 'TTEST13A ⟨|TEST_GROUP_13A|⟩', eq, 'OFF' ],
90             [ 'TTEST15A ⟨|TEST_GROUP_15A|⟩', eq, 'OFF' ]])
91 ##################################################################
92 Step('6', 'MANAGEMENT OF COMPONENT B')
93     Prompt('WARNING: Check something before.', OK)
94     Pause()
95     Prompt('WARNING: Check something else.', OK)
96     Pause()
97     Verify([[ 'TTEST13B ⟨|TEST_GROUP_13B|⟩', eq, 'OFF' ],
98             [ 'TTEST15B ⟨|TEST_GROUP_15B|⟩', eq, 'OFF' ]])
99 ##################################################################
100 Step('7', 'FINAL VERIFICATIONS')
101 # Final checks
102 # - group A
103 GetTM('TTEST13A ⟨|TEST_GROUP_13A|⟩')
104 GetTM('TTEST14A ⟨|TEST_GROUP_14A|⟩')
105 GetTM('TTEST15A ⟨|TEST_GROUP_15A|⟩')
106 # - group B
107 GetTM('TTEST13B ⟨|TEST_GROUP_13B|⟩')
108 GetTM('TTEST14B ⟨|TEST_GROUP_14B|⟩')
109 GetTM('TTEST15B ⟨|TEST_GROUP_15B|⟩')
110 # - Special group
111 GetTM('TTEST12A ⟨|TEST_GROUP_12A|⟩')
112 GetTM('TTEST12B ⟨|TEST_GROUP_12B|⟩')
```

Block I: Declarations (Lines: 1-25) *Lines 1 to 6 contain a multi-line comment. In SPELL, comments start with a #-symbol and finish at the end of the line. Lines 7 to 18 define variables and arguments. Those lines will be omitted in the SPELL-Flow model, i.e., they will be ignored during the translation from SPELL to SPELL-Flow (cf. Sec. 6.1.1). Lines 19 to 25 contain a multi-line comment. In the remainder of this description, we do not explicitly indicate further comments anymore.*

Block II: Step INIT (Lines: 26-30) *Line 26 indicates the actual start of the SPELL procedure. It consists of a* Step *statement which can be seen as jump label and structuring element of the code. A* Step *statement has two parameters: A label (here:* 'INIT') *and a description (here:* 'Confirm procedure execution'*). The next lines (27 - 30) hold an* if *construction. In executing the condition, a* Prompt *will be invoked asking for a user input. Depending on the user's input (*YES *or* NO*), the* if *statement branches to line 29 and executes the* Finish *statement, i.e., the procedure ends. Otherwise, the execution of the procedure continues with line 32, which holds a* Step *statement.*

Block III: Step 1 (Lines: 31-36) *This* Step *statement only structures the code, because it only encompasses comments (lines 33 -36). It is followed by another* Step *in line 37.*

Block IV: Step 2 (Lines: 37-60) *The* Step *statement in line 37 comprises a list of* GetTM *statements (lines 38-60) followed by* Prompt *and* Pause *statements and again a list of* GetTM *statements followed by a* Prompt *and a* Pause*. A* GetTM *statement holds a parameter and is used for retrieving engineering telemetry data with the name indicated by the parameter from the satellite. The* Prompt *statements invoke prompts to the user that need to be answered. In this block,* Prompt*s only offer the option* OK*, i.e., the user has to confirm that she has read this message. The* Pause *statement pauses the execution of the procedure.*

Block V: Step 3 (Lines: 61-64) *The next part (lines 61 to 64) contains a* Step *statement. It is followed by the assignment of the resulting values from the function* SomeFunctionCall() *to two variables (*IVARS*). The names of the variables are provided by parameters, e.g., the first variable is called* $VAR3*.*

Block VI: Step 4 (Lines: 65-82) *Lines 65 to 82 encompass the next block that starts again with a* Step *statement. In this block, the user is prompted for selecting between three options in line 69. The selection result is assigned to variable* user_choice*. Afterwards, the value of* user_choice *will be evaluated and according to the user's decision, the execution pointer of the procedure jumps to a different position of the SPELL source code using the* Goto *statement. The target of the jump is always a* Step *statement, i.e., the parameter given in the* Goto *statement should correspond to a label of an existing* Step *statement. Otherwise, jumping to another position in the source code is not possible.*

Block VII: Steps 5 and 6 (Lines: 83-98) *The following two blocks (lines 83 - 90 and lines 91 - 98) are very similar. Each of them consists of an introducing* Step *statement, followed by a list of* Prompt, Pause *and* Verify *statements, i.e., a user prompt followed by a pause is evoked. Afterwards, the* Verify *is used for retrieving and comparing telemetry parameters of the satellite with the values given in the list of parameters. This statement is able to handle a list of parameters and possible assignements (cf. page 39 in [CNB15]). In many cases, the comparisons can be executed in parallel.*

Block VIII: Step 7 (Lines: 99-112) *Finally, the last part starts at line 99 and ends at line 112. The* Step *statement in line 100 is followed by a list of* GetTM *statements for retrieving different telemetry values from the satellite.* △

The SPELL source code that we provided in Ex. 1.3.1 was to be translated to a SPELL-Flow model, which we will now illustrate in Ex. 1.3.2. It is a hierarchical model: The first layer is the most abstract layer. The underlying layers contain more detailed information on the model. The model also includes comments and SPELL source code snippets that are not directly visible in the SPELL-Flow model, i.e., this information is implicictly available, but explicitly not shown in the screenshots we use in Ex. 1.3.2. This implicit information will be shown as tooltips in the *SPELL-Flow visualisation tool*, i.e., the tool-support we implemented for visualising the SPELL-Flow models (cf. Sec. 6.1.3).

Example 1.3.2 (SPELL-Flow model - target model)**.** *The screenshot in Fig. 1.6 illustrates the first (main) layer of the SPELL-Flow model that corresponds to the SPELL code in Listing 1.1. The main layer contains:*

- *All* Step*s on the first indentation level (in general, this contains all* Step*s in the source code).*
- *All if/elif/else conditions (also for or while loops, and try/catch expressions) on the first indentation level.*
- *All* Goto *statements (i.e., jumps).*

The screenshots in Fig. 1.7 and Fig. 1.8 illustrate the second layer which belongs to the first Step *statement with parameters* INIT, Confirm procedure execution*. In this case, the model in the second layer is very similar to the part of the model that belongs to the same* Step *on the first layer.*

The screenshot in Fig. 1.8 contains only the Step *statement, because only this* Step *was used for structuring and includes just comments, which are not displayed in the SPELL-Flow model.*

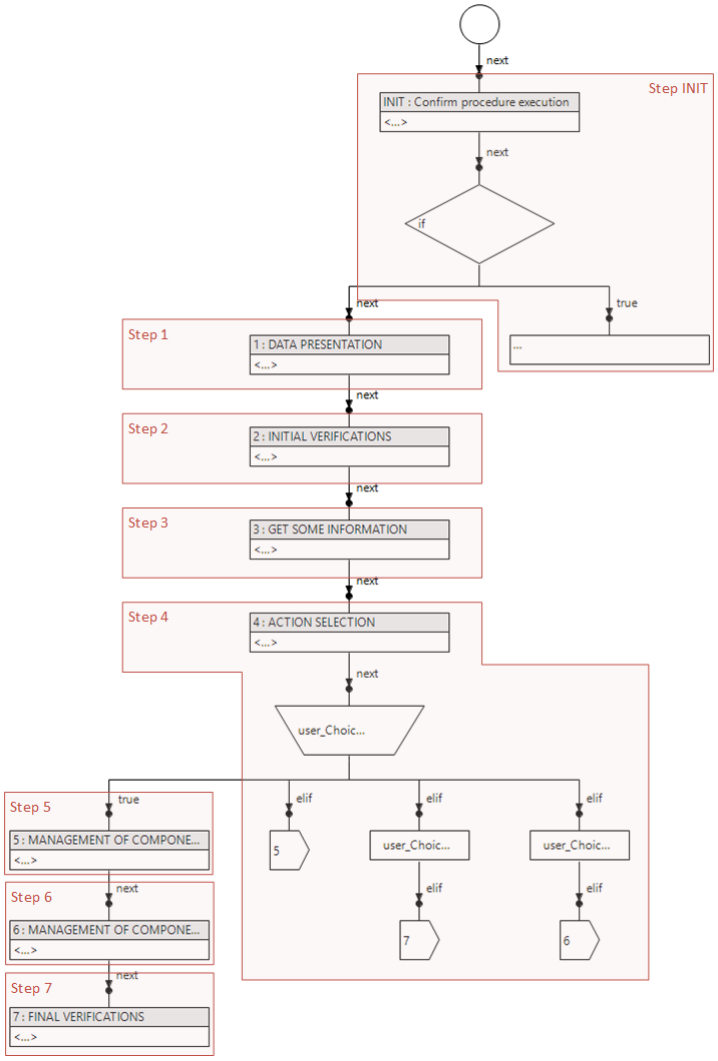


Figure 1.6: Screenshot of first layer of SPELL-Flow model

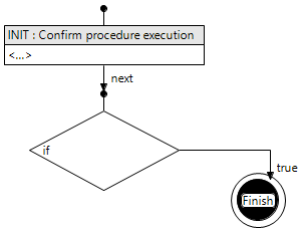


Figure 1.7: Second layer of SPELL-Flow model - part that belongs to Step INIT

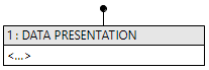


Figure 1.8: Second layer of SPELL-Flow model - part that belongs to Step 1

In Fig. 1.6 Step 2 *is visualised by one shape on the main layer. In contrast, the corresponding second layer of this* Step *statement includes all underlying statements, i.e., all* GetTM, Prompt *and* Pause *statements, as shown in Fig. 1.9.*

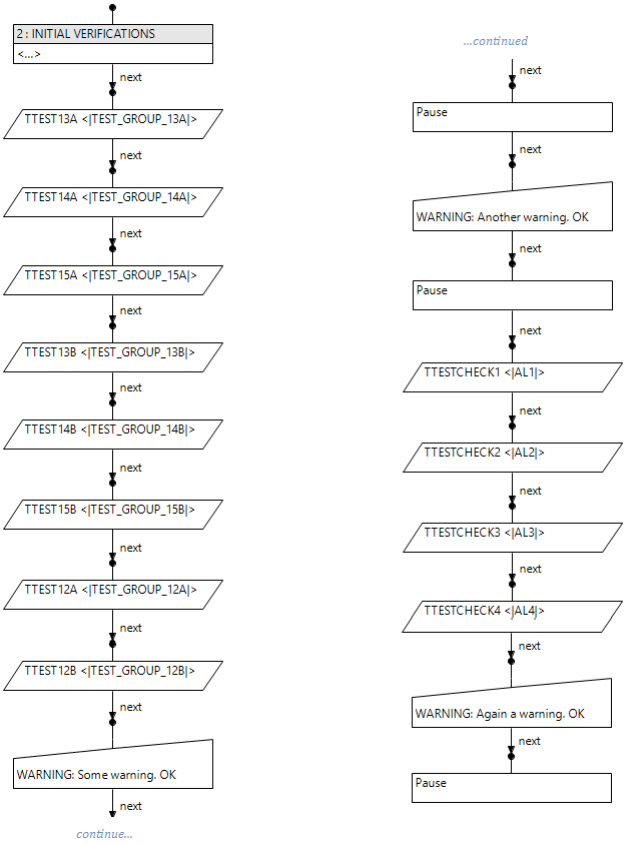


Figure 1.9: Second layer of SPELL-Flow model - part that belongs to Step 2

Step 3 *comprises one assignment which is explicitly shown on the second layer that belongs to this* Step *(cf. Fig. 1.10).*

Fig. 1.11 shows the second layer that belongs to Step 4*. The* if*-condition is also reflected on the main layer, because, according to the rules mentioned above, it is on the first indentation level in the SPELL source code. Still, the second layer is more detailed in the sense that assignment of the user input (*Prompt*) to variable* user_choice *is reflected on this layer, whereas the*

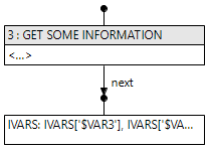


Figure 1.10: Second layer of SPELL-Flow model - part that belongs to Step 3

same line of code is omitted on the main layer.

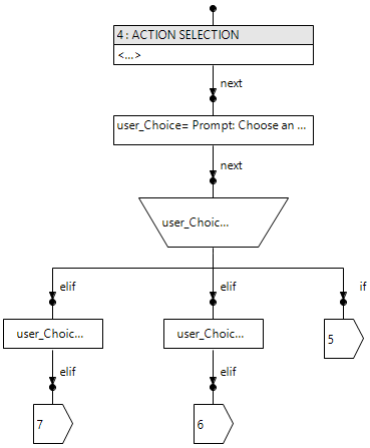


Figure 1.11: Second layer of SPELL-Flow model - part that belongs to Step 4

Fig. 1.12 shows the detailed list of statements that are contained in Step 5*, while the main level only represents the corresponding* Step*.*

Fig. 1.12 represents Step 6 *which is very similar to* Step 5 *due to very similar code blocks (cf., Ex. 1.3.1).*

Finally, Fig. 1.13 illustrates the detailed list of statements that belong to Step 7*. The main layer only contains the* Step *statement.*

△

Example 1.3.3 (SPELL-Flow meta-model)**.** *The instance model of Ex. 1.3.2 is typed over the meta-model in Fig. 1.14 (also called type graph). We implemented this meta-model using Eclipse EMF [EMF16]. For a screenshot showing a visual meta-model we refer to Appendix A.3.*

The node Root *is the container of the instance model. All other nodes in the instance model are contained by this node or by its children. The containment structure is determined by Eclipse EMF. The node* Root *provides several containment edges for different node types in the SPELL-Flow meta-model (*to_StartNode*,* to_EndNode*,* to_Element*,* to_Connector *and* hasComment*). The* Root *node contains one attribute* valueXMIID*.*

Another special node is TargetSpellFlow*, all other nodes (except node* Root*) are derived from this node. This node contains an edge* tgt_opp

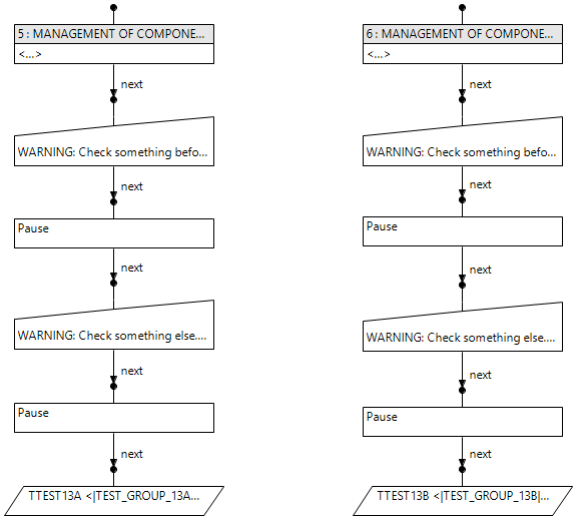


Figure 1.12: Second layer of SPELL-Flow model - part that belongs to Step 5 (left) and part that belongs to Step 6 (right)

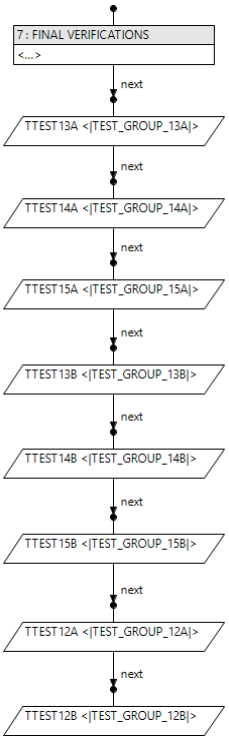


Figure 1.13: Second layer of SPELL-Flow model - part that belongs to Step 7

which is used for the correspondence meta-model. We represent the correspondence meta-model in Appendix A.4. Node Statement *is derived from*

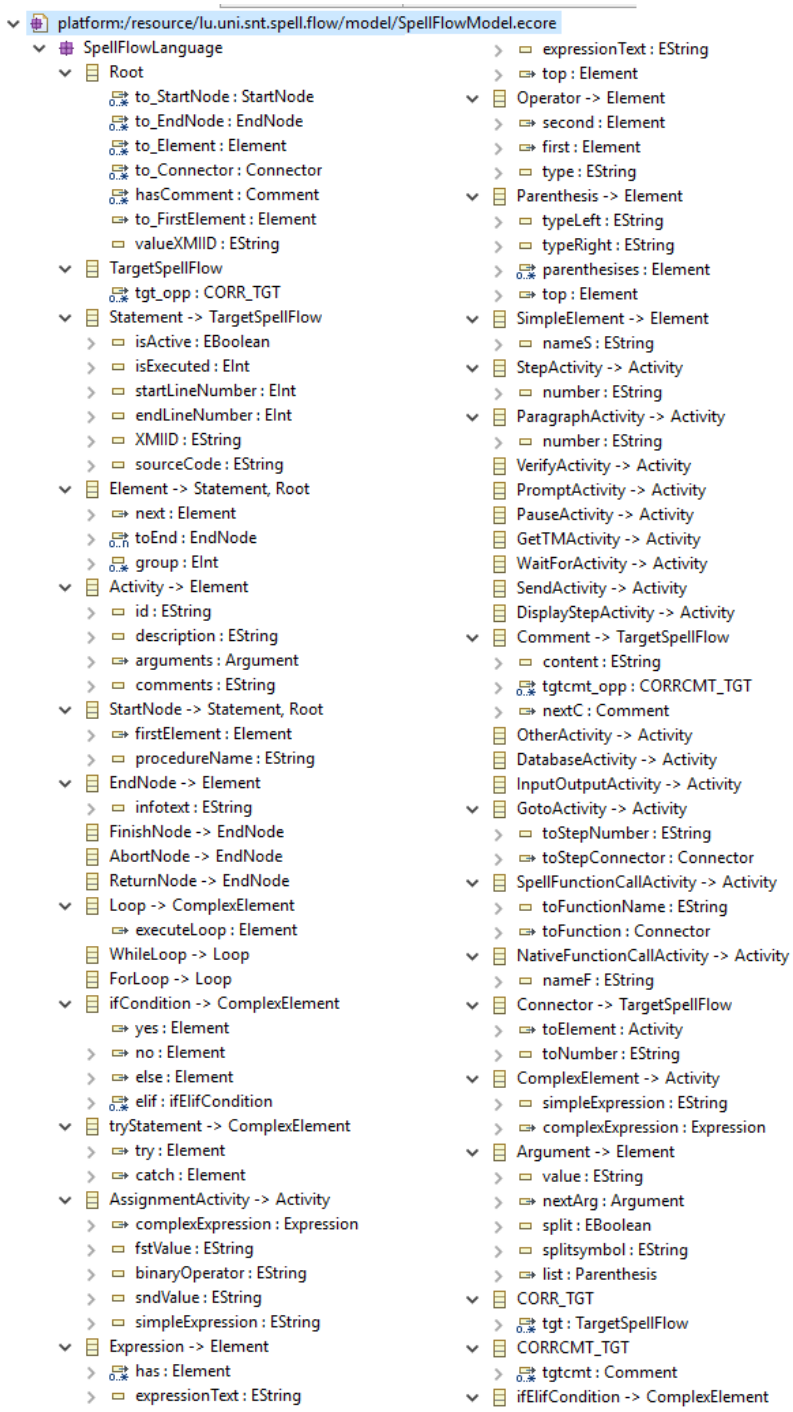


Figure 1.14: SPELL-Flow meta-model, screenshot of expanded Eclipse ecore file

TargetSpellFlow *and provides general attributes for nearly all children nodes, like e.g.,* startLineNumber*,* endLineNumber *or the corresponding* sourceCode*. The* Element *node is parent of all nodes that are visible in the SPELL-Flow instance model, except for the* StartNode *which uses a different set of attributes and edges than the element provided by node* Element*. All other nodes are specialisations (or in-between-nodes used for further specialisations) which are finally used within the instance model, e.g.,* StepActivity*,* VerifyActivity*,* GotoActivity*,* ForLoop*,* ifCondition*.*

Nodes CORR_TGT *and* CORRCMT_TGT *are helper nodes which are necessary for the correspondence meta-model which is used in the triple graph transformation step, i.e., the main part of the unidirectional and also the bidirectional translation.* △

1.4. Overview of the Thesis

The thesis is structured as described below: We will introduce the underlying formal graph transformation framework in Chap. 2. In that chapter, we will address the following main points in order to be able to intorduce model transformations based on triple graph grammars: First, we will repeat the general concepts of graphs so that we are able to extend them to attributed graphs, typed graphs, typed attributed graphs and the corresponding graph transformation concepts. Furthermore, we will discuss (negative) application conditions. Then, we are able to introduce triple graphs, triple graph grammars and model transformations based on triple graph grammars.

In Chap. 3, we will present the methodology for unidirectional and bidirectional model translations which evolved out of the practical work on the industrial projects. We will show its applicability on both industrial case studies.

In the next chaper (Chap. 4), we will review the model synchronisation framework based on triple graph grammars and the extensions regarding concurrency and the semi-automated conflict resolution which we elaborated during the work on this PhD project.

In Chap. 5 the derived propagation framework for propagating model updates in multi-view models, which is an extension of the model synchronisation framework based on triple graph grammars, is presented in detail.

During the work on the PhD project, the model synchronisation framework based on TGGs was applied to the industrial case study at SES. In Chap. 6, we summarise the work on that project and present and evaluate the developed prototypes.

Finally, we analyse related work in Chap. 7, conclude the thesis and discuss perspectives for future work in Chap. 8.

For better clarity, the following Table 1.1 summarises which chapter is based on which research articles and books. (For a full list of all publications

we refer to Appendix A.7.)

Chapter	Articles / Books	Pages
Chap. 2 Formal Framework	∘ *Fundamentals of Algebraic Graph Transformation* by Hartmut Ehrig et al. [EEPT06] ∘ *Graph and Model Transformation - General Framework and Applications* by Hartmut Ehrig et al. [EEGH15]	34
Chap. 3 Methodology for Model Translations	∘ *On an Automated Translation of Satellite Procedures Using Triple Graph Grammars* by Frank Hermann et al. [HGN+13] ∘ *Triple Graph Grammars in the Large for Translating Satellite Procedures* by Frank Hermann et al. [HGN+14a, HGN+14b]. ∘ *Towards Bidirectional Engineering of Satellite Control Procedures Using Triple Graph Grammars* by Susann Gottmann et al. [GHE+13]	10
Chap. 4 Model Synchronisation	∘ *Model synchronization based on triple graph grammars: correctness, completeness and invertibility* by Frank Hermann et al. [HEO+15] ∘ *Correctness and Completeness of Generalised Concurrent Model Synchronisation Based on Triple Graph Grammars* by Susann Gottmann et al. [GHN+13a] ∘ *Optimisation and Customisation of Concurrent Model Synchronisation Based on Triple Graph Grammars - Extended Version* by Susann Gottmann et al. [GHN+13b]	40
Chap. 5 Derived Propagation Framework	∘ *Towards the Propagation of Model Updates along different Views in Multi-View Models* by Susann Gottmann et al. [GNE+16b]	56
Chap. 6 Case Study at SES	∘ *Towards Bidirectional Engineering of Satellite Control Procedures Using Triple Graph Grammars* by Susann Gottmann et al. [GHE+13] ∘ *On an Automated Translation of Satellite Procedures Using Triple Graph Grammars* by Frank Hermann et al. [HGN+13] ∘ *Triple Graph Grammars in the Large for Translating Satellite Procedures* by Frank Hermann et al. [HGN+14a, HGN+14b].	48

Table 1.1: Structure of the Thesis Related to Publications

Formal Framework

2

The model transformation approach we use in this work is based on the (extended) formal algebraic graph transformation framework, which we will introduce in this chapter. Our comprehension of the formal framework and the definitions that we present in this chapter are mainly based on [EEPT06] and [EEGH15].

For further reading on algebraic (typed) (attributed) graph transformation, we refer to the handbook series of Rozenberg et al. [Roz97, EEKR99] and also to [RS97]. A more actual overview of the state of the art in 2006 is given in [Hec06] and also an introduction to graph transformation is available [BH02]. For details on algebraic specification we refer to [EMC+01, EM85].

First, we will define graphs and the algebraic graph transformation approach which forms the basis for the following sections. Using this, we are able to define typed graphs and typed graph transformation [EEPT06, EEGH15], then we define typed attributed graphs and typed attributed graph transformation (TAGT) [GLEO12a] and some further constructions based on TAGT. In order to control the transformation, our transformation rules may be equipped with nested application conditions, which we will describe in a separate section [EEHP06, HP05]. Finally, we introduce all formal details to be able to draw the line to triple graph grammars (TGGs), that is the underlying concept of the subsequent chapters [Sch95, SK08, HEGO14]. We conclude this chapter with the formalisation of transformation units that are used for structuring model transformations [Kus00, KKR08] which were used for implementing the case study (cf. Sec. 6.1).

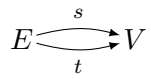
2.1. (Typed) (Attributed) Graphs and Algebraic (Typed) (Attributed) Graph Transformation

The theory on algebraic graph transformation emerged in the late 1960s and in the the early 1970s. In the USA the first works on algebraic graphs and algebraic graph transformation were published by Rosenfeld and Pfaltz [PR69]. In Europe, the first works on that theory emerged by Ehrig, Pfender and Schneider [EPS73, SE77, Ehr79]. The idea behind algebraic graph transformation is inspired by Chomsky grammars that were developed in the late 1950s [Cho59]. We will now review the notions of graphs, graph morphisms, and of constructions like pushouts in order to define the term of algebraic graph transformations.

2.1.1. Graphs

In the algebraic approach, a graph consists of a set of nodes and a set of edges (as sorts of the algebra) and two mapping functions (as operations of the algebra). Both mapping functions define the source and target nodes of each edge (cf. Sec. 1.1.2 in [EEGH15]). The following definitions considers directed edges, i.e., the source and target node of each edge is indicated.

Definition 2.1.1 (Graph (cf. Def. 2.1 in [EEPT06] or in [EEGH15]))**.** *A graph $G = (V, E, s, t)$ consists of a set V of nodes (also called vertices), a set E of edges, and two functions $s, t : E \to V$ mapping source and target nodes to each edge.* △



Example 2.1.1 (Graph)**.** *We have given graph $G = (V, E, s, t)$, that we visualised below, with the following:*

- *set of nodes $V = \{A, B, C, D, E, F\}$*
- *set of edges $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$*
- *source mapping $s : E \to V$ with $e_1, e_2 \mapsto A$, $e_3 \mapsto B$, $e_4, e_5, e_6 \mapsto C$, $e_7 \mapsto D$ and $e_8 \mapsto E$*
- *target mapping $t : E \to V$ with $e_1 \mapsto B$, $e_2, e_3 \mapsto C$, $e_4 \mapsto D$, $e_5, e_7 \mapsto E$ and $e_6, e_8 \mapsto F$* △

A graph morphisms between two graphs maps the edges and nodes from one graph to the other while preserving the source and target functions (i.e., s and t). Graph morphisms that are special kinds of algebra homomorphisms, and with graphs form the category *Graphs* (cf. Sec. 1.1.2 in [EEGH15]). According to Fact 2.5 in [EEPT06], graph morphisms g and f can be composed and will result again in a graph morphism $g \circ f$ which also preserves the source and target functions.

Definition 2.1.2 (Graph Morphism (cf. Def. 2.4 in [EEPT06] or Def. 2.1 in [EEGH15]))**.** *Given graphs G_1, G_2 with $G_i = (V_i, E_i, s_i, t_i)$ for $i = 1, 2$.*

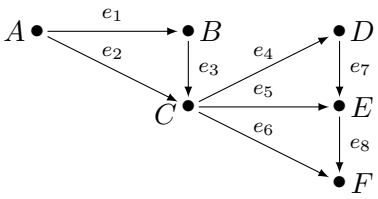
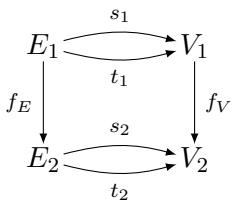


Figure 2.1: Example Graph G

A graph morphism $f : G_1 \to G_2, f = (f_V, f_E)$ consists of two functions $f_V : V_{G_1} \to V_{G_2}$ and $f_E : E_{G_1} \to E_{G_2}$ that preserve the source and target functions, i.e., $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$ (i.e., the diagram on the right commutes). △

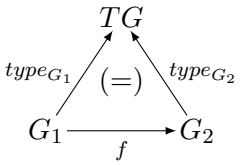


2.1.2. Typed Graphs

We will now extend the definitions of graphs and of graph morphisms to typed graphs and typed graph morphims. It introduces the typing of models. For that, a special graph, called *type graph* is introduced (which is also called *meta-model*) and special morphism, called *type graph morphism*. A type graph defines types for nodes and edges that can be used by typed graphs in order to assign types to its nodes and edges. The typed graph is also called *instance graph*. A type graph morpism is a graph morphism that is additionally type preserving. That means, nodes and edges are mapped to nodes and edges of the same type.

Definition 2.1.3 (Typed Graph and Typed Graph Morphism (cf. Def. 2.6 in [EEPT06] or Def. 2.2 in [EEGH15]))**.** *A type graph is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$. V_{TG} and E_{TG} are called the vertex and the edge type alphabets, respectively.* △

A tuple $(G, type)$ of a graph G together with a graph morphism $type : G \to TG$ is then called a typed graph. Given typed graphs $G_1^T = (G_1, type_1)$ and $G_2^T = (G_2, type_2)$. A typed graph morphism $f : G_1^T \to G_2^T$ is a graph morphism $f : G_1 \to G_2$ such that $type_{G_2} \circ f = type_{G_1}$ (i.e., the diagram on the right commutes).



Example 2.1.2 (Type Graph)**.** *In Ex. 1.3.3 we introduce the complete meta-model, i.e., type graph, of our running example. It is much more complex than a type graph which is possible due to Def. 2.1.3, because it also contains attributes, inheritance, different kinds of edges (e.g., containment edges vs. "normal" edges), cardinalities, etc.. If we shrink this type graph so that it corresponds to our definition, we obtain a type graph which only*

defines types for nodes and edges. In the example type graph, which we illustrate below, we also reduced the number of types of our running example to five different types for nodes in order to increase readability. (Note, in contrast to the SPELL-Flow meta-model in Ex. 1.3.3, we add indicies to edges next *and* to_Element *for better readability.)*

Type graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$ *consists of the following:*

- *set of nodes* $V_{TG} = \{$ *Root, StartNode, StepActivity, PromptActivity, PauseActivity* $\}$
- *set of edges* $E_{TG} = \{$ *to_StartNode, firstElement,* $to\_Element_1$, $to\_Element_2$, $to\_Element_3$, $to\_Element_4$, $next_1$, $next_2$, $next_3$, $next_4$ $\}$
- *source mapping* $s_{TG} : E_{TG} \to V_{TG}$ *with*
  - $to\_StartNode, to\_Element_1 \mapsto Root$,
  - $firstElement \mapsto StartNode$,
  - $next_3 \mapsto PromptActivity$,
  - $next_4 \mapsto PauseActivity$ *and*
  - $to\_Element_2, to\_Element_3, to\_Element_4, next_1, next_2 \mapsto StepActivity$
- *target mapping* $t_{TG} : E_{TG} \to V_{TG}$ *with*
  - $to\_StartNode \mapsto StartNode$,
  - $firstElement, to\_Element_1, to\_Element_2 \mapsto StepActivity$,
  - $next_1, next_4, to\_Element_3 \mapsto PromptActivity$ *and*
  - $next_2, next_3, to\_Element_4 \mapsto PauseActivity$ △

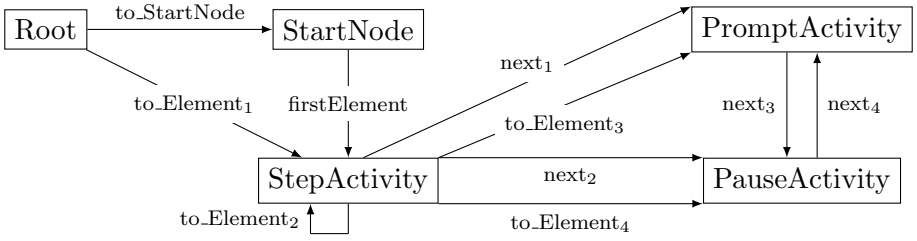


Figure 2.2: Example Type Graph / Meta-Model

Example 2.1.3 (Typed Graph)**.** *We extend graph G from Fig. 2.1 by types. The graph morphism* type *has the following mappings:*

- $type_V : V \to V_{TG}$ *with* $A \mapsto Root$, $B \mapsto StartNode$, $C, D \mapsto StepActivity$, $E \mapsto PromptActivity$ *and* $F \mapsto PauseActivity$

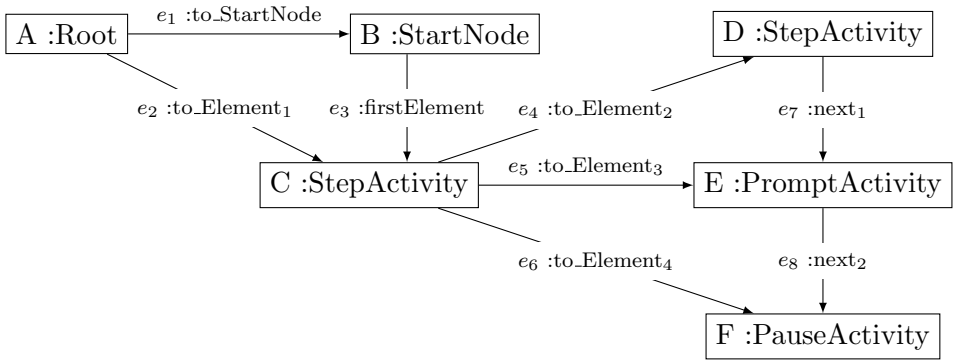


Figure 2.3: Example Instance Graph / Typed Graph $G^T = (G, type)$

- $type_E : E \to E_{TG}$ *with* $e_1 \mapsto to\_StartNode$, $e_2 \mapsto to\_Element_1$, $e_3 \mapsto firstElement$, $e_4 \mapsto to\_Element_2$, $e_5 \mapsto to\_Element_3$, $e_6 \mapsto to\_Element_4$, $e_7 \mapsto next_1$ *and* $e_8 \mapsto next_3$

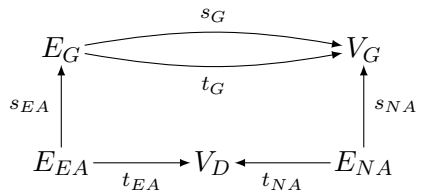
The resulting graph $G^T = (G, type)$ *is shown in Fig. 2.3.* △

Remark 2.1.1 (Visual Notation for Typing)**.** *The visual notation uses* x : TYPE*, where* : TYPE *denotes the type of the corresponding node or edge.* x *denotes the name of the element. We often omit node and edge names, because they are formally irrelevant.* △

2.1.3. Typed Attributed Graphs

The graphs we want to use in practice contain attributes, too. Thus, we extend the definitions of typed graphs and typed graph morphisms to typed attributed graphs and typed attributed graph morphisms [EPT04]. First, we introduce typed graphs and typed graph morphisms. Then, both definitions can be merged to typed attributed graphs and tpyed attributed graph morphisms. The definition is based on *E-graphs*, that are extended kinds of graphs. E-graphs include the definitions of graphs but are expanded by nodes, source and target functions that describe attributes and their data.

Definition 2.1.4 (E-graph and E-graph Morphism (cf. Def. 2.4 in [EEGH15] or Def. 8.1 in [EEPT06]))**.** *An E-graph* $G^E = (V_G, V_D, E_G, E_{NA}, E_{EA}, (s_i, t_i)_{i \in \{G,NA,EA\}})$ *consists of graph nodes* V_G*, data nodes* V_D*, graph edges* E_G*, node attribute edges* E_{NA}*, edge attribute edges* E_{EA}*, and the source and target functions* $s_G : E_G \to V_G$ *and* $t_G : E_G \to V_G$ *for graph edges,* $s_{NA} : E_{NA} \to V_G$ *and* $t_{NA} : E_{NA} \to V_D$ *for node attribute edges, and* $s_{EA} : E_{EA} \to E_G$ *and* $t_{EA} : E_{EA} \to V_D$ *for edge attribute edges.*



For E-graphs G_1^E and G_2^E, an E-graph morphism $f : G_1^E \to G_2^E$ is a tuple $f = ((f_{V_i} : V_i^{G_1} \to V_i^{G_2})_{i\in\{G,D\}}, (f_{E_j} : E_j^{G_1} \to E_j^{G_2})_{j\in\{G,NA,EA\}})$ such that f commutes with all source and target functions. △

Attributed graphs are E-graphs that are extended by a data structure that is provided by an algebra. The underlying algebra includes a data signature (DSIG) for defining the sorts that are necessary for the attribution. An attributed graph uses attribute values that belong to the carrier sets of the algebra, i.e., that correspond to the data signature (cf. [EEPT06, EMC+01]).

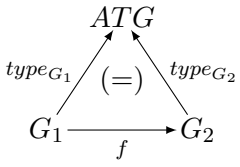
Definition 2.1.5 (Attributed Graph and Attributed Graph Morphism (cf. Def. 2.4 in [EEGH15] or Def. 8.4 in [EEPT06]))**.** *An attributed graph G over a data signature $DSIG = (S_D, OP_D)$ with attribute value sorts $S'_D \subseteq S_D$ is given by $G = (G^E, D_G)$, where G^E is an E-graph and D_G is a $DSIG$-algebra such that $\dot\cup_{s\in S'_D} D_{G,s} = V_D^G$.*

For attributed graphs $G_1 = (G_1^E, D_{G_1})$ and $G_2 = (G_2^E, D_{G_2})$, an attributed graph morphism $f : G_1 \to G_2$ is a pair $f = (f_G, f_D)$ with an E-graph morphism $f_G : G_1^E \to G_2^E$ and an algebra homomorphism $f_D : D_{G_1} \to D_{G_2}$ such that $f_{G,V_D}(x) = f_{D,s}(x)$ for all $x \in D_{G_1,s}, s \in S'_D$. △

For the definition of typed attributed graphs, the term of a final DSIG-algebra is used. It is defined in Def. B.11 in [EEPT06].

Definition 2.1.6 (Typed Attributed Graph and Typed Attributed Graph Morphism (cf. Def. 2.5 in [EEGH15] or Def. 8.7 in [EEPT06]))**.** *An attributed type graph is a distinguished attributed graph $ATG = (TG, Z)$, where Z is the final DSIG-algebra. A tuple $G^T = (G, type_G)$ of an attributed graph G together with an attributed graph morphism $type_G : G \to ATG$ is then called a typed attributed graph.*

Given typed attributed graphs $G_1^T = (G_1, type_{G_1})$ and $G_2^T = (G_2, type_{G_2})$, a typed attributed graph morphism $f : G_1^T \to G_2^T$ is an attributed graph morphism $f : G_1 \to G_2$ such that $type_{G_2} \circ f = type_{G_1}$. △



Example 2.1.4 (Typed Attributed Graph: Explicit Notation)**.** *The attributed type graph ATG, which we visualise in Fig. 2.4 (top) is the type graph from Fig. 2.2 plus nodes that represent the attribute types and edges*

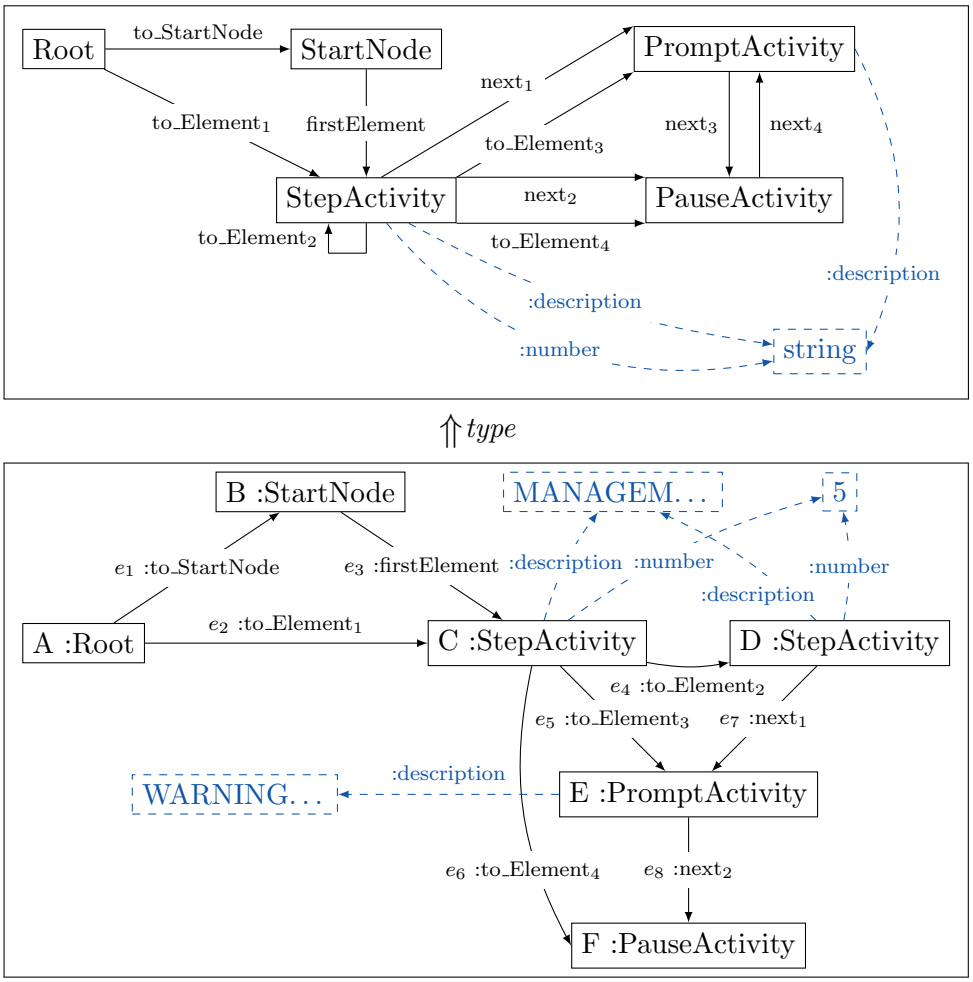


Figure 2.4: Example Typed Attributed Graph $G^T = (G, type)$ (bottom) with Attributed Type Graph ATG (top)

that refer to the attribute types. In the current example, we added one type node string*. In addition, three edges pointing to that tye were added: Two edges named* description *and* number *from the* StepActivity *node and one edge named* description *from the* PromptActivity *node. This extended type graph defines that* StepActivity *nodes my be equipped with two attributes* description *and* number*. Values assigned to both attributes are of type* string*. Furthermore, nodes of type* PromptActivity *may be equipped one attribute* description *of type* string*.*

The example instance graph (attributed typed graph G^T) that we show in Fig. 2.4 (bottom) is typed over type graph ATG. It is the graph from Fig. 2.3 but extended by the following attributes: Both : StepActivity *nodes are equipped with two attributes* description *and*

number. *Both* description *attributes are assigned with string attributes* MANAGEMENT OF COMPONENT A *and* 5*, respectively. Furthermore, node* E : PromptActivity *is extended by attribute* description *that holds the string value* WARNING : Check something before. OK.

In ATG as well as in G^T, we visualise attribute nodes and edges in blue and dashed lines to improve readability. △

Remark 2.1.2 (Visual Notation for Attributes)**.** *In Fig. 2.4 we visualise the attributed type graph ATG as well as the typed attributed graph G^T in explicit notation that corresponds to the E-graph notation. This means that attributes are represented by separate nodes and the attribute names are represented by edges that carry the attribute names. Furthermore, every configuration of all attributes, i.e., all elements of the carrier sets defined by the underlying algebra are available in the typed attributed graph, too. The concrete assignment of an attribute is denoted by an edge. Elements of the carrier sets that are not target of an attribute edge are not assigned. The number of elements from the carrier sets can be infinite. Therefore, we omit elements that are not assigned in the visualisation of the typed attributed graph G^T.*

In Fig. 2.5 we illustrate graph G^T in compact visualisation. There, the attributes and the concrete assignments are part of the graph node, e.g., node C : StepActivity *also includes both attributes and their current assignment. In the remainder of this work, we will use this compact notation.* △

Example 2.1.5 (Typed Attributed Graph: Compact Notation)**.** *In Fig. 2.5 we visualise the same typed attributed graph G^T which is shown in Fig. 2.4 in compact notation.* △

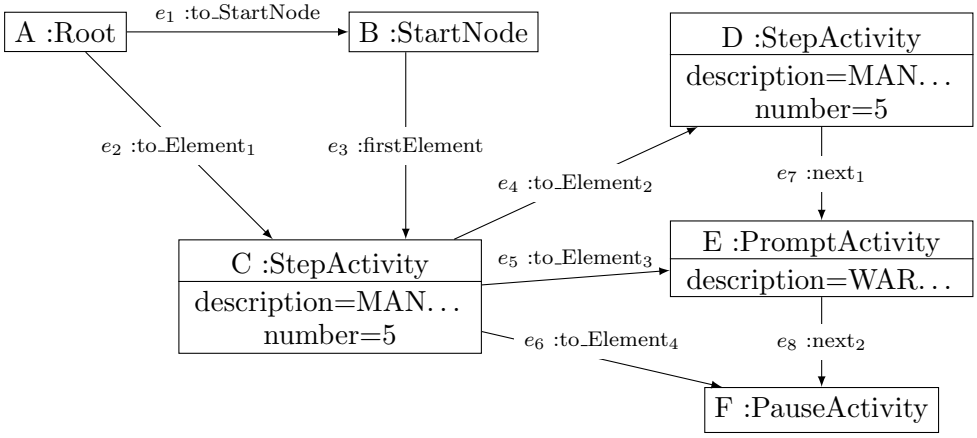


Figure 2.5: Example Typed Attributed Graph $G^T = (G, type)$

2.1.4. $\mathcal{M}$-adhesive Categories and Pushouts & Pullbacks

Eilenberg and Mac Lane established the foundation of the *category theory* in 1945 as result on their research in the mathematical field of algebraic topology (cf. [EM45]). They introduced the concepts of categories, functors and natural transformations. Since then, these concepts were refined and extended in many research papers and books and category theory became an important field in mathematics, logics and theoretic computer science.

The categorial foundation of the algebaric graph transformation approaches, that form the basis for this work, are $\mathcal{M}$-adhesive categories, which are based on a class of $\mathcal{M}$-morphisms and that fulfill a the so-called *van Kampen property* (cf. Def. 4.1 in [EEGH15]). In this abstract class of categories, the rule-based transformation of structures is possible for which numerous desireable theoretical properties hold. (Typed) (attributed) graphs and (typed) (attributed) triple graphs have shown to be $\mathcal{M}$-adhesive categories.

In this section, we will give a short introduction into $\mathcal{M}$-adhesive categories and related concepts, like pushouts and pullbacks, that are necessary in this work. Thus, we omit the categorial foundations and deeper information on $\mathcal{M}$-adhesive categories. For more details on algebraic category theory and $\mathcal{M}$-adhesive categories we refer to [AHS90, EGRW96, Lac05, EEPT06, EEGH15].

We will now cite the general definition of a category. For a short formal introduction to category theory we recommend Appendix A in [EEPT06] or Appendix A in [EEGH15].

Definition 2.1.7 (Category (cf. Def. A.1 in [EEPT06] or Def. A.1 in [EEGH15]))**.** *A category* $\mathbf{C} = (Ob_C, Mor_C, \circ, id)$ *is defined by*

- *a class* Ob_C *of objects;*
- *for each pair of objects* $A, B \in Ob_C$*, a set* $Mor_C(A, B)$ *of morphisms;*
- *for all objects* $A, B, C \in Ob_C$*, a composition operation* $\circ(A, B, C) : Mor_C(B, C) \times Mor_C(A, B) \to Mor_C(A, C)$*; and*
- *for each object* $A \in Ob_C$*, an identity morphism* $id_A \in Mor_C(A, A)$*,*

such that the following conditions hold:

1. *Associativity. For all objects* $A, B, C, D \in Ob_C$ *and morphisms* $f : A \to B$*,* $g : B \to C$ *and* $h : C \to D$*, it holds that* $(h \circ g) \circ f = h \circ (g \circ f)$*.*
2. *Identity. For all objects* $A, B \in Ob_C$ *and morphisms* $f : A \to B$*, it holds that* $f \circ id_A = f$ *and* $id_B \circ f = f$*.* △

The category **Sets** is a category with object class of all sets and with all functions as morphisms (cf. Ex. A.3 [EEGH15]). The composition for

functions $f : A \to B$ and $g : B \to C$ is defined by $(g \circ f)(x) = g(f(x))$ for all $x \in A$. The identity is the identical mapping $id(x) = x$ with $id : A \to A$.

We will now regard special morphisms of a category $\mathbf{C}$: monomorphisms, epimorphisms and isomorphisms. In all categories that we regard in this work, e.g., the categories of typed attributed graphs and their typed attributed graph morphisms ($\mathbf{AGraphs_{ATG}}$) or triple graphs and their triple graph morphisms (**TrGraphs**), the following statements hold, because the categories we use in this work are based on **Sets**.

- monomorphisms are componentwise injective,
- epimorphisms are componentwise surjective, and
- isomorphisms are componentwise bijective.

In general, monomorphisms, epimorphisms and isomorphisms do not correspond to injective, surjective or bijective morphisms, respectively. We will now cite their definitions.

Definition 2.1.8 (Epimorphism (cf. Def. A.12 in [EEGH15] or Def. 2.13 in [EEPT06]))**.** *Given a category* $\mathbf{C}$*, a morphism* $e : A \to B$ *is called a epimorphism if, for all morphisms* $f, g : B \to C \in Mor_{\mathbf{C}}$*, it holds that* $f \circ e = g \circ e$ *implies* $f = g$*.* △

$$A \xrightarrow{e} B \underset{g}{\overset{f}{\rightrightarrows}} C$$

Definition 2.1.9 (Monomorphism (cf. Def. A.12 in [EEGH15] or Def. 2.13 in [EEPT06]))**.** *Given a category* $\mathbf{C}$*, a morphism* $m : B \to C$ *is called a monomorphism if, for all morphisms* $f, g : A \to B \in Mor_{\mathbf{C}}$*, it holds that* $m \circ f = m \circ g$ *implies* $f = g$*.* △

$$A \underset{g}{\overset{f}{\rightrightarrows}} B \xrightarrow{m} C$$

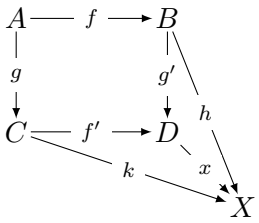
Definition 2.1.10 (Isomorphism (cf. Def. A.9 in [EEGH15] or Def. 2.13 in [EEPT06]))**.** *A morphism* $i : A \to B$ *is called an isomorphism if there exists a morphism* $i^{-1} : B \to A$ *such that* $i \circ i^{-1} = id_B$ *and* $i^{-1} \circ i = id_A$*. Two objects* A *and* B *are isomorphic, written* $A \cong B$*, if there exists an isomorphism* $i : A \to B$*.* △

$$A \underset{i^{-1}}{\overset{i}{\rightleftarrows}} B$$

Intuitively, a pushout (PO) is obtained by gluing two objects via a common subobject. The resulting object is unique up to isomorphism. It is written as $D = B +_A C$, where D is the unique pushout object which is obtained by gluing B and C over a common A. The dual construction is called pullback (PB). Intuitively, it is the intersection of two objects via a common object. It is written as $D = B +_A C$, where D is the pushout object which is obtained by gluing B and C over a common A. We will now present the definitions of both constructions according to [EEPT06, EEGH15]. For more details on the pushout construction and a comparison between the single and the double pushout approach and their results in graph transformations, we refer to [EHK+97].

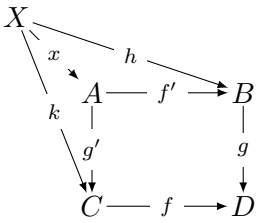
Definition 2.1.11 (Pushout (cf. Def. A.17 in [EEGH15] or Def. 2.16 in [EEPT06]))**.** *Given morphisms $f : A \to B$ and $g : A \to C$ in a category* **C**.

A pushout (D, f', g') over f and g is defined by a pushout object D and morphisms $f' : C \to D$ and $g' : B \to D$ with $f' \circ g = g' \circ f$, such that the following universal property is fulfilled: for all objects X with morphisms $h : B \to X$ and $k : C \to X$ with $k \circ g = h \circ f$, there is a unique morphisms $x : D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$. △



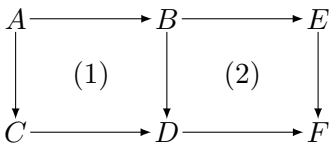
Definition 2.1.12 (Pullback (cf. Def. A.22 in [EEGH15] or Def. 2.22 in [EEPT06]))**.** *Given morphisms $f : C \to D$ and $g : B \to D$ in a category* **C**.

A pullback (A, f', g') over f and g is defined by a pullback object A and morphisms $f' : A \to B$ and $g' : A \to C$ with $g \circ f' = f \circ g'$, such that the following universal property is fulfilled: for all objects X with morphisms $h : X \to B$ and $k : X \to C$ with $f \circ k = g \circ h$, there is a unique morphisms $x : X \to A$ such that $f' \circ x = h$ and $g' \circ x = k$. △



According to Lemma A.21 in [EEGH15] or Fact 2.20 in [EEPT06], pushouts can be composed and decomposed if a commutative diagram like the one on the bottom is given and the following conditions hold:

- If (1) and (2) are pushouts, then the composition of both (1)+(2) is also a pushout.
- If (1) and (1)+(2) are pushouts, then (2) is also a pushout as a result of the decomposition.



Similarly, pullbacks can be composed and decomposed in the following way:

- If (1) and (2) are pullbacks, then the composition of both (1)+(2) is also a pullback.
- If (2) and (1)+(2) are pullbakcs, then (1) is also a pullback as a result of the decomposition.

The composition and decomposition of pullbacks is shown in Lemma A.24 in [EEGH15] or in Fact 2.27 in [EEPT06].

An important property which is part of the definition of $\mathcal{M}$-adhesive categories is the *van Kampern square* property. It defines special compatibilities of pushouts and pullbacks in a commutative cube. The main idea is that a pushout shall be stable under pullbacks and vice versa, i.e., that pullbacks shall be stable under combined pushouts and pullbacks (cf. [EEPT06, EEGH15]). For more details on the van Kampen square property, we refer to Def. 4.1 in [EEPT06] or Def. 4.1 in [EEGH15].

Another property that is necessary for the introduction of $\mathcal{M}$-adhesive categories is the *pushout-pullback compatibility* property. Its definition is provided in the following. Finally, we are able to define $\mathcal{M}$-adhesive categories.

Definition 2.1.13 (PO-PB Compatibility (cf. Def. 4.2 in [EEGH15]))**.** *A morphism class $\mathcal{M}$ in a category $\mathbf{C}$ is called POPB compatible if*

1. *$\mathcal{M}$ is a class of monomorphisms, contains all identities, and is closed under composition ($f : A \to B \in \mathcal{M}, g : B \to C \in M \Rightarrow g \circ f \in \mathcal{M}$).*
2. *$\mathbf{C}$ has pushouts and pullbacks along $\mathcal{M}$-morphisms, and $\mathcal{M}$-morphisms are closed under pushouts and pullbacks.* △

Definition 2.1.14 ($\mathcal{M}$-adhesive Category (cf. Def. 4.4 in [EEGH15]))**.** *A category $\mathbf{C}$ with a POPB compatible morphism class $\mathcal{M}$ is called an $\mathcal{M}$-adhesive category if pushouts in $\mathbf{C}$ along $\mathcal{M}$-morphisms are $\mathcal{M}$-van Kampen squares.* △

For more details, examples and more properties of $\mathcal{M}$-adhesivce categories, we refer to Chapters 4 and 5 in [EEGH15].

A special type of pushout which we use within this work, especially in Chap. 5, is the *effective pushout*. As also visualised in Fig. 2.6, intuitively in **Sets**, an effective pushout E is the union of B and C over a common D, whereas D is the intersection of B and C over a common A. For further reading and more details on effective pushouts, we refer to [Lac05, Gol11, SEM+12, EEGH15].

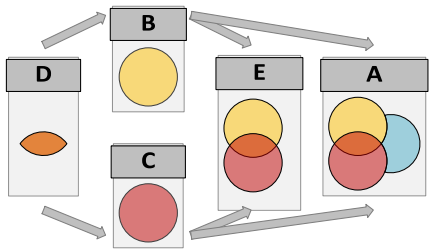
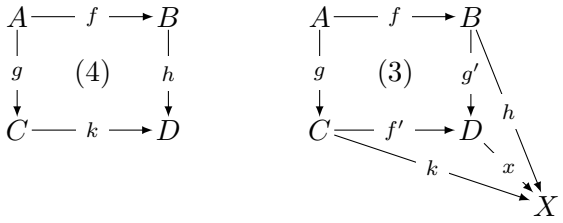


Figure 2.6: Example Effective Pushout in Sets

Definition 2.1.15 (Effective Pushout (cf. Def. 2.3 in [SEM+12] or Def. 4.23 in [EEGH15]))**.** *Given M-morphisms $h : B \to X$, $k : C \to X$ in an $\mathcal{M}$-adhesive category $(\mathbf{C}, \mathcal{M})$ and let (A, f, g) be obtained by the pullback (4) of h and k. Then pushout (3) of f and g is called effective, if the unique morphism $x : D \to X$ induced by pushout (3) is an $\mathcal{M}$-morphism.* △



2.1.5. (Negative) Application Conditions

Before we define algebraic typed attributed graph transformation (TAGT), we introduce graph conditions. Transformation rules form the basis of the TAGT. Each rule define a change on the graph. In practice, many rules shall be applied, if a certain context exist in the graph, or if a given context is not part of the graph. Those restrictions of the rule application are provided by *application conditions (AC)* (cf. [Wag95, EH86]. If a single application condition forbids a certain context in a graph, then it is called *negative application condition (NAC)* (cf. [HHT96, HP05]. Application conditions can be nested, i.e., different application conditions can be combined via logical operators. If a graph condition is combined with a graph rule, then it is called application condition (cf. Def. 2.1.18).

In the following, we cite the definition of graph conditions and the satisfaction of graph conditions. A graph condition is satisfied, if a morphism from the graph condition to a graph exists.

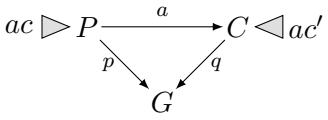
Definition 2.1.16 (Graph Condition (cf. Def. 2.7 in [EEGH15]))**.** *A (nested) graph condition ac over a graph P is of the form*

- $ac = true$,

- $ac = \neg ac'$,
- $ac = \exists(a, ac'')$,
- $ac = \wedge_{i \in \mathcal{I}} ac_i$, *or*
- $ac = \vee_{i \in \mathcal{I}} ac_i$,

where ac' *is a graph condition over* P, $a : P \to C$ *is a morphism,* ac'' *is a graph condition over* C, *and* $(ac_i)_{i \in \mathcal{I}}$ *with an index set* $\mathcal{I}$ *are graph conditions over* P. △

Definition 2.1.17 (Satisfaction of Graph Conditions (cf. Def. 2.8 in [EEGH15]))**.** *Given a graph condition* ac *over* P, *a morphism* $p : P \to G$ *satisfies* ac, *written* $p \models ac$, *if*



- $ac = true$,
- $ac = \neg ac'$ *and* $p \not\models ac'$,
- $ac = \exists(a, ac')$ *and there exists an injective morphism* q *with* $q \circ a = p$ *and* $q \models ac'$,
- $ac = \wedge_{i \in \mathcal{I}} ac_i$ *and* $\forall i \in \mathcal{I} : p \models ac_i$, *or*
- $ac = \vee_{i \in \mathcal{I}} ac_i$ *and* $\exists i \in \mathcal{I} : p \models ac_i$. △

2.1.6. Algebraic (Typed) (Attributed) Graph Transformation & Graph Grammar

First ideas for algebraic typed attributed graph transformation was developed in [EKMR99] (among others), which was later formalised in [EPT04].

We will now present the formalisation of algebraic graph transformation based on typed attributed graps. Similar definitions exist for attributed graphs, typed graphs or graphs in general (cf. [EEPT06, EEGH15]).

A graph transformation rule consists of three graphs: A left-hand side (LHS) which describes a pattern that shall be found in a graph. The gluing part (middle) shows, which elements will be deleted by the rule, i.e., it is a real subset of the LHS if the rule shall delete something, or it is equal to the LHS, if the rule deletes nothing. The right-hand side (RHS) indicates the result of the rule application, i.e., if we compare the gluing part with the RHS, we will notice that if the RHS and the gluing part are equal, then nothing will be added by the rule. If the gluing part is a real subset of the RHS, then elements will be added by the rule. Rules might be equipped with application conditions, in order to restrict the rule application. Graph transformation rules are also called *graph productions*.

Definition 2.1.18 (Rule (cf. Def. 2.9 in [EEGH15] or Def. 3.1 in [EEPT06]))**.** *A rule* $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R; ac)$ *consists of graphs* L, K, *and* R, *called left-hand side, gluing, and right-hand side, respectively, two injective morphisms* l *and* r, *and a graph condition* ac *over* L, *called application condition.* △

$$ac \triangleright L \xleftarrow{\quad l \quad} K \xrightarrow{\quad r \quad} R$$

The gluing condition prevents rule applications, if the application will, result in dangling edges. Dangling edges are edges with a missing source node or a missing target node or both. Note, edges without a source or target node cannot exist in a graph.

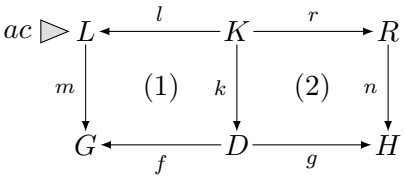
Definition 2.1.19 (Gluing Condition (cf. Def. 3.9 in [EEPT06]))**.** *Given a (typed) graph production* $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, *a (typed) graph* G, *and a match* $m : L \to G$ *with* $X = (V_X, E_X, s_X, t_X)$ *for all* $X \in \{L, K, R, G\}$, *we can state the following definitions:*

- *The gluing points* GP *are those nodes and edges in* L *that are not deleted by* p, *i.e.,* $GP = l_V(V_K) \cup l_E(E_K) = l(K)$.
- *The identification points* IP *are those nodes and edges in* L *that are identified by* m, *i.e.,* $IP = \{v \in VL \mid \exists w \in V_L, w \neq v : m_V(v) = m_V(w)\} \cup \{e \in E_L \mid \exists f \in E_L, f \neq e : m_E(e) = m_E(f)\}$.
- *The dangling points* DP *are those nodes in* L *whose images under* m *are the source or target of an edge in* G *that does not belong to* $m(L)$, *i.e.,* $DP = \{v \in V_L \mid \exists e \in E_G \setminus m_E(E_L) : s_G(e) = m_V(v)$ *or* $t_G(e) = m_V(v)\}$.

p *and* m *satisfy the gluing condition if all identification points and all dangling points are also gluing points, i.e.,* $IP \cup DP \subseteq GP$. △

The graph transformation defines the process of applying a rule to a graph. Morphism $m : L \to G$ is called *match*. Morphism $n : R \to H$ is called *comatch*. The direct transformation of a graph via one rule is constructed using the double-pushout approach (cf. [EHK+97, EEPT06]).

Definition 2.1.20 (Algebraic (Typed) (Attributed) Graph Transformation (cf. Def. 2.10 in [EEGH15] or cf. Def. 3.2 in [EEPT06]))**.** *Given a rule* $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R; ac)$, *a graph* G, *and a match* $m : L \to G$ *such that* $m \models ac$ *then a direct transformation* $G \overset{p,m}{\Longrightarrow} H$ *from* G *to a graph* H *is given by the pushouts (1) and (2). A sequence of direct transformations is called a* transformation. △



Example 2.1.6 (Rule with Negative Application Condition)**.** *In Fig. 6.13 we illustrate the typed attributed graph transformation rule* P2_merge empty Argument node with SimpleElement in Parenthesis_case 2*. In the screenshot taken in Henshin [Hen16b], the gluing graph is not illustrated. It is illustrated implicitly by colors: If an element is marked with red in the LHS, then it will be deleted by the rule. If an element is marked in green in the RHS, then it will be created by the rule. In both cases, the colored elements are not available in the gluing part of the rule.*

The example rule in Fig. 6.13 is equipped with one negative application condition AC0 *which forbids the application of the rule, if node* : SimpleElement *is followed by a node* : Element*.*

The mapping from node : SimpleElement *to the same element in the NAC is visualised by mapping number* [2]*.*

Note, in theory, this rule is not applicable in the case described by the NAC, even if the NAC is missing, due to a violation of the gluing condition: If node : SimpleElement *will be deleted, then edge* : next *would be a dangling edge. In the implementation of Henshin, the developers had different options in dealing with violations of the gluing conditions, e.g., abort the transformation if the gluing condition is not fulfilled or another possibility is to delete the dangling edge automatically. For this example rule, the current implementation would lead to an undesired effect, which is the reason why we defined this NAC.* △

Example 2.1.7 (Rule with Nested Application Condition)**.** *The graph transformation rule* set to_Element Step next *which is illustrated in Fig. 6.30, contains a nested application condition:* $\neg(AC0 \wedge AC1)$*, i.e., if a morphism from* $AC0$ *or* $AC1$ *(or from both) can be found to the current graph, then the rule is not applicable (because of the* $\neg$*).* △

Finally, we review the terms of algebraic (typed) (attributed) graph transformation system, graph grammar and language.

Definition 2.1.21 (Graph Transformation System, Graph Grammar and Language (cf. Def. 7 in [EPT04] or Def. 3.4 in [EEPT06]))**.** *A typed attributed graph transformation system* $GTS = (DSIG, ATG, S, P)$ *based on* $(\mathbf{AGraphs}_{\mathbf{ATG}}, \mathcal{M})$ *consists of a data type signature* $DSIG$*, an attributed type graph* ATG*, a typed attributed graph* S*, called start graph, and a set* P *of productions, where*

1. *a production* $p = (L \xleftarrow{l} K \xrightarrow{r} R; ac)$ *consists of typed attributed graphs* L, K *and* R *attributed over the term algebra* $T_{DSIG}(X)$ *with variables* X, *called left hand side* L, *gluing object* K *and right hand side* R *respectively, and morphisms* $l, r \in \mathcal{M}$, *i.e.,* l *and* r *are injective and isomorphisms on the data type* $T_{DSIG}(X)$,

2. *a direct transformation* $G \overset{p,m}{\Longrightarrow} H$ *via a production* p *and a morphism* $m : L \to G$ *is given by the double pushout diagram (cf. Def. 2.1.20), where (1) and (2) are pushouts in* $\mathbf{AGraphs_{ATG}}$,

3. *a typed attributed graph transformation (short transformation) is a sequence* $G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$ *of direct transformations, written* $G_0 \overset{*}{\Rightarrow} G_n$,

4. *the language* $\mathcal{L}(GTS)$ *is defined by* $\mathcal{L}(GTS) = \{G \mid \exists$*(typed) graph transformation* $S \overset{*}{\Rightarrow} G\}$. △

2.1.7. Additions: Inheritance & Edge Types

Inheritance and different edge typed, e.g., containment edges vs. "normal" edges are interesting concepts for algebraic typed attributed graph transformation, especially, when the theoretical concepts shall be applied in practice. For the practical implementaion of case studies and industrial projects, many tools supporting algebraic typed attributed graph transformation were implemented [ABJ+10, EHGB12, Tae04, NNZ00, LAS14, VBH+16].

Inheritance for graph transformation approaches was introduced, formalised and extended in [BEL+03, TR05, LBE+07, GLEO12b]. In [BEL+03, TR05], a concept for type graphs with *inheritance* was introduced which leads to a new concept of typed graphs with inheritance. The algebraic graph transformation rules were divided into abstract rules and derived concrete rules. Consequently, the graph transformation approach was extended to the new notation of inheritance. In addition, *multiplicities* of edges in graphs were introduced and formalised. These concepts were extended and formal analysis techniques were applied in later research work, e.g. in [LBE+07, GLEO12b].

The distinction between different kinds of edges, i.e., between normal edges and containment edges was introduced in [BET08] and extended in [BET12]. The authors developed a new kind of graph, called *C-graph*, for graphs with containment edges. In addition, the term *rooted graph* was defined which fulfills the EMF-specification (cf. [EMF16]), that each graph has to contain a root node and each other node in this graph has to be derived from the root node.

2.2. Triple Graph Grammars

Triple graph grammars (TGGs) were introduced in 1994 by Schürr et al. [Sch95] as a new technique for specifying graph translators. They are used for determining a pair of models that are in relationship with each other. The relationship between both models is given by means of an intermediate model, the correspondence model. Therefore, TGGs provide a bidirectional model transformation language [SK08, ALS15] that are used for uni- and bidirectional model transformations between two models, for model integrations and for model synchronisations (cf. [KW07]). Model integration means, given two models and a TGG, then the correspondence part will be derived, in order to relate both models with each other. Given a triple grah and a TGG, then model synchronisation describes the process of propagating changes perfromed in the source model to the target model, or vice versa, respectively.

According to the algebraic definition Def. 3.3 in [HEO+15], a TGG is defined by a set of triple rules, a triple start graph, which is usually empty, and a triple type graph. The language $\mathcal{L}(TGG)$ of triple graphs is generated by the TGG, i.e., $\mathcal{L}(TGG)$ is the set of all possible triple graphs that can be derived out of the start graph in applying the set of triple rules. Each triple graph of $\mathcal{L}(TGG)$ can be typed over the triple type graph.

A triple graph consists of three graphs: A source graph, a target graph and a correspondence graph which establishes links between nodes of the source graph and nodes of the target graph. A triple graph is typed over a three-component meta-model. A triple graph morphism from one triple graph to another is composed of three morphisms: one between the source, one between the target and one between the correspondece graphs.

TGGs are formalised using the set-theoretical approach in [KS06]. This formalisation was extended in [EEE+07, EEH08] to typed attributed graphs. In Thm. 7.2 of [EEGH15], it is shown that the categories of triple graphs **TrGraphs** (triple graphs), $\mathbf{TrGraphs_{TG}}$ (typed triple graphs), **ATrGraphs** (attributed triple graphs), and $\mathbf{ATrGraphs_{TG}}$ (typed attributed tryiple graphs) are $\mathcal{M}$-adhesive, i.e., the properties of $\mathcal{M}$-adhesive categories also hold for those triple graph categories.

In the following, we will provide the formal definitions of the terms we introduced informally above. The definitions are mainly based on [EEGH15]. In our work, we use typed attributed triple graphs.

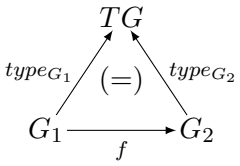
Definition 2.2.1 (Triple Graph (cf. Sec. 3.3 [EEGH15] or Def. 1 in [EEH08]))**.** *A triple graph* $G = (G^S \xleftarrow{s_G} G^C \xrightarrow{t_G} G^T)$ *consists of graphs* G^S, G^C, *and* G^T, *called source, correspondence, and target graphs, and two graph morphisms* $s_G : G^C \to G^S$ *and* $t_G : G^C \to G^T$ *mapping the correspondence to the source and target graphs. G is called empty, if* G^S, G^C, *and* G^T *are empty graphs.* △

Definition 2.2.2 (Triple Graph Morphism (cf. Sec. 3.3 [EEGH15] or Def. 1 in [EEH08]))**.** *Given triple graphs* $G = (G^S \overset{s_G}{\leftarrow} G^C \overset{t_G}{\rightarrow} G^T)$ *and* $H = (H^S \overset{s_H}{\leftarrow} H^C \overset{t_H}{\rightarrow} H^T)$*. A triple graph morphism* $m = (m_S, m_C, m_T) : G \to H$ *consists of graph morphisms* $m_S : G^S \to H^S$*,* $m_C : G^C \to H^C$ *and* $m_T : G^T \to H^T$ *such that* $m_S \circ s_G = s_H \circ m_C$ *and* $m_T \circ t_G = t_H \circ m_C$*, i.e., both parts of the diagram commute (annotated in the diagram with* $(=)$*). The triple graph morphism* m *is injective, if morphisms* m_S*,* m_C*, and* m_T *are injective.* △

$$\begin{array}{ccccccc} G = & (G^S & \xleftarrow{s_G} & G^C & \xrightarrow{t_G} & G^T) \\ \downarrow m & \downarrow m_S & (=) & \downarrow m_C & (=) & \downarrow m_T \\ H = & (H^S & \xleftarrow[s_H]{} & H^C & \xrightarrow[t_H]{} & H^T) \end{array}$$

Definition 2.2.3 (Typed Triple Graph and Typed Triple Graph Morphism (cf. Sec. 3.3 [EEGH15]))**.** *A typed triple graph* $(G, type_G)$ *is given by a typing morphism* $type_G : G \to TG$ *from the triple graph* G *into a given triple type graph* TG*.*

Given triple graphs $G_1 = (G_1^S \overset{s_{G_1}}{\leftarrow} G_1^C \overset{t_{G_1}}{\rightarrow} G_1^T)$ *and* $G_2 = (G_2^S \overset{s_{G_2}}{\leftarrow} G_2^C \overset{t_{G_2}}{\rightarrow} G_2^T)$*. A typed triple graph morphism* $f : (G_1, type_{G_1}) \to (G_2, type_{G_2})$ *is a triple graph morphism* f *such that* $type_{G_2} \circ f = type_{G_1}$*, i.e., the diagram on the right commutes.* △



Example 2.2.1 (Triple Type Graph and Triple Graph)**.** *The triple type graph of our running example consists of the following three components:*

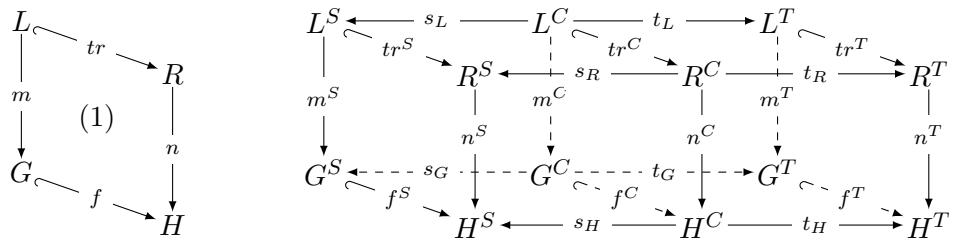
- *the source meta-model (SPELL meta-model), which is explicitly given in Appendix A.2,*
- *the target meta-model (SPELL-Flow meta-model), which is explicitly given in Appendix A.3, and*
- *the correspondence meta-model (CORR meta-model), which is explicitly given in Appendix A.4.*

In Ex. 5.1.3, we present a reduced version of the triple graph of our running example. A detailed example triple graph is given in Appendix A.6. The example triple graph is typed over the type graph that is given by the SPELL meta-model, the SPELL-Flow meta-model and the CORR meta-model. △

Triple rules extend the triple graph in creating elements in the source, target and correspondence part simultaneously. They are non-deleting and therefore, the triple rule tr can be applied using the single pushout approach,

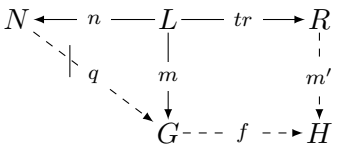
i.e., $tr : L \to R$. If we apply the general definition of rules to triple rules, i.e., the double-pushout approach, then we obtain $tr : L \xleftarrow{l} K \xrightarrow{r} R$, where l is the identical morphisms and thus, $L = K$. Hence, the double pushout approach and single pushout approach are equivalent for triple rules. Triple rules can be equipped with application conditions ac that demand for a certain context (positive application condition) or forbid a specified context (negative application condition) in a triple graph in order to be applicable. In the following, the definitions for triple rules, triple rule transformation and negative application conditions for triple rules are given. Finally, we are able to define triple graph grammars (TGGs) and the language $\mathcal{L}(TGG)$ formed by the TGG. The language $\mathcal{L}(TGG)^S$ denotes the source language, i.e., $\mathcal{L}(TGG)$, but restricted to the source component. Dually, $\mathcal{L}(TGG)^T$ denotes the target language, i.e., $\mathcal{L}(TGG)$, but restricted to the target component.

Definition 2.2.4 (Triple Rule and Transformation (cf. Def. 3.7 in [EEGH15] or Def. 2 in [EEH08]))**.** *A triple rule $tr = (tr : L \to R, ac)$ consists of triple graphs L and R, an $\mathcal{M}$-morphism $tr = (tr_S, tr_C, tr_T) : L \to R$, i.e., $tr_S : L^S \to R^S$, $tr_C : L^C \to R^C$, and $tr_T : L^T \to R^T$ (see upper part of the diagram below) and an application condition ac over L. The morphism $tr = (tr_S, tr_C, tr_T)$ is injective on the graph component and isomorphic on the data part.*



Given a triple graph G, a triple rule $tr = (tr, ac)$ and a match $m : L \to G$ with $m \models ac$, then a direct triple transformation $G \overset{tr,m}{\Longrightarrow} H$ of G via tr and m is given by pushout (1) in the diagram above, which is constructed as the component-wise pushouts in the S -, C-, and T -components, where the morphisms s_H and t_H are induced by the pushout of the correspondence component. In addition to H, we obtain comatch $n : R \to H$ and transformation inclusion $f : G \to H$. A direct transformation is also called triple graph transformation (TGT) step. △

Definition 2.2.5 (Negative Application Conditions in Triple Rules (cf. Def. 1 in [EEHP09]))**.** *Given a triple graph G and a triple rule $tr : L \to R$. A general negative application condition (NAC) (N, n) consists of a triple graph N and an injective triple graph morphism $n : L \to N$.*



A NAC with $n = (n^S, id_{L_C}, id_{L_T})$ is called source NAC and a NAC with $n = (id_{L_S}, id_{L_C}, n^T)$ is called target NAC. This means that source-target NACs, i.e. either source or target NACs, prohibit the existence of certain structures either in the source or in the target part only. A match $m : L \to G$ is NAC consistent if there is no injective $q : N \to G$ such that $q \circ n = m$. A triple transformation $G \Rightarrow H$ is NAC consistent if all matches are NAC consistent. △

Definition 2.2.6 (Triple Graph Grammar (Def. 3.3 in [HEO+15] or Def. 3.11 in [EEGH15]))**.** *A triple graph grammar $TGG = (TG, TR, S)$ consists of a triple type graph TG, a set TR of triple rules and a triple graph S called triple start graph. (Note that usually, S is empty, i.e., $S = \varnothing$.)*

A language of triple graphs generated by TGG is given by $\mathcal{L}(TGG) = \{G \mid \exists$ triple transformation $S \overset{*}{\Rightarrow} G$ via rules in $TR\}$. The source language $\mathcal{L}(TGG)^S = \{G^S \mid (G^S \overset{s_G}{\leftarrow} G^C \overset{t_G}{\rightarrow} G^T) \in \mathcal{L}(TGG)\}$ contains all graphs that are the source component of a derived triple graph. Similarly, the target language $\mathcal{L}(TGG)^T = \{G^T \mid (G^S \overset{s_G}{\leftarrow} G^C \overset{t_G}{\rightarrow} G^T) \in \mathcal{L}(TGG)\}$ contains all derivable target components. The model transformation relation $MT_R = \{(G^S, G^T) \in \mathcal{L}(TGG)^S \times \mathcal{L}(TGG)^T \mid \exists G = ((G^S \overset{s_G}{\leftarrow} G^C \overset{t_G}{\rightarrow} G^T) \in \mathcal{L}(TGG)\}$ defines the set of all consistent pairs (G^S, G^T) of source and target models. △

Example 2.2.2 (Triple Graph Grammar and Triple Rules)**.** *The triple graph grammar of our running example is given by:*

- *The triple type graph (cf. Ex. 2.2.1),*
- *an empty start graph $\varnothing$, and*
- *the set of triple rules. In Ex. 4.1.6 and Ex. 6.1.1, we present a small subset of the triple rules of our running example and of our case study.*

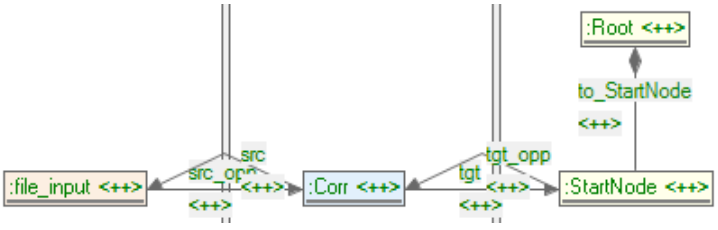


Figure 2.7: Triple Rule T_file_input-2-StartNode

We show explicitly the screenshot of one triple rule, which we will use as basis example in the remainder of this sub-section. The triple rule

T_file_input-2-StartNode *in Fig. 2.7 contains one node* file_input *in the source domain, i.e., the SPELL domain, and two nodes* Root *and* StartNode *connected by edge* to_StartNode *in the target domain, i.e., the SPELL-Flow domain. The correspondence part establishes a relation between nodes* file_input *and* StartNode*. This triple rule creates all elements in all three domains (indicated by green* $< ++ >$ *symbols), thus it does not imply any existing structures.* △

Remark 2.2.1 (Visual Notation for Rules)**.** *In the examples and screenshots we illustrate within this thesis, we use a compact visual notation. For triple and operational rules it means that we will not show the left-hand side and the right-hand side of the rule separately. Instead, they are visualised together. There, the following notation is used:*

- *A green* $< ++ >$ *marker indicates that the corresponding element shall be created by the rule. Consequently, the LHS of that rule is the given rule without the elements marked with* $< ++ >$*. The RHS of that rule is identical to the LHS plus all elements marked with* $< ++ >$*.*

In Chap. 6, we also include screenshots of plain graph rules taken in Henshin. Those screenshots visualise the LHS and the RHS of the rule. Elements marked in red indicate that they will be deleted by the plain graph rule. Elements marked in green indicate additions. △

2.2.1. Operational Rules for Model Transformations via TGGs

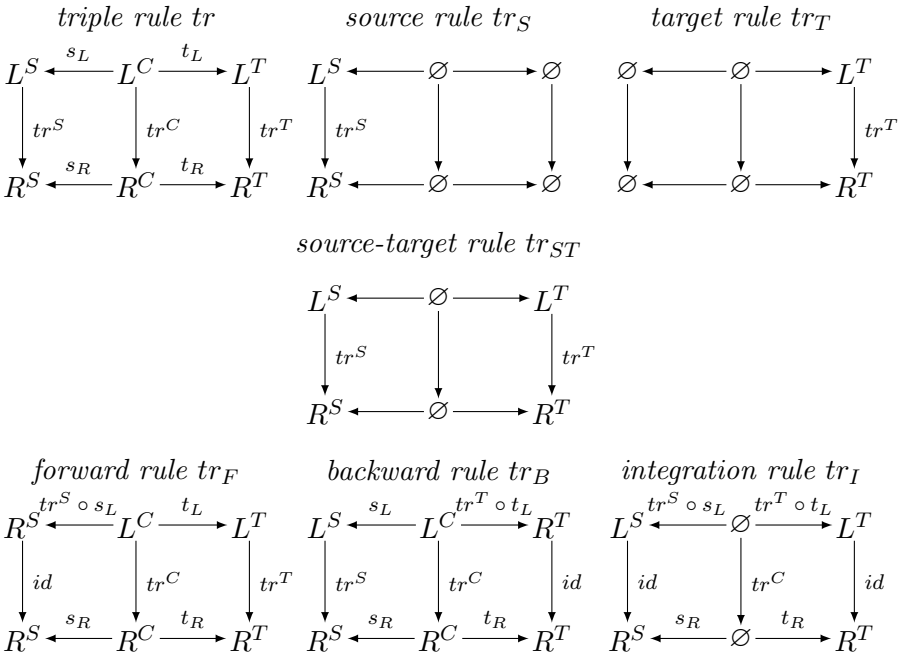
In the practical part of our work (cf. Chap. 6), we use TGGs for unidirectional and bidirectional model transformations. In future work, it is also desired to extend this approach to model synchronisations (cf. Chap. 4). In detail uni- or bidirectional model transformations using TGGs means, that given a source model (or target model, repectively), we derive the target model (or source model, respectively) using the given TGG. In order to be able to derive a model in another domain, we need special kinds of triple rules: the so-called operational rules. They are derived out of the given set of triple rules. We distinuguish between the following kinds of operational rules that are derived out of a triple rule tr:

- source rule tr_S,
- forward rule tr_F,
- target rule tr_S,
- backward rule tr_B,
- source-target rule tr_{ST},
- integration rule tr_I,

- forward transformation rule tr_{FT},
- backward transformation rule tr_{BT},
- integration translation rule tr_{IT}, and
- consistency creating rule tr_{CC}.

The concept of operational rules is presented in [KS06] and formalised and extended in [EEE+07, EEHP09]. In the following, we will define the above-mentioned operational rules. The given definitions are taken from [EEE+07, EEHP09, EEGH15]. Note, in this work, we illustrate the definitions of operational rules without application conditions. For the definitions of operational rules with application conditions, we refer to Sec. 7.2 in [EEGH15], (especially Defs. 7.6, 7.7., 7.8, 7.12 and Lemma 7.9).

Definition 2.2.7 (Derived Operational Rules Without Application Conditions (cf. Def. 7.11 in [EEGH15]))**.** *Given a triple rule* $tr = (tr : L \to R, ac)$*. We derive its operational rules* tr_S *,* tr_T *,* tr_{ST}*,* tr_F*,* tr_B*,* tr_I *without application conditions according to the diagram below. The data component for each inclusion* $\varnothing \to X$ *for a triple graph* X *is given by an identity, i.e., the construction does not change the data component.* △



It is obvious, that the construction of source and target rules, as well as of forward and backward rules is symmetric. Hence, it is sufficient to explicitly show the definition of forward model transformations based on source and forward rules. The backward case is dually. In definition Def. 2.2.8, we

use the term *source consistency*. First, we need to explain the term *match consistency*. Given a sequence of source rule and a sequence of forward rules that are both derived from the same triple rules. Then, match consistency means that the match of each forward rule is determined by the comatch of the corresponding source rule, i.e., the forward rule matches and extends the same structure in the source component of the triple graph that was built up by its corresponding source rule. Then, source consistency means that if we have a source transformation sequence, i.e., a graph was created using source rules, only. Then, the forward transformation sequence is match consistent. For more detailed definitions we refer to Def. 4 (which includes source consistency) and Def. 3 (match consistency) in [EEH08].

Definition 2.2.8 (Model Transformations Based on Source and Forward Rules (cf. Def. 2 in [EEHP09] or Def. 7.23 in [EEGH15]))**.** *A model transformation sequence* $(G^S, G_0 \overset{tr*_F}{\Longrightarrow} G_n, G^T)$ *is given by a source graph* G^S*, a target graph* G^T*, and a source consistent forward transformation* $G_0 \overset{tr*_F}{\Longrightarrow} G_n$ *with* $G_0 = (G^S \overset{\varnothing}{\leftarrow} \varnothing \overset{\varnothing}{\rightarrow} \varnothing)$ *and* $G_n^T = G^T$*.*

A (forward) model transformation $MT_F : \mathcal{L}(TGG)^S \Rrightarrow \mathcal{L}(TGG)^T$ *is defined by all (forward) model transformation sequences.* △

The model integration is defined similarily: Given a source and a target model, then the integrated model, i.e., the correspondence graph, is created using the set of source-target rules and model integration rules that are generated out of the set of triple rules. For a detailed definition, we refer to Sec. 4 in [EEH08].

Based on the model transformation using source and forward rules an extension to forward and backward translation rules, respectively, was developed in [HEOG10] and extended in [HEGO10a, HEGO14]. The main idea is to combine the source and forward rule into one rule: the forward translation rule. The backward case is similar. In order to keep track of elements in the source model that were already translated, so-called *translation attributes* were introduced. These are markers that indicate the translation status of each element, i.e., if the marker is set to **F** (false), then the element is not translated. If the element is set to **T** (true), then this element has been translated and shall not be translated, again. In [HEOG10] it is additionally shown, that the model transformation using source and forward rules is equivalent to the model transformation using forward translation rules. The latter is also implemented in HenshinTGG, that we were using in the applied part of this work (cf. Chap. 6).

Definition 2.2.9 (Graph with Translation Attributes (cf. Def. 2 in [HEOG10]))**.** *Given an attributed graph* $AG = (G, D)$ *and a subgraph* $G_0 \subseteq G$ *we call* AG' *a graph with translation attributes over* AG *if it extends* AG *with one boolean-valued attribute* tr_x *for each element* x *(node,*

edge or attribute) in G_0 and one boolean-valued attribute tr_{x_a} for each attribute associated to such an element x in G_0. The set of all these additional translation attributes is denoted by Att_{G_0}. $Att^v_{G_0}$, where $v = \mathbf{T}$ or $v = \mathbf{F}$, denotes a translation graph where all the attributes in Att_{G_0} are set to v. By $AG' = AG \oplus Att_{G_0}$ we specify that AG is extended by the attributes in Att_{G_0}. Moreover, we define $Att^v(AG) := AG \oplus Att^v_G$. △

In the definition we provide above, the translation markers belong to the set of data structures. In [HEGO14], a separation of translation markers from the data structures was worked out. For definitions of this so-called *family with translation attributes* and a modification of the above-mentioned definition of a graph with translation attributes according to the new developments, we refer to Defs. 2.11 and 2.12 in [HEGO14] or also to Defs. 7.25 and 7.26 in [EEGH15].

Example 2.2.3 (Graph with Translation Attributes)**.** *In Fig. 2.8 we illustrate a screenshot of the SPELL abstract syntax graph (ASG) that we translated using HenshinTGG. It is an excerpt of our running example in Ex. 1.3.1. It visualises the sub-tree of* Step 5 *that belongs to source code lines 84 - 90. The following figure shows a screenshot of the same ASG but extended by translation markers. In this screenshot, all elements are marked with translation markers that are set to* **T***, except for all* NEWLINE *nodes and edges* nl_post *that are connected to those nodes. They are equipped with* **F***-markers. In HenshinTGG, the notation is as follows: Elements with green boxes, lines and green textcolors and that contain marker* [tr] *are equipped with translation markers that are set to* **T***. Elements with red boxes, lines and red text color and that contain marker* [!tr] *are equipped with translation markers that are set to* **F***.* △

A forward translation rule is an operational rule that includes translation attributes. It can be seen as a combination of the corresponding source rule and forward rule. In the source domain, the forward translation rule is equipped with translation attributes, i.e., it does not create any element in the source domain. Similar to the forward rule, it creates elements in the correspondence and the target domain. Due to the use of translation attributes that change markers in the mapped model from **F** to **T**, forward translation rules are deleting, but only for translation attributes. This mean that the value **F** will be deleted and replaced by **T**. The following definitions of operational rules are based on [HEOG10]. Due to the extensions of family with translation attributes (as described above), those definitions were adapted, too. For the updated definitions, we refer to [HEGO14] or [EEGH15].

Definition 2.2.10 (Forward Transformation Rule (cf. Def. 3 in [HEOG10]))**.** *Given a triple rule $tr = (L \to R)$. The forward translation*

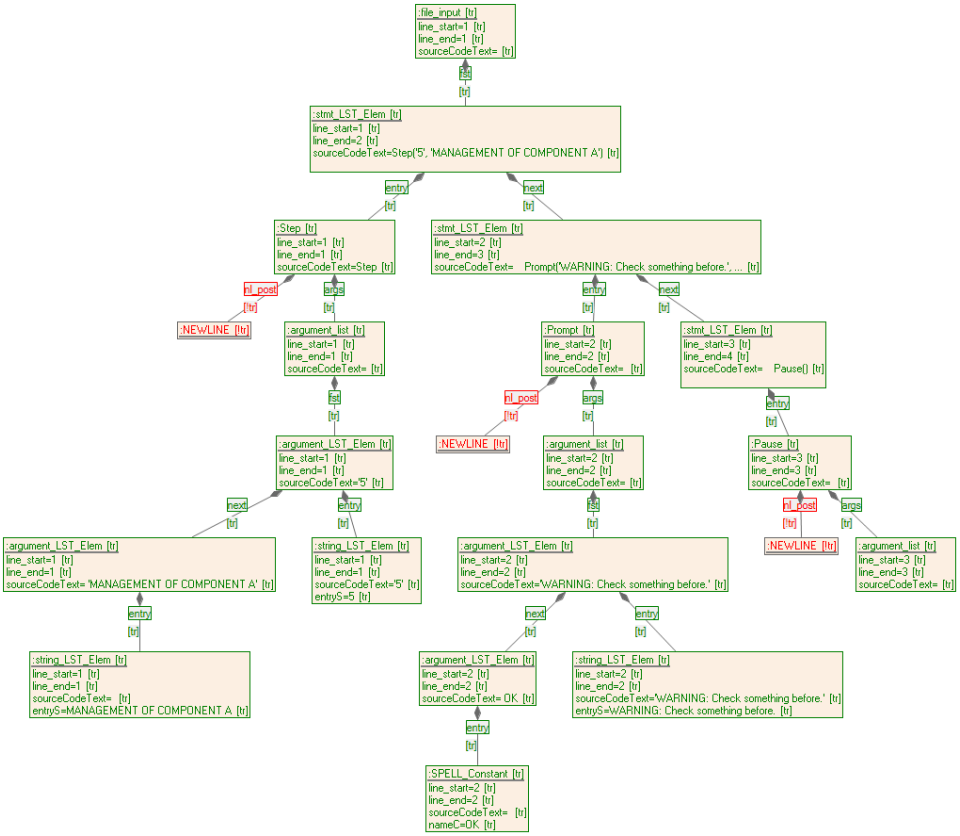


Figure 2.8: Example SPELL ASG with Translation Markers

rule of tr is given by $tr_{FT} = (L_{FT} \overset{l_{FT}}{\leftarrow} K_{FT} \overset{r_{FT}}{\rightarrow} R_{FT})$ defined as follows using the forward rule $(L_F \overset{tr_F}{\rightarrow} R_F)$ and the source rule $(L_S \overset{tr_S}{\rightarrow} R_S)$ of tr, where we assume w.l.o.g. that tr is an inclusion:

- $K_{FT} = L_F \oplus Att^{\mathbf{T}}_{L_S}$,
- $L_{FT} = L_F \oplus Att^{\mathbf{T}}_{L_S} \oplus Att^{\mathbf{F}}_{R_S \setminus L_S}$,
- $R_{FT} = R_F \oplus Att^{\mathbf{T}}_{L_S} \oplus Att^{\mathbf{T}}_{R_S \setminus L_S} = R_F \oplus Att^{\mathbf{T}}_{R_S}$,
- *l_{FT} and r_{FT} are the induced inclusions.* △

Note that the gluing condition for all kinds of operational rules with translation attributes, (i.e., forward translation rules backward translation rules, integration translation rules and consistency creating rules) is always fulfilled, because they are only deleting for translation attributes and therefore, no dangling edges will occur. For the formal proof we refer to Fact 7.31 in [EEGH15].

Example 2.2.4 (Forward Transformation Rule)**.** *The forward transformation rule* FT_T_file_input-2-StartNode *(cf. Fig. 2.9) is derived from triple rule* T_file_input-2-StartNode *(cf. Fig. 2.7). The node in the source domain is equipped with translation attribute* $\mathsf{tr}_{\mathsf{file\_input}} = \mathbf{F} \to \mathbf{T}$*. This means that a match from this FT-rule is only possible to a source model which contains a node* file_input *that hasn't been translated, yet. Then, in applying this FT-rule, the structures in the correspondence and target domains will be created.* △

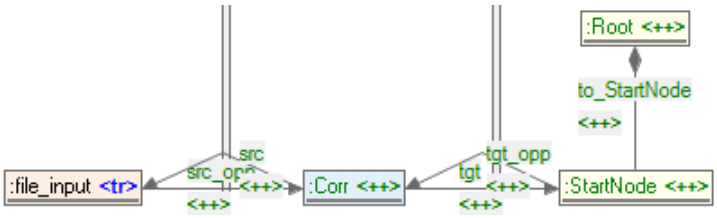


Figure 2.9: Forward Translation Rule FT_T_file_input-2-StartNode

Definition 2.2.11 (Model Transformation Based on Forward Translation Rules (cf. Def. 5 in [HEOG10]))**.** *A model transformation sequence* $(G_S, G'_0 \overset{tr*_{FT}}{\Longrightarrow} G'_n, G_T)$ *based on forward translation rules consists of a source graph* G_S*, a target graph* G_T*, and a complete TGT-sequence* $G'_0 \overset{tr*_{FT}}{\Longrightarrow} G'_n$ *with almost injective matches,* $G'_0 = (Att^{\mathbf{F}}(G_S) \leftarrow \varnothing \to \varnothing)$ *and* $G'_n = (Att^{\mathbf{T}}(G_S) \leftarrow G_C \to G_T)$*.*

A model transformation $MT : \mathcal{L}(TGG)^{S0} \Rrightarrow \mathcal{L}(TGG)^{T0}$ *based on forward translation rules is defined by all model transformation sequences* $(G_S, G'_0 \overset{tr*_{FT}}{\Longrightarrow} G'_n, G_T)$ *based on forward translation rules with* $G_S \in \mathcal{L}_{S0}$ *and* $G_T \in \mathcal{L}(T0)$*. All these pairs* (G_S, G_T) *define the model transformation relation* $MTR_{FT} \subseteq \mathcal{L}(TGG)^{S0} \times \mathcal{L}(TGG)^{T0}$*. The model transformation is terminating if there are no infinite TGT- sequences via forward translation rules and almost injective matches starting with* $G_0 = (Att^{\mathbf{F}}(G_S) \leftarrow \varnothing \to \varnothing)$ *for some source graph* G_S*.* △

The language $\mathcal{L}(TGG)^{S0}$ is the set of models that can be generated out of (and parsed by) the set of source rules TR_S, i.e., $\mathcal{L}(TGG)^{S0} = \{G_S \mid \varnothing \overset{*}{\Rightarrow} (G_S \leftarrow \varnothing \to \varnothing) via TR_S\}$. $\mathcal{L}(TGG)^{S0}$ is possibly larger than $\mathcal{L}(TGG)^{S}$, still the following relation holds: $\mathcal{L}(TGG)^{S} \subseteq \mathcal{L}(TGG)^{S0}$. The similar relation $\mathcal{L}(TGG)^{T} \subseteq \mathcal{L}(TGG)^{T0}$ holds for $\mathcal{L}(TGG)^{T0}$ which is the language that is generated out of (and parsed by) the set of target rules TR_T.

In Thm. 1 in [HEOG10] or in Fact 7.36 in [EEGH15], it is shown, that both model transformation concepts, i.e., model transformation using forward rules and model transformation based on forward translation rules, are equivalent. The *backward transformation rules* are definied dually to forward translation rules. Hence, we will not present the definitions for

the backward transformation case based on backward transformation rules, explicitly.

In Thm. 2 in [HEOG10] it is shown that each model transformation based on forward transformation rules is terminating, correct and complete. Termination means that each FT-rule changes at least one translation attribute. Correctness means that each triple graph resulting out of the TGT-sequence belongs to the language $\mathcal{L}(TGG)$. Completeness means that if we have a source model being element of the source language $\mathcal{L}(TGG)^S$, then there exists also a triple graph being element of the language generated by all triple rules $\mathcal{L}(TGG)$ with the source model in the source domain of the triple graph. In addition, the triple graph can be generated out of the source model by a TGT-sequence based on FT-rules.

For better clarity on completeness of a forward translation sequence, we cite the following definition:

Definition 2.2.12 (Complete Forward Translation Sequence (cf. Def. 3.11 in [GHN+13b]))**.** *A forward translation sequence* $G_0 \overset{tr^*_{FT}}{\Longrightarrow} G_n$ *with almost injective matches is called* complete *if* G_n *is* completely translated*, i.e., all translation attributes of* G_n *are set to true (“***T***”).* △

Example 2.2.5 (Backward Transformation Rule)**.** *The backward transformation rule* BT_T_file_input-2-StartNode *(cf. Fig. 2.10) that is generated out of triple rule* T_file_input-2-StartNode *(cf. Fig. 2.7), is dual to the corresponding forward translation rule.* △

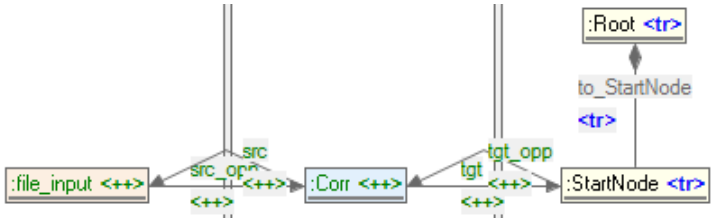


Figure 2.10: Backward Translation Rule BT_T_file_input-2-StartNode

The set of model *integration translation rules* are gernerated out of the set of triple rules. Given a source model and a target model, then the correspondence model will be derived using the set of integration translation rules (IT-rules, TR_{IT}). IT-rules are operational rules that are equipped with translation attributes in the source and target domains. Thus, they are similar to forward and backward rules. The model transformation based on IT-rules is defined analogously to the model transformation based on FT-rules or BT-rules, respectively (cf. Remark 7.43 in [EEGH15]).

Example 2.2.6 (Integration Translation Rule)**.** *We visualise a screenshot of the integration translation rule* IT_T_file_input-2-StartNode *in Fig. 2.11.*

It is again generated out of triple rule T_file_input-2-StartNode *(cf. Fig. 2.7). The generation of the IT-rule is similar to the generation of FT- or BT-rules. The elements in the source and target domain are equipped with translation markers, whereas the correspondence part will be created by this rule.* △

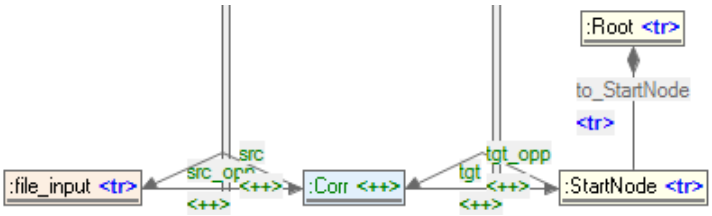


Figure 2.11: Integration Translation Rule IT_T_file_input-2-StartNode

The last type of operational rules that we want to introduce are *consistency creating rules* (CC-rules). They are defined formally in Def. 7.44 in [EEGH15]. The set of consistency creating rules (TR_{CC}) is generated out of the set of triple rules. All consistency creating rules are equipped with translation markers in all three domains, i.e., in the soucre, target and correspondence domain. Therefore, they only change translation markers from **F** to **T**, and do not create any elements in the triple graph. They are used for checking the consistency of a triple graph, i.e., given a triple graph (= integrated model), then can all markers of the triple graph be changed from **F** to **T**? If all markers of the triple graph are set to **T** after applying the set of CC-rule, i.e., after executing the model transformation based on CC-rules, then the triple graph is consistent. Otherwise, it is not consistent. For formal details we refer to Sec. 7.5.3 in [EEGH15].

Example 2.2.7 (Consistency Creating Rule)**.** *The consistency creating rule* CC_T_file_input-2-StartNode *in Fig. 2.12 is generated out of triple rule* T_file_input-2-StartNode *(cf. Fig. 2.7). It contains translation markers* $\mathsf{tr}_x = \mathbf{F} \to \mathbf{T}$ *for all elements in all domains. This rule does not create any element, instead it only changes the translation attributes in order to check whether the given triple graph is consistent.* △

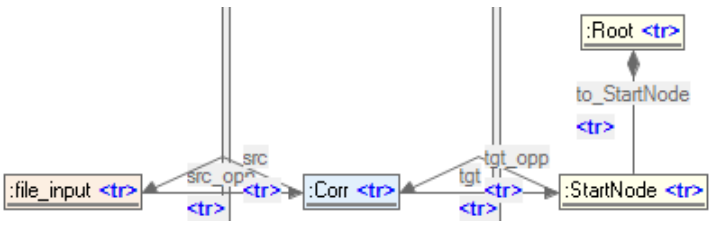


Figure 2.12: Consistency Creating Rule CC_T_file_input-2-StartNode

Remark 2.2.2 (Visual Notation for Rules: Translation Attributes)**.** *A blue* $<\mathsf{tr}>$ *marker (or violet* $[\mathsf{F}>\mathsf{T}]$ *marker, e.g., in Fig. 4.15) in operational rules indicates translation attributes, i.e., the translation attribute shall be changed from* **F** *to* **T** *when applying this rule.* △

In general, during the model transformation, it might occur, that there exist different paths for the translation. Due to non-determinism of rule applications, it cannot be ensured that the path will be chosen that results into a completely translated model. If the model transformation results in a partially translated graph, backtracking may be necessary to reverse applied rules and to be able to use other rules in order to get a fully translated graph. Paths, which lead to a model which cannot be fully translated are called *misleading*. In order to minimise or even prevent the occurence of misleading paths, *filter NACs* were developed. Filter NACs are NACs that are only valid for operational rules, e.g., a filter NAC for a forward translation rule may include content in the source domain, that will be created by the corresponding triple rule, i.e., that cannot be matched by the triple rule (cf. Ex. 2.2.8).

In Fact 3.7 [HEGO14] (or in Fact 8.18 in [EEGH15]), an algorithm is presented for the automatic generation of filter NACs. Note, the automated generation of filter NACs is not implemented in HenshinTGG. Therefore, we had to create all filter NACs in our application scenario manually (cf. Sec. 6.1). Thus, the set of filter NACs is not complete, in the sense of Fact 3.7.

In the following, we will only cite the definitions of misleading graphs and filter NACs. For more details on filter NACs, the generation of filter NACs, or formal analysis results of model transformations with filter NACs, we refer to [HEGO14, HEGO10a, EEGH15].

Definition 2.2.13 (Translatable and Misleading Graphs (cf. Def. 8.14 in [EEGH15] or Def. 3.3 in [HEGO14]))**.** *A triple graph with translation attributes G is translatable if there is a transformation sequence $G \overset{tr*_{FT}}{\Longrightarrow} H$ via forward translation rules such that H is completely translated.*

A triple graph with translation attributes G is misleading, if every triple graph G' with translation attributes that contains $G (G \subseteq G')$ we have that G' is not translatable. △

Definition 2.2.14 (Filter NACs (cf. Def. 8 in [HEGO10a] or Def. 8.16 in [EEGH15]))**.** *A filter NAC n for a forward translation rule $tr_{FT} : L_{FT} \leftarrow K_{FT} \rightarrow R_{FT}$ is given by a morphism $n : L_{FT} \rightarrow N$, such that there is a TGT step $N \overset{tr_{FT},n}{\Longrightarrow} M$ with M being misleading. The extension of tr_{FT} by some set of filter NACs is called forward translation rule tr_{FN} with filter NACs.* △

Example 2.2.8 (Filter NAC of Forward Rule)**.** *The forward rule* FT_T_NEWLINE_expr_stmt-2-empty_POST *that we describe and illustrate in Fig. 6.6 is extended by a filter NAC.* △

2.2.2. Additions: Inheritance & Edge Types

Similar additions which we introduced in Sec. 2.1.7, can be applied to triple graph grammars. In [GL06], the node type inheritance for TGGs is defined in explicit.

2.2.3. Analysis

Model transformations based on triple graph grammars provide a wide variety of analysis techniques and also techniques for verifying certain properties. In this section, we will introduce the most relevant properties for this work, because in Chap. 5, we will show that certain gerneral properties for TGGs still hold in our newly developed construction (cf. Thm. 5.6.1).

We will present the properties of *functional behaviour of model transformations* (cf. [HEGO10a, HEOG10]), *syntactical correctness and completeness* (cf. [EEHP09]) and *termination* (cf. [EEHP09]) for TGGs. Note that we will cite the definitions in [EEGH15], even if we still mention other references.

Syntactical correctness means that, if we have a model transformation sequence via forward rules that lead to an integrated model, then this integrated model belongs to $\mathcal{L}(TGG)$, i.e., the language set up by the triple graph grammar. *Completeness* means that, if we have an integrated model which belongs to $\mathcal{L}(TGG)$, then there exists a forward model transformation sequence that builds up this integrated model. The same properties hold for backward transformations.

Definition 2.2.15 (Syntactical Correctness and Completeness (cf. Def. 8.3 in [EEGH15] or Thm. 2 in [EEHP09]))**.** *A model transformation* $MT : \mathcal{L}(TG^S) \Rrightarrow \mathcal{L}(TG^T)$ *based on forward rules is*

- *syntactically correct, if for each model transformation sequence* $(G^S, G_0 \overset{tr*_F}{\Longrightarrow} G_n, G^T)$ *there is* $G \in \mathcal{L}(TGG)$ *with* $G = (G^S \leftarrow G^C \rightarrow G^T)$ *implying further that* $G^S \in \mathcal{L}(TGG)^S$ *and* $G^T \in \mathcal{L}(TGG)^T$*, and it is*

- *complete, if for each* $G^S \in \mathcal{L}(TGG)^S$ *there is* $G = (G^S \leftarrow G^C \rightarrow G^T) \in \mathcal{L}(TGG)$ *with a model transformation sequence* $(G^S, G_0 \overset{tr*_F}{\Longrightarrow} G_n, G^T)$ *and* $G_n = G^T$*. Vice versa, for each* $G^T \in \mathcal{L}(TGG)^T$ *there is* $G = (G^S \leftarrow G^C \rightarrow G^T) \in \mathcal{L}(TGG)$ *with a model transformation sequence* $(G^S, G_0 \overset{tr*_F}{\Longrightarrow} G_n, G^T)$ *and* $G_n = G$ △

According to Corollary 8.5 in [EEGH15], the syntactical correctness and completeness is also valid for model transformations using forward translation rules. Similarily, according to Def. 8.8 in [EEGH15], the syntactical correctness and completeness is also valid for model integrations.

The next important property that we want to define in the following is called *functional behaviour*. Intuitively functinal behaviour includes *termination* and *local confluence*. Termination means that each transformation sequence will terminate, i.e., the transformation system is designed so that no endless loops will occur. Local confluence describes the property that if a graph K can be transformed in one step into two graphs P_1 and P_2, then, both graphs can be transformed to K', i.e., each transformation leads to the same result graph. In the remainder of this subsection, we will cite the definitions for functional behaviour, termination and local confluence. For a more detailed view on these terms, we refer to Sec. 5.2.4 in [EEGH15].

Definition 2.2.16 (Functional Behaviour of a Transformation System (cf. Def. 8.10 in [EEGH15]))**.** *A transformation system $TS = (R)$ with transformation rules R has functional behaviour, if for each two terminated transformation sequences $G \overset{*}{\Rightarrow} H_1$ and $G \overset{*}{\Rightarrow} H_2$ via TS and starting at G the resulting graphs are isomorphic, i.e., $H_1 \cong H_2$.* △

Definition 2.2.17 (Functional Behaviour of Model Transformations (cf. Def. 8.11 in [EEGH15] or Def. 6 in [HEGO10a] or Thm. 3 in [HEOG10]))**.** *Given a source domain specific language (DSL) $\mathcal{L}_S \in \mathcal{L}(TGG)_S$, then a model transformation MT based on forward translation rules has functional behaviour if each execution of MT starting at a source model $G^S \in \mathcal{L}_S$ leads to a unique (up to isomorphism) target model $G^T \in L(TGG)_T$.* △

Definition 2.2.18 (Termination (cf. Def. 8.12 in [EEGH15] or Thm. 3 in [EEHP09]))**.** *A system of operational translation rules TR_X with $X \in F_{CC;FT;BT}$ is terminating, if each transformation sequence via TR_X is terminating, i.e., the sequence ends at a graph to which no further translation rule $(FT; BT; CC)$ is applicable.* △

According to Fact 8.13 in [EEGH15], termination is ensured for all operational rules using translation markers, if and only if all input graphs are finite on the graph part and each operational rule using translation markers changes at least one translation attribute from **F** to **T**. Note that the data part of the graph is possibly infinite (cf. Def. 2.1.4 and Rem. 2.1.2).

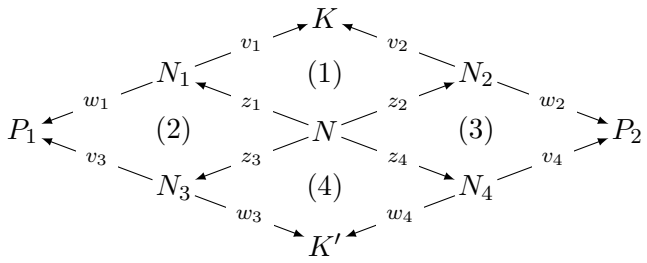
In order to define local confluence, the term of *strict AC-confluence* needs to be clarified. It uses the notion of *critical pairs*. A critical pair denotes a minimal conflict between two rules w.r.t. a graph. This means that critical pairs describe overlappings between two rules but only in the context of a concrete graph (cf. Def. 5.39 in [EEGH15]).

Definition 2.2.19 (Strict AC-Confluence (cf. Def. 5.42 in [EEGH15]))**.** *A critical pair $P_1 \xleftarrow{\bar{p}_1,o_1} K \xrightarrow{\bar{p}_2,o_2} P_2$) with induced application conditions ac_K is strictly AC-confluent if it is*

1. *confluent without application conditions, i.e., there are plain transformations $P_1 \overset{*}{\Rightarrow} K'$ and $P_2 \overset{*}{\Rightarrow} K'$,*

2. *strict, i.e., given derived spans $der(P_i \xRightarrow{\bar{p}_i,o_i} K_i) = K \overset{v_i}{\leftarrow} N_i \overset{w_i}{\rightarrow} P_i$) and $der(P_i \overset{*}{\Rightarrow} K') = P_i \overset{v_{i+2}}{\leftarrow} N_{i+2} \overset{w_{i+2}}{\rightarrow} K'$) for $i = 1, 2$ and pullback (1) then there exist morphisms $z3$, $z4$ such that diagrams (2), (3), and (4) commute, and*

3. *for $\bar{t}_i : K = \xRightarrow{\bar{p}_i,o_i} P_i \overset{*}{\Rightarrow} K'$ it holds that $ac_K \Rightarrow ac(\bar{t}_i)$ for $i = 1, 2$.* △



Theorem 2.2.1 (Local Confluence Theorem (cf. Thm 5.43 in [EEGH15]))**.** *A transformation system is locally confluent if all its critical pairs are strictly AC-confluent.* △

2.3. Transformation Units

In general, the rule-base graph transformation is not structured, i.e., out of the set of transformation rules, one rule will be selected and checked, if it is applicable to the current graph, or not. If it is applicable, then the direct transformation will be executed. Then, the next rule will be arbitrarily selected, and so on. The graph transformation will finish, if no rule is applicable anymore.

Therefore, the aim came up to introduce structures that will be layed above of the set of rules for guiding the rule application and for dividing the set of rules into separate units in order to structure the rule application or reuse different components. These structures are called *transformation units* which were introduced in [KK96]. For detailed overviews on transformation units, we recommend [Kus00] and [KKR08]. Transformation units can be nested, i.e., a transformation unit can contain one or a set of transformation units, again.

Transformation units are implemented in EMF Henshin [Hen16b] and a very limited implementation was included in HenshinTGG [Hen16a] for the industrial case study (Sec. 6.1.2).

In the following, we will informally review those transformation units, that are implemented in EMF Henshin and describe their functionality.

Rules Transformation rules (cf. Def. 2.1.18 for plain graph rules and Def. 2.2.4 for triple rules) form the smallest transformation unit. They contain no other units. Thus, rules are the only type of unit that is applicable during the model transformation.

Independent Unit An independent unit contains a set of subunits. Out of all subunits, one is selected arbitrarily and will be applied.

SequentialUnit All subunits of a sequential unit will be applied after each other with regard to the given order. If one subunit is not applicable, the application of the sequential unit is aborted and all changes performed by this unit will be rolled back.

LoopUnit All subunits of a loop unit will be applied as long as possible.

ConditionalUnit A conditional unit is similar to an if-condition. It consists of a condition (if), a then-branch and an else-branch. According to the evaluation of the condition, either the then-branch will be applied or the else-branch. If the else-branch is empty, then nothing will be applied, if the condition is not satisfied. In Henshin, the condition is usually defined by means of a transformation rule which evaluates to true if applicable.

PriorityUnit Subunits of a priority unit are listed according to their priorities, i.e., the subunit which comes first in the list has the highest priority, the last subunit has the lowest priority. The subunit with the highest priority and which is applicable, will be applied next.

Example 2.3.1 (Transformation Unit)**.** *In Fig. 6.8 a screenshot is given which illustrates the structure of the main transformation unit of the graph grammar* Refactor_SPELL-Flow *which we developed for the industrial case study (cf. Chap. 6). The following screenshots taken in Henshin illustrate the structure of the corresponding subunits:*

- *Fig. 6.9 shows sequential unit* PreProcessing*,*
- *Fig. 6.10 shows sequential unit* Refactoring*, and*
- *Fig. 6.11 shows sequential unit* CleanUp*.*

Fig. 6.21 is a screenshot of the main transformation unit and its direct subunits of graph grammar SPELL-Flow-2-Hierarchy*. The subunits are provided in the following screenshots:*

- *Fig. 6.22 shows sequential unit* PreProcessing*,*
- *Fig. 6.23 shows sequential unit* Hierarchies*,*
- *Fig. 6.24 shows sequential unit* PostProcessing*, and*
- *Fig. 6.25 shows sequential unit* CleanUp*.* △

Methodology for Model Translations 3

This chapter presents a methodology for unidirectional software translations, which will be later extended to a methodology for bidirectional software translations. Both kinds of methodologies are based on the following papers about two industrial projects with SES, in which we discussed both methodologies and in which we applied them in practice:

- Unidirectional translation (project PIL2SPELL):
  - *On an Automated Translation of Satellite Procedures Using Triple Graph Grammars* [HGN+13]
  - *Triple Graph Grammars in the Large for Translating Satellite Procedures* [HGN+14a, HGN+14b].
- Bidirectional translation (project SPELL-2-SPELL-Flow):
  - *Towards Bidirectional Engineering of Satellite Control Procedures Using Triple Graph Grammars* [GHE+13]

At the end of this chapter, we will show the application of both methodologies in practice in referring to the industrial project in which we applied those methodologies for a prototype software translation of from SPELL to SPELL-Flow (unidirectional) and vice versa (bidirectional).

3.1. Methodology for General Software Translations

First, we discuss the unidirectional software transformation methodology, i.e., given a source language $\mathcal{L}1$ and a target language $\mathcal{L}2$, then the question which we want to answer is: *According to which concept is it possible to transform model $\mathcal{L}1$ to model $\mathcal{L}2$?*.

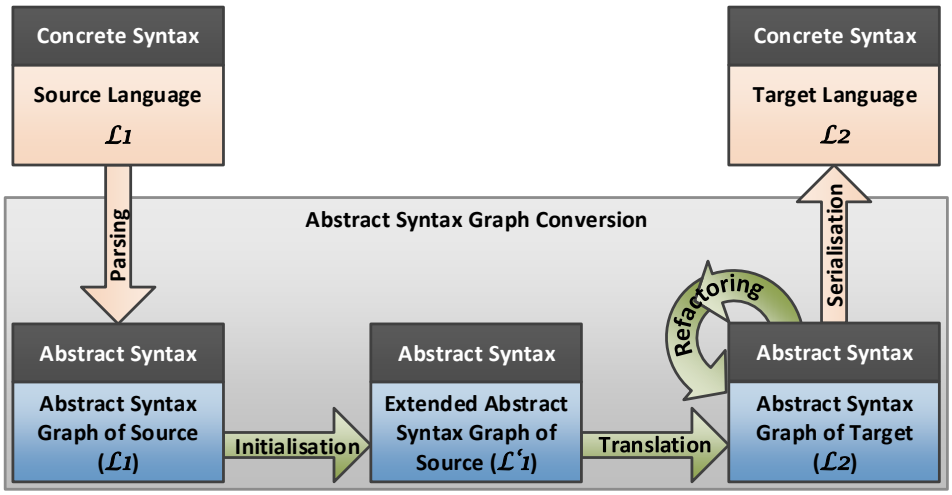


Figure 3.1: Concept for software translation (adapted from [HGN+14a])

We consider Fig. 3.1. The left box on the top row represent an instance (e.g., source code) given in the source language $\mathcal{L}1$, i.e., the *concrete syntax* of the instances is given. For the transformation from the instance in the source language $\mathcal{L}1$ to an instance in the target language $\mathcal{L}2$, we undergo the following main phases: *parsing*, *abstract syntax graph conversion* (main phase) and *serialisation*.

In detail, in the first phase *parsing* the concrete syntax in $\mathcal{L}1$ is parsed to an *abstract syntax graph* (ASG) by means of a predefined parser. The parsing phase is followed by the main phase, i.e., the abstract syntax graph conversion, which is again divided into three sub-phases: *initialisation*, *translation* and *refactoring*, wheras the two sub-phases initialisation and refactoring are optional.

During the initialisation phase, the abstract syntax graph is enriched by additional elements in order to perform a pre-processing of elements or in order to add further helper strutures which store additional information locally in the abstract syntax tree. Both help to reduce the complexity and to simplyify the specification of the rules used in the next phase - the translation phase. For the initialisation phase an in-place transformation technique is used, namely plain graph transformation (GT) (cf. Sec. 2.1.3).

The translation phase is executed using a triple graph grammar (TGG) (cf. Sec. 2.2). It iteratively applies its translation rules to a substructure the source ASG in order to translate this substructure to a substructure in the target ASG. Furthermore traceability links are created between corresponding substructures of the source and the target model. Finally, all substructures in the target ASG are connected according to the source ASG. To ensure, that each element will be translated exactely once, TGGs use trans-

lation attributes. They mark all untranslated elements with **F**. During the translation step, the marking of the elements that are translated is changed to **T**. The translation finishes, when no triple rule is applicable anymore or when all translation markers are set to **T**. Due to the fact that TGGs are non-deleting, the complete source ASG is preserved.

During the refactoring phase, the target ASG is enriched or reduced by elements in order to fullfil given guidelines required in the target language. Again, the refactoring phase uses in-place transformation, i.e., plain graph transformation, also because it may contain deleting operations on the target ASG.

The last phase is the serialisation phase in which the target ASG is converted to an instance of the target language $\mathcal{L}2$, i.e., the target ASG is serialised to concrete syntax. This target instance given in language $\mathcal{L}2$ is the final result of the translation that started at the source instance given in language $\mathcal{L}1$.

3.2. Methodology for Bidirectional Software Translations

The methodology for bidirectional software translations is an extension of the unidirectional approach. Given a source language $\mathcal{L}1$ and a target language $\mathcal{L}2$, the research question is extended to the following one: *According to which concept is it possible to transform model* $\mathcal{L}1$ *to model* $\mathcal{L}2$ *and vice versa?*.

Sec. 3.2 illustrates the scheme for bidirectional software translations. It includes the scheme for unidirectional software translations in Fig. 3.1. In detail, the *forward* translation from an instance in language $\mathcal{L}1$ to an instance in language $\mathcal{L}2$ is identical to Sec. 3.1. The *backward* direction, i.e., the translation from an instance in langauge $\mathcal{L}2$ to an instance in $\mathcal{L}1$ is equivalent, i.e., it is performed in three phases: the initialisation phase, the abstract syntax graph conversion and the serialisation phase. The abstract syntax graph conversion is again divided into the sub-phases: initialisation, translation and refactoring.

The main difference between unidirectional and bidirectional software translations is that they use different sets of graph transformation rules for the initialisation and refactoring phases because both phases are based on different meta-models. In the forward translation, the initialisation phase is performed on an ASG that is typed over a meta-model that belongs to $\mathcal{L}1$. The refactoring step is executed on an ASG that is typed over a meta-model that belongs to $\mathcal{L}2$. In contrast, during the backward translation, the initisalisation phase is executed on an ASG that is typed over a meta-model that belongs to $\mathcal{L}2$, wheras the basis for the refactoring phase is an ASG that is typed over a meta-model of $\mathcal{L}1$. In general, the translation

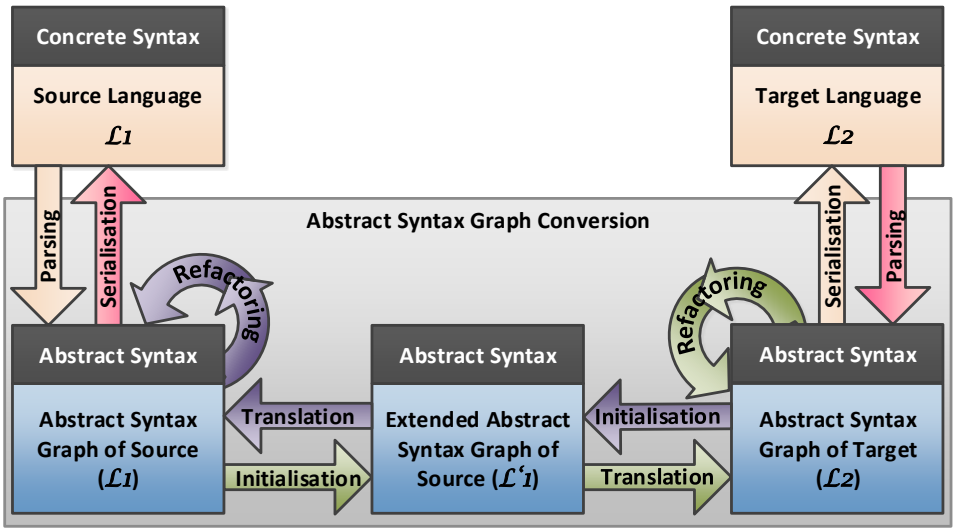


Figure 3.2: Concept for bidirectional software translation (adapted and merged from [HGN+14a, GHE+13])

phase is based on the same TGG for both, the forward and the backward direction, because TGGs are defined simultaneously for the whole triple model, i.e., for the forward and backward translation the same rules are used as basis to derive the corresponding forward and backward translation rules (cf. Sec. 2.2).

3.3. Industrial Case Study: SPELL ⇔ SPELL-Flow

We will now show the application of the unidirectional software translation methodology to the industrial case study which was executed in cooperation with SES [GHE+13]. The unidirectional approach is used for the software translation from SPELL to SPELL-Flow. This forward direction deals with the *visulisation of source code*. The bidirectional approach is applied to the translation from SPELL to SPELL-Flow and back. The backward translation can also be seen as *code generation*.

3.3.1. Unidirectional: SPELL to SPELL-Flow

During the case study with our industrial partner SES, we developed a prototype for the software translation of a sub-set of SPELL statements into their visualisation, i.e., to a SPELL-Flow model. It is desired to extend this to a software translation of all SPELL statements in order to be able to translate an arbitrary SPELL procedure into its SPELL-Flow models. This can then be integrated into the SPELL GUI, which can be seen as the SPELL

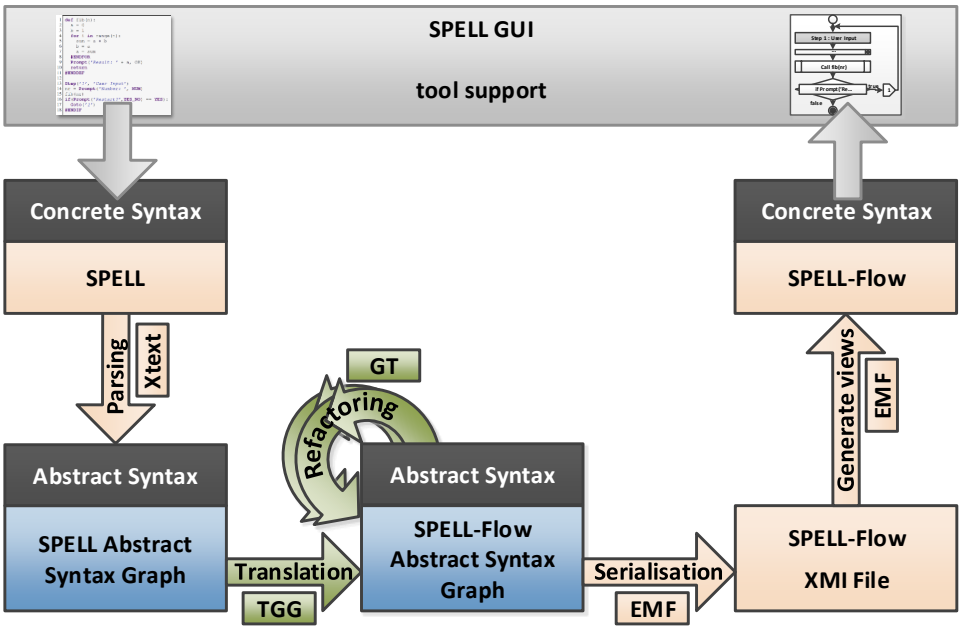


Figure 3.3: Software translation in SPELL to SPELL-Flow (adapted from [GHE+13])

execution environment. The SPELL GUI is an application at SES used by satellite operators. It contains different views on a SPELL procedure, e.g., the source code view or a view containing all execution statuses. The visualisation of the SPELL source code shall be integrated as additional view to the SPELL GUI in order to improve the work of the satellite operators. Note that the SPELL GUI is not a development environment. This means that the satellite operators do not change existing SPELL procedures. As a consequence, the visualisation will not change at runtime. The software translation from SPELL to SPELL-Flow will be executed offline.

We illustrate the application of the methodology for unidirectional software translations from Sec. 3.1 in Fig. 3.3. The box on the top of this image is labeled with *SPELL GUI - tool support*, i.e., it represents the SPELL GUI application. The SPELL source code view and the SPELL-Flow visualisation are illustrated explicitly, too. They are indicated by small icons: The SPELL source code view is on the left and the corresponding SPELL-Flow visualisation is on the right.

Parsing

For the software translation from SPELL source code to a SPELL-Flow model, we start with a SPELL source code instance that is in addition illustrated using the box *Concrete Syntax - SPELL*. As a first step, this SPELL

source code instance is parsed (see arrow *Parsing*. In our case study, we used Xtext [xte16] for deriving a parser and a serialiser for SPELL (cf. Sec. 6.1.3 for a more detailed introduction to Xtext). In order to be able to generate a parser and a serialiser, Xtext needs a grammar definition. For that, we reused the SPELL grammar which was defined for the *PIL2SPELL* project [HGN+13] and adapted it slightly to the needs of the current case study. The adaptions mainly concern redefinitions of newlines and multi-line comments. In Appendix A.1 we will show the definition of the SPELL grammar. The syntax is very related to the EBNF notation (extended backus naur form). The parsing step results in an abstract syntax graph (ASG) of the given SPELL source code. It is stored in a separate XMI file. Note, due to the fact that the SPELL source code file is a text file and the parsing is performed line-by-line, the abstract syntax graph is an abstract syntax tree (AST) in reality, i.e., it contains no cycles.

Translation

In our general methodology, the next step is possibly the initialisation phase. This step is omitted in our case study. Consequently, the next step is the translation step. For that, we applied triple graph transformation which was possible due to the use of *HenshinTGG* [Hen16a], an Eclipse-based tool for applying triple graph transformation (cf. Sec. 6.1.2 for more details on HenshinTGG). In practice, the SPELL ASG file is imported to HenshinTGG and the set of forward translation rules are applied to the SPELL ASG in order to derive a SPELL-Flow ASG. Again, in reality, the SPELL-Flow ASG is an AST, which is ensured by our specification of the triple rules, i.e, we omitted to create cycles. Due to the missing initialisation step and due to different requirements set up by SES (e.g., mark elements that cannot be translated), the triple graph transformation step is divided into several substeps which is realised by transformation units. For details on that phase we refer to Sec. 6.1.2.

Refactoring

The next step in Fig. 3.3 is the refactoring phase. The refactoring phase uses plain graph transformation. The graph transformation rules are defined in and will be executed using the Henshin-Editor for plain graph transformation [Hen16b] (cf. Sec. 6.1.2 for an introduction into the Henshin-Editor we used). For the refactoring phase, we defined two different plain graph grammars. Note, in Fig. 3.3 we illustrated the use of more than one refactoring graph grammar by means of a double arrow. We use the technique of plain graph transformation, because both refactoring grammars include deletion operations on the abstract syntax graph (ASG).

- The first graph grammar performs general refactoring, e.g., it merges

shapes which belong together (e.g., multi-line comments, list of attributes). It removes nodes that remain empty, i.e., statements that have no benefit for the SPELL-Flow instance model, e.g., statement without arguments. Furthermore, this grammar reduces the number of FIXME statements in merging those together that belong to the same SPELL statement.

- The second graph grammar introduces the hierarchical model. The hierarchical model will be created in copying nodes that shall occur on the next level of abstraction and creating containment edges between corresponding nodes. In detail, containment edges will be created between the parent node on the more abstract layer to all children nodes on the more detailed layer.

Serialisation and generation of views

The resulting ASG of the refactoring phase is serialised in order to store it into an XMI file. The serialiser which is used in that step is provided by Xtext. The serialisation step is illustrated in Fig. 3.3. The final XMI file will be opened within the *SPELL-Flow visualisation tool*, which we implemented in the PhD project (cf. Sec. 6.1.3). This tool is based on Eclipse GMF, which includes Eclipse EMF. In opening the final XMI file, the visualisation will be generated by EMF. This visualisation is called *SPELL-Flow conrete syntax* and is visualised in the box with the same label in Fig. 3.3. As already mentioned at the beginning of this section, it is desired to include this visualisation in the SPELL GUI application of SES (box on top of Fig. 3.3).

3.3.2. Bidirectional: SPELL to SPELL-Flow and vice versa

The bidirectional implementation of a prototype software translation between SPELL source code instances and SPELL-Flow visualisations shall be integrated into the SPELL development environment. It is the environment used by SES engineers for developing SPELL procedures. In contrast to the application area of the unidirectional approach, which is read-only, the bidirectional software translation takes place "online", i.e., when a developer changes the SPELL source code, then the changes shall be propagated in real-time to the corresponding SPELL-Flow visualisation, and vice versa. Note, the SPELL-Flow visualisation is more abstract and less detailed than the SPELL source code. Thus, the software translation from a SPELL-Flow model to SPELL source code derives a skeletton of the source code which needs manual completion performed by the SPELL developer.

The bidirectional implementation of a prototype includes the unidirectional approach from Sec. 3.3.1 and is illustrated in Fig. 3.4. Thus, the

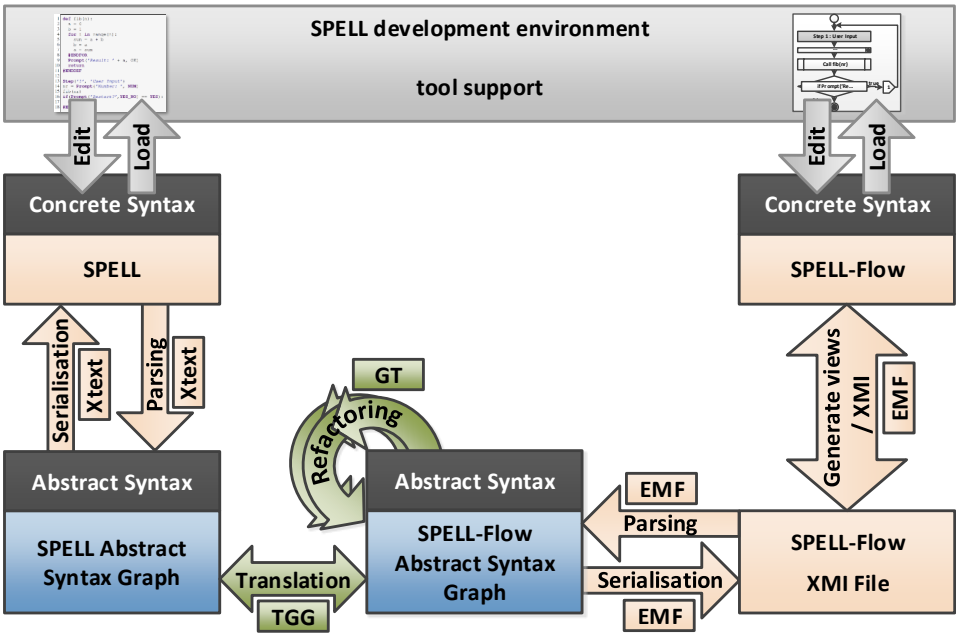


Figure 3.4: Bidirectional software translation between SPELL and SPELL-Flow (adapted from [GHE+13])

forward direction is nearly identical to the unidirectional approach for software translation of SPELL source code to a SPELL-Flow visualisation. Due to the changed application environment, i.e., the usage within the SPELL development environment, we allow edit operations. This is illustrated on top of Fig. 3.4 by the changed box, which is now labeled with *SPELL development environment - tool support* and also by the new incoming and outgoing arrows *edit* and *load* of that box.

Nevertheless, the backward direction is similar to the forward direction, too. It is divided into the following steps:

- An *edit* operation changes the SPELL-Flow model resulting an a modified concrete syntax of a SPELL-Flow visualisation. This change shall be propagated to the corresponding SPELL source code instance.
- As next step, a new XMI file will be *generated* using EMF resulting in a new SPELL-Flow XMI file.
- The XMI file will be *parsed* using EMF resulting in an abstract syntax graph (ASG) of the SPELL-Flow visualisation.
- To the SPELL-Flow ASG, *refactorings* will be applied which remove the hierarchies so that we finally obtain an ASG representing the most detailled level of the SPELL-Flow model. For the refactoring step(s), we apply plain graph transformation using Henshin.

- The main step is the *translation* phase which is applied via TGGs using HenshinTGG. It transforms the SPELL-Flow ASG into a SPELL ASG.
- Finally, the new SPELL ASG is *serialised* via Xtext so that the new SPELL source code can be *loaded* to and executed in the SPELL development environment, i.e., the view containing the corresponding SPELL source code instance will be updated.

4 Model Synchronisation

In this chapter, we will review the model synchronisation framework based on triple graph grammars (TGGs) [HEO+15, HEO+11b]. After the formal introduction of this model synchronisation framework, we will introduce an extension of this framework regarding concurrency. Afterwards, we present a generalisation of the concurrent approach to the non-deterministic case. In general, the triple rules in a TGG can be non-deterministic. This means that different triple rules may cover similar contexts. Consequently, the derived operational rules may be applicable to the same graph which lead to different results. One result may be that the graph cannot be fully translated so that backtracking is necessary. We introduce an optimisation of this non-deterministic concurrent model synchronisation framework in which we use filter NACs Def. 2.2.14 for reducing the necessary backtracking steps. Summarising, we discuss the following research question:

In which way is it possible to treat a non-deterministic set of rules in concurrent model synchronisations? (Q3)

During the whole chapter, we use the running example in Listing 4.1 for illustrating the formal model synchronisation framework based on TGGs as well as for illustrating the following section in which we discuss the different kinds of conflicts and the strategies for solving them semi-automatically.

The first and second part of this chapter, i.e., the model synchronisation framework based on TGGs and the concurrent model synchronisation framework based on TGGs, is mainly based on [HEO+15, HEO+11b]. The third part, i.e., the extension of the concurrent model synchronisation framework to the non-deterministic case, is mainly based on [GHN+13a, GHN+13b]. In addition, we refer to the other related articles and books discussing these research areas, like [HEEO12, EEGH15].

4.1. Introduction of the Running Example

In this chapter, we use a small excerpt of the running example for illustrating the functionality of the model synchronisation framework and its extensions presented in this chapter. The excerpt is taken from the running example which we introduced in Ex. 1.3.1 and Ex. 1.3.2. We introduce the SPELL source code and its corresponding SPELL abstract syntax graph (ASG) in this subsection, first. Then, we show the corresponding hierarchical SPELL-Flow visualisation and its ASG. In order to propagate (domain) model updates along different views, we present our example domain model update in the target domain, i.e., in the SPELL-Flow visualisation. Finally, we illustrate relevant triple rules which we need for propagating the given (domain) model update.

Example 4.1.1 (SPELL source code and corresponding ASG)**.** *The following SPELL source code snippet illustrates the running example of this chapter.*

Listing 4.1: Running example: SPELL code (source instance)

```
1 ARGS['$ARG1']=Var(Type=ABSTIME, Confirm=False)
2 ARGS['$ARG2']=Var(Type=ABSTIME, Confirm=False)
3 ##################################################################
4 Step('5', 'MANAGEMENT OF COMPONENT A')
5     Prompt('WARNING: Check something before.', OK)
6     Pause()
```

In the next screenshot (Fig. 4.1), we visualise the abstract syntax graph (ASG) that corresponds to the listing above. The screenshot is taken from the tool HenshinTGG.

The node : file_input *is the topmost node in each SPELL ASG. It is the main container of all SPELL ASG nodes. In the illustration, we highlighted the the main container part in red. The node* file_input *is followed by a* : stmt_LST_Elem *node via containment edge* fst*. This* : stmt_LST_Elem *node is the starting node of an* : expr_stmt *structure, which represents SPELL source code line 1. We highlighted this subtree in blue. The first* : stmt_LST_Elem *is followed by another* : stmt_LST_Elem *via edge* : next *which is again the starting node of an* : expr_stmt *structure. This structure represents SPELL source code line 2. The comment in line 3 is connected to the second* : expr_stmt *node. We highlighted the corresponding subtree in green, which reflects the assignment of the* ARGS*-structure. Note, the assignment* ARGS[...] = Var(...) *shall be omitted during the translation from SPELL to SPELL-Flow (cf. Sec. 6.1.1).*

The second : stmt_LST_Elem *node is connected with a* Step *statement via edge* : next*. The whole set of elements that reflect the* Step *statement in source code line 4 from Listing 4.1 is highlighted with yellow. It*

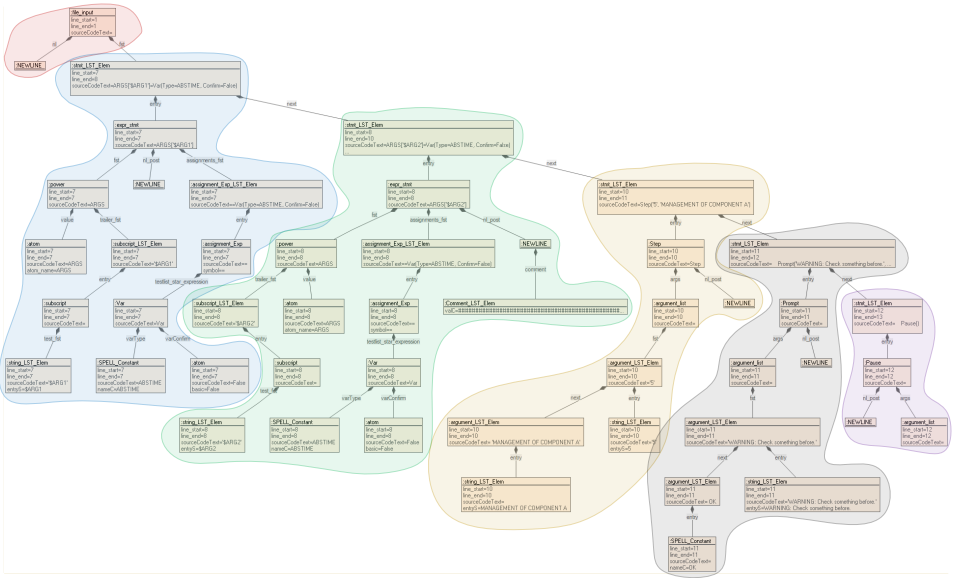


Figure 4.1: Running example: Snippet of SPELL ASG (source ASG)

is the biggest part containing eight nodes. The first node which is contained by the stmt_LST_ELem *node, is a node called* : Step. *It has three attributes containing the corresponing lines numbers in the source code and the source code text of that node. Each node in the SPELL ASG has those three attributes with corresponding assignments. Node* : NEWLINE *indicates a linebreak in the corresponding SPELL source code. The arguments of the* Step *statement are stored in a list of nodes which are contained by the* : Step *node (cf. node* : argument_list. *The first argument is the Step number which is of type* string. *This is indicated by the node* : argument_LST_Elem *followed by the leaf node* : string_LST_Elem *with attribute* entryS = 5. *The second argument is the description of the* Step *statement which again is of type* string. *This is represented by the second node* : argument_LST_Elem *and its leaf node* : string_LST_Elem *with attribute* entryS = MANAGEMENT OF COMPONENT A.

The fifth SPELL source code line of Listing 4.1 is reflected by the elements highlighted in grey in the SPELL ASG (Fig. 5.2). Again, the topmost node of this block is the node : stmt_LST_Elem *which is contained by the* : stmt_LST_Elem *of the previous statement (i.e.,* Step*) via containment edge* next. *The first node contained by* : stmt_LST_Elem *is node* : Prompt *which has two arguments. This is also visible in SPELL source code line 5 in Listing 4.1. In the SPELL ASG, both arguments are represented as argument list (*: argument_list*) with two entries: The first entry is a* : string_LST_Elem *node which is contained by* : argument_list *via containment edge* fst *to* : argument_LST_Elem *and via* entry *edge to the* : string_LST_Elem *leaf node.*

The leaf node contains the string WARNING : Check something before. *assigned to attribute* entryS. *The second argument is a* : SPELL_Constant *with attribute* nameC = OK. *This node is connected to the* : argument_list *node via the previous* : argument_LST_Elem *node to its own* : argument_LST_Elem *via containment edge* next *and finally to the leaf node via edge* entry.

The SPELL source code line 6 contains the Pause *statement. It is represented in the SPELL ASG by the four nodes highlighted in violet. The corresponding* : stmt_LST_Elem *which is connected to the previous statement via containment edge* : next *has one entry, the node* : Pause. *Due to the* Pause *statement having no arguments, the node* : argument_list *has no successors.* △

Example 4.1.2 (SPELL-Flow visualisation and corresponding ASG)**.** *The screenshot in Fig. 4.2, which is taken in the SPELL-Flow visualisation tool (cf. Sec. 6.1.3), shows the hierarchical SPELL-Flow visualisation of the SPELL source code in Listing 4.1. On the left, the first layer of the SPELL-Flow visualisation is given, i.e., the main layer, which is the most abstract one. On the right side, the second layer is visualised which is contained by the* : Step *node on the first layer.*

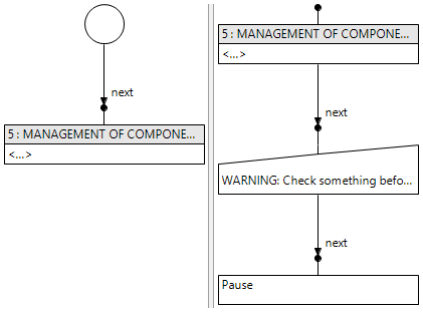


Figure 4.2: Running example: Snippet of SPELL-Flow model (target instance)

The following graph illustrates the SPELL-Flow ASG which corresponds to the visualisation in Fig. 4.2. This screenshot is taken in Henshin.

The SPELL-Flow ASG has a different structure than the SPELL ASG: It has a node : Root *which is the main container which contains all other nodes, either directly, or by children nodes. In contrast to the SPELL ASG, successive nodes are connected via normal edges and not by containment edges.*

The structure of the SPELL-Flow ASG as depicted in Fig. 4.3 is very similar to the final visualisation in Fig. 4.2. The two nodes connected directly via containment edge with node : Root *(the first is highlighted in red, the second one is one of both nodes highlighted in yellow) are the ASG nodes that correspond to both elements on the main layer of the SPELL-Flow vi-*

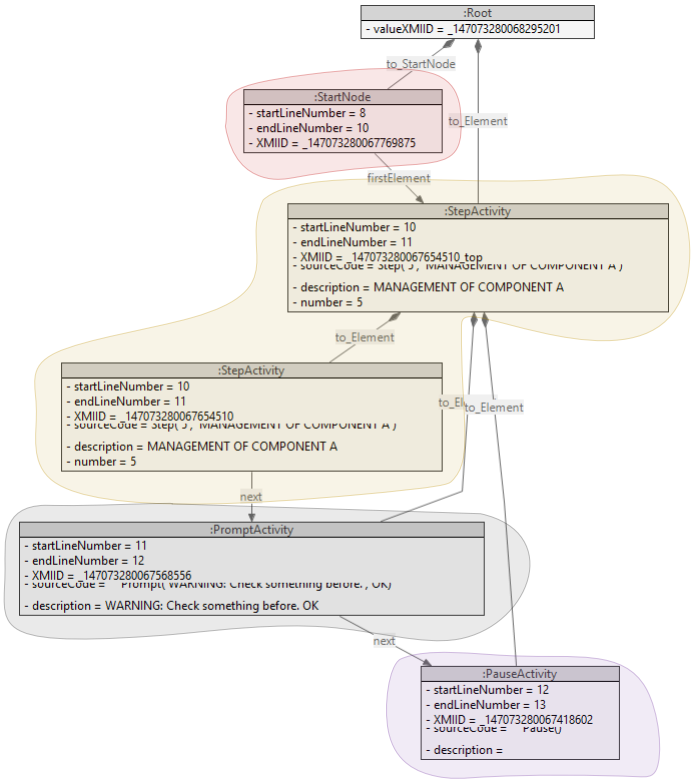


Figure 4.3: Running example: Snippet of SPELL-Flow ASG (target ASG)

sualisation. The first node corresponds to the start node (i.e., the circle) in the visualisation, wheras the second node is the topmost node of the Step *statement in the ASG.*

All other nodes constitute the second hierarchy level, which consists of a : StepActivity *node (the second node highlighted in yellow), a* : PromptActivity *node (highlighted in grey) and a* : PauseActivity *node (highlighted in violet). The list of attributes of each node determine the details that will be shown in the SPELL-Flow visualisation. All nodes have the same types of attributes:*

- startLineNumber *and* endLineNumber *contain the line numbers of the original SPELL statement.*
- XMIID *is an internal ID of each element. This is necessary for the visualisation using GMF (cf. Sec. 6.1.3).*
- *Attribute* sourceCode *gets the original SPELL source code line.*
- *The* description *attribute contains the list of parameters. It is of type string and all original parameters are concatenated into one string. This is an example of an element in which detailed information got*

lost during the software translation process. It is not possible anymore to restore the correct number of attributes and the correct split of the string into the correct attributes.

Note, the first three SPELL source code lines (cf. Listing 4.1) are not reflected in the SPELL-Flow model. △

Example 4.1.3 (SPELL ⇔ SPELL-Flow Triple Graph)**.** *The illustration in Fig. 4.4 shows the triple graph which contains a reduced version of the SPELL ASG (left, source model) from Fig. 4.1 as well as a reduced version of the SPELL-Flow ASG (right, target model) from Fig. 4.3 and the correspondence model which connects the SPELL ASG with the SPELL-Flow ASG. The triple graph is typed over a reduced SPELL ⇔ SPELL-Flow meta-model. The full meta-model consiste of the meta-models in Appendix A.2 to A.4. The target model is only reduced by some attributes w.r.t. the SPELL-Flow ASG which we described above. In contrast, the source model is reduced by nodes, edges and attributes that do not have further semantic meaning in order to increase readability. We provide a detailed list of omitted elements in the following. In the correspondence model, it is obvious, that the* ARGS *subtree has no connection to the SPELL-Flow model, i.e., those elements are not reflected in the target model.*

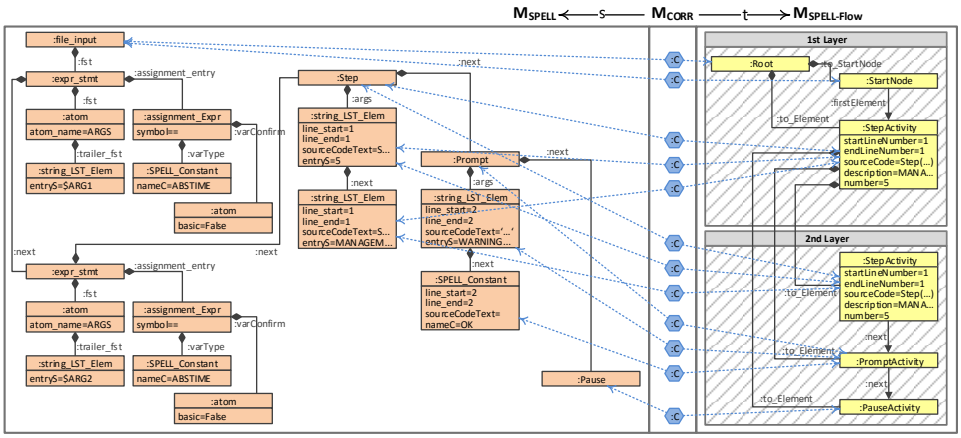


Figure 4.4: Running example: Triple Graph

1. *In the SPELL domain (source domain) we omitted:*

   - *All attributes, except the ones of the three* : string_LST_Elem *nodes and of the* : SPELL_Constant *node. The* XMIID *attribute is missing in all nodes.*
   - *The SPELL model contains many nodes which are in-between "relevant" nodes, e.g.,* : stmt_LST_Elem, : argument_list *or*

: argument_LST_Elem. *We removed those nodes and corresponding containment edges whereas the structure of the ASG has been preserved.*

- : NEWLINE *nodes and their* : nl_post *edge will not be translated, i.e., we left them out in Fig. 5.5.*

2. *In the SPELL-Flow domain (target domain) we omitted:*

   - *Similar to the SPELL domain, we removed all attributes, except the ones in both* : StepActivity *nodes, which are relevant for the running example in this chapter. The* XMIID *does not appear in any node.*

3. *In the CORR domain (correspondence domain) we omitted:*

   - *Helper nodes.*
   - *Due to the reduced SPELL ASG, many correspondence nodes and their links to the source and target domain are missing. In addition, the correspondence nodes are of type* : C *in our illustration. In reality, we have to use different types of correspondence nodes, due to some implementation details (cf. Sec. 6.1.3, but they do not differ in their semantics.* △

The domain model update which is used for illustrating the basic model synchronisation framework based on triple graph grammars (Sec. 4.2) is the source model update presented below. In contrast, the model update for illustrating the concurrent model synchronisation framework, which is an extension of the basic version (Sec. 4.3), consists of two parts which are executed simultaneously. It consists of a source model update which changes the SPELL domain and a target model update that modifies the SPELL-Flow domain of the model.

Example 4.1.4 (Source Model Update)**.** *The source model update is visualised in the following figure. In the SPELL model, the statement* Pause() *will be deleted, i.e., source code line 6 will be deleted in Listing 4.1. In the SPELL ASG (cf. Fig. 4.1), the* Pause() *statement is highlighted in violet. The following figure shows the reduced SPELL ASG from that was described in Fig. 4.4. The components to be deleted are marked with a red border and with markers −−.* △

Example 4.1.5 (Target Model Update)**.** *The target model update which is part of the whole model update adds a comment as attribute to the node* : Prompt *in the SPELL-Flow domain. This node belongs to the second layer of the visual SPELL-Flow model. Therefore, Fig. 4.6 only shows the second layer of the SPELL-Flow model. The attribute to be added is marked with a green border and with marker ++.* △

The propagation of the source and target model updates is executed using the underlying triple graph grammar. In the following, we will introduce a subset of triple rules that are relevance for this chapter and for the next chapter.

Example 4.1.6 (Relevant Triple Rules)**.** *The first triple rule, which we want to mention explicitly, is* T_Step-2-StepActivity*. It simultaneously creates a* : Step *node in the SPELL domain (source domain) and two* : StepActivity *nodes in the SPELL-Flow domain (target domain). Both* : StepActivity *nodes are connected via the containment edge* : to_Element*. Another triple rule, which we do not mention here in detail, is creating the containment edge of the corresponing root node to the parent* : StepActivity *node in the SPELL-Flow domain. The corresponding root node is either node* : Root*, if the parent* : StepActivity *node is situated on the first layer, or it is another activity (in most cases another* : StepActivity*). Furthermore, triple rule* T_Step-2-StepActivity *creates two correspondence nodes: one from the* : Step *to the parent* : StepActivity *and one from the* : Step *to the child* : StepActivity*.*

In gerneral, each : Step *node is followed by two* : string_LST_Elem *nodes in the SPELL domain which indicate the* Step *number and description. They are of type* string*. The second triple rule* T_Step_args-2-attrs *creates those nodes with attributes indicating the step number description, and the corresponding containment edges. At the same time, the description and number will be added in the target domain to both corresponding*

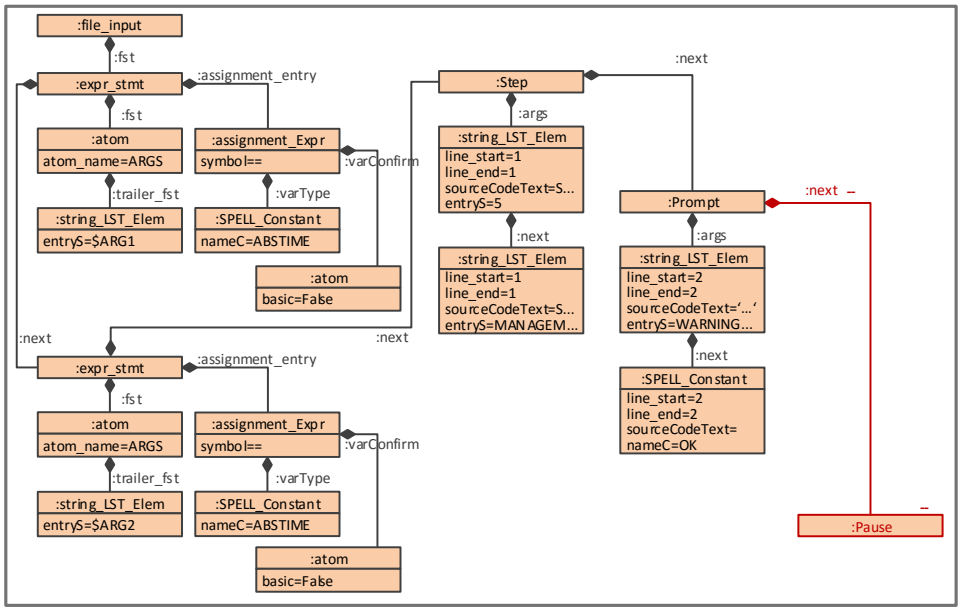


Figure 4.5: Running example: Source Model Update

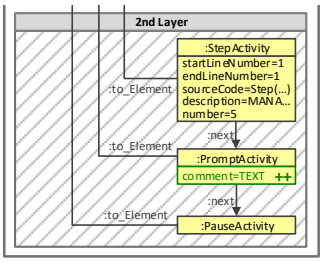


Figure 4.6: Running Example: Target Model Update

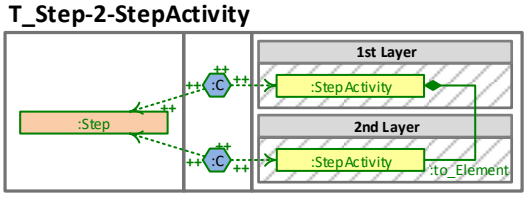


Figure 4.7: Triple rule: T_Step-2-StepActivity

: StepActivity *nodes. This triple rule also adds correspondence nodes between both* : string_LST_Elem *nodes and the corresponding* : StepActivity *nodes.*

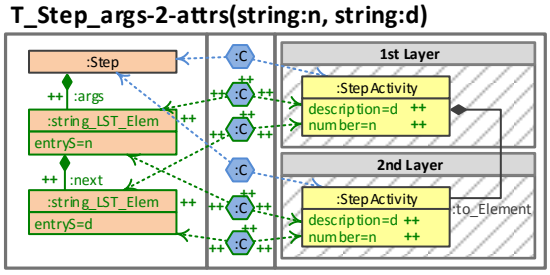


Figure 4.8: Triple rule: T_Step_args-2-attrs

The third triple rule, which we want to introduce, is T_Pause-2-PauseActivity. *It creates a* : Pause *statement connected via containment edge to another statement (*: stmt*) in the SPELL domain. Simultaneously, a* : PauseActivity *is created in the SPELL-Flow domain, whereas the* : PauseActivity *is created in a sub-layer, which is the same layer as the one of the* : Activity *which corresponds to the* : stmt. *The superjacent layer of* : Activity *contains a* : Root *node (or another node derived from* : Root, *e.g. another* : Activity*). The new* : PauseActivity *shall be contained by this* : Root *node via the newly created containment edge* : to_Element. *In addition, triple rule* T_Pause-2-PauseActivity *creates the necessary correspondence node* : C *and edges.*

The fourth triple rule, which we use explicitly within this chapter is

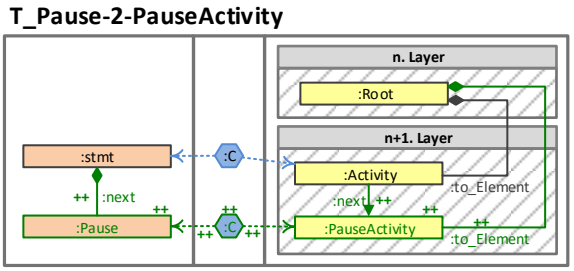


Figure 4.9: Triple rule: T_Pause-2-PauseActivity

T_Comment_LST_Elem-2-comments. *It creates two* comments *attributes in the SPELL-Flow domain: One within an* : StepActivity *on the first layer, one within a corresponding* : StepActivity *on the second layer, i.e., this triple rule requires two* : StepActivity *nodes in the SPELL-Flow domain which are connected with each other via a* : to_Element *edge. Simultaneously, this triple rule creates the necessary* : Comment_LST_Elem *node in the SPELL domain and connects it to the* : stmt *node which corresponds to the two* : StepActivity *nodes in the SPELL-Flow domain. In addition, the necessary correspondence part is created, too.*

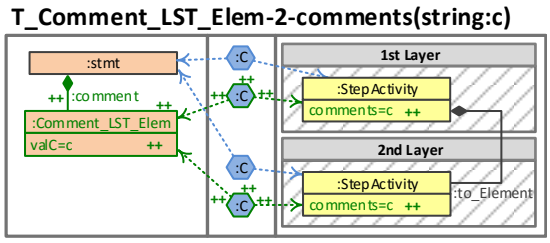


Figure 4.10: Triple rule: T_Comment_LST_Elem-2-comments

The next triple rule called T_Comment_LST_Elem-2-comment_2L *is very similar to the previous one. The difference is that it creates only one* comments *attribute in the SPELL-Flow domain. It will be created in an* : Activity *node which is situated on the second layer. Note, for this rule and the previous one, the correct sequence must be maintained, i.e., triple rule* T_Comment_LST_Elem-2-comment *must be applied before triple rule* T_Comment_LST_Elem-2-comment_2L. *Otherwise, it is possible that* T_Comment_LST_Elem-2-comment_2L *will be applied to a* : StepActivity *node on the second layer and then, the corresponding* : StepActivity *on layer 1 will not be equipped with the* comments *attribute. Another solution would be the introduction of a NAC, but it would slow down the model transformation.*

Triple rule T_sourceCodeText-2-sourceCode *creates one* sourceCodeText *attribute in the SPELL domain as part of an existing* : argument *node. This node is an abstraction of, e.g., node* : string_LST_Elem. *Furthermore, this*

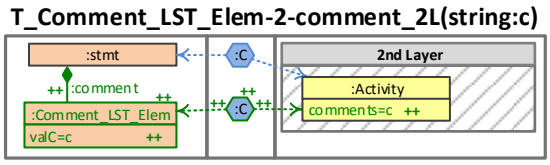


Figure 4.11: Triple rule: T_Comment_LST_Elem-2-comment_2L

triple rule sets two sourceCode *attributes in the SPELL-Flow domain for already existing two* : Activity *nodes that represent the same statement but in different layers. This dependency is given by the correspondence nodes and the containment edge* : to_Element.

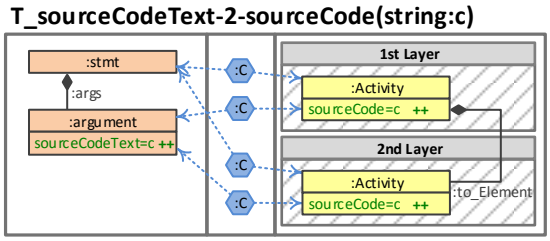


Figure 4.12: Triple rule: T_sourceCodeText-2-sourceCode

The next triple rule, which we want to address explicitly is T_lines-2-lineNumbers. *It is similar to the previous triple rule but creates a* line_start *and a* line_end *attribute for node* : argument *in the SPELL domain as well as the corresponding* startLineNumber *and* endLineNumber *attributes for both existing corresponding* : Activity *nodes in the SPELL-Flow domain.*

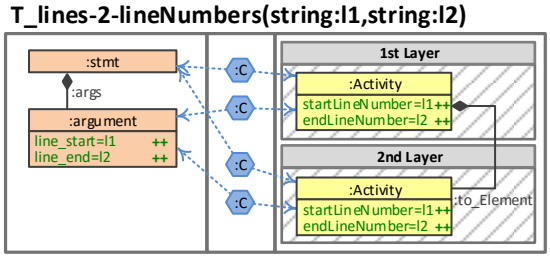


Figure 4.13: Triple rule: T_lines-2-lineNumbers

The last triple rule to be presented is named T_ARGS-2-nothing. *It contains all elements that describe an* ARGS *structure in the SPELL domain, as it is used in the example SPELL source code lines 1 or 2. The correspondence as well as the SPELL-Flow domains are empty. This means that this kind of* ARGS *structure has no relation in the target domain, i.e., it is not reflected in the SPELL-Flow domain. Note, this rule only creates the*

ARGS *structure in the SPELL domain. It does not create any connection to other elements in the SPELL domain, e.g., via* : next *or* : fst *edges. For that, the set of triple rules contain other rules which we will not visualise and describe in explicit, here.* △

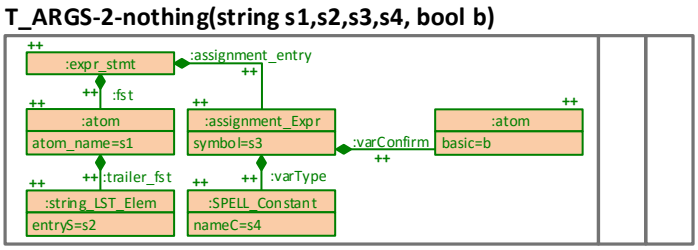


Figure 4.14: Triple rule: T_ARGS-2-nothing

Example 4.1.7 (Triple Graph Grammar TGG and Language $\mathcal{L}(TGG)$)**.** *Given the* $TGG = (MM, \varnothing, TR)$ *over triple type graph* MM *in Ex. 2.2.1 and with triple rules* TR*. A subset of the whole set of triple rules is presented in Ex. 4.1.6. Then the triple graph in Ex. 4.1.3 is contained in* $\mathcal{L}(TGG)$*.* △

Example 4.1.8 (Forward Translation Rules)**.** *The corresponding forward translation rules of the triple rules in Ex. 4.1.6 are shown explicitly in Fig. 4.15. Within the FT-rules, the correspondence and SPELL-Flow domains are identical to the same domains in the triple rules. In the SPELL domain, the translation markers are modified in the sense that elements with marker* $< ++ >$ *in the triple rule received translation attribute* $\mathsf{tr} = \mathbf{F} > \mathbf{T}$ *in the corresponding FT rule (cf. Sec. 2.2).* △

4.2. Formal Background on Basic Model Synchronisation Framework Based on Triple Graph Grammars

The model synchronisation framework based on triple graph grammars was introduced by Hermann et al. in [HEO+11b, HEEO12, HEO+15, EEGH15]. It was inspired by the replica framework presented in [Dis11].

The main idea of the model synchronisation framewok which we will introduce in this section is as follows: Given a consistent triple graph that includes a source and a target model and correspondences between them. The triple graph is also called *integrated model*. We will use the following notation: $G^S \overset{r}{\leftrightarrow} G^T$, whereas G^S is the source model, G^T is the target model and r defines the relation between source and target model, i.e., r is given explicitly by the correspondence model. Furthermore, given a consistent domain model update u, whereas the model update might be a source model update or a target model update.

Then, the synchronisation problem deals with the question, how to propagate the consistent domain model update to the other domain? This means, how is it possible to obtain a consistent integrated model $G'^S \overset{r'}{\leftrightarrow} G'^T$? Based on the domain model update u, a new consistent domain model update u' will be derived that will reflect the changes in the opposite domain. In general, the forward model synchronsiation process is called fPpg which stands for forward propagation operation. The dual construction is the backward propagation operation (bPpg). Both operations are *total* and *deterministic*. total means, that both operations derive results for all inputs. Determism means that each operation will allways return unique results for all inputs. Note, we generalised an extended version of this framework to non-deterministic propagatin operations [GHN+13a, GHN+13b] which we will review in Sec. 4.4.

A formal definition of the synchronisation problem and framework is

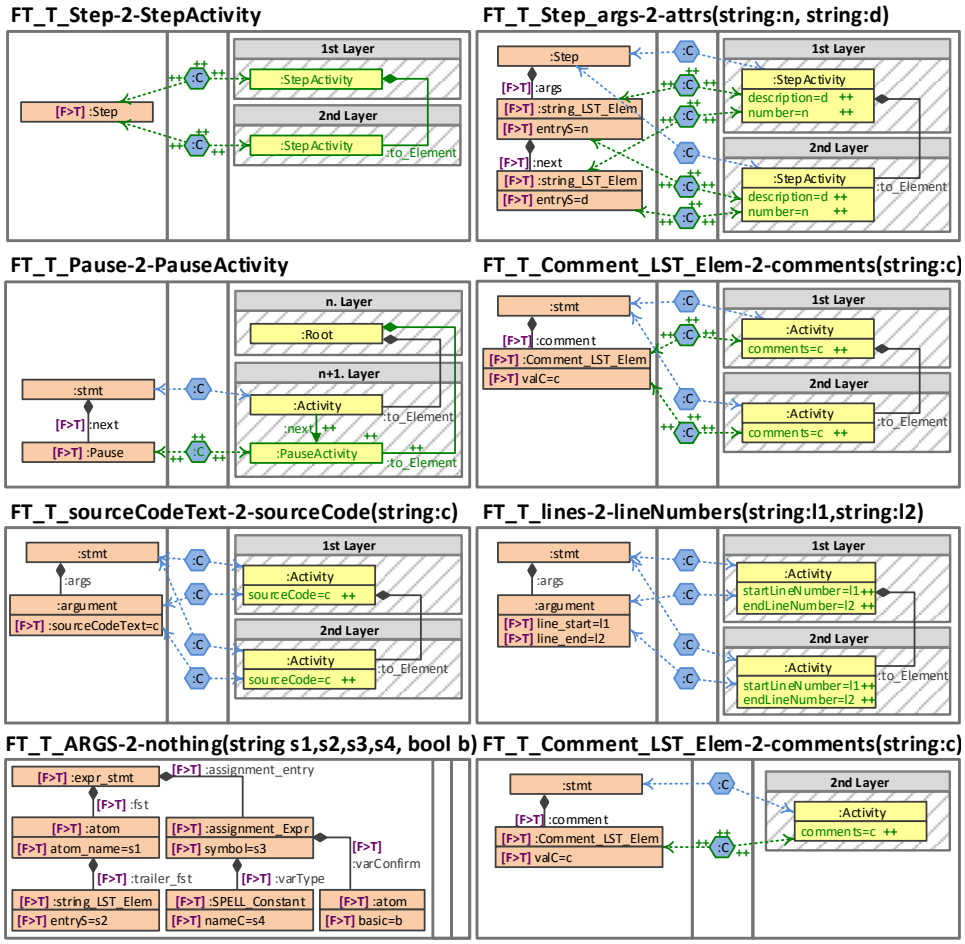


Figure 4.15: Forward Translation Rules of Triple Rules in Ex. 4.1.6

given in Def. 4.2.1. Note, it uses symbol C, which is the correspondence relation of all consistently integrated models, i.e., $C = \mathcal{L}(TGG)$.

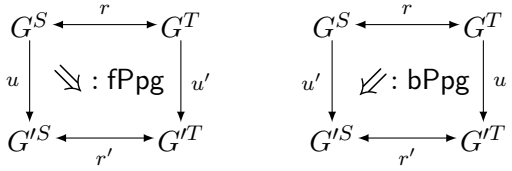


Figure 4.16: fPpg and bPpg

Definition 4.2.1 (Synchronisation Problem and Framework (cf. Def. 3.3 in [HEO+15]) or Def. 9.7 in [EEGH15])**.** *Let* $MF = (\mathcal{L}(TG^S), \mathcal{L}(TG^T), R, C, \Delta_S, \Delta_T)$ *be a TGG model framework. The forward synchronisation problem is to construct an operation* fPpg $: R \otimes \Delta_S \to R \times \Delta_T$ *leading to the left diagram in Fig. 4.16, where* $R \otimes \Delta_S = \{(r, a) \in R \times \Delta_S \mid r : G^S \leftrightarrow G^T, u : G^S \to G'^S\}$*, i.e., u and r coincide on* G^S*. The pair* $(r, u) \in R \otimes \Delta_S$ *is called premise and* $(r, u') \in R \times \Delta_T$ *is called solution of the forward synchronisation problem, written* $\mathsf{fPpg}(r, a) = (r', u')$*. The backward synchronisation problem is to construct an operation* bPpg *leading to the right diagram of Fig. 4.16. Operation* fPpg *is called correct with respect to C, if axioms (a1) and (a2) in Fig. 4.17 are satisfied and, symmetrically,* bPpg *is called correct with respect to C, if axioms (b1) and (b2) are satisfied.* △

The axioms that we mention in Def. 4.2.1 are illustrated in Fig. 4.17. Axiom (a2) means that the fPpg operation always generates a consistent correspondence relation r' from G'^S which is a consistent model resulting out of the source model update u. Axiom (a1) describes the situation, if the source model update is the identity. Then, fPpg changes nothing. Axioms (b1) and (b2) describe similar situations w.r.t. the dual operation bPpg.

The forward propagation operation (fPpg) consists of a composition of the following auxiliary operations (cf. Def. 4.2.2 and diagrams in Figs. 4.18 and 4.19). Note that the backward model synchronisation process bPpg is dual.

1. The first step is called *forward alignment* (fAln), where the deletion part of the source model update is performed. The fAln operation includes the deletion of the correspondence components that are affected by deleted elements in G'^S.

2. The next step named Deletion (Del), the maximal consistent subgraphs $G_k^S \subseteq G^S$ and $G_k^T \subseteq G^T$ are calculated, so that $G_k = (G_k^S \leftrightarrow G_K^T)$ are consistent, i.e., $G_k \in \mathcal{L}(TGG)$. G_k is calculated so that it takes source model update u into account. G_k^T is the new target model. In this step consistence creating rules are used.

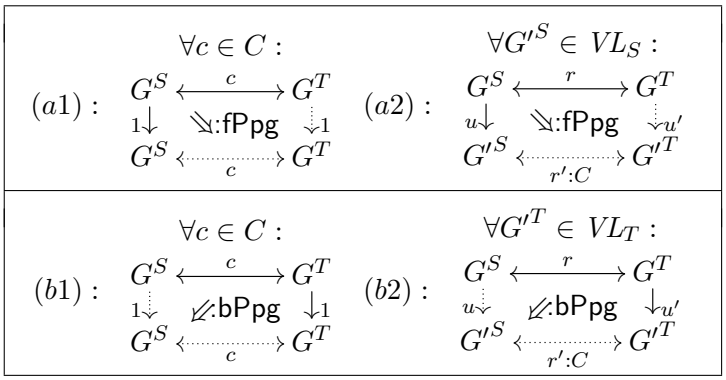


Figure 4.17: Axioms for fPpg and bPpg

3. The last auxiliary operation is the *addition* step (fAdd). G_k^T is extended to G'^T so that the resulting triple graph $G'^S \leftrightarrow G'^T$ is consistent. In detail, the extension is performed in the sense that all elements in G'^S that extend G_k^S are used for the forward transformation to G'^T. This step of the propagation applies forward translation rules (cf. Def. 2.2.10)

In general, graph transformation is non-deterministic. In order to be deterministic, we require that all operational translation rules are deterministic, so that unique resuts can be ensured for the fPpg operation (cf. [EEGH15]). The bPpg is defined analogously. The fPpg operation is now defined formally in Def. 4.2.2.

Definition 4.2.2 (Auxiliary TGG Operations (cf. Def. 7.1 in [HEO+15]) or Def. 9.13 in [EEGH15])**.** *Let $TGG = (TG, \varnothing, TR)$ be a TGG with deterministic sets TR_{CC}, TR_{FT} and TR_{BT} of operational translation rules and let further $MF(TGG)$ be the derived TGG model framework.*

1. *The auxiliary operation* fAln *computing the forward alignment remainder is given by* $\mathsf{fAln}(r, a) = r'$*, as specified in the upper part of Fig. 4.18. The square marked by (PB) is a pullback, meaning that D^C is the intersection of D^S and G^C.*

2. *Let $r = (s, t) : G^S \leftrightarrow G^T$ be a correspondence relation, then the result of the auxiliary operation* Del *is the maximal consistent subgraph $G_k^S \leftrightarrow G_k^T$ of r, given by* $\mathsf{Del}(r) = (u, r', u')$*, which is specified in the middle part of Fig. 4.18.*

3. *Let $r = (s, t) : G^S \leftrightarrow G^T$ be a consistent correspondence relation, $u = (u1, u2) : G^S \to G'^S$ be a source modification and $G'^S \in \mathcal{L}(TGG)^S$. The result of the auxiliary operation* fAdd*, for propagating the additions of source modification u, is a consistent model $G'^S \leftrightarrow G'^T$ ex-*

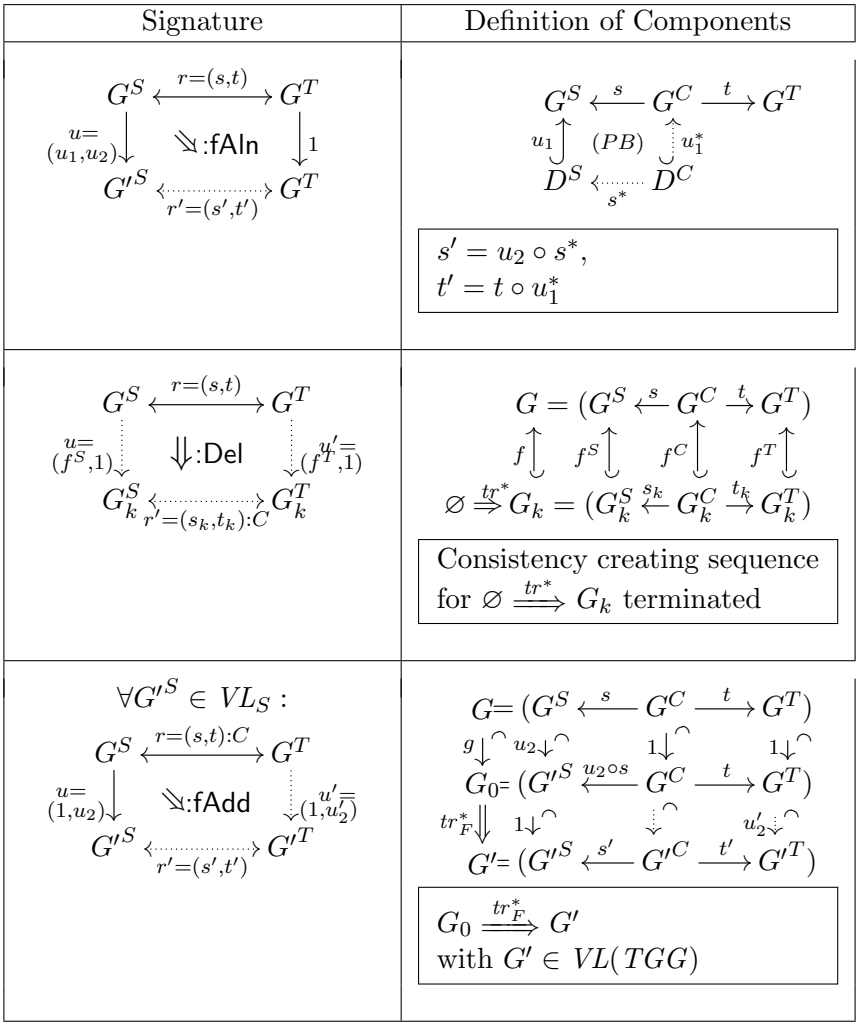


Figure 4.18: Auxiliary operations fAln, Del and fAdd

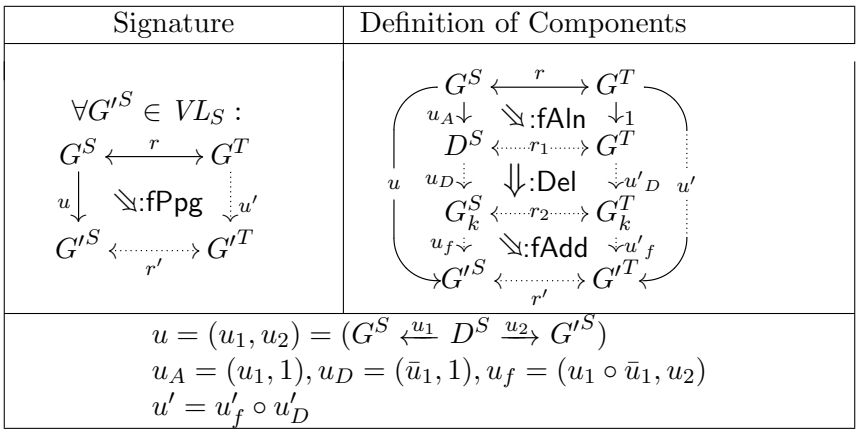


Figure 4.19: Synchronization operation fPpg - formal definition

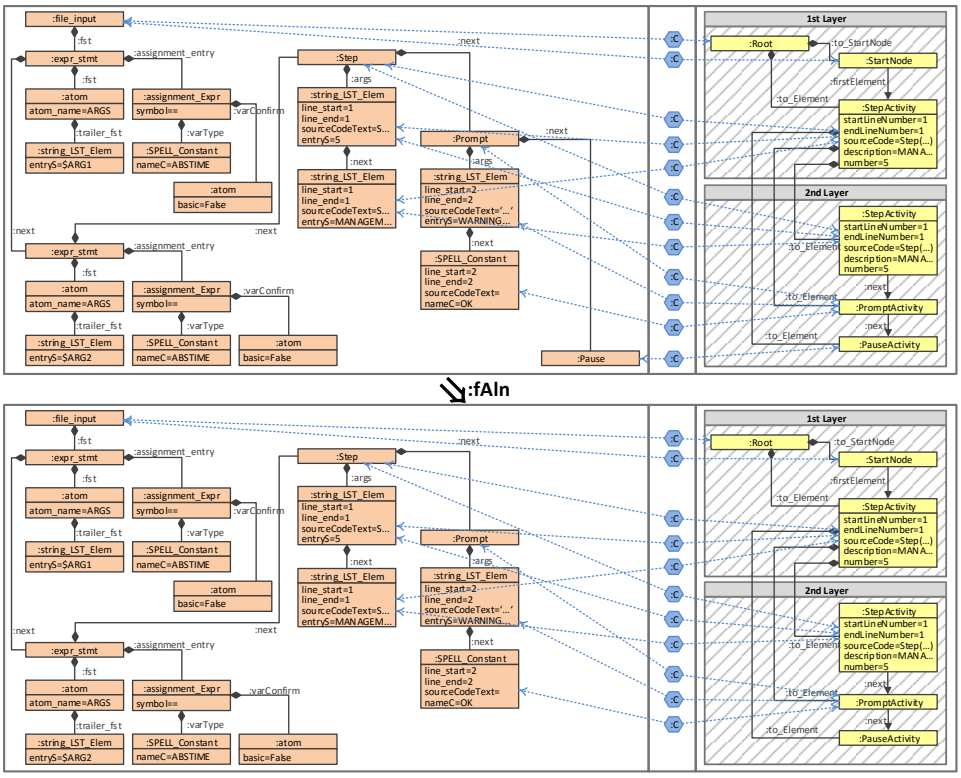


Figure 4.20: Running Example: Forward Alignment (: fAln)

tending $G^S \leftrightarrow G^T$*, and is given by* $\mathsf{fAdd}(r, u) = (r', u')$*, according to the lower part of Fig. 4.18.* △

In applying the source model update from Ex. 4.1.4 to the triple graph in Ex. 4.1.3, we are able to demonstrate the basic model synchronisation framework.

Example 4.2.1 (Running Example: Model Synchronisation)**.** *The source model update u is deleting the* Pause *statement in the SPELL domain, which is the source domain.*

Fig. 4.20 illustrates the initial situation (top), i.e., the triple graph in Ex. 4.1.3. On the bottom, the situation after executing the auxiliary operation fAln *is visualised. There, the* : Pause *node is deleted in the source model as well as the* : next *edge between nodes* : Prompt *and* : Pause*. In addition, the correspondence node which created a relation between the* : Pause *node and the corresponding node in the target model is removed. In addition, both edges that were connected to this correspondence node are deleted, too.*

*In the deletion step (*Del*), a consistent integrated model is derived with regard to the model on top of Fig. 4.21. The consistent intergated model is illustrated on the bottom of Fig. 4.21, where the source and the correspon-*

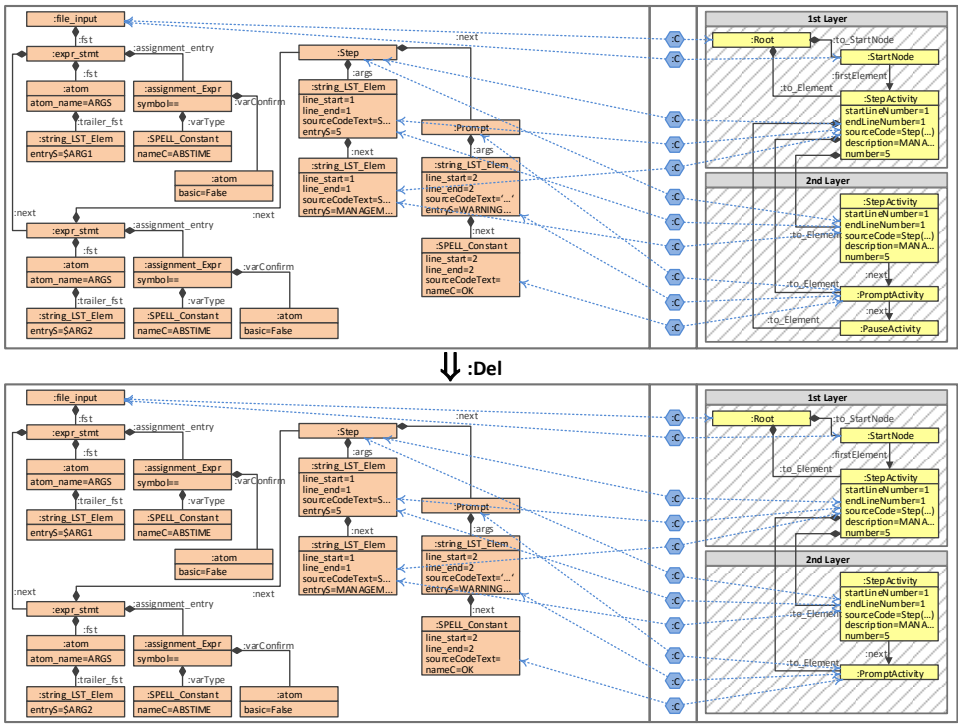


Figure 4.21: Running Example: Deletion (: Del)

dence model stay unchanged, but node : PauseActivity *and edge* : next *are omitted in the SPELL-Flow domain.*

Finally, the auxiliary operation fAdd *is applied to the resulting model of step* Del. *There, forward translation rules will be applied in order to propagate additions performed by the source model update. In our example, the* fAdd *step changes nothing, because there are no additions performed in the source model update. The resulting triple graph is illustrated in Fig. 4.22 (bottom).* △

The basic model synchronisation framework based on TGGs that we introduced in this section, ensures correctness and completeness if the TGG fulfills certain properties. For details, we refer to Sec. 9.2.3, and especially to Thm. 9.25 in [EEGH15].

4.2.1. Invertibility and Weak Invertibility

With the basic model synchronisation framework, another property of model synchronisations was introduced: invertibility [HEO+15]. Intuitively, invertibility means that the propagation operations fPpg and bPpg are inverse to each other in the given context. A attenuated version of invertibility is the weak invertibility. It means that the propagation operations fPpg and bPPg

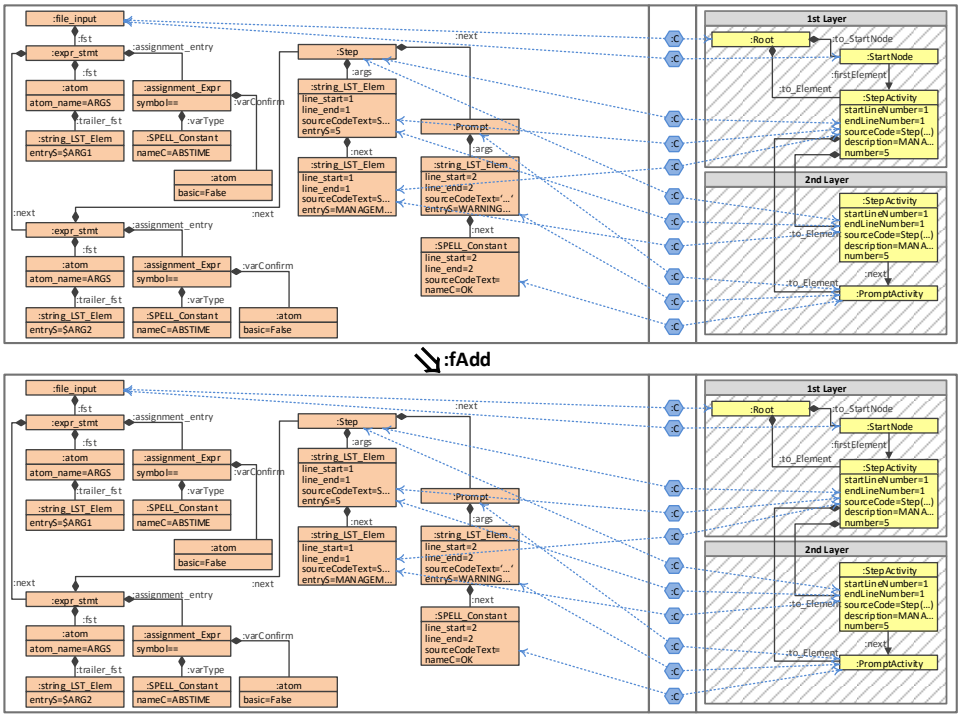


Figure 4.22: Running Example: Forward Addition (: fAdd)

are not directly inverse to each other, instead, the application of operations (fPpg, bPpg, fPpg) or (bPpg, fPpg, bPpg), respectively, lead to the same integrated model. Intuitively, it means that information might get lost during the application of these operations.

Formally, weak invertibility is ensured, if laws (c1) or (c2) in Fig. 4.23 are fulfilled. Invertibility is ensured, if additionally axioms (d1) or (d2) are satisfied.

Invertibility and weak invertibility are formalised by Thm. 8.6 in [HEO+15] or by Thm. 1 in [HEO+11b]. The full proof is given in [HEO+11b].

Example 4.2.2 (Invertible or Weakly Invertible?)**.** *Our running example is not invertible in general, due to the fact that information get lost during the forward model transformation from SPELL to SPELL-Flow, which cannot be restored in the backward transformation process. Therefore, our TGG cannot be invertible. Examples for the loss of information are: the* ARGS *statements are omitted in the SPELL-Flow model or the merging of statements, e.g., comments or list of arguments.*

In order to ensure weak invertibility, the TGG, among other properties, has to be tight. Tightness means that all triple rules of the TGG are creating on the source and on the traget component, so that all derived operational

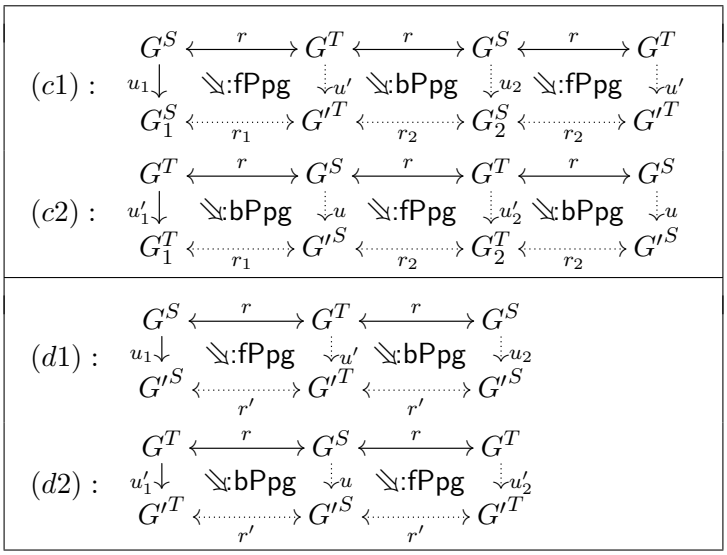


Figure 4.23: Laws for (Weak) Invertible Synchronisation Frameworks

rules change at least one translation attribute. In our TGG, this is not the case: Let us consider triple rule T_NEWLINE_expr_stmt-2-empty_POST *which is illustrated in Fig. 6.6 (top). This rule is only creating on the source component and the target and correspondence domains are empty. Thus, the corresponding backward transformation rule does not change any translation attribute, because it is empty in the target domain.*

To sum it up, the TGG we use in our running example is neither invertible, nor weakly invertible in general. △

4.3. Concurrent Model Synchronistaion

An extension of the basic model synchronisation framework based on triple graph grammars deals with concurrency and was formalised in [HEEO12]. Concurrency means, that we have a model update which performs changes in the source and in the target model simultaneously. All changes shall be propagated to the opposite domain models. Furthermore, the model update does not need to be consistent anymore, because the concurrent model synchronisation framework includes consistency creating operations. They calculate a consistent model out of a possibly inconsistent one.

The general idea of this approach is to combine the forward propagation operation and the backward propagation operation so that the changes in one domain will be propagated to the second domain and then, both changes of the update in the second domain will be "merged" and propagated back to the first domain. In order to procduce consistent results and also in order to reduce conflicts that may occur due to the complexity of model

updates in two domains, the concurrent model synchronisation framework defines new steps: source and target consistency creating operations as well as a semi-automated conflict resolution strategy which will be applied when both domain updates will be "merged" in the second domain. Finally, in [HEEO12], the authors discussed results w.r.t. determinisms, correctness and compatibility with the basic model synchronisation framework, that we will review at the end of this section.

In the remainder of this section, we will introduce the corresponding procedure formally and simulate it by means of our running example from Sec. 4.1. This section is based on the following publications: [HEEO12, GHN+13a, GHN+13b, EEGH15].

Let us consider a model update u which is a combination of the source model update in Ex. 4.1.4 and the target model update Ex. 4.1.5. This means that both domain model updates are performed simultaneously, therefore, they both shall be propagated to our integrated model in Ex. 4.1.3 without any priority for one domain model update.

The concurrent model synchronisation framework provides a solution for solving the question on how to propagate a consistent model update u to a triple graph, so that a consistent integrated model is derived which reflects the changes of u. The model update changes a the triple graph in the source and in the target domain simultaneously. The concurrent model synchronisation operation is called CSynch and reuses the fPpg and bPpg operations.

The formal definition of the concurrent synchronisation problem and framework is given below in Def. 4.3.1.

Definition 4.3.1 (Concurrent Synchronisation Problem and Framework (cf. Def. 2.3 in [HEEO12] or Def. 9.33 in [EEGH15]))**.** *Given a triple graph grammar TGG, the* concurrent synchronisation problem *is to construct a total and non-deterministic operation* $\mathsf{CSynch} : (R \otimes \Delta_S \otimes \Delta_T) \rightsquigarrow (R \times \Delta_S \times \Delta_T)$ *leading to the signature diagram in Fig. 4.24 called concurrent synchronisation tile with concurrent synchronisation operation* CSynch*. Given a pair* $(prem, sol) \in \mathsf{CSynch}$ *the triple* $prem = (r_0, d_1^S, d_1^T) \in R \otimes \Delta_S \otimes \Delta_T$ *is called premise and* $sol = (r_2, d_2^S, d_2^T) \in R \times \Delta_S \times \Delta_T$ *is called a solution of the synchronisation problem, written* $sol \in \mathsf{CSynch}(p)$*. The operation* CSynch *is called* correct *with respect to consistency relation C, if laws (a) and (b) in Fig. 4.24 are satisfied for all solutions. Given a concurrent synchronisation operation* CSynch*, the derived* concurrent synchronisation framework *CSynch is given by* $CSynch = (TGG, \mathsf{CSynch})$*. It is called* correct*, if operation* CSynch *is correct.* △

Law (a) in Fig. 4.24 means that the concurrent model synchronisation framework returns a consistent integrated model. Law (b) describes the case that if the consistent integrated model stays unchanged, then CSynch returns the identity, i.e., the same model which is given as input.

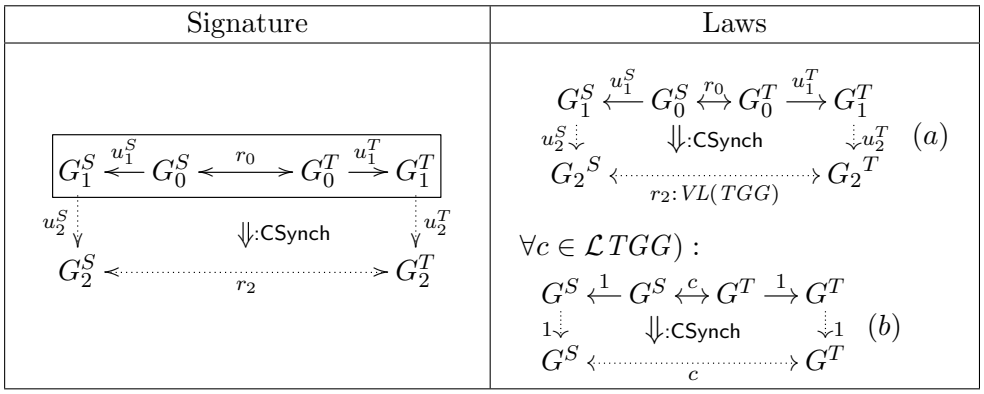


Figure 4.24: Signature and Laws for Correct Concurrent Synchronisation Frameworks

In detail, the concurrent synchronisation framework consists of the following auxiliary operations (cf. Fig. 4.25 and Fig. 4.26).

- CCS & CCT: Consistentcy creating operations of source and of target model. Given a source model update $u_1^S : G_0^S \to G_1^S$. The CCS operation calculates a maximal consistent submodel $G_{1,C}^S$ out of the updated source model G_1^S. For the submodel it holds that $G_{1,C}^S \in \mathcal{L}(TGG)^S$. Operation CCT is dual to CCS and runs on the target model.

- fPpg & bPpg: The forward and backward propagation operations are reused from the basic model synchronisation framework. For details we refer to the previous section Sec. 4.2.

- Res: The conflict resolution operation as it is proposed in [HEEO12] is also called *tentative merge construction*. If both concurrent domain model updates are conflict-free, then both can be merged to one update in order to execute them at the same time. Otherwise, the authors of [HEEO12] distinguish between two kinds of conflicts: *delete-delete* and *delete-insert* conflicts. All conflicts can occur several times simultaneously. A delete-delete conflict is a conflict in which both concurrent domain updates delete the same element. A delete-insert conflict describes the situation in which one domain model update deletes a node, which is also the source or the target of an edge that will be inserted by the other domain model update. Delte-insert conflicts can be solved automatically in prioritising insertion over deletion. Anyway, the resolution strategy is tentative, because the possible solution will be proposed to the user who decides manually how the conflict resolution will be applied.

For more formal details introducing the basis of concurrent graph rules and conflicting graph rules, we refer to Sec. 2.3 in [EEGH15].

Fact 4.3.1 (Conflict Resolution by Tentative Merge Construction (cf. Fact 4.1 in [HEEO12] or Thm. 3 in [EET11a])). *Given two conflicting graph modifications $m_i = G \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$ (i.e., they are not conflict-free). The tentative merge construction yields the merged graph modification $m = (G \leftarrow \bar{D} \rightarrow H))$ and resolves conflicts as follows:*

1. *If (m_1, m_2) are in* delete-delete conflict*, with both m_1 and m_2 deleting $x \in G$, then x is deleted by m.*

2. *If (m_1, m_2) are in* delete-insert conflict*, there is an edge e_2 created by m_2 with $x = s(e_2)$ of $x = t(e_2)$ preserved by m_2, but deleted by m_1. Then x is preserved by m (and vice versa for (m_2, m_1) being in delete-insert conflict).* △

The construction of the derived concurrent synchronisation framework is visulised by the diagrams Fig. 4.25 and Fig. 4.26. In the following, we will cite the construction of the CSynch operation, as it was introduced in [HEEO12].

Construction 4.3.1 (Operation fSynch and CSynch (cf. Constr. 5.2 in [HEEO12] or Constr. 9.37 in [EET11a])). *In the first step (operation CCS), a maximal consistent subgraph $G^S_{1,C} \in \mathcal{L}(TGG)^S$ of G^S_1 is computed. In step 2, the update $u^S_{1,CC}$ is forward propagated to the target domain via operation fPpg. This leads to the pair $(r_{1,F}, u^T_{1,F})$ and thus, to the pair $(u^T_{1,F}, u^T_1)$ of target updates, which may show conflicts. Step 3 applies the conflict resolution operation Res including optional manual modifications. In order to ensure consistency of the resulting target model $G^T_{2,FC}$ we apply the consistency creating operation CCT for the target domain and derive target model $G^T_{2,FCB} \in \mathcal{L}(TGG)^T$ in step 4. Finally, the derived target update $u^T_{2,CC}$ is backward propagated to the source domain via operation bPpg leading to the source model $G^S_{2,FCB}$ and source update $u^S_{2,CB}$. Altogether, we have constructed a nondeterministic solution (r_2, u^S_2, u^T_2) of operation fSynch for the premise (r_0, u^S_1, u^T_1) with $(r_2, u^S_2, u^T_2) = (r_{2,FCB}, u^S_{2,FCB}, u^T_{2,FCB})$ (see Fig. 4.25). The concurrent synchronisation operation bSynch is executed analogously via the dual constructions. Starting with CCT in step 1, it continues via bPpg in step 2, Res in step 3, CCS in step 4, and finishes with fPpg in step 5. The non-deterministic operation CSynch = (fSynch ∪ bSynch) is obtained by joining the two concurrent synchronisations operations fSynch bSynch.* △

Finally, we are able to refer to the definition of the derived concurrent triple graph grammar synchronisation framework.

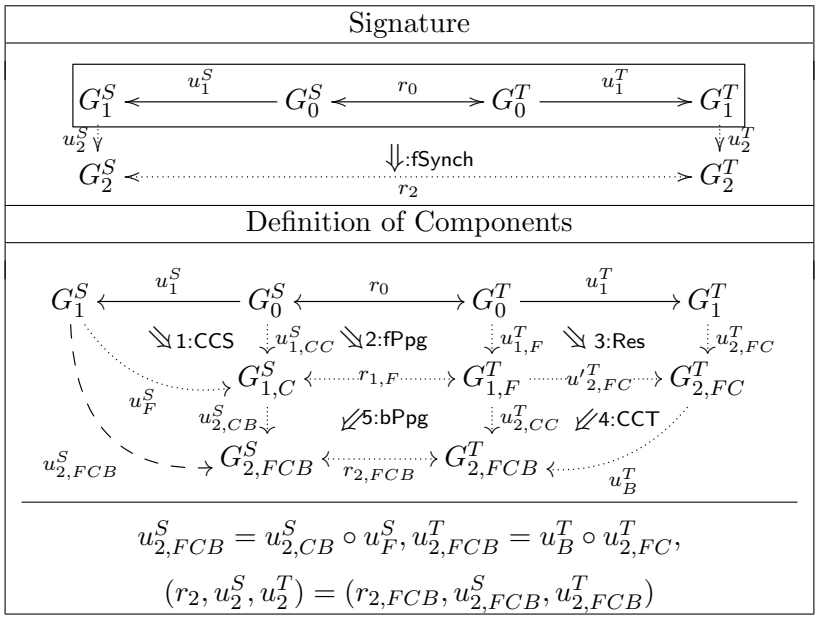


Figure 4.25: Concurrent Model Synchronization With Conflict Resolution (Forward Case: fSynch)

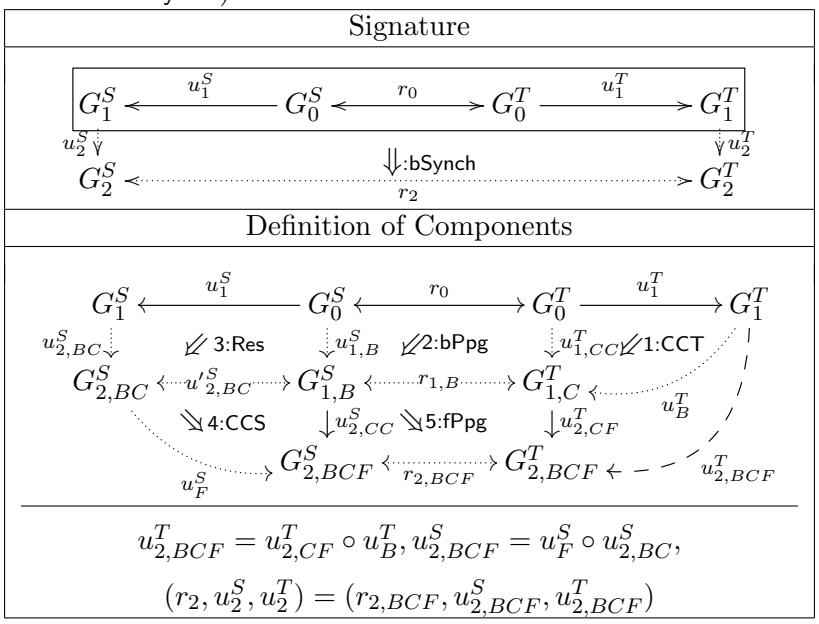


Figure 4.26: Concurrent Model Synchronization With Conflict Resolution (Backward Case: bSynch)

Definition 4.3.2 (Derived Concurrent TGG Synchronisation Framework (cf. Def. 5.5 in [HEEO12] or Def. 9.40 in [EEGH15])). *Let* fPpg *and* bPpg *be correct basic synchronisation operations for a triple graph grammar TGG*

and let operation CSynch *be derived from* fPpg *and* bPpg *according to Constr. 4.3.1. Then, the derived concurrent TGG synchronisation framework is given by* $CSynch = (TGG; \mathsf{CSynch})$. △

In [HEEO12] as well as in [EEGH15], it was examined whether the following properties are met by the framework:

- determinism,
- correctness, and
- compatibility with the basic model synchronisation framework (cf. Sec. 4.2)

For proofs and more details, we refer to Sec. 6 in [HEEO12] or Sec. 9.3.3 in [EEGH15].

As already indicated in Constr. 4.3.1, operation CSynch is nondeterministic in general. The reasons for this behaviour lie in the fact that the choice of using fSynch or bSynch is not fixed and both may result in different integrated models. In addition, the consistiency creating steps CCS and CCT may derive different submodels from the same domain model. Further, the tentative conflict resolution step Res is not deterministic, too, due to the manual user intervention being the solution strategy of the tentative merge construction (cf. Fact 4.3.1).

The derived concurrent TGG synchronisation framework always yields a consistent integrated model. Therefore, it ensures *correctness*, because it fulfills laws (a) and (b) in Fig. 4.24 (cf. Thm. 6.3 in [HEEO12]).

The concurrent model synchronisation framework based on TGGs is compatible with the basic model synchronisation framework which is shown in Thm. 6.4 in [HEEO12]. In detail this means that both frameworks will yield the same integrated models as result of the same domain model update u. Note, in that case, update u changes only one domain, i.e., the update on the other domain is the identity.

Example 4.3.1 (Running Example with Concurrent Model Update)**.** *We demonstrate the concurrent model synchronisation framework by means of the example introduced in Sec. 4.1. Given a concurrent model update* $u = (u_1^S, u_1^T)$ *in the source and target domain (cf. Examples 4.1.4 and 4.1.5) which we want to apply to the triple graph* $G_0 = (G_0^S \leftarrow G_0^C \rightarrow G_0^T)$ *in Ex. 4.1.3.*

For the current example, we decided to apply the forward case of CSynch *(*fSynch*, cf. Fig. 4.25). The first substep of the* fSynch *operation is the application of the source model update* u_1^S *to the source component* G_0^S *of triple graph* G_0*. The top row of Fig. 4.27 shows the application of* u_1^S *to* G_0^S*. The source model update deletes node* : Pause *as well as the corresponding* : next *edge from the source model. The resulting graph is* G_1^S.

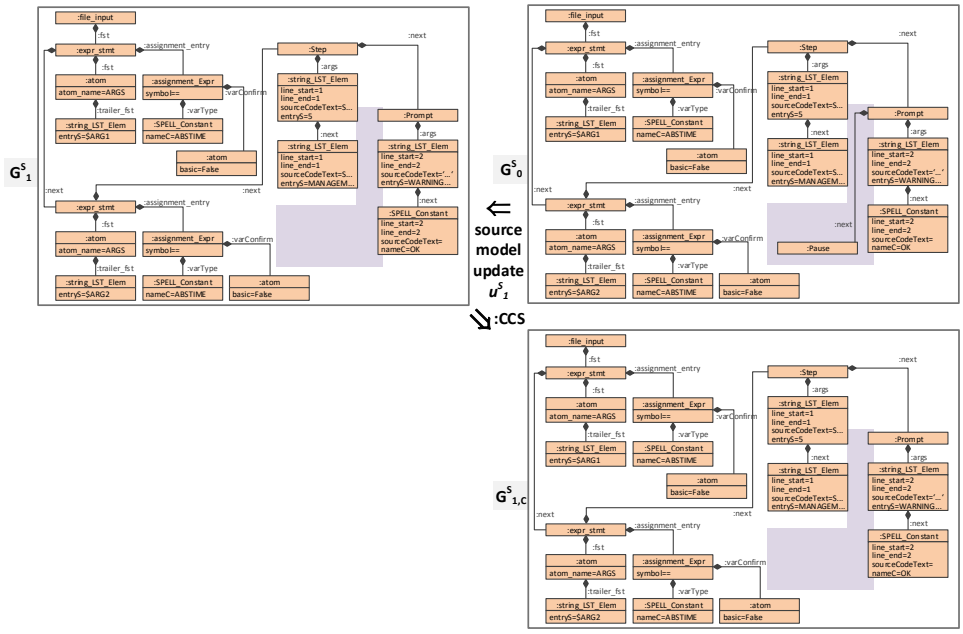


Figure 4.27: Source Model Update u_1^S and CCS Operation

Afterwards the source consistency creating operation : CCS *is executed. For that, a maximal consistent source graph* $G_{1,C}^S$ *will be derived, where* $G_{1,C}^S \subseteq G_1^S$ *is maximal, and* $G_{1,C}^S \in \mathcal{L}(TGG)^S$*. Maximal means that there exists no bigger graph that is subgraph of* G_1^S*. In the current case, it is true that* $G_1^S \in \mathcal{L}(TGG)^S$*, therefore, the* CCS *step returns a graph which is equal to the updated graph, i.e.,* $G_{1,C}^S = G_1^S$*. The* CCS *step is visualised in Fig. 4.27, too.*

Using source model $G_{1,C}^S$ *and triple graph* G_0 *as input, the next auxiliary operation, which is called* fPpg *(forward propagation), derives the corresponding triple graph* $G_{1,F} = (G_{1,C}^S \leftarrow G_{1,F}^C \rightarrow G_{1,F}^T)$ *in using the forward translation rules of the TGG. We depicted the most relevant ones for this running example in Ex. 4.1.8. In detail, we applied the following forward translation rules for translating* $G_{1,C}^S$ *to* $G_{1,F}$ *(cf. Fig. 4.15.*

- FT_T_Step-2-StepActivity *is used for translating the* : Step *node to both* StepActivityNodes*.*

- FT_T_Step_args-2-attrs *is applied for translating the desctiption and the number of the* Step *statement.*

- FT_T_sourceCodeText-2-sourceCode *is applied in the SPELL-domain to all attributes* sourceCode *that will be translated to* sourceCodeText *attributes in the SPELL-Flow domain. In the current case, it translates*

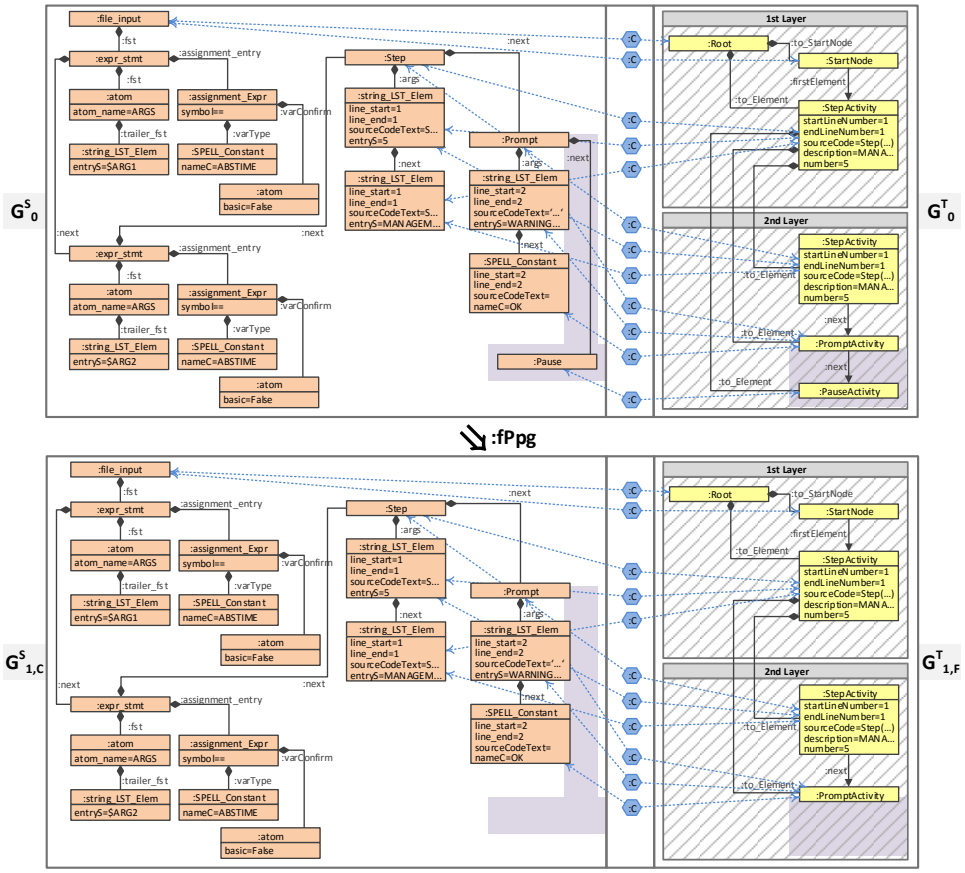


Figure 4.28: fPpg Operation

all sourceCode *attributes of node* : string_LST_Elem *connected to node* : Step.

- *Rule* FT_T_lines-2-lineNumbers *is simuilar to the previous one, but it translates the* start− *and* endLineNumber.
- *FT rule* FT_T_ARGS-2-nothing *will be applied twice. It changes markers of the* ARGS *subtree from* **F** *to* **T**. *This means, that both* ARGS *statements are omitted in the SPELL-Flow domain.*
- *Other FT rules are used, which are not illustrated explicitly:*
  - *For translating the* : fileinput *node to nodes* : Root *and*
  - *For creating the correct connection of the* : StepActivity *node on layer 1 with the* : StartNode.
  - *For translating the* : Prompt *subtree. The FT rule that translates node* : Prompt *is similar to FT rule* FT_T_Pause-2-PauseActivity.

The other rules that translate the arguments of the Prompt *statements resemble FT rule* FT_T_Step_args-2-attrs.

- *In order to get a fully-translated model, some further FT rules are necessary that translate attributes that were not translated, yet, e.g., attributes* sourceCode, line_end *or* line_start *of node* : Prompt.
- *Some FT rules need to translate the edges, that were not touched, yet, e.g., edge* : fst *in the SPELL domain. Those rules may create connections of unattached elements in the SPELL-Flow domain.*

After applying the target model update u_1^T to the original target model G_0^T, we receive target model G_1^T. The target model update adds a comment "TEXT" to the : Prompt *node. We illustrate the target model update application in the top row of Fig. 4.29.*

The next substep of fSynch *is the conflict resolution step* Res. *We show the* Res *step in Fig. 4.29. Source model update u_1^S and target model update u_1^T of model update u are not in conflict, because the deletion of node* : Pause *in the SPELL domain does not affect the addition of the attribute* comment *for node* : PromptActivity *in the SPELL-Flow domain. Therefore, both updates can be merged into one update by the* Res *step. No manual interrogation is necessary, therefore, the* Res *step is deterministic for the current example. The* Res *step reuses target models $G_{1,F}^T$ (result of* fPpg*) and G_1^T (result of target model update application). The outcome of* Res *is target model $G_{2,FC}^T$.*

After the execution of the conflict resolution step, target model $G_{2,FC}^T$ is used as input for the consistency creating step of the target model CCT. *It is executed similar to the* CCS *step on the target model. In the current example, which is illustrated in Fig. 4.30, the* CCT *step returns target model $G_{2,FCB}^T \in \mathcal{L}(TGG)^T$. $G_{2,FCB}^T$ is equal to $G_{2,FC}^T$, because the latter graph is already consistent, i.e., $G_{2,FC}^T \in \mathcal{L}(TGG)^T$.*

Finally, the backward propagation operation fPpg *is applied using triple graph $G_{1,F}$ and the target model $G_{2,FCB}^T$. The backward propagation step uses the backward translation rules that were derived out of the set of triple rules of the TGG (for examples of the triple rules, we refer to Ex. 4.1.6). The BT-rules are not presented in explicit in this work. One example BT-rule is given in Ex. 2.2.5.*

The bPpg *step results in triple graph $G_{2,FCB} = (G_{2,FCB}^S \leftarrow G_{2,FCB}^C \rightarrow G_{2,FCB}^T)$, which we illustrate on the bottom row of Fig. 4.31.*

Note, the backward transformation rule that corresponds to triple rule T_ARGS-2-nothing *is empty on the correspondence and target component, i.e., it changes no translation attributes. Therefore, it is applicable endlessly during the backward transformation process. In order to achieve termination, we decided to remove this BT-rule from the set of backward transformation rules in practice. So, both* ARGS *structures cannot be restored during the* bPpg *operation. Thus, they are missing in the source model $G_{2,FCB}^S$ of*

triple graph $G_{2,FCB}$.

For our running example, the whole forward synchronisation operation fSynch *returns triple graph* $G_{2,FCB}$ *as result of the concurrent model synchronisation process based on TGGs.* △

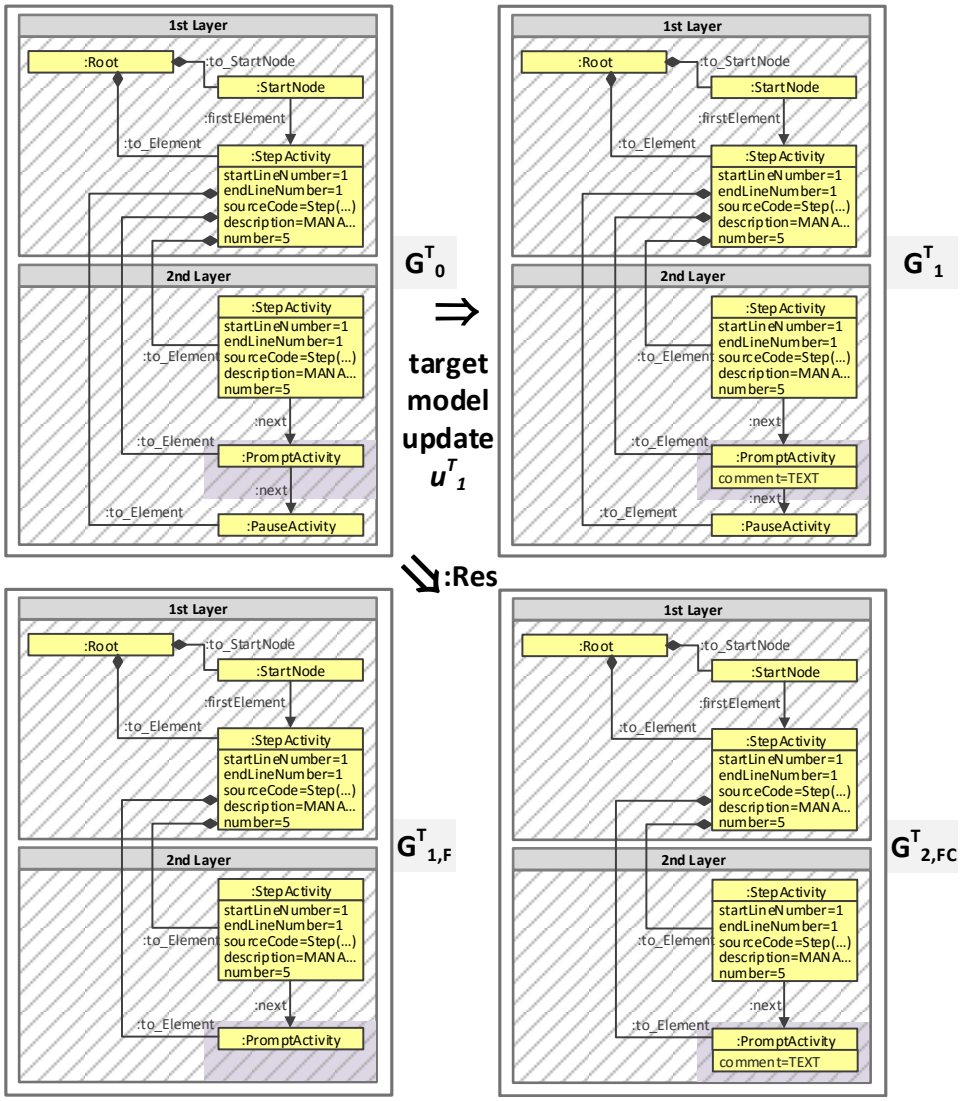


Figure 4.29: Target Model Update u_1^T and Res Operation

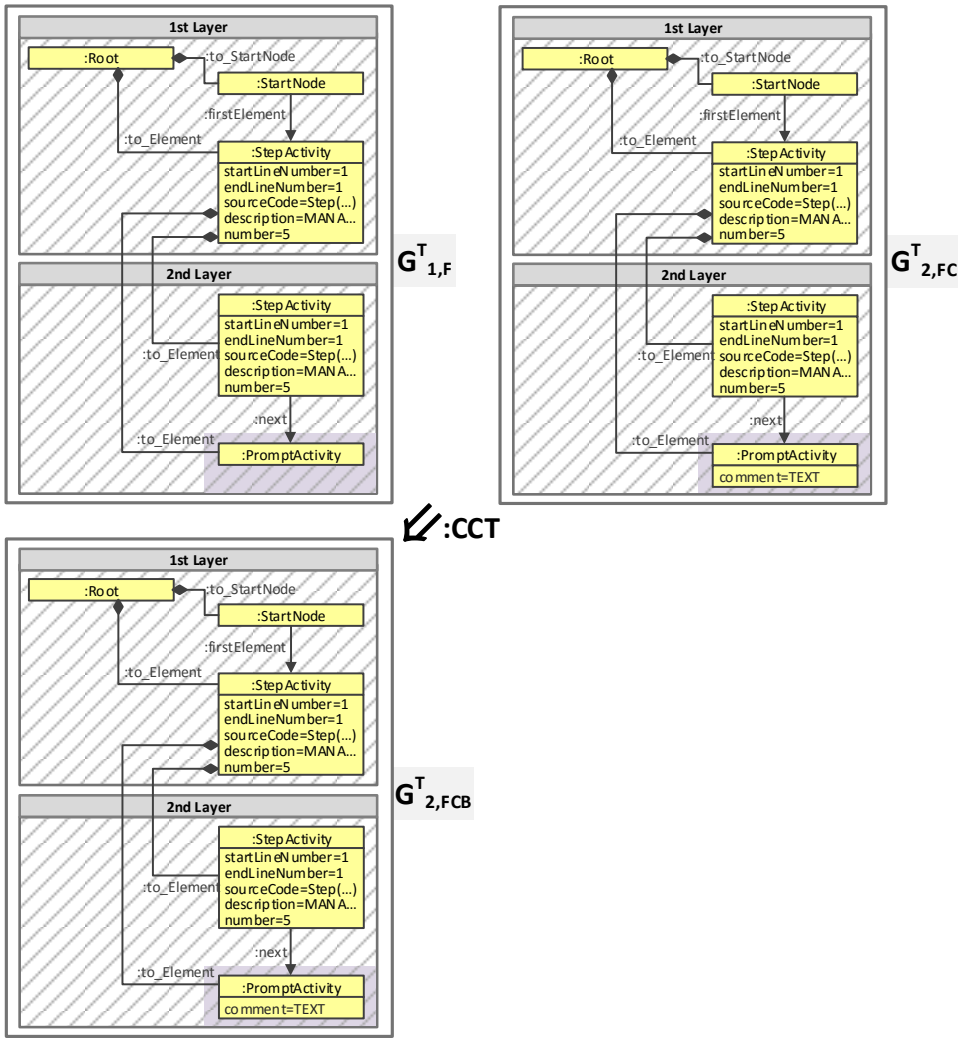


Figure 4.30: CCT Operation

4.4. Non-Deterministic Concurrent Model Synchronisations and Efficiency Improvement

We generalised the concurrent model synchronisation framework that we introduced in the prevoius section to the non-deterministic case [GHN+13a, GHN+13b]. In addition, we applied optimisations to the (non-deterministic) concurrent model synchronisation framework: The second main result of those publications is the efficiency improvement in which we show the applicability of filter NACs Def. 2.2.14 to the concurrent model synchronisation framework. In this section, we will summarise the formalisations and the results of the above-mentioned works.

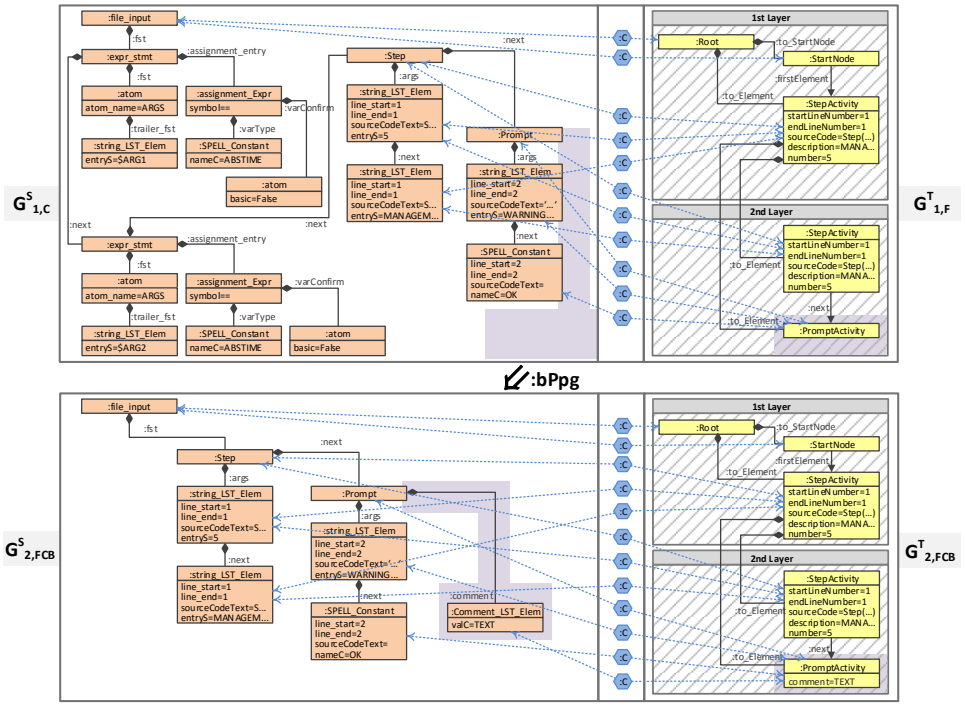


Figure 4.31: bPpg Operation

4.4.1. Non-Deterministic Concurrent Synchronisation Framework

The basic and concurrent model synchronisation framework that we introduced in the previous sections is limited to deterministic propagation operations. In [GHN+13a], we generalised this framework to non-deterministic propagation operations and also to arbitrary TGGs.

First, we define the non-deterministic concurrent synchronisation problem and framework which is the extension of Def. 4.3.1.

Definition 4.4.1 (Non-Deterministic Concurrent Synchronisation Problem and Framework (cf. Def. 2.3 in [GHN+13a] or Def. 3.3 in [GHN+13b]))**.** *Given a triple type graph TG, the* concurrent synchronisation problem *is to construct a non-deterministic operation* CSync *leading to the signature diagram in Fig. 4.24 with concurrent synchronisation operation* CSync*. Given a triple graph grammar* $TGG = (TG, TR)$ *and a concurrent synchronisation operation* CSync*, the* non-deterministic concurrent synchronisation framework *CSynch*$(TGG,$ CSync$)$ *is called* correct*, if laws* (a) *and* (b) *in Fig. 4.24 are satisfied and it is called* complete*, if operation* CSync *is a left total relation.* △

The difference between the concurrent synchronisation framework (cf. Sec. 4.3) and the generalisation to the non-deterministic case is that we do

not require deterministic TGGs. The propagation is executed via the same auxiliary operations of CSynch Figs. 4.25 and 4.26. Due to non-determinism, the each of the steps CCS, CCT and fPpg, bPpg may lead to different results and may require backtracking (cf. Concept 1 in [GHN+13b]).

The non-deterministic concurrent synchronisation framework yields a consistent result for every valid input. This is shown in the following theorem regarding correctness and completeness (cf. Thm. 4.4.1). The theorem and its proof are originally published in [GHN+13b, GHN+13a]. In addition, the generalisation ensures termination, if each operational translation rule changes at least one translation attribute Def. 2.2.18.

Theorem 4.4.1 (Correctness and Completeness of Non-Deterministic Concurrent Synchronisation Framework (cf. Thm. 1 in [GHN+13b] or Thm. 2.5 in [GHN+13a]))**.** *Given a triple graph grammar TGG, the derived non-deterministic concurrent synchronisation framework $CSynch(TGG, \mathsf{CSynch})$ is correct and complete.* △

Proof. By Thm. 1 in [HEEO12] we know that the concurrent synchronisation framework is correct and complete for deterministic sets $TR_{CC}, TR_{FT}, TR_{BT}$ of operational rules. We have to show that the extended operations fPpg and bPpg based on non-deterministic sets $TR_{CC}, TR_{FT}, TR_{BT}$ are correct and complete as well. Since operations fPpg and bPpg are defined symmetrically, it is sufficient to show orrectness and completeness of operation fPpg. By Rem. 3.1 in [GHN+13b] (we cited this remark in Rem. A.8.1), we described the execution of operation fPpg and defined the difference due to backtracking. Using the correctness and completeness result for model transformations based on forward rules (Thm. 1 in [EEHP09]), we know that there is a source consistent forward sequence yielding a consistent integrated model $G' \in \mathcal{L}(TGG)$, if $G'^S \in \mathcal{L}(TGG)^S$. According to the preconditions in laws (a) and (b) in Fig. 4.24 we have that $G'^S \in \mathcal{L}(TGG)^S$ holds and thus, operation fPpg is correct and complete. By symmetry of the definitions, we derive that operation bPpg is correct and complete.

We now consider the remaining steps in Fig. 4.25. Consistency checking operations (CCS, CCT) compute maximal sub models. These operations may also require backtracking, because the sets of operational rules are not necessarily deterministic. Their execution is performed by constructing a corresponding model transformation sequence. Hence, as in the deterministic case, these operations lead to the consistent models $G^S_{1,C} \in VL_S$ and $G^T_{2,FCB} \in \mathcal{L}(TGG)^T$ as required for operations fPpg and bPpg. Moreover, if the given models are consistent already, then there is a corresponding model transformation sequence that translates this model due to the completeness result of model transformations based on TGGs (Thm. 1 in [EEHP09]). This ensures law (b) in Fig. 4.24. Finally, operation Res does not have to ensure special properties. All together, operation fSynch always yields a consistent

integrated model G_2 (correctness law (a) in Fig. 4.24), does not change anything for consistent inputs with identical updates (correctness law (b) in Fig. 4.24). and it provides an output for any input (completeness).

By symmetry of the definitions, we can also conclude that operation bSynch is correct an complete (Fig. 4.26), such that operation CSynch = bSynch ∪ bSynch is correct and complete. □

4.4.2. Efficiency Improvement via Filter NACs

The generalisation of the concurrent model synchronisation framework to the non-deterministic case allows arbitrary TGGs. This may lead to the situation where backtracking is necessary because the framework will produce several possible results. In order to reduce the backtracking steps and therefore reduce conflicts between the operational rules of the underlying TGG, we propose using filter NACs, that we already introduced in Def. 2.2.14.

In [GHN+13a, GHN+13b] we extended the non-deterministic concurrent model synchronisation framework by filter NACs for the following kinds of operational translation rules: forward translation rules, backward translation rules and concistency creating rules. Thus, we are able to improve the efficiency of the model synchronisation in reducing the backtracking steps. Consequently, the synchronisation operations needed to be extended so that they are compatible with filter NACs. Afterwards, it is shown that the correctness and completeness results in Thm. 4.4.1 still hold for the extension by filter NACs.

Intuitively, a filter NAC specifies a pattern of an operational translation rule which will lead to an incomplete translation and therefore require backtracking. Thus, a filter NAC prevents the application of the translation rule in such situations, in order to reduce the backtracking steps.

We will now review the details and illustrate this extension be means of an example at the end of this section.

If we consider the forward propagation operation fPpg in Fig. 4.19 and its sub-steps in Def. 4.2.2. In the basic model synchronisation framework and in the concurrent model synchronisation framework, the second sub-step Del is defined so that it calculates a maximal consistent subgraph w.r.t. G^S and G^T (cf. Sec. 4.2). In the generalisation, we require the Del sub-step to be terminating without necessarily deriving the maximal sub-graph. The reason for this is that operational translation rules that are equipped with filter NACs may reduce the possible triple sequences. Thus, the same triple sequence without filter NACs may not satisfy the maximality condition.

For ensuring consistency of a triple graph g' that was derived with the aid of forward (or backward) translation rules, the set of consistency creating rules TR_{CC} will be equipped with filter NACs, too. If consistency creating rules would not contain the corresponding filter NACs, then the consistency check of G' may fail or backtracking would be necessary, because other CC-

rules would be applicable in contrast to the corresponding FT- (or BT-) rules that were setting up G'.

In the following, we review the definition of filter NACs for the non-deterministic concurrent model synchronisation framework CSynch. Note, we restrict filter NACs to be domain-specific, i.e., they forbid structure on the target domain or on the source domain only. In detail it means, that a domain-specific NAC of a forward translation rule extends the LHS in the source domain and a domain-specific NAC of a backward translation rule extends the LHS in the target domain.

Definition 4.4.2 (Filter NACs for CSynch (cf. Def. 4.5 in [GHN+13b]))**.** *Let tr be a triple rule, tr_{FT} be its derived forward translation, tr_{CC} its derived consistency creating rule and $i : L_{FT} \hookrightarrow L_{CC}$ be the corresponding inclusion of the left hand sides. Let $(n_{FN} \colon L_{FT} \hookrightarrow N_{FN})$ be a domain specific filter NAC, i.e., with $N_{FT} = (N^S_{FT} \leftarrow L^C_{FT} \rightarrow L^T_{FT})$. Then, the consistency creating rule with propagated filter NAC tr_{CN} extends tr_{CC} by the additional NAC $(n_{CN} \colon L_{CC} \rightarrow N_{CN})$ with: $N_{CN} = (N^S_{FN} \xleftarrow{s'} L^C_{CC} \xrightarrow{t'} L_C C^T)$, $s' = s \circ i^S$, $t' = t_{L_{CC}}$, $n^S_{CN} = n^S_{FN}$, $n^C_{CN} = id$, $n^T_{CN} = id$. In the case of a backward translation rule, the construction is performed symmetrically. Let TGG be a triple graph grammar and TR_{FN} and TR_{BN} be the derived sets of forward and backward translation rules possibly extended by filter NACs. Then, $\mathsf{CSynch}(TGG, TR_{FN}, TR_{BN})$ is derived by extending the consistency creating rules TR_{CC} with all propagated filter NACs from TR_{FN} and TR_{BN} leading to a set TR_{CN} and performing the construction as for* CSynch(TGG) *on these sets of extended rules $(TR_{CN}, TR_{FN}, TR_{BN})$.* △

The following result in Thm. 4.4.2, shows that the correctness and completeness results from the concurrent model synchronisation framework (Sec. 4.3) still hold for the non-deterministic case that is extended by filter NACs.

Theorem 4.4.2 (Correctness of Concurrent Synchronization Frameworks with Efficiency Improvement by Filter NACs (cf. Thm. 2 in [GHN+13b]))**.** *Given a triple graph grammar TGG and a set of domain specific filter NACs for the operational translation rules that have been propagated to the consistency creating rules TR_{CC}. Then, the derived non-deterministic concurrent synchronisation framework with domain specific filter NACs $\mathsf{CSync}(TGG, \mathsf{CSynch}_{FN})$ is correct and complete.* △

Proof Idea 4.4.1 (Thm. 4.4.2 (cf. Thm. 2 in [GHN+13b]))**.** *According to Thm. 2 in [GHN+13b], the proof of the theorem uses Thm. 1 in [HEO+15] by which we know that the deterministic concurrent model synchronisation framework (Sec. 4.3) is correct, complete and invertible. The proof of Thm. 4.4.2 is similar but it must be shown that the operational translation rules that are equipped with filter NACs do not affect the correctness property. The full proof is recited in the following [GHN+13b].* △

First, the following facts need to be defined because they are used in the full proof of Thm. 4.4.2.

Fact 4.4.1 (Equivalence of Triple and Extended Consistency Creating Sequences (cf. Fact 10 in [HEO+11b]))**.** *Let $TGG = (TG, \varnothing, TR)$ be a triple graph grammar with derived consistency creating rules TR_{CC} and given $G \in \mathcal{L}(TG)$. Then, the following are equivalent for almost injective matches*

1. *There is a TGT-sequence $s = (\varnothing \overset{tr^*}{\Longrightarrow} G_k)$ via TR with injective embedding $f : G_k \to G$.*

2. *There is a consistency creating sequence $s' = (G'_0 \overset{tr^*_{CC}}{\Longrightarrow} G'_k)$ via TR_{CC} with $G'_0 = Att^{\mathbf{F}}(G)$.*

Moreover, the sequences correspond via $G'_k = H \oplus Att^{\mathbf{T}}_{G_k} \oplus Att^{\mathbf{F}}_{H \setminus G_k}$. △

Proof of Fact 4.4.1. For the full proof we refer to Fact 10 in [HEO+11b]. □

Fact 4.4.2 (Induced Triple Sequence of Consistency Creating Sequence with Additinal NACs (cf. Fact 4.2 in [GHN+13b]))**.** *Let $TGG = (TG, \varnothing, TR)$ be a triple graph grammar with derived consistency creating rules TR_{CC} and let TR'_{CC} be obtained from TR_{CC} by possibly adding some NACs to some of the rules. Then, each consistency creating sequence $s' = (G'_0 \overset{tr^*_{CC}}{\Longrightarrow} G'_k)$ via TR'_{CC} with $G'_0 = Att^{\mathbf{F}}(G)$ induces a triple sequence $s = (\varnothing \overset{tr^*}{\Longrightarrow} G_k)$ via TR with $G_k \subseteq G$.* △

Proof of Fact 4.4.2 (cf. Fact 4.2 in [GHN+13b]). Consistency creating sequence s' via TR'_{CC} is also a consistency creating sequence via TR_{CC}, because disregarding NACs the sets TR'_{CC} and TR_{CC} are the same and all NACs of a rule tr_{CC} in TR_{CC} are also NACs of the corresponding rule tr'_{CC} in TR'_{CC}. Thus, we can apply Fact 4.4.1 to the sequence s' via TR_{CC} and derive the corresponding triple sequence $s = (\varnothing \overset{tr^*}{\Longrightarrow} G_k)$ via TR. □

Fact 4.4.3 (Consistency Creating Sequence with Propagated Filter NACs and Induced Forward Translation Sequence with Filter NACs (cf. Fact 4.3 in [GHN+13b]))**.** *Let TGG be a triple graph grammar and TR_{FN} and TR_{BN} be the derived sets of forward and backward translation rules possibly extended by domain specific filter NACs, and let TR_{CN} be the set of consistency creating rules derived from TR_{CC} by propagating all filter NACs of TR_{FN} and TR_{BN}.*

Let $s = (H'_0 \overset{tr^*_{CN}}{\Longrightarrow} H'_k)$ via TR_{CN} with $H'_0 = Att^{\mathbf{F}}(H)$. Then, there is a triple graph $G \subseteq H$ and a forward translation sequence $s_{FN} = (I'_0 \overset{tr^*_{FN}}{\Longrightarrow} I'_k)$ via TR_{FN}, such that $I'_0 = (H'^S_0 \leftarrow \varnothing \rightarrow \varnothing)$ and $I'_k = (H'^S_k \leftarrow G^C \rightarrow G^T)$. △

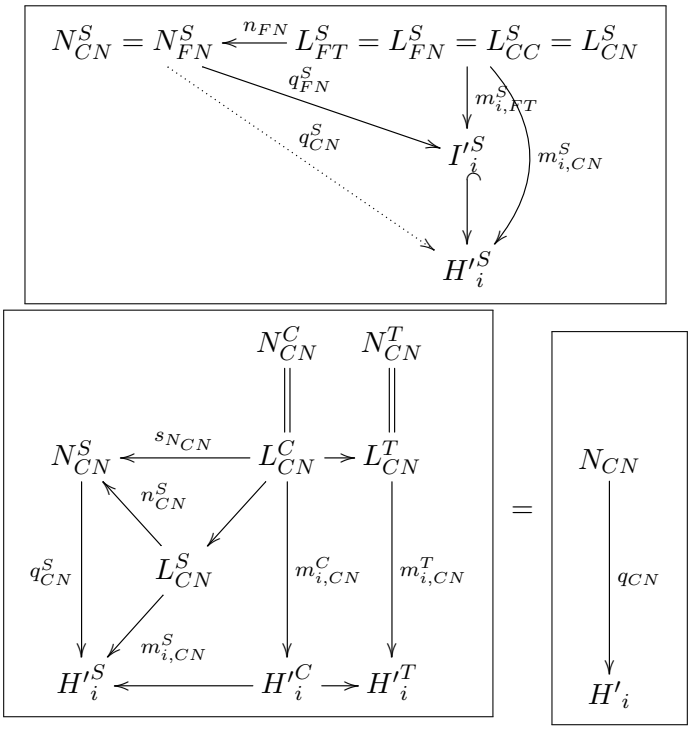


Figure 4.32: Constructions for the proof of Fact 4.4.3

Proof of Fact 4.4.3 (cf. Fact 4.3 in [GHN+13b]). Consistency creating sequence $s = (H'_0 \overset{tr^*_{CN}}{\Longrightarrow} H'_k)$ via TR_{CN} with $H'_0 = Att^{\mathbf{F}}(H)$ implies a corresponding triple sequence $s = (\varnothing \overset{tr^*}{\Longrightarrow} G)$ via TR by Fact 4.4.2. By the composition and decomposition result for TGGs with application conditions (Thm. 1 in [GEH11]) this implies a source consistent forward sequence $s_F = (G_0 \overset{tr^*_F}{\Longrightarrow} G_k)$ via TR_F with $G_0 = (G^S \leftarrow \varnothing \rightarrow \varnothing)$. By Fact 1 in [HEGO10b] (Equivalence of complete forward translation sequences with source consistent forward sequences), there is a corresponding forward translation sequence $s_{FT} = (G'_0 \overset{tr^*_{FT}}{\Longrightarrow} G'_k)$ via TR_{FT} with $G'_0 = (Att_{\mathbf{F}}(G^S) \leftarrow \varnothing \rightarrow \varnothing)$ and $G'_k = (Att_{\mathbf{T}}(G^S) \leftarrow G^C \rightarrow G^T)$.

By Fact 6 in [HEO+11b] (Extension of FT-sequences), this sequence implies the existence of the corresponding extended sequence via forward translation rules $s'_{FT} = (I'_0 \overset{tr^*_{FT}}{\Longrightarrow} I'_k)$ via TR_{FT} with $I'_0 = (Att_{\mathbf{F}}(H^S) \leftarrow \varnothing \rightarrow \varnothing)$ and $I'_k = (H^S \oplus Att^{\mathbf{T}}_{G^S} \oplus Att^{\mathbf{F}}_{H^S \setminus G^S} \leftarrow G^C \rightarrow G^T)$.

It remains to show that sequence s'_{FT} satisfies in each step also the additional filter NACs in TR_{FN}. We show this by contraposition. Assume that there is a domain specific filter NAC $(n_{FN}\colon L_{i,FN} \rightarrow N_{FN})$ of rule $tr_{i,FN}$ and there is a step $I'_i \overset{tr_{i,FT}, m_{i,FT}}{\Longrightarrow} I'_{i+1}$ not satisfying NAC n_{FN}. This means that there is an almost injective morphism $q_{FN}\colon N_{FN} \rightarrow I'_i$

compatible with $m_{i,FT}$, i.e., $q_{FN} \circ n_{FN} = m_{i,FT}$.

We extend morphism q_{FN} as shown in the two diagrams above. On the source component, we obtain $q^S_{CN} = i^S \circ q^S_{FN}$ using the inclusion $i\colon I_i \hookrightarrow H_i$. On the correspondence and target components we can take the match $m_{i,CN}$, because the NAC is domain specific concerning the source domain.

Morphism q_{CN} is a triple graph morphism, because all diagrams above commute. It is almost injective, because q_{FN} is almost injective and $i\colon I_i \hookrightarrow H_i$ is an inclusion. Finally, $q_{CN} \circ n_{CN} = m_{i,CN}$ as depicted in the diagrams above, where the source component is depicted right and commutativity on the correspondence and target component holds by: $q^C_{CN} \circ n^C_{CN} = m^C_{CN} \circ id = m^C_{CN}$ and $q^T_{CN} \circ n^T_{CN} = m^T_{CN} \circ id = m^T_{CN}$. Thus, morphism $q_{CN}\colon N_{CN} \to H_i$ violates a NAC of the step $H_i \overset{tr_{i,CN} H_{i+1}}{\Longrightarrow}$ that corresponds to forward translation step $s_{FN} = (I'_0 \overset{tr^*_{FN}}{\Longrightarrow} I'_k)$. This is a contraction to the given NAC-consistent consistency creating sequence $s = (H'_0 \overset{tr^*_{CN}}{\Longrightarrow} H'_k)$ via TR_{CN}. Therefore, the assumption that there is a forward translation step is invalid. Thus, forward translation sequence $s'_{FT} = (I'_0 \overset{tr^*_{FT}}{\Longrightarrow} I'_k)$ via TR_{FT} is NAC consistent for all NACs of TR_{FN} and therefore, it is a consistent sequence via TR_{FN}. □

Proof of Thm. 4.4.2 (cf. Fact 4.3 in [GHN+13b]). Operations fAIn and bAIn are defined for all inputs, because they are based on pullback constructions. Operation Del is given by a terminating execution of a consistency creating sequence via TR_{CN}. This condition is ensured by the precondition that the set TR_{CN} ensures termination. Finally, operations fAdd and bAadd are not ensured to yield the required results for all possible inputs, because the additional NACs may cut off some transformation seuqences, i.e. they become shorter. Thus, we need to show that the composed operation fPpg is still defined for all inputs and the resulting output is as required.

From the computed consistency creating sequence via operation Del, we derive the corresponding forward translation sequence $G'_0 \overset{tr^*_{FN}}{\Longrightarrow} G'_k$ via TR_{FN} using Fact 4.4.3. This sequence can be extended to a terminated forward translation sequence $G'_0 \overset{tr^*_{FN}}{\Longrightarrow} G'_k \overset{tr'^*_{FN}}{\Longrightarrow} G'_n$ via TR_{FN}. In the general case, we may have to backtrack till we derive a complete forward translation sequence. But, due to the completeness result for forward translation sequences with filter NACs [HEGO10a], we know that there is at least one such sequence. This ensures completeness for the synchronisation operation using the completeness result for concurrent synchronisation without filter NACs [HEEO12].

By Def. 6 in [HEGO10a], a model transformation based on forward translation rules is based on complete forward translation sequences. Thus, by Thm. 1 in [HEGO10a] (correctness), we can conclude that $G = (G^S \leftarrow$

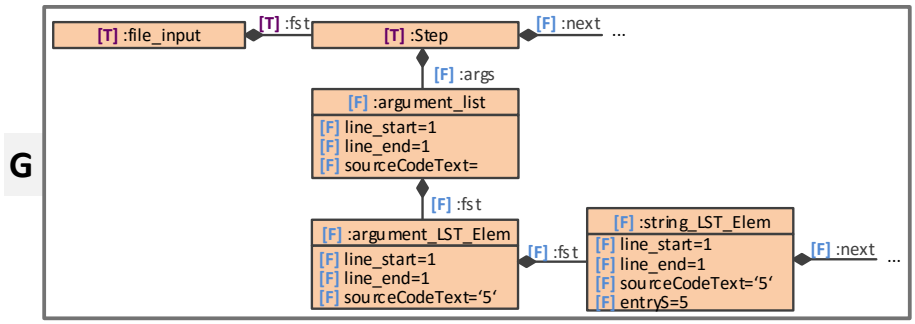


Figure 4.33: Example Triple Graph

$B^C \rightarrow G^T) \in VL$. Therefore, operation fPpg is correct and symmetrically, we derive that operation bPpg is correct. This implies that the concurrent synchronization operation CSynch is correct as well. □

The forward propagation operation fPpg does not require backtracking, if the sets TR_{CN} and TR_{FN} do not require backtracking, according to the proof mentioned above. The similar case holds for the backward propagation operation bPpg. In Thm. 4.4.2, we have shown that the correctness and completness results still hold for the generalised approach, i.e., the non-deterministic concurrent model synchronisation framework.

We will now illustrate the effect by means of the following example (Ex. 4.4.1).

Example 4.4.1 (Filter NACs)**.** *If we consider the triple graph in Fig. 4.33, which is an excerpt taken from the abstract syntax graph of the SPELL running example (cf. Fig. 5.2). Nodes* : file_input *and* : Step *as well as edge* : fst *between both nodes, are already translated during the* fPpg *step. Now, the forward translation operation is able to apply on of the following FT-rules: Either* FT_T_argument_list-2-Argument *or* FT_T_argument_list_fst_argument_LST_Elem-2-Argument*. Both are depicted in Fig. 4.33 and are taken out of the set of triple and FT-rules of our case study (Chap. 6).*

The choice which FT-rule will be applied is non-dterministic. If FT-rule FT_T_argument_list_fst_argument_LST_Elem-2-Argument *is applied, then the translation can be completed. In contrast, if FT-rule* FT_T_argument_list-2-Argument *is applied, then, the* fPpg *operation will lead to a non-consistent model, i.e., a model which cannot be translated completely, because there exists no FT-rule in our set which translates only node* argument_LST_Elem*. Thus, in case FT-rule* FT_T_argument_list_fst_argument_LST_Elem-2-Argument *is applied, backtracking is necessary. If we add a filter NAC for FT-rule* FT_T_argument_list-2-Argument*, which prevents the application of that rule*

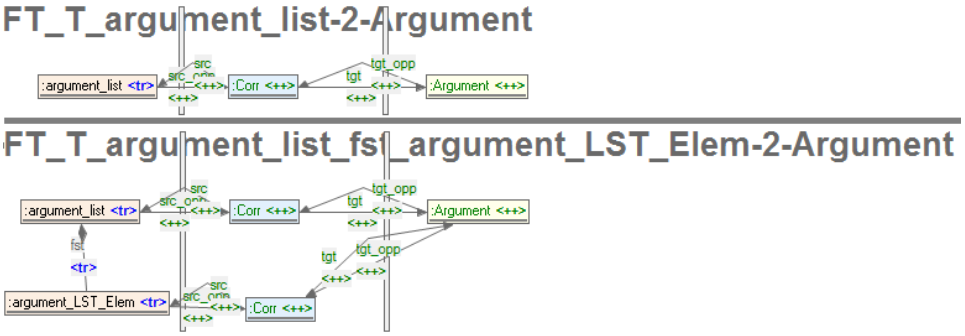


Figure 4.34: Triple Rules

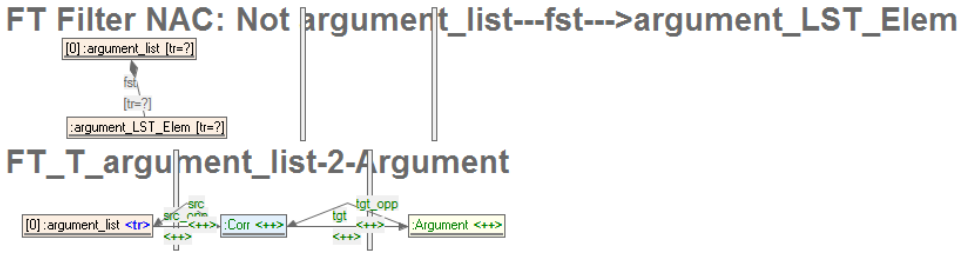


Figure 4.35: Forward Translation Rules

in the given context, then we are able to achieve determinism for the given case. Note, the case study in Chap. 6 does not use generated filter NACs, instead, we created each filter NAC manually. △

Propagation of Model Updates in Multi-View Models

5

In the previous chapter, we presented the bidirectional model synchronisation framework for triple graphs which is defined for 1:1 models. During the work on the bidirectional *SPELL* ↔ *SPELL-Flow* software translation project in cooperation with the industrial partner SES, we realised that the underlying model in this project can be seen as 1:m model. Furthermore, the different levels of hierarchy in the SPELL-Flow model contain duplicates of the same data, due to the different abstraction layers. We decided to call the different abstraction layers in the SPELL-Flow domain *views*. Thus, we extended this model synchronisation framework in order to answer the following research question:

If a model update in one view is performed, then how is it possible to

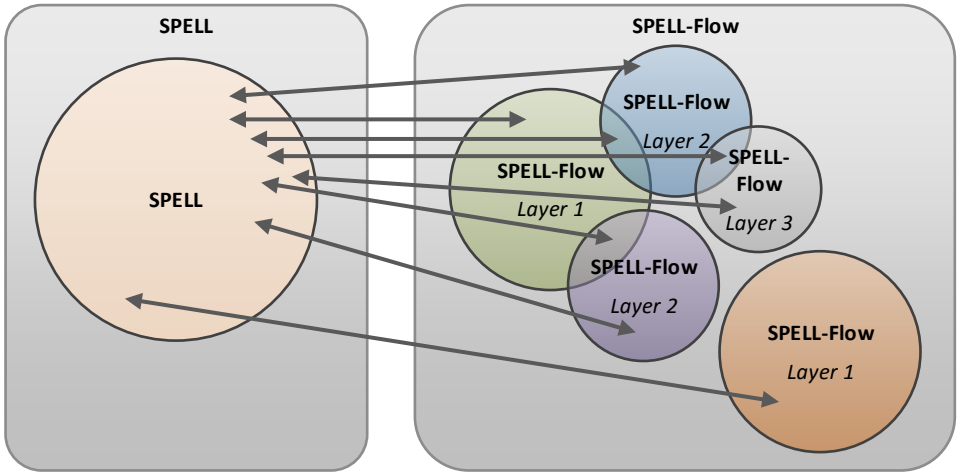


Figure 5.1: The bidirectional software translation between SPELL and SPELL-Flow uses views in the target model

consistently propagate this model update to all other views and also to the other domain?.

This section is structured as follows: First, we will introduce the *derived propagation framework* in presenting the formal details and in using our running example from Chap. 5.

This chapter is based on the article in [GNE+16b] and the corresponding technical report [GNE+16a]. The *derived propagation framework* we developed takes a 1:1 model as input, i.e., a triple graph. The structure of this triple graph is not relevant. As already mentioned at the beginning, we want to define a new propagation framework for n:m models. In detail, we realise this fact in our running example by a 1:1 model that may contain different sub-models. In the running example, the sub-models are represented by the different views in the SPELL-Flow model.

5.1. Introduction of Running Example

In the current section, we present an excerpt of the running example in Sec. 1.3 which we use throughout the current chapter. The excerpt is similar to the one in Sec. 4.1, i.e., the SPELL-Flow model is identical, whereas the SPELL source code is slightly smaller, because the first three source code lines are omitted here.

Example 5.1.1 (SPELL source code and corresponding ASG)**.** *The following listing shows the excerpt of the SPELL source code. It is the same source code as in Ex. 4.1.1, but the first three source code lines are missing.*

Listing 5.1: Running example: SPELL code, part of Step 5 (source instance)

```
1 Step('5', 'MANAGEMENT OF COMPONENT A')
2     Prompt('WARNING: Check something before.', OK)
3     Pause()
```

The corresponding SPELL abstract syntax graph (ASG) of the given source code is visualised in the following Fig. 5.2. This screenshot is taken in HenshinTGG.

The SPELL ASG in this chapter is a very similar to the SPELL ASG in Fig. 4.1. The big difference is that the subtrees representing the ARGS *assignments (and the comment in line 3) are missing. In the current visualisation of the SPELL ASG, we use the same highlighting as in Fig. 4.1: The topmost node* file_input*, i.e., the main container of this ASG, is highlighted in red. In contrast to Fig. 4.1, the* file_input *node is followed by the subtree that represents the* Step *statement (highlighted in yellow). The* Step *is followed by the* Prompt *structure (highlighted in grey) and finally followed by the* Pause *subtree (highlighted in violet).* △

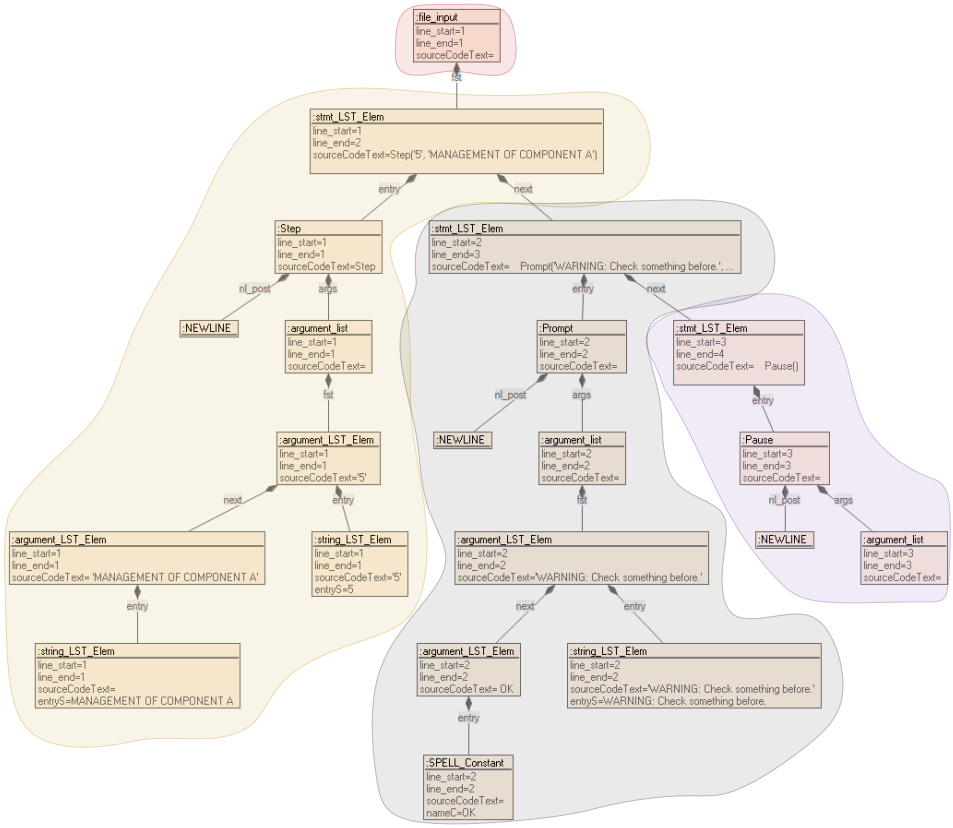


Figure 5.2: Running example: SPELL ASG, part of Step 5 (source instance)

Example 5.1.2 (SPELL-Flow visualisation and corresponding ASG)**.** *The screenshot taken in HenshinTGG of the SPELL-Flow visualisation that corresponds to the SPELL model in Ex. 5.1.1, is identical to the SPELL-Flow visualisation in Ex. 4.1.2. Consequently, the SPELL-Flow ASG depicted in Fig. 5.4 is identic to the one in Ex. 4.1.2, too.*

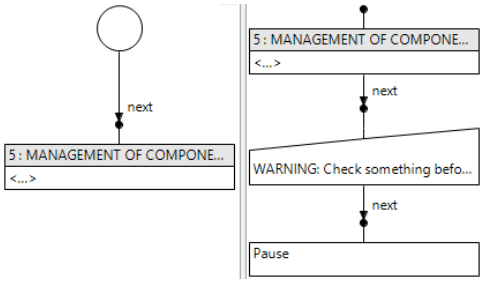


Figure 5.3: Running example: SPELL-Flow visulisation, part of Step 5 (target instance)

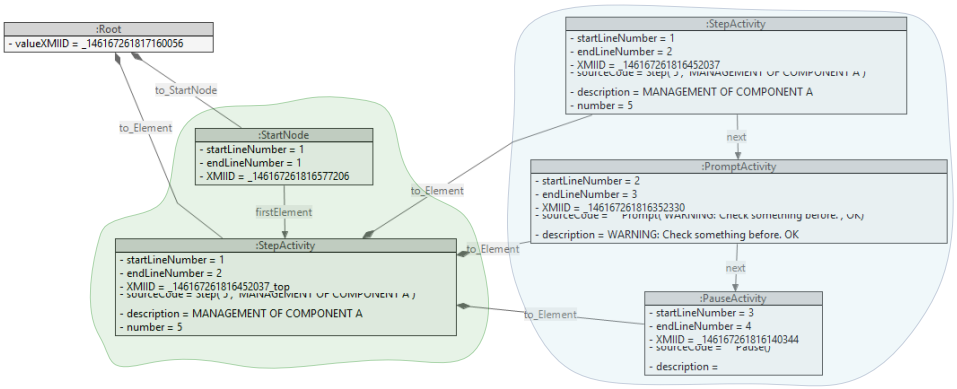


Figure 5.4: Running example: SPELL-Flow model, part of Step 5 (target ASG)

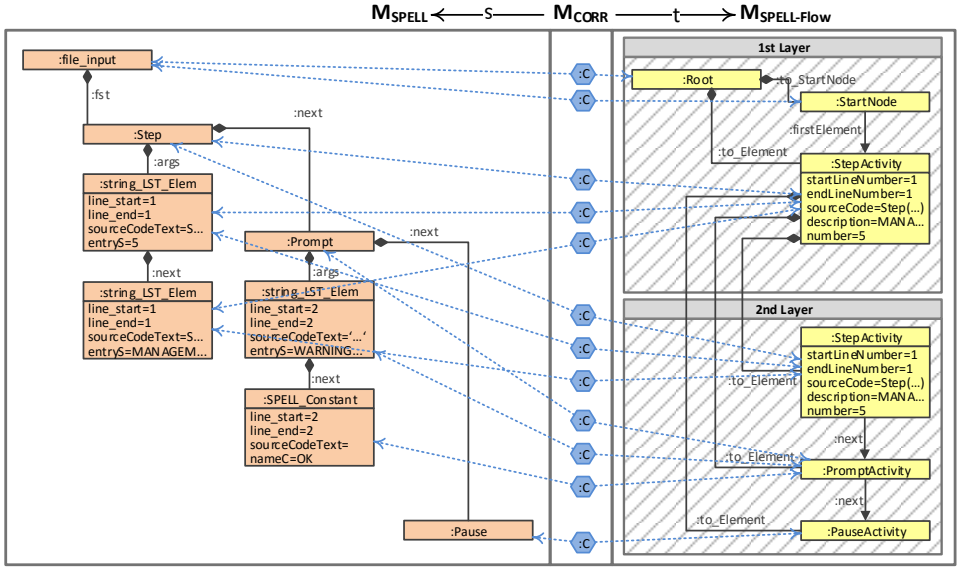


Figure 5.5: Running Example: Triple Graph

The two nodes highlighted in green are the ASG nodes that correspond to both elements on the main layer of the SPELL-Flow visualisation. The nodes highlighted in blue constitute the second hierarchy level. △

Example 5.1.3 (SPELL ⇔ SPELL-Flow Triple Graph)**.** *The following figure shows the triple graph M that is typed over a reduced SPELL ASG and over the SPELL-Flow ASG. Moreover, this triple graph includes important correspondences. In order to improve the readability, we omitted elements in this image. Note, the whole running example in this chapter uses the reduced version, similar to the example in Ex. 4.1.3. The full triple graph, which is still reduced by attributes, that are not important in this chapter, is*

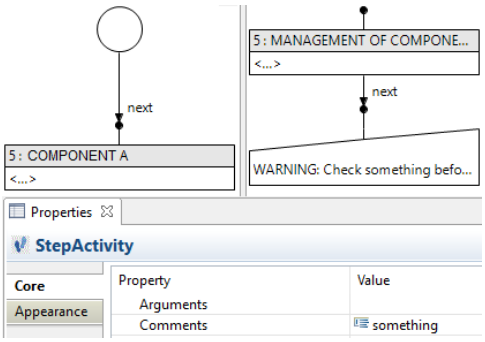


Figure 5.6: Running Example: Result of Target Domain Model Update in Visualisation

illustrated in Appendix A.6. △

Example 5.1.4 (Domain model update in target domain)**.** *We want to execute the following model update in the target domain, i.e., in the visual model of SPELL-Flow:*

- *Delete* Pause *node in the second layer of the SPELL-Flow visualisation.*
- *Change the description of the* Step *node in the main layer (1st layer) from* 5, MANAGEMENT OF COMPONENT A *to* 5, COMPONENT A.
- *Add a comment "something" to the* : Step *node on first layer.*

In Fig. 5.6 the intermediate result of the target model update in the SPELL-Flow visaulisation tool is illustrated. The left side of the screenshot shows the first (main) layer. The right side illustrates the second layer which belongs to the Step *statement. In the screenshot, the additional comment is only visible in the properties view of the corresponding* Step *node (bottom).* △

Example 5.1.5 (Relevant Triple Rules)**.** *The subset of relevant triple rules that we use in this chapter is already described in detail in Ex. 4.1.6.* △

Example 5.1.6 (Triple Graph Grammar TGG and Language $\mathcal{L}(TGG)$)**.** *Given the* $TGG = (MM, \varnothing, TR)$ *over triple type graph* MM *in Ex. 2.2.1 and with triple rules* TR*. Then the triple graph in Fig. 5.5 is contained in* $\mathcal{L}(TGG)$*.* △

5.2. (Domain) Model Update

We focus on consistent propagations of model updates. Therefore, we clarify the notion of consistency first. Given a TGG over type graph

$MM = (MM_{D_1} \xleftarrow{s} MM_C \xrightarrow{t} MM_{D_2})$, then all graphs in $\mathcal{L}(TGG)$ are called consistent integrated models. All graphs in $\mathcal{L}(TGG)^{D_1}$ are called consistent models of domain D_1 and all graphs in $\mathcal{L}(TGG)^{D_2}$ are called consistent models of domain D_2. A model update leads to state changes in models and is given by a span of inclusions $u\colon M \xleftarrow{u_1} M_P \xrightarrow{u_2} M'$. Equivalent to rules, all elements in $M \setminus u_1(M_P)$ are deleted, all elements in M_P are preserved and all elements in $M' \setminus u_2(M_P)$ are created. An update is consistent, if M' is a consistent model. An update is called D_1-update, if it leads to changes in models of domain D_1 only and it is called D_2-update, if it leads to changes in models of domain D_2 only. If the model update is unknown, existing methods of difference compution [EEGH15] can be used to obtain the update from the state-changes of models.

Definition 5.2.1 ((Consistent) (Domain) Model Update)**.** *Given models M, M', then the update $u = (u_1, u_2)$ between M and M' is given by a span of inclusions $u\colon M \xleftarrow{u_1} M_P \xrightarrow{u_2} M'$ with $u_1, u_2 \in \mathcal{M}$. Given a TGG over triple type-graph $MM = (MM_{D_1} \xleftarrow{s} MM_C \xrightarrow{t} MM_{D_2})$, then u is called a D_i-domain update, short D_i-update, if $M, M_P, M' \in \mathcal{L}(MM_{D_i})$ $(i = 1, 2)$. A D_i-update is consistent, if $M' \in \mathcal{L}(TGG)^{D_i}$. With $\Delta_{D_i} = \{u \mid u$ is a D_i-update$\}$ we denote all updates in domain D_i.* △

Example 5.2.1 (Domain Model Update)**.** *We already informally introduced the model update which is performed in the SPELL-Flow domain (target domain) in Ex. 5.1.4. In Fig. 5.7 (top) we illustrate the target model update $u : M \leftarrow M_P \rightarrow M'$ with model $M = M_{SPELL-Flow}$ using the SPELL-Flow graph representation. Due to readability, we illustrate only the SPELL-Flow domain, i.e., the target domain, but implicitly assume the whole model. Furthermore, the image uses the compact notation (cf. Rem. 2.2.1). Parts that are deleted, are marked in red and with $--$. They are contained in M, but not in M_P and M. Parts that will be added by the update, are marked in green and with $++$. Those elements are contained in M', but not in M and M_P. All unmarked parts are contained in all, M, M_P and M'.*

Note that the update is not consistent w.r.t. the TGG (cf. Ex. 5.1.6), i.e., $M' \notin \mathcal{L}(TGG)^{SPELL-Flow}$.

The result which we want to obtain using the derived propagation framework *shall be consistent in the language $\mathcal{L}(TGG)$ of the whole triple graph grammar. Then, the final triple graph shall be changed as listed below:*

- *The deletion of the* : PauseActivity *shall be reflected in the SPELL domain (source domain), i.e., the* : Pause *node with its containment edge* : next *needs to be removed. Furthermore, the cooresponding* : C *node with edges have to be deleted, too.*
- *The change of the* description *in the* : StepActivity *on first layer needs to be reflected in the* : StepActivity *on second layer and also in the corresponding* : string_LST_Elem *that contains this description, too.*

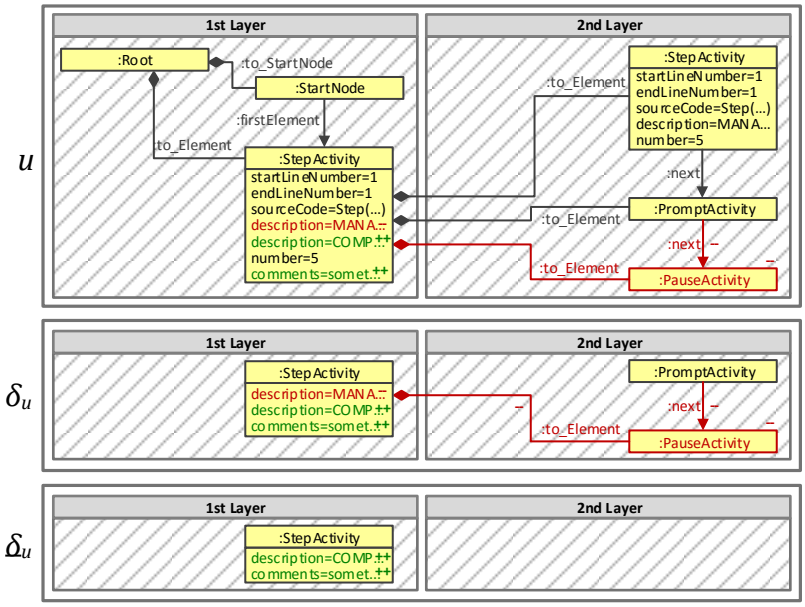
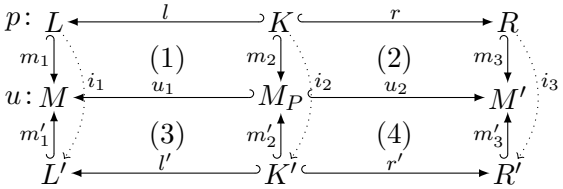


Figure 5.7: SPELL-Flow Domain Model Update (u), Delta (δ_u) & Creating Delta ($\underline{\delta}_u$)

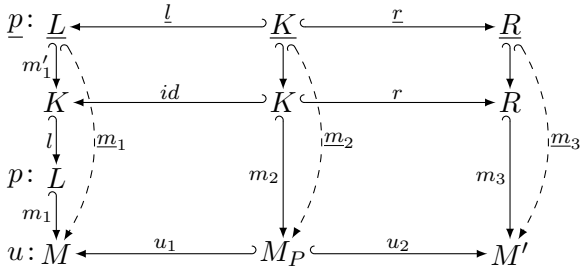
- *The addition of a comment for the* : StepActivity *on the first layer shall be propagated to the corresponding* : StepActivity *node on the second layer and also to the SPELL domain.* △

The delta δ_u of an update u is u but restricted to those elements only that are touched by the update. This includes all elements that are created and deleted by u as well as the elements that are directly connected to them. Therefore, the delta is minimal in the sense that there exists no smaller rule that reflects the update. This concept is based on the concept of minimal rules (cf. Def. 6 in [EET11b]). The creating delta $\underline{\delta}_u$ of an update u is delta δ_u but restricted to the created elements of u only. We will now define the update delta δ_u in Def. 5.2.2 and the creating delta $\underline{\delta}_u$ in Def. 5.2.3.

Definition 5.2.2 (Update Delta)**.** *Given a model update* $u\colon M \xleftarrow{u_1} M_P \xrightarrow{u_2} M'$*, then the* delta $\delta_u = (p, m_1)$ of u *is given by a rule* $p\colon L \xleftarrow{l} K \xrightarrow{r} R$ *with* $l, r \in \mathcal{M}$ *and match* $m_1\colon L \to M \in \mathcal{M}$ *whose application to* M *via* m_1 *leads to* M' *with pushouts (1), (2) such that for all rules* $L' \xleftarrow{l'} K' \xrightarrow{r'} R'$ *with* $l', r' \in \mathcal{M}$ *and match* $m_1'\colon L' \to M \in \mathcal{M}$ *whose application to* M *via* m_1' *also leads to* M' *with pushouts (3), (4) it is true that there exists inclusions* i_1, i_2, i_3 *with* $(m_j = m_j' \circ i_j)_{j \in \{1..3\}}$*.* △



Definition 5.2.3 (Creating Update Delta)**.** *Given the delta δ_u of u, the* creating delta $\underline{\delta_u} = (\underline{p}, \underline{m_1})$ of u *is given by the delta* $(\underline{p}\colon \underline{L} \xleftarrow{\underline{l}} \underline{K} \xrightarrow{\underline{r}} \underline{R}, m_1' \colon \underline{L} \to K)$ *of update* $K \xleftarrow{id} K \xrightarrow{r} R$ *with* $\underline{m_1} = m_1 \circ l \circ m_1'$. △



Remark 5.2.1 ((Creating) Update Delta)**.** *In $\mathcal{M}$-adhesive categories, morphisms $m_2\colon K \to M_P$ and $m_3\colon R \to M'$ for the update delta as well as morphisms $\underline{m_2}\colon \underline{K} \to M_P$ and $\underline{m_3}\colon \underline{R} \to M'$ for the creating update delta are uniquely induced by the uniqueness of pushout complements and pushout objects (cf. Thm. 4.22 in [EEGH15]).* △

Example 5.2.2 (Delta & Creating Delta)**.** *The rule $p\colon L \leftarrow K \rightarrow R$ in Fig. 5.7 (middle) illustrates the delta $\delta_u = (p, m)$ of model update u (cf. Ex. 5.2.1). It contains only those elements, that are touched by u. In detail these are:*

- : StepActivity *on the first layer with attributes* description *and* comments.
- : PromptActivity *and* : PauseActivity *on the second layer with edges* : to_Element *and* : next.

Analogously to the update, all elements not marked with ++ are contained in L, all unmarked elements are contained in K and all elements not marked with −− are contained in R. The match $m\colon L \to M$ is uniquely determined.

The image on the bottom shows the corresponding creating delta $\underline{\delta_u} = (\underline{p}, \underline{m})$ of u with $\underline{p}\colon \underline{L} \leftarrow \underline{K} \rightarrow \underline{R}$. The creating delta is δ_u but restricted to the created elements of u. In detail, this is the : StepActivity *on the first layer with its attributes* description *and* comments. △

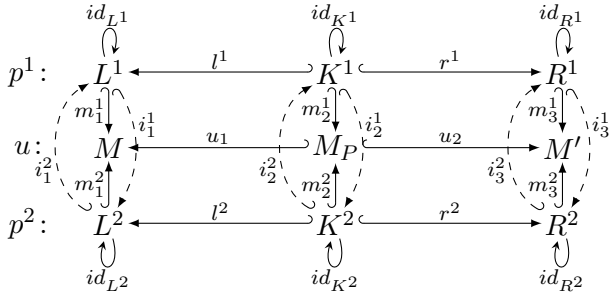
Note that the delta of a model update is unique up to isomorphism. Therefore, in the following, with the delta δ_u of an update u we implicitly

refer to the equivalence class of all isomorphic deltas of u with δ_u being a representant of this class. The same holds for the creating update delta $\underline{\delta}_u$. Two deltas $(\delta_u^i = (p^i \colon L^i \leftarrow K^i \rightarrow R^i, m_1^i))_{i=1,2}$ are isomorphic if both rules p^i with matches m_1^i are isomorphic, i.e., both rule applications via matches m_1^i delete, preserve and create the same elements.

Definition 5.2.4 (Isomorphic Rules with Matches)**.** *Given a model M and two tuples $(p^i, m^i)_{i=1,2}$ of rules $p^i \colon L^i \xleftarrow{l^i} K^i \xrightarrow{r^i} R^i$ with corresponding matches $m^i \colon L^i \rightarrow M$. Tuple (p^1, m^1) is isomorphic to (p^2, m^2), written $(p^1, m^1) \cong (p^2, m^2)$, if there exists isomorphisms $i_1 \colon L^1 \rightarrow L^2, i_2 \colon K^1 \rightarrow K^2$ and $i_3 \colon R^1 \rightarrow R^2$ such that $i_1 \circ l^1 = l^2 \circ i_2$, $i_3 \circ r^1 = r^2 \circ i_2$ and $m^1 = m^2 \circ i_1$.* △

Proposition 5.2.1 (Uniqueness of Update Delta)**.** *Given a model update $u \colon M \xleftarrow{u_1} M_P \xrightarrow{u_2} M'$ in an $\mathcal{M}$-adhesive category, then delta δ_u of u is unique up to isomorphism, i.e., for any two deltas $(\delta_u^i)_{i=1,2}$ of u it is true that $\delta_u^1 \cong \delta_u^2$.* △

Proof. Let $(\delta_u^i = (p^i \colon L^i \xleftarrow{l^i} K^i \xrightarrow{r^i} R^i, m_1^i \colon L^i \rightarrow M))_{i=1,2}$ be two deltas of u. By the definition of deltas, there exists inclusions $i_1^1 \colon L^1 \rightarrow L^2, i_2^1 \colon K^1 \rightarrow K^2, i_3^1 \colon R^1 \rightarrow R^2$ with $(m_j^2 \circ i_j^1 = m_j^1)_{j \in \{1..3\}}$. Conversely, there exists inclusions $i_1^2 \colon L^2 \rightarrow L^1, i_2^2 \colon K^2 \rightarrow K^1, i_3^2 \colon R^2 \rightarrow R^1$ with $(m_j^1 \circ i_j^2 = m_j^2)_{j \in \{1..3\}}$. By $m_1^1 \circ id_{L^1} = m_1^2 \circ i_1^1 = m_1^1 \circ i_1^2 \circ i_1^1$ and $m_1^1 \in \mathcal{M}$, m_1^1 being a monomorphism implies that $id_{L^1} = i_1^2 \circ i_1^1$. Conversely, by $m_1^2 \circ id_{L^2} = m_1^1 \circ i_1^2 = m_1^2 \circ i_1^1 \circ i_1^2$ and $m_1^2 \in \mathcal{M}$, m_1^2 being a monomorphism implies that $id_{L^2} = i_1^1 \circ i_1^2$. Thus, $i_1^1 \colon L^1 \rightarrow L^2$ is an isomorphism. Analogously, it is shown that $i_2^1 \colon K^1 \rightarrow K^2$ and $i_3^1 \colon R^1 \rightarrow R^2$ are isomorphisms. Furthermore, by the commutativity of pushouts (1) and (3) it follows that $m_1^2 \circ i_1^1 \circ l^1 = m_1^2 \circ l^2 \circ i_2^1$. By $m_1^2 \in \mathcal{M}$ being a monomorphism it follows that $i_1^1 \circ l^1 = l^2 \circ i_2^1$. Analogously, the commutativity of pushouts (2) and (4) implies that $i_3^1 \circ r^1 = r^2 \circ i_2^1$. Thus, $(p^1, m^1) \cong (p^2, m^2)$. □



□

Corollary 5.2.1 (Uniqueness of Creating Update Delta)**.** *By Prop. 5.2.1, it directly follows that given a model update u, then also the creating delta $\underline{\delta}_u$ of u is unique up to isomorphism.* △

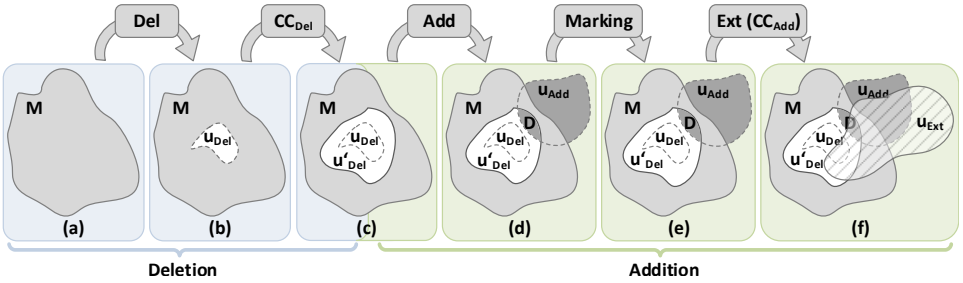


Figure 5.8: Propagation of Domain Model Updates: (a) - (f) illustrating sub-steps

5.3. Propagation Problem & Framework

The problem of propagating domain updates according to a given TGG, is to extend them such that they "fit" to the TGG. In our running example, this means that the target model update in the SPELL-Flow domain should also cover the change of the description of the : StepActivity in the second layer as demanded by triple rule T_Step_args-2-attrs. In addition, the deletion of : PauseActivity shall be reflected in the source domain, too, as demanded by triple rule T_Pause-2-PauseActivity. Both triple rules are provided in Ex. 4.1.6.

The extension is performed by applying propagation operations. The propagation framework according to a given TGG is given by two total and deterministic propagation operations, one for each domain of the TGG. An operation is total, if for each valid input it leads to a result. An operation is determistic, if it has functional behaviour, i.e., it terminates and leads to an unique result for each valid input, and the operation does not require backtracking.

We outline the sub-steps of the *derived propagation framework* of model updates $Ppg(TGG)$ in Fig. 5.8 that we will introduce in detail in the remainder of this chapter. Note, $Ppg(TGG) = Ppg(MM, u, TR)$, where MM is the triple meta-model, TR the set of triple rules and u describes the model update which shall be propagated. The general idea behind the derived propagation framework is: First, the framework gets a model M and a model update u: $M \xleftarrow{u_1} M_P \xrightarrow{u_2} M'$ as input. Then, the deleting part will be applied to model M and out of the result, a consistent model that includes thoses changes will be derived. Afterwards, the creating part of the update will be applied and then, all necessary elements will be added and recreated (out of M), that are necessary to finally derive a consistent model.

In the following, we summarise all steps of Fig. 5.8:

(a) This image illustrates the initial situation, i.e., triple graph M.

(b) The deletion part u_{Del} of the update u is executed on M, i.e., $M \setminus u_{Del}$.

(c) The deletion may affect more elements u'_{Del} so that a consistent intermediate model can be derived. Those elements will be deleted, too, i.e., $M \setminus u'_{Del}$.

(d) Creates all elements that are created by the model update, i.e., $(M \setminus u'_{Del}) \cup u_{Add}$. This step may recreate elements that were previously deleted (D).

(e) The model will be marked in order to guide the extension sub-step - the next step. (The marking is not explicitly visible, here. For details see Sec. 5.5.2.)

(f) In order to derive a consistent model, the model from (e) will be extended according to the TGG, i.e., $(M \setminus u'_{Del}) \cup u_{Add} \cup u_{Ext}$. Note, u_{Ext} is not part of the model update, anymore.

In detail, the formalisation of the derived propagation framework is summarised in Listing 5.2 and defined in Def. 5.6.10. Note, the derived propagation framework is based on the model synchronisation framework [HEEO12, HEO+15] that we introduced in Chap. 4. The propagation operation Ppg_{D_i} consists of two main steps. Fig. 5.9 formally illustrates both steps Del and Add of operation Ppg_{D_2} for propagating model updates in domain D_2. Operation Ppg_{D_1} for propagating model updates in domain D_1 is defined analogously. We will now summarise both main steps.

- **Step Deletion (Del):** Given a model M and a model update u, then the Del step deletes everything from M that is deleted by update u, first. Then, Del deletes everything that is related to the update in order to obtain a maximal consistent integrated sub-model (namely M') of M w.r.t. the given TGG. Thus, step Del propagates the deletion of elements along different views.

- **Step Addition (Add):** The Add step works with two models: Model M (no deletion performed) and M' (the result of the Del step). First of all, the Add step adds everything to M and M' that is created by update u. Then, markings are added to both models. Each element that is added by update u is marked with **F** (False). All other (untouched) elements are marked with **T** (True). Afterwards, in the Ext sub-step, two special kinds of triple rules, are iteratively applied to M' and M, in order to change the **F** markers to **T**. At the same time, elements may be reconstructed that were deleted during the Del step, but that are necessary to derive a consistent model. The special kinds of rules we use are called *shifted rules* and *consistency creating rules* (CC rules). The latter are a special kind of shifted rules, which only change markers. The iteration stops when no **F** marker is available anymore (successful), or when no shifted (or CC) rule could be found that is

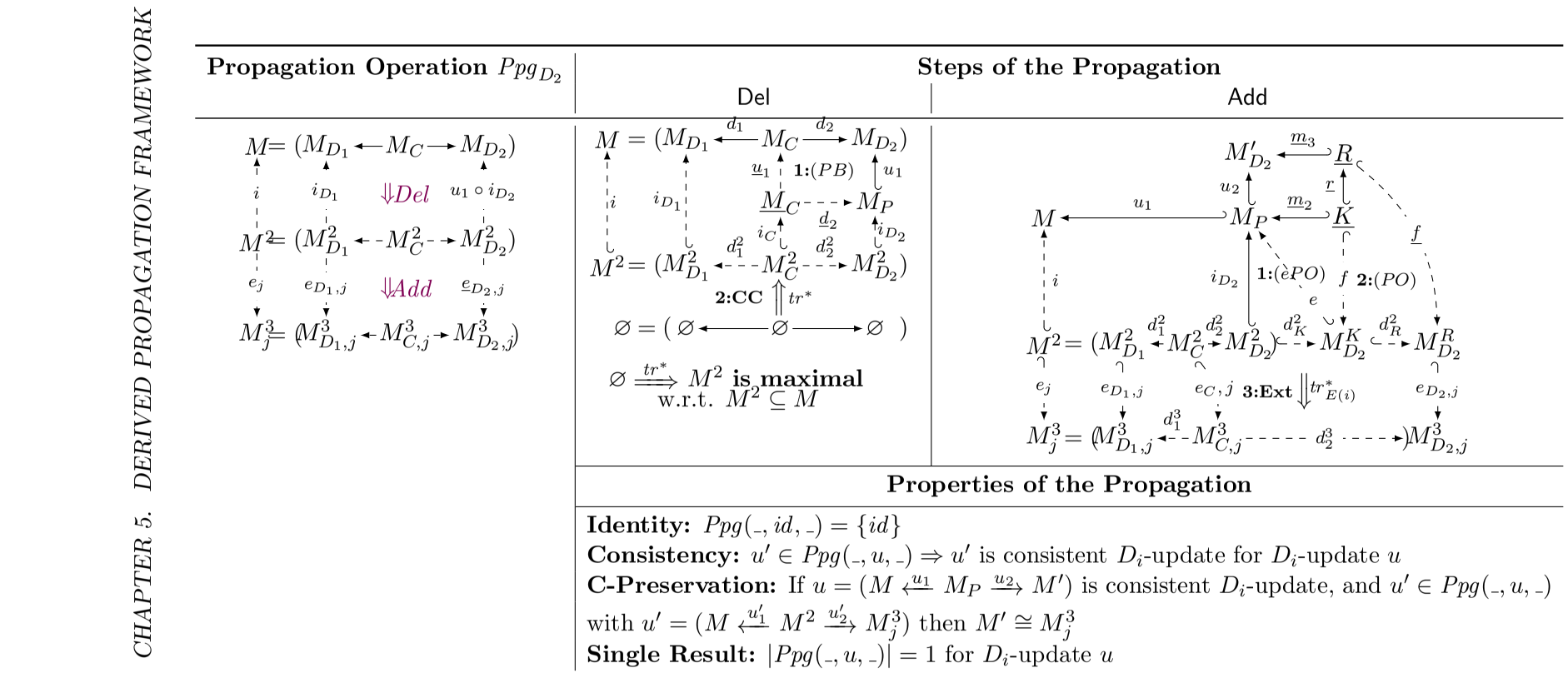


Figure 5.9: Propagation operation Ppg_{D_2} (left), Steps of the Propagation (right, top) and Properties of the Propagation (right, bottom)

applicable (abort). The derived propagation framework $Ppg(TGG)$ returns the modified model M'. If the derived propagation framework finishes successfully, then it returns a consistent model, otherwise the propagation aborts.

Definition 5.3.1 (Propagation Problem & Framework)**.** *Let* $TGG = (MM, \varnothing, TR)$ *be a triple graph grammar over triple type-graph* $MM = (MM_{D_1} \xleftarrow{s} MM_C \xrightarrow{t} MM_{D_2})$ *for domains* D_1, D_2*. A triple graph* $M = (M_{D_1} \leftarrow M_C \rightarrow M_{D_2}) \in \mathcal{L}(MM)$ *coincides with an update* $u\colon M \leftarrow M_P \rightarrow M' \in \Delta_{D_i}$*, if* $M = M_{D_i}$ *(*$i = 1, 2$*). The* D_i*-propagation problem is to construct an operation* $Ppg_{D_i}\colon UD \rightarrow \Delta_{D_i}$ *from tuples* $UD = \{(M, u, \delta_u) \mid u \in \Delta_{D_i}, M$ *coincides with* u, δ_u *is delta of* $u\}$ *of* D_i*-updates* Δ_{D_i} *on triple graphs with corresponding deltas to extended* D_i*-updates (*$i = 1, 2$*). The propagation framework* $Ppg(TGG) = (\Delta_{D_1}, \Delta_{D_2}, Ppg_{D_1}, Ppg_{D_2})$ *is given by all updates* $\Delta_{D_1}, \Delta_{D_2}$ *of domains* D_1 *and* D_2 *as well as total and deterministic propagation operations* Ppg_{D_1}, Ppg_{D_2} *for both domains.* △

Listing 5.2 describes the algorithm of the derived propagation framework by means of pseudocode. Both main steps Add and Del are described in greater detail in the following sections and in Listings 5.3 and 5.4.

```
1  /* --- Propagation Framework (Ppg) --- */
2  operation Ppg:
3
4  Input:
5    Given triple graph grammar TGG = (MM, ∅, TR) with
6      - triple type graph (meta model)
           MM = MM_D1 ↩ MM_C ↪ MM_D2
7      - a set of triple rules TR
8      - an empty start graph ∅
9    Given a triple graph M = M_D1 ↩ M_C ↪ M_D2
10   Given a set of domain updates Δ_Di (i ∈ {1,2}), containing
         updates u = (u1, u2) ∈ Δ_Di
11 Output:
12   consistent triple graph
13
14   foreach u ∈ Δ_Di
15     // Calculate update delta
16     δ_u ⇐ minimal rule out of TR restricted to all elements
           that are touched by u
17
18     // Calculate creating update delta
19     δ_u ⇐ δ_u restricted to creating elements only
20
21     // Apply both steps of Ppg
22     perform operation Del on M
23     perform operation Add on M
```

Listing 5.2: Propagation Framework - *Ppg* operation

5.4. Step Deletion (Del) (Sub-steps (a - c))

As illustrated in Fig. 5.8 (a+b), intuitively, given a model M and a model update u: $M \xleftarrow{u_1} M_P \xrightarrow{u_2} M'$, then step Del first deletes everything $u_{Del} = M \setminus u_1(M_P)$ from M that is deleted by u. Then, Del additionally deletes everything u'_{Del} from M that is related to u_{Del} in order to obtain a maximal consistent integrated sub-model $M \setminus u'_{Del}$ of M w.r.t. the given TGG. Thus, step Del propagates the deletion of elements along different views.

We will now present the algorithm of the Del step in Listing 5.3.

```
/* --- Deletion (Del) --- */
operation Del:

Input:
  Given triple graph M = M_D1 ↩ M_C ↪ M_D2
  // M before propagation: see Fig. 5.8 (a)
  Given a domain update u = (u1, u2) (i ∈ {1,2})
Output:
  triple graph (M²)

  // Perform deleting part of update
  // Scheme illustrating this step: Fig. 5.8 (b)
  M ⇐ M \ u_Del, i.e., delete everything that is deleted by u

  // We want to obtain a maximal consistent
  // sub-model of M
  // M² ⇐ M \ u'_Del, i.e., delete everything that
  // is interweaved with u and that is
  // inconsistent w.r.t. TGG
  // Scheme illustrating this step: Fig. 5.8 (c)
  M² ⇐ ∅
  while M² ⊆ M and a triple rule tr ∈ TR is applicable to
      M²:
    M² ⇐ apply tr ∈ TR to M²

  return triple graph M²
```

Listing 5.3: Deletion operation Del

In Fig. 5.9 (middle) the formal construction of step Del is visualised. Using this diagram, we are able to describe the technical details of the Del step in Rem. 5.4.1.

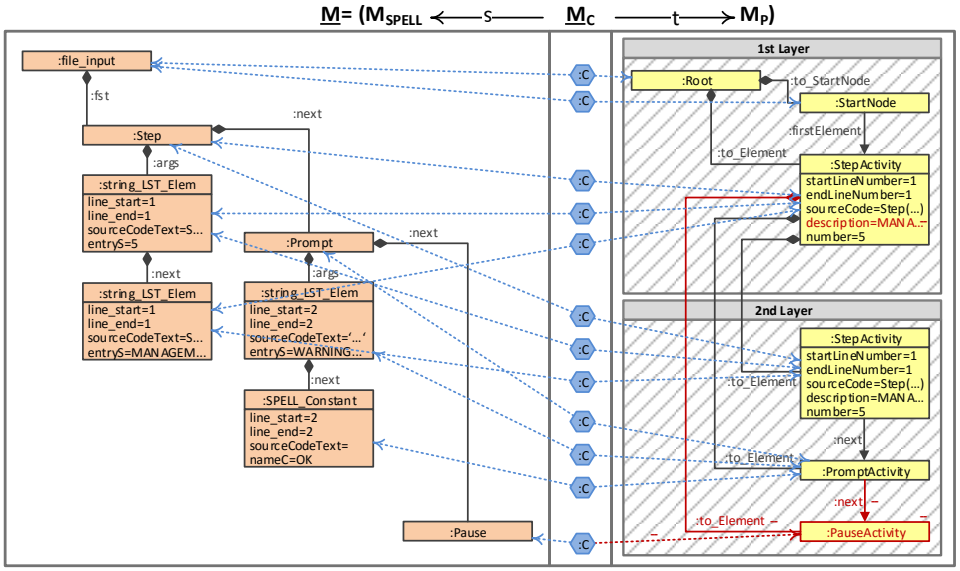


Figure 5.10: Intermediate model $\underline{M}$ of Del step

Remark 5.4.1 (Construction of Del Step)**.** *Technically, the* Del *step is identical to the* fAIn *and* Del *steps in the model synchronisation framework [HEO+15]. Given a D_2-update $u\colon M_{D_2} \xleftarrow{u_1} M_P \xrightarrow{u_2} M'_{D_2}$ applied to model $M = (M_{D_1} \xleftarrow{d_1} M_C \xrightarrow{d_2} M_{D_2})$, then model $\underline{M}_C$ is computed at first as intersection of M_C and M_P via common M_{D_2} by pull-back (PB) construction (cf. Fig. 5.9 (middle)). This leads to triple graph $\underline{M} = (M_{D_1} \xleftarrow{d_1 \circ \underline{u}_1} \underline{M}_C \xrightarrow{\underline{d}_2} M_P)$ where all correspondences have been deleted from M_C to $\underline{M}_C$ that are related to those elements $M_{D_2} \setminus u_1(M_P)$ that have been deleted by the update. The construction for D_1-updates is defined analogously. Afterwards, in the consistency creating (CC) step, the maximal consistent integrated sub-model $M^2 = (M^2_{D_1} \leftarrow M^2_C \rightarrow M^2_{D_2})$ of the possibly inconsistent model $\underline{M}$ is computed with inclusion $i\colon M^2 \rightarrow \underline{M}, i = (i_{D_1}\colon M^2_{D_1} \rightarrow M_{D_1}, i_C\colon M^2_C \rightarrow \underline{M}_C, i_{D_2}\colon M^2_{D_2} \rightarrow M_P)$ by step-wise application of triple rules $tr \in TR$ starting at the empty model $\varnothing$. Maximal means that there is no larger model M'^2 ($M^2 \subset M'^2$) that is also a consistent integrated sub-model of $\underline{M}$ ($M'^2 \subseteq \underline{M}$).* △

Scheme (c) in Fig. 5.8 illustrates the state of the propagation framework after performing the Del step. It is visible, that the Del step might delete more elements u'_{Del} from model M than indicated in model the update u_{Del}, i.e., $u_{Del} \subseteq u'_{Del}$.

Finally, we will show the application of the Del step by means of our running example in Ex. 5.4.1.

Example 5.4.1 (Del Step)**.** *Applying the SPELL-Flow domain model update*

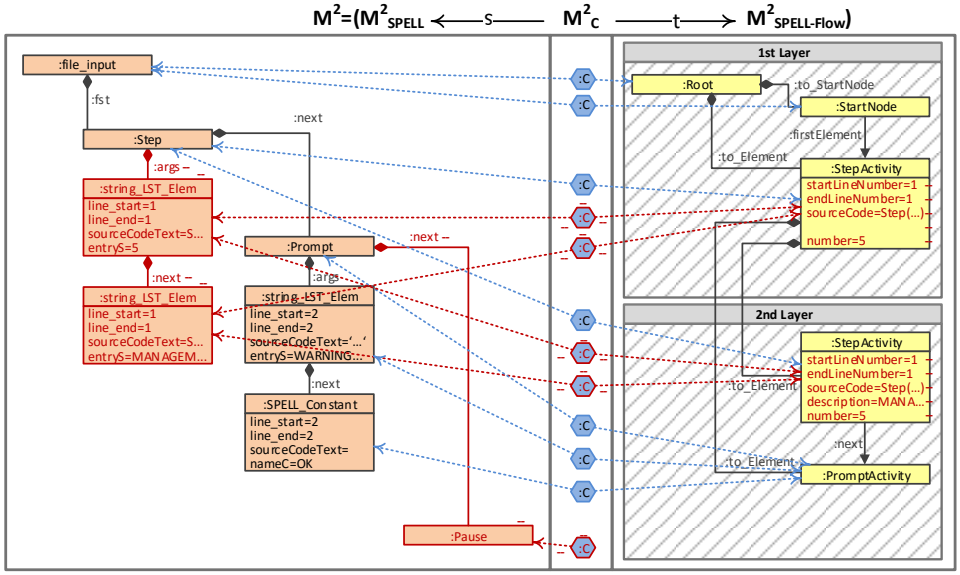


Figure 5.11: Resulting model M^2 of Del step M^2

in Ex. 5.2.1 and Fig. 5.7 (top) to model M Fig. 5.5 will delete the description *of node* : StepActivity *in the first layer. Moreover, it will delete the* : Pause *node in the second layer and all edges that are connected to this node, i.e.,* : to_Element, : next *and the edge from the correspondence node to this activity, because the gluing condition is violated, otherwise due to a dangling edge ([EEPT06], Def. 3.9 for graph structures; and Def. 6.3 for an abstract definition on categories). Finally, triple graph* $\underline{M} = (M_{SPELL} \leftarrow \underline{M_C} \rightarrow M_P)$ *reflects the effect of applying the deleting part of the SPELL-Flow domain model update u to the triple graph. We denoted the deleting part of u with* u_{Del} *in Sec. 5.3. Triple graph is not necessarily consistent w.r.t. the given TGG (*$\underline{M} \notin \mathcal{L}(TGG)$*). We visualise this intermediate model* $\underline{M}$ *in Fig. 5.10. Note, the graph in the SPELL domain stays unchanged (*M_{SPELL}*), whereas the target model and the correspondence model changed.*

Considering Fig. 5.9 (middle), then the second sub-step of Del *is to derive a consistent triple graph* $M^2 \subseteq I$ *which reflects* u'_{Del} *(cf. Sec. 5.3). That means,* u'_{Del} *includes* u_{Del} *but it might be extended by more elements, so that a consistent triple graph can be derived. In our running example, triple graph* M^2 *is illustrated in Fig. 5.11. Here,* $M^2 \subset I$ *because the two* : string_LST_Elem *nodes that are connected to the* : Step *node, all connected edges and the correspondence part that is directly connected to those nodes will be deleted, due to the deletion of the corresponding attributes in the SPELL-Flow domain, according to triple rules Fig. 4.8, Fig. 4.12 and Fig. 4.13. Furthermore, node* : PauseActivity *was deleted that leads to the deletion of the corresponding* : Pause *node, all connected edges and the*

corresponding elements in the correspondence domain according to triple rule Fig. 4.9. Model M^2 can be created by a terminating sequence of rule applications with the triple rules. Thus, the sub-model is consistent w.r.t. the given TGG ($M^2 \in \mathcal{L}(TGG)$). Note, only three triple rules out of the comprehensive set of all triple rules are given in Ex. 4.1.6. △

5.5. Step Addition (Add) (Sub-steps (d - f))

The result of Del serves as input for step Add. The Add step is illustrated intuitively in Fig. 5.8 (c-f), and the corresponding pseudo-code of the whole Add step is given in Listing 5.4.

The Add step is divided into three sub-steps: The first sub-step extends the model $\underline{M}$ (intermediate result from Del) and M^2 (result from Del) by the update delta $\underline{\delta_u}$ resulting in models M^R and M'.

Afterwards, in the second sub-step, the extended model M^R and M' will be marked via graph $\underline{M}$. This results in a model M^I that contains the model update u and all elements that were not deleted by the update, but, in general, this model is inconsistent.

In the third sub-step, we will continue working on model M'_{marked} and M^R_{marked}. Then, the framework iteratively extends both models according to the given set of triple rules TR in order to derive a consistent model out of M^R_{marked} satisfying the model update u and the corresponding triple graph grammar. The second graph which is based on M'_{marked} will be discarded. It was used as "helper" triple graph for deriving a consistent model. Note that it may occur, that no consistent model can be derived. Then, the propagation will abort returning an inconsistent model.

The following Listing 5.4 shows all sub-steps of the Add step in pseudo-code, which we will introduce in detail in the subsequent sub-sections.

```
1  /* --- Addition (Add) --- */
2  operation Add:
3
4  Input:
5    Given consistent integrated triple graph M^2 (result of
       Del)
6    // Scheme in Fig. 5.8 (c)
7    Given a domain update u = (u_1, u_2) (i ∈ {1,2})
8    Given update delta δ_u derived from u
9    Given creating delta δ_u underline derived from δ_u
10 Output:
11   (consistent) triple graph
12
13   // ----------------------
14   // --- Add sub-step 1 ---
15
```

```
// Add user update: first sub-step
M' ⇐ M² ∪ u_Add
// Add those elements to triple graph M² that are created
     by update u

M^K ⇐ (M²_D1, M²_C, M^K_D2), where M^K_D2 is obtained via effective
    pushout over M_P, M²_D2 and K
// cf. Fig. 5.9 (Step Add, 1:(ePO))
// For effective pushout: cf. ~Fig. 2.6

M^R ⇐ (M²_D1, M²_C, M^R_D2), where M^R_D2 is obtained via pushout
    over M^K_D2, R and K
// cf. Fig. 5.9 (Step Add, 2:(PO))
// Intuitively: in sets a pushout is a union

// --------------------------------
// --- Add sub-step 2: Marking ---

// Scheme illustrating this step in Fig. 5.8 (d).
// Part D contains those elements that were
// deleted previously but recreated by this
// step.

// Extend M^R by markings
M^R_marked ⇐ M^R

// e is an element in model M^R_marked,
// i.e., e is an edge, node or attribute
// if e has a making ma, then a tuple (e,ma)
// exists, otherwise ma = ∅, i.e., e has no
// second component
while ∃e ∈ M^R_marked without marking
  if ∃ morphism from an f ∈ M² to e ∈ M^R:
    ma ⇐ T
  else:
    ma ⇐ F
  // Add marking for e to M^R_marked
  replace e by tuple (e,ma) in M^R_marked

// The same for M'_marked
M'_marked ⇐ M'
while ∃e ∈ M'_marked without marking
  if ∃ morphism from an f ∈ M² to e ∈ M':
    ma ⇐ T
  else:
    ma ⇐ F
  replace e by tuple (e,ma) in M'_marked

// --------------------------------
```

```
// --- Add sub-step 3: Extension ---

// Calculate consistency creating rules
Calculate all CC rules out of TR

while elements in M^R_marked still have flag F and
      ∃r ∈ CC rules that is applicable to M^R_marked:

  // case 1: standard extension match
  sr ⇐ Calculate a maximal relevant shifted rule w.r.t.
      M^R_marked
  // if #1
  if sr ≠ ∅ and
     ∃ morphism from sr → M'_marked:
    M^R_marked ⇐ apply sr to M^R_marked
    M'_marked ⇐ apply sr to M'_marked
  else
    // case 2: guided extension match
    // if #2
    if sr ≠ ∅:
      sr_2 ⇐ calculate maximal relevant shifted rule sr_2 w.r
          .t. sr and M'_marked
      // if #3
      if sr_2 ≠ ∅:
        // For details cf. Rem. 5.6.6
        O^R_marked ⇐ construct ePO via M^R_marked, sr_2 and M'_marked
        M^R_marked ⇐ apply sr_2 to O^R_marked
        M'_marked ⇐ apply sr_2 to M'_marked
      else:
      // case 3: CC extension match
        for all CC rules:
          // if #4
          if ∃r ∈ CC rules that is applicable to M^R_marked:
            remember match m
            M^R_marked ⇐ apply r to M^R_marked
            M'_marked ⇐ apply r to M'_marked
          // end if #4
        // end for
      // end if #3
    // end if #2
  // end if #1

// end while

// The resulting triple graph is illustrated
// in Fig. 5.8 (e)
// Note: If the resulting triple graph M^R_marked
// still contains F markings, then we cannot
// derive a consistent model.
```

```
109  // Otherwise the model is consistent w.r.t.
110  // the given TGG.
111  return triple graph after Add (M^R_marked)
```

Listing 5.4: Addition operation Add

5.5.1. First Sub-Step of Add (Sub-step (d))

Given the maximal consistent integrated model $M^2 = M \setminus u'_{Del}$, from Del (cf. Fig. 5.11) and the creating update delta $\underline{\delta_u} = (\underline{p}, \underline{m})$ of update u with $\underline{p}$: $\underline{L} \leftarrow \underline{K} \rightarrow \underline{R}$ (cf. Fig. 5.7).

Then in a first sub-step, step Add adds those elements $u_{Add} = \underline{R} \setminus (M \setminus u'_{Del})$ to the model that are created by the update leading to model $M' = (M \setminus u'_{Del}) \cup u_{Add}$. Note that previously deleted elements $D = u_{Add} \cap u'_{Del}$ by Del may be recreated by this sub-step of Add. This sub-step of Add is illustrated in diagram Fig. 5.9 (d).

In detail, the first sub-step of Add consists of the following graph operations: Given the model M^2 that is obtained by the Del step in Ex. 5.4.1 and the update with corresponding creating update delta $\underline{\delta_u}$ from Fig. 5.7. Then, Add adds those elements to the model M^2 that are created by the update resulting in model M'. The diagram Fig. 5.9 (right) mentions explicitly model M'_{D_2}, i.e., in our running example this is the *SPELL-Flow* domain of triple graph M'. Implicitly, the whole model $M' = (M_{D_1} \leftarrow M_C \rightarrow M'_{D_2}))$ is available, where M'_{D_2} is resulting out of the application of update $u = (u_1, u_2)$. Model M^K is the triple graph $(M^2_{D_1}, M^2_C, M^K_{D_2})$, where graph $M^K_{D_2}$ is obtained via effective pushout using M_P (obtained in Del as part of $\underline{M}$), $M^2_{D_2}$ and $\underline{K}$. The latter is the graph that is part of $\underline{\delta_u} = (\underline{p}, \underline{m_1})$ with $\underline{p}$: $\underline{L} \leftarrow \underline{K} \rightarrow \underline{R}$. Afterwards, $M^R = (M^2_{D_1}, M^2_C, M^R_{D_2})$ is derived where $M^R_{D_2}$ results out of a pushout construction via $M^K_{D_2}$, $\underline{R}$ and $\underline{K}$.

Example 5.5.1 (Add Step - Sub-Step I)**.** *In Ex. 5.2.2 we already described the creating update delta, which creates the attribute* description *of node* : StepActivity *with new value* COMPONENT A *as well as attribute* comments *with new value* something *on the first layer of the model in the SPELL-Flow domain.*

First, the creating upadte delta $\underline{\delta_u} = (\underline{p}, \underline{m_1}$ *is applied to* $\underline{M} = (M_{SPELL} \leftarrow \underline{M_C} \rightarrow M_P)$ *resulting in* $M' = M_{SPELL} \leftarrow \underline{M_C} \rightarrow M'_{SPELL-Flow}$. *Note,* $\underline{p} = \underline{R} \leftarrow \underline{K} \rightarrow \underline{R}$, *i.e., it is the production which reflects the changes of the creating update delta. We illustrate triple graph* M' *in Fig. 5.12.* M' *differs from* $\underline{M}$ *in the additional attributes* description = COMPONENT A *and* comments = something *of node* : StepActivity *on the first layer, i.e., triple graph* M' *includes the changes of* $\underline{\delta_u}$.

The next triple graph which we derive in the first sub-step of Add *is* $M^K = (M^K_{SPELL} \leftarrow M^K_C \rightarrow M^K_{SPELL-Flow})$, *wheras* $M^K_{SPELL} = M^2_{SPELL}$ *and*

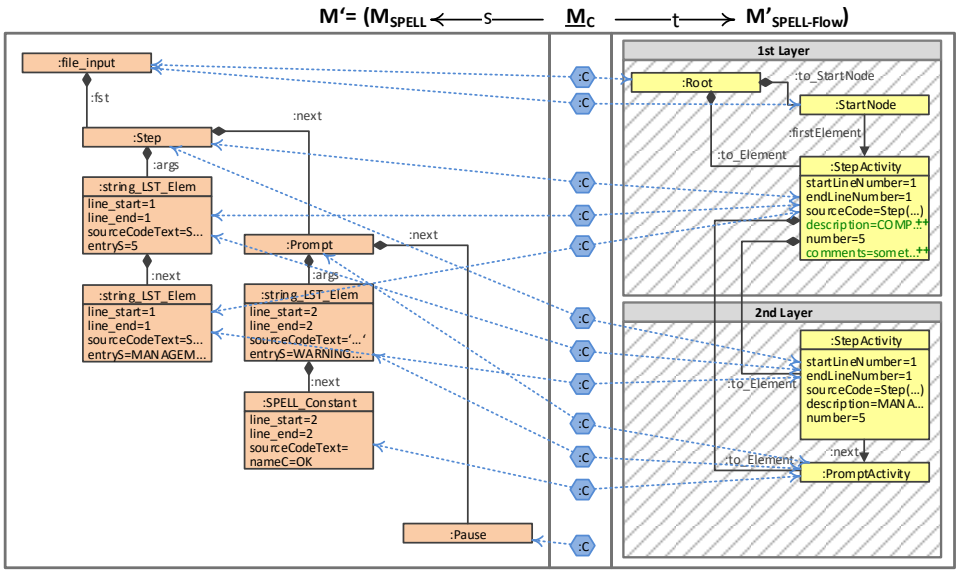


Figure 5.12: Triple Graph M'

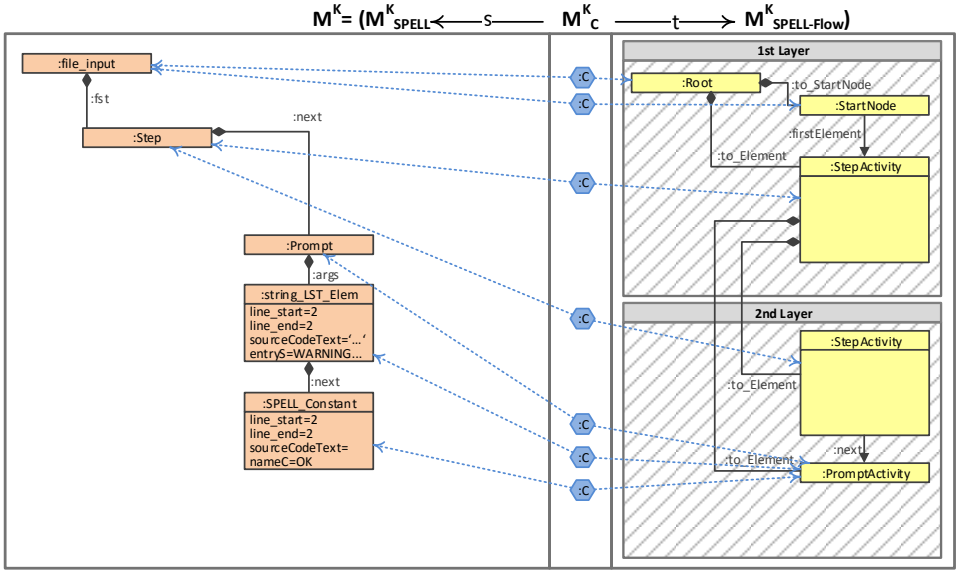


Figure 5.13: Triple Graph M^K

$M^K_C = M^2_C$. *The third part of this triple graph, i.e.,* $M^K_{SPELL-Flow}$*, is constructed via effective pushout using* $M^2_{SPELL-Flow}$*,* M_P *and* $\underline{K}$ *(cf. Fig. 5.9). In the given running example,* $M^K_{SPELL-Flow}$ *equals* $M^2_{SPELL-Flow}$*, but this is not the general case. Thus, in the current example,* $M^K = M^2$*. We illustrate model* M^K *in Fig. 5.13. (In the running example which we use in [GNE+16a],* M^K_{D2} *differs from* M^2_{D2} *and therefore,* $M^K \neq M^2$*.)*

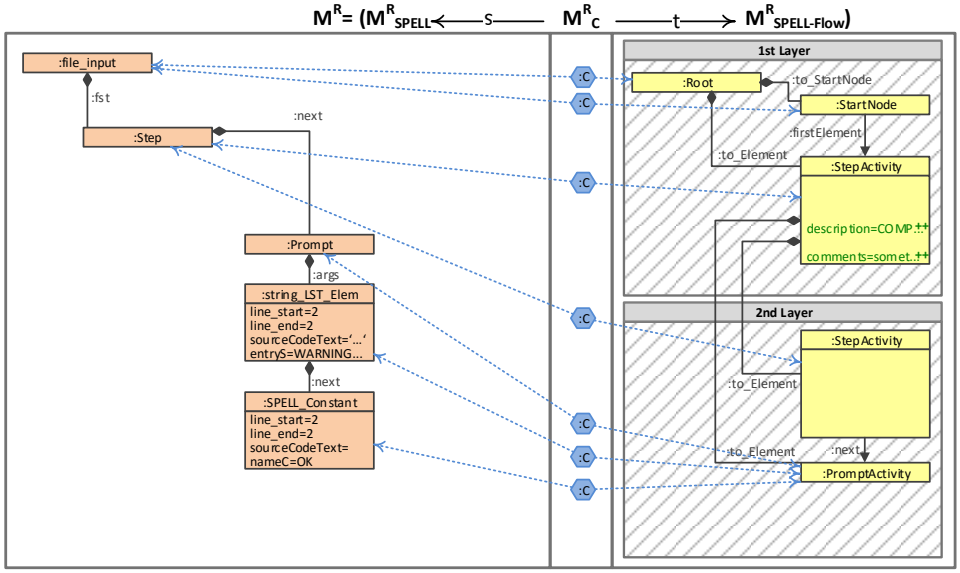
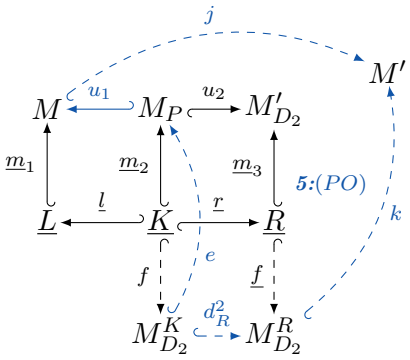


Figure 5.14: Triple Graph M^R

The last model which is calculated in the current sub-step of Add *is model* $M^R = (M^R_{SPELL} \leftarrow M^R_C \rightarrow M^R_{SPELL-Flow})$. *Again,* M^R_{SPELL} *equals* M^2_{SPELL} *and* M^R_C *equals* M^2_C. *Component* $M^R_{SPELL_Flow}$ *is constructed as pushout via graphs* $M^K_{SPELL-Flow}$, $\underline{K}$ *and* $\underline{R}$. *The resulting model is shown in Fig. 5.14.* $M^R_{Spell-Flow}$ *is* $M^K_{SPELL-Flow}$ *extended by attributes* description = COMPONENT A *and* comments = something *within node* : StepActivity *which is contained in the first layer, i.e.,* $M^K_{SPELL-Flow} \subseteq M^R_{SPELL-Flow}$. △

Remark 5.5.1 (Add Step - Sub-Step I)**.** *Technically, the first sub-step of* Add *is defined as follows. Given the consistent integrated model* $M^2 = (M^2_{D_1} \xleftarrow{d^2_1} M^2_C \xrightarrow{d^2_2} M^2_{D_2})$ *from* Del *with inclusion* $i_{D_2}: M^2_{D_2} \rightarrow M_P$, *the* D_2*-update* $u: M_{D_2} \xleftarrow{u_1} M_P \xrightarrow{u_2} M'_{D_2}$ *(cf. Rem. 5.4.1) and the corresponding creating delta* $\underline{\delta}_u = (\underline{p}, \underline{m}_1)$ *with* $\underline{p}: \underline{L} \xleftarrow{\underline{l}} \underline{K} \xrightarrow{\underline{r}} \underline{R}$ *and induced morphisms* $\underline{m}_2: \underline{K} \rightarrow M_P$ *and* $\underline{m}_3: \underline{R} \rightarrow M'_{D_2}$ *(cf. Rem. 5.2.1).*



In a first sub-step, those elements are added to M^2 that are created by update u. As, the preserved elements of the update delta in $\underline{K}$ may be deleted by the previous Del *step, all deleted elements of $\underline{K}$ are recreated at first. This is done by an effective pushout (ePO) construction over $\underline{m}_2, i_{D_2}$ leading to model $M^K_{D_2}$ with morphisms f, d^2_K and induced morphism $e\colon M^K_{D_2} \to M_P \in \mathcal{M}$ (cf. diagram above that is taken from cf. Fig. 5.9 (right) and extended).*

Afterwards, the remaining elements of the delta that are created by the update are added. This is done by a pushout (PO) construction over $\underline{r}, f$ leading to model $M^R_{D_2}$ with morphisms $d^2_R, \underline{f}$. This results in triple graph $M^R = (M^R_{D_1} \leftarrow M^R_C \rightarrow M^R_{D_2})$ with $M^R_{D_1} = M^2_{D_1}$ and $M^R_C = M^2_C$.

The construction for D_1-updates is defined analogously. Note that we explicitly use the creating delta of update u and not the delta with deleted elements in order to avoid that elements are added in this first sub-step that only refer to deleted elements of u.

The application of the creation delta extension $\underline{\delta}_u$ to the "original" model M is performed via PO (3) and (4), i.e., the rule $\underline{p}$ is applied to M via match $\underline{m}_1$ resulting in M'. The details of the application of $\underline{p}$ to M via match $\underline{m}_1$ is depicted in the diagram below. The diagram is an excerpt from the diagram above.

$$
\begin{array}{ccccc}
M' = (M'_{D_1} & \xleftarrow{d'_1} & M'_C & \xrightarrow{d'_2} & M'_{D_2}) \xleftarrow{u_1 \circ m_3} \underline{R} \\
\uparrow j & & & & \uparrow (4) \uparrow \underline{r} \\
 & & & & M^K_{D_2} \xleftarrow{u_1 \circ m_2} \underline{K} \\
 & & & & \uparrow (3) \uparrow \underline{l} \\
M = (M_{D_1} & \xleftarrow{d_1} & M_C & \xrightarrow{d_2} & M_{D_2}) \xleftarrow{\underline{m}_1} \underline{L}
\end{array}
$$

If we consider the first diagram, again. 5 : PO *(also marked in blue) can be constructed via morphisms: $e \circ u_1 \circ j = d^2_R \circ k$ with interface graph $M^K_{D_2}$, whereas morphisms j and k are $\mathcal{M}$ morphisms (due to unchanged graphs in the correspondence and D_1 domain in both triple graphs M and M^2). We will reuse this property in Prop. 5.5.1.* △

5.5.2. Second Sub-Step of Add: Marking (Sub-step (e))

The second sub-step of Add marks the relevant models M^R and M' via interface triple graph M^2. In both models, all elements from u_{Add} are marked with **F** (False - not extended). All other elements are marked with **T** (True - extended). In the following, we define the marking operation *mark*.

Definition 5.5.1 (Markings)**.** *Given models M, M' and I and morphisms $i : I \to M$ and $i' : I \to M'$. The marking operation is defined $mark(X) = X \oplus Att^{\mathbf{F}}_{X \setminus x(I)} \oplus Att^{\mathbf{T}}_{x(I)}$, where $(X, x) \in \{(M, i)(M', i')\}$. Technically, the marking is performed via PO construction, as defined in Def. 7.26 in [EEGH15].* △

$$\begin{array}{ccc} & & M \oplus Att^{\mathbf{F}}_{M \setminus i(I)} \oplus Att^{\mathbf{T}}_{i(I)} \\ & \overset{i}{\nearrow} & \overset{mark(M)}{\dashrightarrow} \\ I & & \\ & \underset{i'}{\searrow} & \underset{mark(M')}{\dashrightarrow} \\ & & M' \oplus Att^{\mathbf{F}}_{M' \setminus i'(I)} \oplus Att^{\mathbf{T}}_{i'(I)} \end{array}$$

Note, that the interface graph I is given by M^2 in the right diagram of Add in Fig. 5.9. Furthermore, models M^R and M' will be marked. Both are constructed in the first sub-step of Add. The resulting models of the second Add sub-step are:

- $M'_{marked} = M' \oplus Att^{\mathbf{F}}_{M' \setminus (u_2 \circ i)(M^2)} \oplus Att^{\mathbf{T}}_{(u_2 \circ i)(M^2)}$
- $M^R_{marked} = M^R \oplus Att^{\mathbf{F}}_{M^R \setminus (d_R \circ d_K)(M^2)} \oplus Att^{\mathbf{T}}_{(d_R \circ d_K)(M^2)}$ with $d_R : M^K \to M^R$ and $d_K : M^2 \to M^K$

Example 5.5.2 (Sub-step 2: Markings)**.** *Let us consider Fig. 5.15. Triple graphs M' and M^R are marked via interface graph M^2 that is illustrated on the top.*

*During marking, graph M' is enriched by the following markers: all elements that can be mapped by M^2 are marked with **T** all other elments are marked with **F**. We denote the marked triple graph with M'_{marked} (cf. Fig. 5.15 (middle)). In detail, the following nodes, edges and attributes are marked with **T** in the SPELL domain:*

- Nodes: : file_input, : Step, : Prompt, : string_LST_Elem *which is directly connected to* : Prompt *and* : SPELL_Constant.
- Attributes: *All attributes of the nodes we mentioned in the item before are marked with* **T**.
- Edges: : fst, *the* : next *edge between nodes* : Step *and* : Prompt, *and the* : next *edge between nodes* : string_LST_Elem *and* : SPELL_Constant, *and edge* : args *which has node* : Prompt *as source node.*

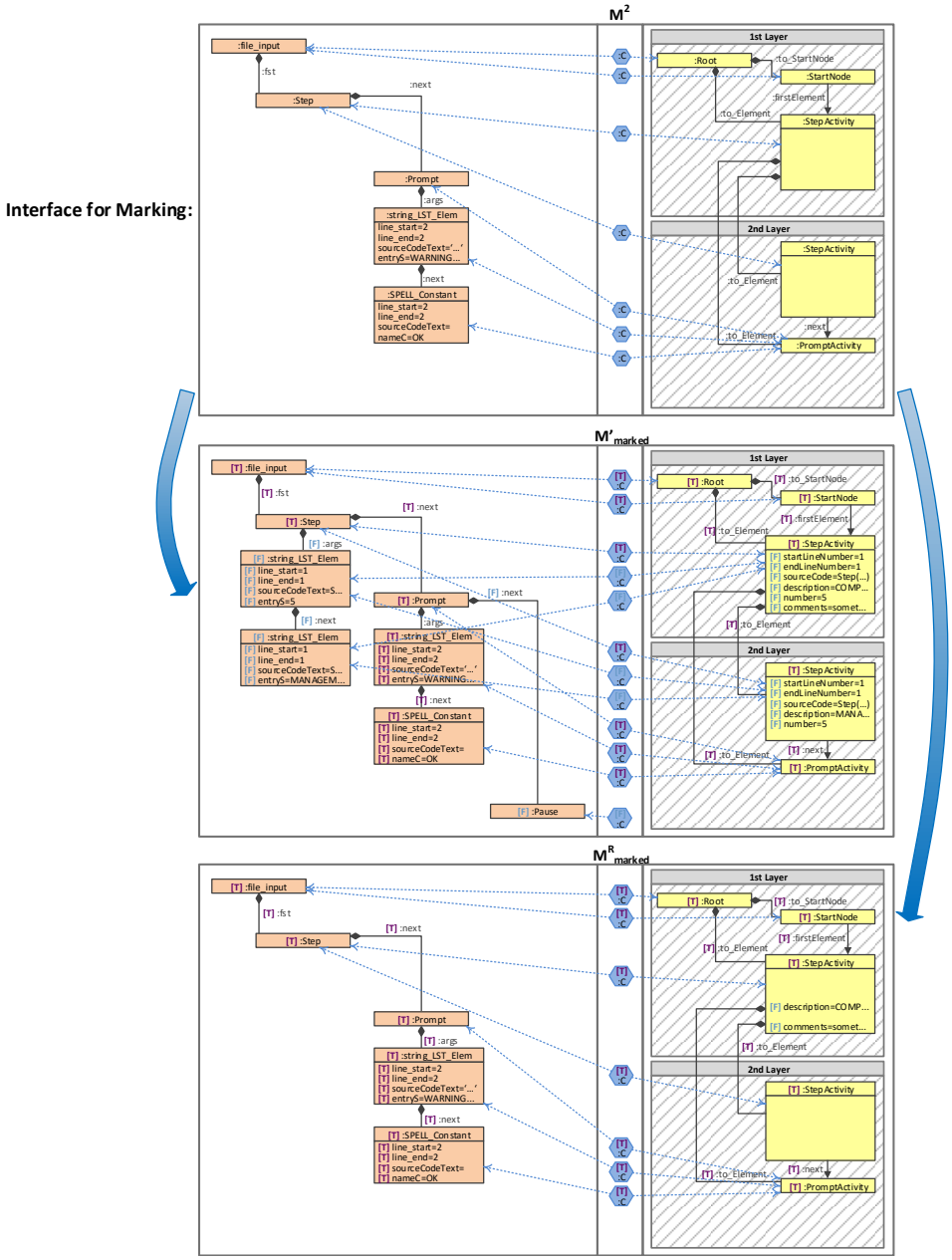


Figure 5.15: Models M' and M^R are marked resulting in M'_{marked} (middle) and M^R_{marked} (bottom) via Interface Triple Graph M^2 (top)

In the SPELL-Flow domain, the following elements are marked with **T***:*

- Nodes: *All nodes on both layers are marked with* **T**.
- Attributes: *No attributes are marked with* **T**.

- Edges: *All edges in all layers.*

In the correspondence domain, seven correspondence nodes and the corresponding edges get marker **T**. *Those nodes point to those nodes in the SPELL domain (and also SPELL-Flow domain) that get marker* **T**, *too. For better readability, we omitted the markers for all edges which have a correspondence node in their source within the visualisation provided in Fig. 5.15. Those edges are marked with the same marker as their source node. All other elements of* M' *are marked with* **F**.

Triple graph M^R *is enriched by the* **T** *and* **F** *markers accordingly (cf. Fig. 5.15 (bottom)). In detail: All elements in all domains of the triple graph* M^R_{marked} *are marked with* **T** *except attributes* description *and* comments *contained in node* : StepActivity *on the first layer of the SPELL-Flow model. Those attributes get marker* **F**. △

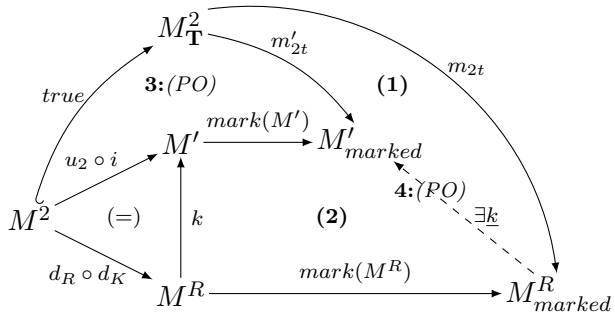
Both triple graphs that were marked are required for the third sub-step of Add: The extension step Ext. Still, we need a morphisms from M^R_{marked} to M'_{marked}. This is achieved with the aid of $\mathcal{M}$-morphism $k : M^R \to M'$ resulting out of PO (5) in Rem. 5.5.1. In the following propsition, we will review the details.

Proposition 5.5.1 (Morphism Between Marked Triple Graphs)**.** *Given triple graphs* M' *and* M^R *with* $\mathcal{M}$*-morphism* $k : M^R \to M'$. *Furthermore, given marked triple graphs* M'_{marked} *and* M^R_{marked} *obtained via marking operation mark defined in Def. 5.5.1. Then, the* $\mathcal{M}$*-morphism* $\underline{k} : M^R_{marked} \to M'_{marked}$ *exists, too.*

$$\begin{array}{ccc} M' & \xrightarrow{mark(M')} & M'_{marked} \\ \uparrow{\scriptstyle k} & & \uparrow{\scriptstyle \underline{k}} \\ M^R & \xrightarrow{mark(M^R)} & M^R_{marked} \end{array}$$

△

Proof. The $\mathcal{M}$-morphism $k : M^R \to M'$ is given (see PO (5) in Rem. 5.5.1) and a commuting triangle (=) with $k \circ d_R \circ d_K = u_2 \circ i$. According to Def. 7.26 in [EEGH15], the marking of graphs is technically defined using pushouts. In the diagram below, the marking of M' is constructed via PO (1) and the marking of M^R via PO (2). Triple graph $M^2_{\mathbf{T}}$ is triple graph M^2 extended by markers **T** for each element, i.e., $true : M^2 \to M^2_{\mathbf{T}}$.



By construction of the marking according to Def. 7.26 in [EEGH15], the following two POs exist:

- $\exists$ PO $(m'_{2T}, mark(M'))$ over $(true, u_2 \circ i)$ with PO object M'_{marked} (in the diagram, this PO is marked with (3))
- $\exists$ PO $(m_{2T}, mark(M^R))$ over $(true, d_R \circ d_K)$ with PO object M^R_{marked} (in the diagram, this PO is marked with (4))

Due to the universal property of PO (4), a morphism $\underline{k} : M^R_{marked} \to M'_{marked}$ exists with commuting (1) and (2). □ □

5.6. Third Sub-Step of Add: Extension (Ext) (Sub-step (f))

The third sub-step Ext extends the given domain update u such that the update fits to the given TGG. Therefore, Ext takes models M^R_{marked} and M'_{marked} from the second sub-step of Add as input. Note, triple graph M'_{marked} reflects the user update. In contrast, the unmarked triple graph M^R that corresponds to M^R_{marked} is consistent w.r.t. the given TGG.

Sub-step Ext analyses the applicability of the triple rules $tr \in TR$ from the given TGG to M^R_{marked} such that each application maximally overlaps with that part of M^R_{marked} which is marked with **F**. From each overlapping triple rule that is applicable via a match, an operational rule, called shifted rule sr_1, with extended match is derived which creates only those elements of the triple rule that do not overlap. Afterwards, Ext extend the maximal overlapping to M'_{marked} and a second shifted rule sr_2 is derived, in order to recreate necessary elements. With the help of the shifted rule sr_2 and models M'_{marked} and M^R_{marked} an intermediate model $\underline{M}^R_{marked}$ is derived. Finally, sr_2 will be applied to $\underline{M}^R_{marked}$ and also to M'_{marked} leading to models $M^{R'}_{marked}$ and M''_{marked}, where the markings of overlapping elements will be changed from **F** to **T** ($[\mathbf{F} > \mathbf{T}]$). Furthermore, shifted rules create missing elements. (For details, see Ex. 5.6.2).

If such a shifted rule cannot be found, then the Ext step checks for complete overlappings, i.e., no elements will be created. In that case, a special kind of operational rule is applied which is called consistency creating

rule (CC rule) (cf. Def. 7.44 in [EEGH15]). Those rules only change markers from **F** to **T**.

Applications with empty overlappings are omitted unless they completely overlap with previously deleted elements $D \subseteq M \setminus M^R$. If the overlapping contains previously deleted elements, then the application is additionally maximally overlapped with previously deleted elements $D \subseteq M \setminus M^R_{marked}$ in order to guide the search for applicable matches.

Example 5.6.1 (Triple Graph D)**.** *Triple graph D of our running example is given in the diagram below (cf. Fig. 5.16). It represents all elements which were deleted by the domain model update u. Note, in our example, D contains also node* : StepActivity *even if it is not deleted by u. It is contained in D, because the attributes of that node which were deleted, cannot be part of a graph without a containing node. The same holds for nodes* : Prompt, : PromptActivity *and* : Step *which need to be part of D because of dangling edges, otherwise.* △

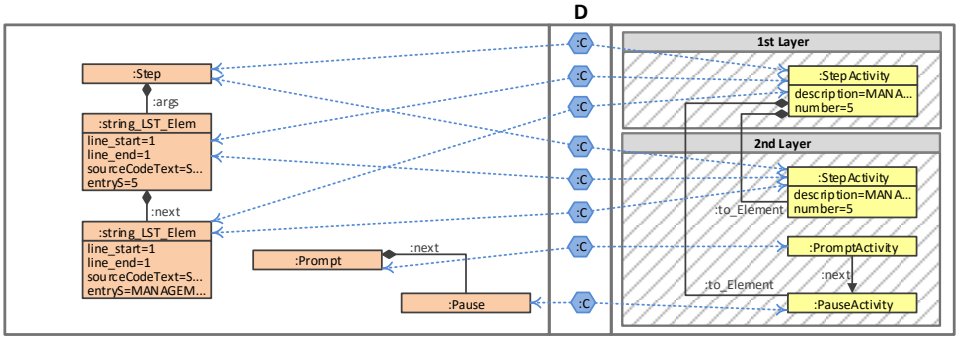


Figure 5.16: Model $D \subseteq M \setminus M^R$

Models $M^{R'}_{marked}$ and M''_{marked} are taken as input for sub-step Ext again, as long as all elements are not marked with **T** or no rules are applicable any more. In Sec. 5.6.1, we illustrate the Add step in great detail by means of our running example.

Note that previously deleted elements by Del are not explicitly recreated as part of the matches itself. Previously deleted elements are only used to guide the search for matches or to find complete overlappings and therefore, may be recreated in the extension process. As, in all sub-steps of Add, previously deleted elements may be recreated, the proposed propagation framework prioritises creation over deletion. In contrast to propagating the deletion of elements by Del, step Add propagates the creation of elements along different views.

For defining the Ext sub-step of Add we introduce the notion of shifted rules as operational rules. Based on the application of shifted rules the notions of the update extension step and the update extension sequence

are defined. Given a triple rule $tr : L \to R$, then a shifted rule of tr is a rule $tr_d\colon \underline{L} \leftarrow \underline{K} \to \underline{R}$ where $\underline{L}$ not only contains all elements of L but also elements of R. $\underline{R}$ contains the remaining elements of R that are not contained in $\underline{L}$. Therefore, a shifted rule is derived from a triple rule by shifting elements from R to L resulting in new left- and right-hand sides $\underline{L}$ and $\underline{R}$. All elements of L in $\underline{L}$ and all elements of $\underline{R}$ are marked with $\mathbf{T}$ while the markings of the shifted elements is changed from $\mathbf{F}$ to $\mathbf{T}$, i.e., $\underline{K}$ is $\underline{L}$ without F-markings. Thus, in contrast to the underlying triple rule $tr\colon L \to R$, the match from a shifted rule of tr to a model not only involves elements of L but also the shifted elements of R (extended match) while only the remaining elements of R that were not shifted are created by applying the shifted rule. The successive application of shifted rules extends an update step-wise in the sense that the elements which are created by the update are complemented by those elements of the underlying triple rules which do not overlap with the update and therefore, could not be shifted but would also be created when applying the triple rules. The markings $\mathbf{T}$ and $\mathbf{F}$ are introduced in order to control which part (elements) of an update already have been extended. Technically, the shifting of elements in a triple rule $tr\colon L \to R$ is controlled by a decomposition of morphism tr. As, in the example of attributed graphs we are only interested in shifting structural graph elements without altering data elements, we restrict the decompositions from general decompositions to $\mathcal{M}$-decompositions only.

Definition 5.6.1 ($\mathcal{M}$-decomposition)**.** *An $\mathcal{M}$-decomposition d of an $\mathcal{M}$-morphism $tr\colon L \to R$, in short tr-decomposition, consists of two $\mathcal{M}$-morphisms $d\colon L \xrightarrow{tr_1} L' \xrightarrow{tr_2} R$ with $tr_2 \circ tr_1 = tr$ and $tr_1, tr_2 \in \mathcal{M}$.* △

In the following, with decompositions we implicitly refer to $\mathcal{M}$-decompositions. Based on $\mathcal{M}$-decompositions, shifted rules are defined.

Definition 5.6.2 (Shifted Rule)**.** *Let $(tr\colon L \to R, ac_L)$ be a triple rule tr with application condition ac_L and $d\colon L \xrightarrow{tr_1} L' \xrightarrow{tr_2} R$ be a tr-decomposition. The* shifted rule *$tr_d = (\underline{tr}_d, ac_{\underline{L}})$ of tr w.r.t. d is given by rule $\underline{tr}_d\colon \underline{L} \xleftarrow{l} \underline{K} \xrightarrow{r} \underline{R}$ with $\underline{L} = L' \oplus Att_L^{\mathbf{T}} \oplus Att_{L'\setminus L}^{\mathbf{F}}$, $\underline{K} = L' \oplus Att_L^{\mathbf{T}}$, $\underline{R} = R \oplus Att_R^{\mathbf{T}}$ and application condition $ac_{\underline{L}} = tExt(ac_L, \underline{L}, \{D_1, C, D_2\})$. The morphisms $\underline{l}, \underline{r} \in \mathcal{M}$ are induced by morphism $tr_2 \in \mathcal{M}$.* △

Similar to all kinds of operational rules [EEGH15], shifted rules (and their application conditions (ac)) are extended by markings, too. The following Rem. 5.6.1 describes the technical details.

Remark 5.6.1 (Marking of Shifted Rules)**.** *Given a graph L' with sub-graph $L \subseteq L'$, then $L' \oplus Att_L^v$ is L' but extended by a marking $v \in \{\mathbf{T}, \mathbf{F}\}$ for each element of L in L'. For technical details we refer to Def. 7.26 in [EEGH15]. By $tExt(ac_L, \underline{L}, \{D_1, C, D_2\})$ application condition ac_L is extended by the*

additional elements and markings of $\underline{L}$ *via morphism* $L \xrightarrow{tr_1} L' \xrightarrow{m} \underline{L}$ *where morphism* m *is derived by the marking of* L'*. Furthermore, all additional elements in* ac_L *that are not in* L *are all marked with* **T***. The marking considers elements of both domains* D_1 *and* D_2 *as well as the correspondence part* C*. For technical details we refer to Def. 7.28 in [EEGH15].* △

Shifted rules are a generalisation of consistency creating (CC) rules. CC rules were defined in the theory of model synchronisations to mark consistent integrated sub-models of possibly inconsistent models [HEO+15]. Given a triple rule $tr\colon L \to R$, the CC rule of tr is derived by shifting all elements from R to L. CC rules are used for applications (update extensions) based on complete overlappings of the underlying triple rule with previously deleted elements, i.e., CC rules only change markings from **F** (not translated) to **T** (translated).

Definition 5.6.3 (Consistency Creating (CC) Rule)**.** *Let* $(tr\colon L \to R, ac_L)$ *be a triple rule* tr *with application condition* ac_L*. The* consistency creating rule tr_{CC} *of* tr *is given by the shifted rule of* tr *w.r.t. decomposition* $d\colon L \xrightarrow{tr} R \xrightarrow{id} R$*. Given a set of triple rules* TR*, with* TR_{CC} *we denote the set of all consistency creating rules of* TR*.* △

For update extensions, only shifted rules are of relevance where at least one element is shifted. This omits applications of shifted rules which do not overlap with the elements that are created by the update and therefore are of no importance concerning the update.

Definition 5.6.4 (Relevant Shifted Rule)**.** *Given* tr_d *being a shifted rule w.r.t. decomposition* $d\colon L \xrightarrow{tr_1} L' \xrightarrow{tr_2} R$*. Then,* tr_d *is* relevant*, if* $L \ncong L'$*.* △

Furthermore, shifted rules have to be maximal in the sense that they should shift and match as many elements as possible such that the overlappings are maximal and only the non-existing (non-overlapping) elements of the underlying triple rules are created by their application.

Definition 5.6.5 (Maximal Shifted (Applicable) Rule)**.** *Given a model* M*, a triple rule* $(tr\colon L \to R, ac_L)$*, a* tr*-decomposition* $d\colon L \xrightarrow{tr_1} L' \xrightarrow{tr_2} R$ *and the corresponding shifted rule* $tr_d = (\underline{tr}_d\colon \underline{L} \xleftarrow{l} \underline{K} \xrightarrow{r} \underline{R}, ac_{\underline{L}})$*. Then,* tr_D *is* maximal shifted *w.r.t.* M*, if there exists a match* $m\colon \underline{L} \to M$ *and there does not exist a* tr_2*-decomposition* $L' \xrightarrow{tr_2'} L'' \xrightarrow{tr_2''} R$ *with* $d'\colon L \xrightarrow{tr_2' \circ tr_1} L'' \xrightarrow{tr_2''} R$*, and relevant shifted rule* $tr_{d'} = (\underline{tr}_{d'}\colon \underline{L'} \leftarrow \underline{K'} \to \underline{R}, ac_{\underline{L'}})$ *w.r.t.* d' *and match* $m'\colon \underline{L'} \to M$*. Rule* tr_D *is applicable, if additionally, matches* m *and* m' *are applicable.* △

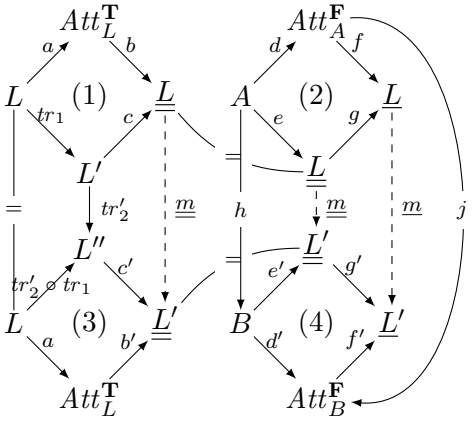
Remark 5.6.2 (Relevant Maximal Shifted Rule & CC Rule)**.** *The maximal shifted rule of* tr *w.r.t. decomposition* $d\colon L \xrightarrow{tr} R \xrightarrow{id} R$ *and some model* M *is*

the CC rule of tr. Conversely, assuming a non-trivial triple rule $tr\colon L \to R$ such that tr creates at least one element (tr is not an epimorphism). Then, the CC rule tr_{CC} of tr with a match m from tr_{CC} to some model M is a relevant maximal shifted rule w.r.t. M, as, tr_{CC} has already shifted all elements of tr and thus, no further elements can be shifted such that a new "extended" match is obtained. △

Given a triple rule tr, then maximal shifted applicable rules of tr w.r.t. some model M are maximally shifted concerning rules that are only applicable to M. This is defined by the fact that not only match m but also match m' in Def. 5.6.5 needs to be an applicable match. Thus, there may exist (maximal) shifted rules of tr (w.r.t. M) with more shifted elements but each of these rules is not applicable to M, since, either there does not exist a match from the rule to M or the match does not satisfy the application condition of the rule. Conversely, given a shifted rule of tr which is applicable to M, then also all shifted rules of tr with less shifted elements are applicable to M as shown in Prop. 5.6.1.

Proposition 5.6.1 (Applicability of Shifted Rules)**.** *Given a model M, a triple rule $tr = (tr\colon L \to R, ac_L)$, a tr-decomposition $d\colon L \xrightarrow{tr_1} L' \xrightarrow{tr_2} R$, a tr_2-decomposition $L' \xrightarrow{tr_2'} L'' \xrightarrow{tr_2''} R$ with $d'\colon L \xrightarrow{tr_2' \circ tr_1} L'' \xrightarrow{tr_2''} R$ and shifted rules $tr_d = (\underline{tr}_d\colon \underline{L} \xleftarrow{l} \underline{K} \xrightarrow{r} \underline{R}, ac_{\underline{L}})$ and $tr_{d'} = (\underline{tr}_{d'}\colon \underline{L}' \xleftarrow{l'} \underline{K}' \xrightarrow{r'} \underline{R}, ac_{\underline{L}'})$ in an $\mathcal{M}$-adhesive category with match $m'\colon \underline{L}' \to M$. If rule $tr_{d'}$ is applicable via m', then there exists a match $m\colon \underline{L} \to M$ such that tr_d is applicable via m.* △

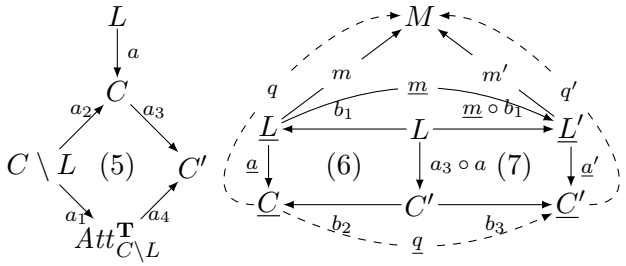
Proof. Let m' be an applicable match, i.e., m' satisfies the gluing condition w.r.t. $tr_{d'}$ and $m' \models ac_{\underline{L}'}$. By the construction of shifted rules, there exists a morphism $\underline{m}\colon \underline{L} \to \underline{L}'$ and match $m = m' \circ \underline{m}, m\colon \underline{L} \to M$ $^{(*1)}$ as shown in the following.



$A = c(L' \setminus tr_1(L))$
$B = c'(L'' \setminus tr_2'(tr_1(L)))$

Models $\underline{L}$ and $\underline{L}'$ are constructed by pushouts (1), (2) and (3), (4) (cf. Def. 5.6.2 and Rem. 5.6.1). By the universal pushout property of (1) and

$b' \circ a = c' \circ tr'_2 \circ tr_1$ (by pushout (3)) it follows that there exists morphism $\underline{\underline{m}}\colon \underline{\underline{L}} \to \underline{\underline{L}}'$. Morphism $\underline{\underline{m}}$ induces morphisms $h\colon A \to B$ and $j\colon Att_A^{\mathbf{F}} \to Att_B^{\mathbf{F}}$ with $e' \circ h = \underline{\underline{m}} \circ e$ $^{(*^2)}$ and $j \circ d = d' \circ h$ $^{(*^3)}$. By $f' \circ j \circ d \overset{(*^3)}{=} f' \circ d' \circ h \overset{PO\,(4)}{=} g' \circ e' \circ h \overset{(*^2)}{=} g' \circ \underline{\underline{m}} \circ e$ and the universal pushout property of (2) it follows that there exists a morphism $\underline{m}\colon \underline{L} \to \underline{L}'$. It remains to show that tr_d is applicable via m. Match m satisfies the gluing condition w.r.t. tr_d, as, only attributes are deleted along $\underline{l}\colon \underline{K} \to \underline{L}$ by the construction of shifted rules. It remains to show that $m \models ac_{\underline{L}}$ by induction over the structure of application conditions (cf. Defs. 5.1 & 5.2 in [EEGH15]).



$b_1 = g \circ c \circ tr_1$

1. For $ac_L = \mathbf{true}$, $ac_{\underline{L}} = \mathbf{true}$ and $m \models ac_{\underline{L}}$.

2. For $ac_L = ac'_L = \exists(a\colon L \to C, ac_C)$, $ac_{\underline{L}} = ac'_{\underline{L}} = \exists(\underline{a}\colon \underline{L} \to \underline{C}, ac_{\underline{C}})$ and $ac_{\underline{L}'} = ac'_{\underline{L}'} = \exists(\underline{a}'\colon \underline{L}' \to \underline{C}', ac_{\underline{C}'})$ are constructed by pushouts (5), (6) and (7) over condition ac_L (cf. Def. 5.6.2 and Rem. 5.6.1). By $m' \models ac_{\underline{L}'}$ it follows that there exists morphism $q'\colon \underline{C}' \to M, q' \in \mathcal{M}$ with $q' \circ \underline{a}' = m'$ $^{(*^4)}$. By the universal pushout property of (6) and $\underline{a}' \circ \underline{m} \circ b_1 \overset{PO\,(7)}{=} b_3 \circ a_3 \circ a$ it follows that there exists morphism $\underline{q}\colon \underline{C} \to \underline{C}'$ with $\underline{q} \circ \underline{a} = \underline{a}' \circ \underline{m}$ $^{(*^5)}$. By $\underline{m} \in \mathcal{M}$ it follows that $\underline{q} \in \mathcal{M}$ By $\mathcal{M}$-composition (cf. Defs. 4.2 & 4.4 in [EEGH15]), $q = q' \circ \underline{q} \in \mathcal{M}$. Furthermore, $m \overset{(*^1)}{=} m' \circ \underline{m} \overset{(*^4)}{=} q' \circ \underline{a}' \circ \underline{m} \overset{(*^5)}{=} q' \circ \underline{q} \circ \underline{a} = q \circ \underline{a}$. Therefore, $m \models ac_{\underline{L}}$ for $ac_C = \mathbf{true}$.

3. Conversely, for $ac_L = \neg ac'_L, ac_{\underline{L}} = \neg ac'_{\underline{L}}$ and $ac_{\underline{L}'} = \neg ac'_{\underline{L}'}$ with $ac'_L, ac'_{\underline{L}}$ and $ac'_{\underline{L}'}$ from item 2, assume that there exists morphism $q\colon \underline{C} \to M$ with $m = q \circ \underline{a}$ $^{(*^6)}$. By the universal pushout property of (7) and $m' \circ \underline{m} \circ b_1 \overset{(*^1)}{=} m \circ b_1 \overset{(*^6)}{=} q \circ \underline{a} \circ b_1 \overset{PO\,(6)}{=} q \circ b_2 \circ a_3 \circ a$, there exists $q'\colon \underline{C}' \to M$ with $m' = q' \circ \underline{a}'$ and $q' \in \mathcal{M}$. Therefore, $m \models ac'_{\underline{L}} \Leftrightarrow m \not\models \neg ac'_{\underline{L}} = ac_{\underline{L}}$ implies that $m' \models ac'_{\underline{L}'} \Leftrightarrow m' \not\models \neg ac'_{\underline{L}'} = ac_{\underline{L}'}$ for $ac_C = \mathbf{true}$. Assume that $m \not\models ac_{\underline{L}}$, then the implication $m' \not\models ac_{\underline{L}'}$ contradicts with assumption $m' \models ac_{\underline{L}'}$. Thus, $m \models ac_{\underline{L}}$ for $ac_C = \mathbf{true}$.

4. For $ac_L = \wedge_{i\in I} ac_{L,i}$ $(\vee_{i\in I} ac_{L,i})$, $m' \models ac_{\underline{L'}} = \wedge_{i\in I} ac_{\underline{L'},i}$ $(\vee_{i\in I} ac_{\underline{L'},i})$ implies that $\forall i \in I$ $(\exists i \in I)$: $m' \models ac_{\underline{L'},i}$. By item 1 to item 3, $\forall i \in I$ $(\exists i \in I)$: $m \models ac_{\underline{L},i}$. Thus, $m \models ac_{\underline{L}} = \wedge_{i\in I} ac_{\underline{L},i}$ $(\vee_{i\in I} ac_{\underline{L},i})$ for $ac_C = \textbf{true}$.

The case for $ac_C \neq \textbf{true}$ is shown inductively over the nestings of the condition by item 1 to item 4. □

Example 5.6.2 (Relevant Maximal Shifted Rule & CC Rule)**.** *Given model* M^R_{marked} *as illustrated in Fig. 5.15 (bottom, right), triple rule* T_Step_args-2-attrs *from Fig. 4.8 and a decomposition* $d1$ *of triple rule* T_Step_args-2-attrs*. The triple rule is again illustrated in Fig. 5.17 (1).*

Intuitively, the shifted rule that results out of the decomposition $d1$ *shall "fit" to triple graph* M^R_{marked}*, i.e., additionally to all elements from the LHS of triple rule* T_Step_args-2-attrs*, it shall contain the* description = COMPONENT A *attribute on the first layer.*

Technically, this is achieved in the following way: Let $d1\colon L \to L' \to R$ *be given by the LHS* L *of triple rule* T_Step_args-2-attrs *and* L' *which additionally contains attribute* description *on the first layer. Therefore, decomposition* $d1$ *controls the shifting of elements from* R *towards* L*. The rule* $\text{T\_Step\_args-2-attrs}_{d1}\colon \underline{L} \leftarrow \underline{K} \to \underline{R}$ *in Fig. 5.17 (3) is a relevant maximal shifted applicable rule of triple rule* T_Step_args-2-attrs *w.r.t.* $d1$ *and* M^R_{marked}*. It is an operational rule derived from the corresponding triple rule (cf. Fig. 5.17 (1)).*

Graph $\underline{L}$ *contains all elements of* L' *where the shifted attribute in* L' *is marked with* **F***. The remaining elements* L *in* L' *are marked with* **T***. Graph*

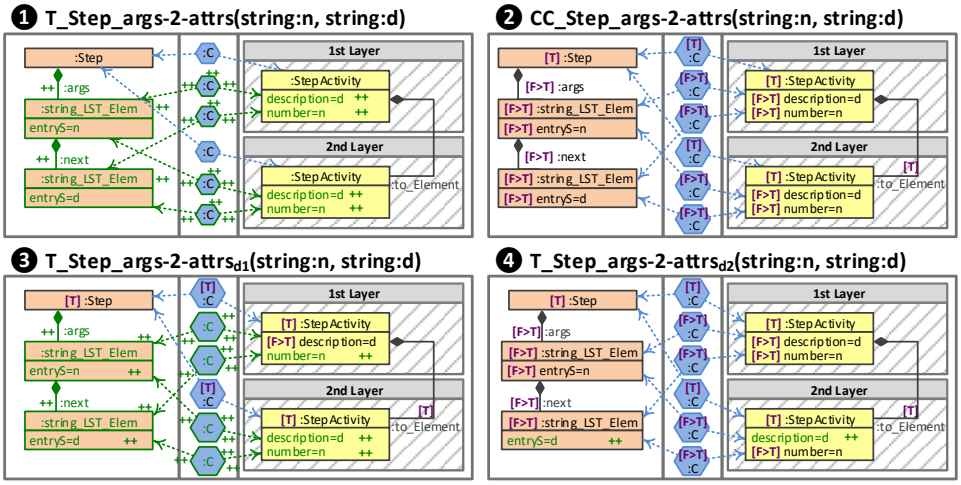


Figure 5.17: Triple Rule T_Step_args-2-attrs (1), the Corresponding CC Rule (2), and two Relevant Maximal Shifted Rules (3) and (4)

$\underline{K}$ *is* $\underline{L}$ *except for the shifted attribute which is unmarked in* $\underline{K}$ *(the marking is going to be updated to* $\mathbf{T}$*). All elements of* $\underline{R}$ *are marked with* $\mathbf{T}$*. Graph* $\underline{R}$ *is* $\underline{K}$*. Additionally it contains the remaining elements of the RHS* R *of triple rule* T_Step_args-2-attrs *that were not shifted and therefore, are created with markings* $\mathbf{T}$ *by applying the shifted rule as indicated by markings* $++$*. Furthermore, the application of the shifted rule updates the markings of the two shifted elements from* $\mathbf{F}$ *to* $\mathbf{T}$ *(*$\mathbf{F} > \mathbf{T}$*). The shifted rule is relevant, as, one elements is shifted, i.e.,* L *and* L' *are not isomorphic (*$L \ncong L'$*). Beyond, the rule is maximal shifted and applicable w.r.t.* M^R_{marked}*, as, there exists an applicable match from* $\underline{L}$ *to* M^R_{marked} *and no further elements can be shifted such that a new "extended" applicable match is obtained.*

In Fig. 5.17 (4) a second shifted rule $\mathsf{T\_Step\_args\text{-}2\text{-}attrs}_{\mathsf{d2}} \colon \underline{\underline{L}} \leftarrow \underline{\underline{K}} \rightarrow \underline{\underline{R}}$ *is illustrated. It is a relevant maximal shifted rule w.r.t.* $O^{R'}_{marked}$ *(cf. Fig. 5.24) and a decomposition* $d2 \colon L \rightarrow L'' \rightarrow R$*. There, elements were shifted to* $\underline{\underline{L}}$*, except attribute* description *of node* : StepActivity *which is part of the second layer of the SPELL-Flow domain and attribute* entryS *which belongs to the second* : string_LST_Elem *node of the list in the SPELL domain. Again, the RHS* $\underline{\underline{R}}$ *of shifted rule* $\mathsf{T\_Step\_args\text{-}2\text{-}attrs}_{\mathsf{d2}}$ *is* $\underline{\underline{K}}$*, but extended by both aforementioned attributes that were not shifted.*

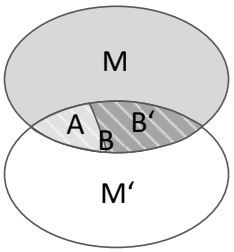
The rule CC_Step_args-2-attrs *in Fig. 5.17 (2) is the corresponding consistency creating (CC) rule of triple rule* T_Step_args-2-attrs*. All elements are shifted from the RHS towards the LHS. Therefore, when applying a CC rule no elements are created but the markings of the shifted elements are updated from* $\mathbf{F}$ *to* $\mathbf{T}$ *while the markings of all other elements remain* $\mathbf{T}$*.* $\triangle$

Update extensions are performed by applying relevant maximal shifted applicable rules and CC rules via extension matches. We distinguish between the following three notions of extension matches in order to technically define the intuition of sub-step Ext (cf. Ex. 5.6.5).

1. A match from a relevant maximal shifted applicable rule to a model M where the match (overlapping) does not contain previously deleted elements is called a *standard extension match.*
2. If the match contains previously deleted elements, then it is additionally guided. We call denote this kind of match as *guided extension match.*
3. If no relevant maximal shifted applicable rule exists for a triple rule tr and w.r.t. model M, then complete overlappings of tr with previously deleted elements may exist with corresponding induced matches from the CC rule of tr. These matches are called *CC extension matches.*

Matches that do not contain previously deleted (i.e., recreated) elements are defined in terms of non-recreating matches (cf. Def. 5.6.6) which in turn are defined based on the notion of recreation correlations which we introduce formally in Rem. 5.6.3.

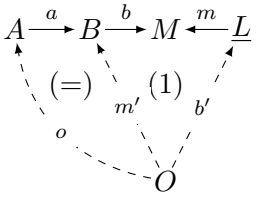
Remark 5.6.3 (Recreation Correlation)**.** *A recreation correlation is given by three morphisms* $\langle a\colon A \to B, b\colon B \to M, c\colon B \to M'\rangle$ *and outlines the following situation (see illustration on the right).*



Model A is derived from M' *by deleting some elements. Model B is derived from A by recreating some deleted elements* $B' = B \setminus a(A)$*. These recreated elements together with the elements of A are also present in a fourth model M. Therefore, for each pair of elements* (x, y)*, where* $x = b(e) \in M$ *and* $y = c(e) \in M'$ *with* $e \in B'$*, elements x and y are said to be in a recreation correlation via common B.* △

In using the formalisation of the recreation correlation, we are now able to define the term of the non-creating match, i.e., matches that do not contain elements that were deleted previously.

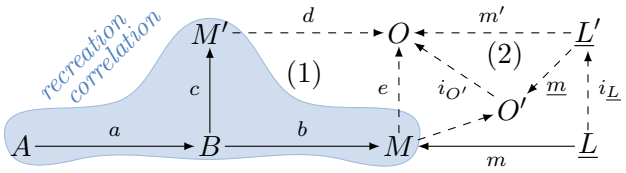
Definition 5.6.6 (Non-Recreating Match)**.** *Given a shifted rule* $tr_d = (\underline{L} \leftarrow \underline{K} \rightarrow \underline{R}, ac_{\underline{L}})$ *with match* $m\colon \underline{L} \to M$ *and recreation correlation* $\langle a\colon A \to B, b\colon B \to M, c\colon B \to M'\rangle$*. Match m is* non-recreating *w.r.t.* $\langle a, b, c\rangle$*, if there exists a morphism* $o\colon O \to A \in \mathcal{M}$ *for the pullback (1) over* b, m *such that* $a \circ o = m'$*.* △



Remark 5.6.4 (Non-Recreating Match)**.** *Model B is derived from A by recreating some previously deleted elements of M (cf. Rem. 5.6.3). Model O is obtained by intersection of B and* $\underline{L}$ *via common M. Therefore, O contains those elements of B that are matched by m in M. Thus, if there exists a morphism* $o\colon O \to A$ *with commuting* $a \circ o = m'$*, then match m does not contain previously deleted (i.e. recreated) elements.* △

Based on non-recreating matches, *standard*, *guided* and *CC extension matches* are defined.

Definition 5.6.7 ((CC & Guided) Extension Match)**.** *Given a triple rule* $(tr\colon L \to R, ac_L)$ *and a recreation correlation* $\langle a\colon A \to B, b\colon B \to M, c\colon B \to M'\rangle$*.*



1. *Let* $tr_d = (\underline{tr}\colon \underline{L} \leftarrow \underline{K} \rightarrow \underline{R}, ac_{\underline{L}})$ *be a relevant maximal shifted applicable rule of tr w.r.t. M with match* $m\colon \underline{L} \to M \in \mathcal{M}$*. If m is*

a non-recreating match w.r.t. $\langle a, b, c\rangle$*, then* m *is called a* standard extension match*.*

2. *Let* O *be obtained by pushout* (1) *over* b, c*. Given a relevant maximal shifted applicable rule* $tr'_d = (\underline{tr}'_d\colon \underline{L}' \leftarrow \underline{K}' \rightarrow \underline{R}', \underline{ac}_{\underline{L}'})$ *of* tr *w.r.t.* O *with match* $m'\colon \underline{L}' \rightarrow O \in \mathcal{M}$ *and inclusion* $i_{\underline{L}}\colon \underline{L} \rightarrow \underline{L}'$*. Furthermore, let* O' *be obtained by an effective pushout* (2) *over* e, m' *with induced morphisms* $i_{O'}\colon O' \rightarrow O \in \mathcal{M}$ *and* $\underline{m}\colon \underline{L}' \rightarrow O'$ *with* $i_{O'} \circ \underline{m} = m'$*. If* m *is not a non-recreating match w.r.t.* $\langle a, b, c\rangle$ *and* $e \circ m = m' \circ i_{\underline{L}}$*, then* $\underline{m}$ *is called a* guided extension match*.*

3. *Considering item 2. If* tr_d *with* m *does not exist and* tr'_d *is a CC rule of* tr*, then* $\underline{m}$ *is called a* CC extension match*.* △

In Def. 5.6.7, PO (2) is required to be an effective pushout, since, induced morphism $i_{O'}$ needs to be an $\mathcal{M}$-morphism for the recursive application of Def. 5.6.7 as performed by the update extension sequences (cf. Def. 5.6.9). In a recursive step with $c = i_{O'} \in \mathcal{M}$, it is true that $e \in \mathcal{M}$ and $e'\colon M \rightarrow O' \in \mathcal{M}$, because $\mathcal{M}$-morphisms are closed under pushouts and pullbacks. This allows $\mathcal{M}$-traces of update extension steps in Def. 5.6.8, with $e' \circ b \circ a \in \mathcal{M}$ by $\mathcal{M}$-composition which leads to valid model updates (span of $\mathcal{M}$-morphisms) as the result of the update propagation (cf. Def. 5.2.1).

Example 5.6.3 (Standard, Guided and CC Extension Match)**.** *Considering Sec. 5.6.1, which discusses a detailed execution of the* Ext *sub-step on our running example, we are able to find all three kinds of matches. For details we refer to the detailed description in the aforemetioned section.*

- *In the first iteration, we discuss the application of a* standard extension match *using models in Fig. 5.22 and shifted rule* $\mathsf{T\_Comment\_LST\_Elem\text{-}2\text{-}comments_d}$ *(cf. Fig. 5.18).*

- *The second iteration uses a* guided extension match *with shifted rules* $\mathsf{T\_Step\_args\text{-}2\text{-}attrs_{d1}}$ *w.r.t. model* $M^{R'}_{marked}$ *and* $\mathsf{T\_Step\_args\text{-}2\text{-}attrs_{d2}}$ *w.r.t. model* $O^{R'}_{marked}$ *(cf. Fig. 5.17 (3) + (4) for both shifted rules and cf. Figs. 5.23 and 5.24 for both models).*

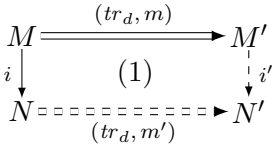
- *An example for a* CC extension match *is provided in the third iteration. There, the CC rule* $\mathsf{CC\_sourceCodeText\text{-}2\text{-}sourceCode}$ *(cf. Fig. 5.19) is applied to triple graphs* M'''_{marked} *and* $O^{R'''}_{marked}$ *in Fig. 5.26.* △

The application of shifted rules can be embedded into a bigger context: In detail that means: If a shifted rule tr_d is applicable to model M, i.e., a morphism from the LHS of tr_d to M exists and the application condition (if it exists) is fullfiled. Then tr_d is also applicable to a model M', if a $\mathcal{M}$-morphism from M to M' exists, i.e., M can be embedded into the bigger model M' and all elements in M' that are not part of M get marker **F**.

Proposition 5.6.2 (Shifted Rules along $\mathcal{M}$-Morphisms)**.** *Let $i\colon M \to N \in \mathcal{M}$ be an $\mathcal{M}$-morphism and $tr_d = (\underline{tr}_d\colon \underline{L} \leftarrow \underline{K} \rightarrow \underline{R}, ac_{\underline{L}})$ be a shifted rule. If the extension $\underline{M} = N \setminus i(M)$ of M along i is exclusively marked with $\mathbf{F}$ ($\underline{M} = G \oplus Att^{\mathbf{F}}_G$ for some unmarked G), then*

1. *There exists an applicable match $m\colon \underline{L} \to M$ if and only if there exists an applicable match $m'\colon \underline{L} \to N$ with $m' = i \circ m$, AND*

2. *The transformation step $M \overset{(tr_d,m)}{\Longrightarrow} M'$ leads to extension diagram (1) with transformation step $N \overset{(tr_d,m')}{\Longrightarrow} N'$ and $i'\colon M' \to N'$.*



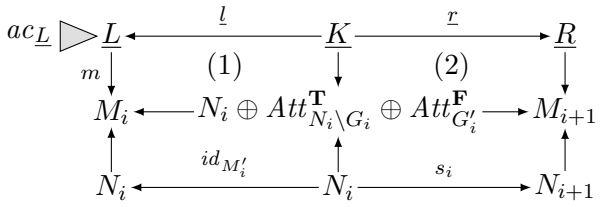
△

Proof. We split the proof of Prop. 5.6.2 in two cases. In the first case the shifted rule tr_d has no application condition $tr_d = (\underline{tr}_d, \varnothing)$. Shifted rules only add elements and change trasnlation attributes from $\mathbf{F}$ to $\mathbf{T}$, i.e., only translation attributes are deleted. Thus, the gluing condition (cf. Def. 6.3 in [EEPT06]) is also fulfilled for the application of the shifted rule tr_d via match $m' : \underline{L} \to N$.

The second case considers shifted rules with application conditions $tr_d = (\underline{tr}_d, ac_{\underline{L}})$. The construction of shifted rules marks all elements in the application condition $ac_{\underline{L}}$ to $\mathbf{T}$ that are not in $\underline{L}$ (cf. Def. 5.6.2 and Rem. 5.6.1). Therefore, the match $m' : \underline{L} \to N$ exists, too, because rule tr_d is applicable to M, and thus, the morphism from $ac_{\underline{L}}$ to an extended N maps the same elements as the one to M. So, the shifted rule tr_d can be applied to N. □

An update extension step can be seen as one iteration step of the update propagation. It defines the application of one shifted rule to the model.

Definition 5.6.8 (Update Extension Step)**.** *An update extension step $s_G = (N, M \overset{(tr_d,m)}{\Longrightarrow} M', N')$ from M to M' w.r.t. an extension relevant subgraph $G \subseteq N$ is given by a transformation step $M \overset{(tr_d,m)}{\Longrightarrow} M'$ via extension match $m\colon \underline{L} \to M$ w.r.t. G from shifted rule $tr_d = (\underline{tr}\colon \underline{L} \overset{l}{\leftarrow} \underline{K} \overset{r}{\Rightarrow} \underline{R}, ac_{\underline{L}})$ to M with recreating morphism $r\colon \underline{M} \to M \in \mathcal{M}$ where $\underline{M} = M' \oplus Att^{\mathbf{T}}_{M' \setminus G} \oplus Att^{\mathbf{F}}_G$ and $\underline{N} = N' \oplus Att^{\mathbf{T}}_{N' \setminus G'} \oplus Att^{\mathbf{F}}_{G'}$ for some $G' \subseteq N'$ (cf. diagram (1)+(2)). Furthermore, by the construction of recreating extension matches $M = \underline{M} \uplus H \oplus Att^{\mathbf{F}}_H$ for some elements H (cf. Def. 5.6.7).*



The trace $trace(s_{G_i}) = (s_i \circ r) \in \mathcal{M}$ *of a step* s_{G_i} *is given by the recreating morphism* r *composed with the induced morphism* $s_i \colon N_i \to N_{i+1}$. △

The trace morphism is a morphism to the "original" model, i.e., the model on which we applied the shifted rule (CC rule). It keeps track of the changes that were applied to the "original" model.

Remark 5.6.5 (Trace $\mathcal{M}$-Morphism)**.** *The trace morpism* $trace(s_{G_i}) = s_i \circ r$ *of an update extension step is an* $\mathcal{M}$*-morphism, as, pushout (2) preserves* $\mathcal{M}$*-morphism* $\underline{r} \in \mathcal{M}$ *with* $s_i \in \mathcal{M}$ *and by the composition of* $\mathcal{M}$*-morphisms* s_i *and* r *it follows that* $s_i \circ r \in \mathcal{M}$ *(cf. Def. 4.4 & Def 4.2 in [EEGH15]).* △

We will now combine all update extension steps to a sequence, which is called *update extension sequence*. In order to get functional behaviour (cf. Thm. 5.6.1), we demand termination of the sequence.

Definition 5.6.9 ((Terminating) Update Extension Sequence)**.** *Let* M *be a sequence start model with* $G \subseteq M$*. An update extension sequence w.r.t.* G *is inductively defined as follows.*

1. *Each empty sequence* $s_G = (M \xrightarrow{id_M} M)$ *is an update extension sequence from* M *to* M*.*

2. *Each update extension step* $s_G = (M \xrightarrow{(tr_d, m)} M')$ *(cf. Def. 5.6.8) is an update extension sequence from* M *to* M'*.*

3. *The composition of update extension steps* $s^1_G = (M \xrightarrow{(tr^1_d, m^1)} M^1) \circ s^2_G = (M^1 \xrightarrow{(tr^2_d, m^2)} M^2) \circ \ldots \circ s^n_G = (M^n \xrightarrow{(tr^n_d, m^n)} M^n)$ *is an update extension sequence from* M *to* M^n*.* △

The order of applying CC and non-CC rules (i.e., general shifted rules) in an update extension sequence is irrelevant when assuming appropriate sufficient conditions as shown by Prop. 5.6.1. Thus, finding applicable matches does not need to be ordered in Def. 5.6.7.

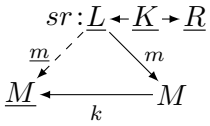
Example 5.6.4 (Extension Step & Sequence)**.** *The application of a shifted rule or CC rule, respcetively, represents one extension step of the update from Ex. 5.2.1 Ex. 5.6.5 in the sense that the elements which are created by the update are extended by those elements of the shifted rule (or CC rule) that do not overlap with the update and therefore, would also be created when*

applying the shifted rule (or CC rule). The update of markings from **F** *to* **T** *are introduced in order to control which elements of the update already have been extended.* △

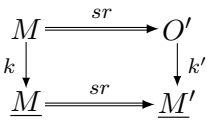
Now, we are able to determine the Ext sub-step based on the details defined before in this section. Finally, the complete derived propagation framework *Ppg* can be defined.

Remark 5.6.6 (Add Step - Sub-Step III)**.** *Given marked triple graphs M and $\underline{M}$ and a morphism $k : M \to \underline{M}$ between them resulting out of the second sub-step of* Add*, i.e., the* Marking *sub-step (cf. Sec. 5.5.2). Furthermore, given a set of triple rules TR.*

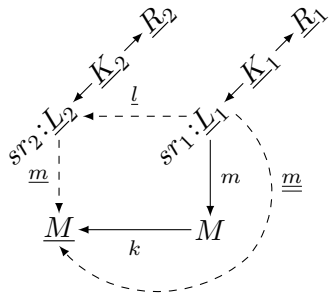
Case 1: standard extension match (using shifted rule) *The* Ext *sub-step computes a maximal relevant shifted rule $sr\colon \underline{L} \leftarrow \underline{K} \rightarrow \underline{R}$ w.r.t. triple graph M, i.e., $\exists m\colon \underline{L} \to M$. Then, morphism m will be extended to $\underline{M}$, i.e., $\underline{m} = k \circ m\colon \underline{L} \to \underline{M}$.*



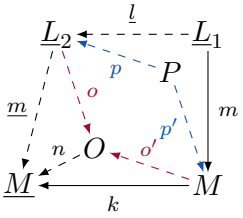
If m is a standard extension match*, then we can apply the shifted rule* sr *to models M and $\underline{M}$ resulting in models O' and $\underline{M}'$.*



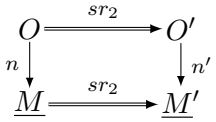
Case 2: guided extension match (using shifted rule) *In case, no standard extension match can be found, the* Ext *sub-step checks for a* guided extension match *in the following way. The* Ext *sub-step computes a maximal relevant shifted rule $sr_1\colon \underline{L}_1 \leftarrow \underline{K}_1 \rightarrow \underline{R}_1$ w.r.t. triple graph M, i.e., $\exists m\colon \underline{L}_1 \to M$ (similar to case 1). Afterwards, the morphism m will be extended to $\underline{M}$, i.e., $\underline{m} = k \circ m\colon \underline{L}_1 \to \underline{M}$. Based on morphism $\underline{m}$, a new relevant maximal shifted rule $sr_2\colon \underline{L}_2 \leftarrow \underline{K}_2 \rightarrow \underline{R}_2$ will be computed w.r.t. model $\underline{M}$. Hence, a morphism $\underline{l}\colon \underline{L}_1 \to \underline{L}_2$ exists and also $\underline{m}\colon \underline{L}_2 \to \underline{M}$, as illustrated in the diagram below.*



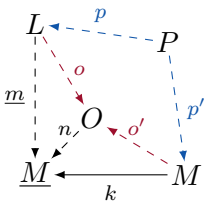
In the next step, a pullback via $(\underline{m}, \underline{M}, k)$ needs to be constructed first (also highlighted in blue). Then, a pushout via (p, P, p') is derived (red). So, the effective pushout construction is used (cf. Def. 2.1.15).



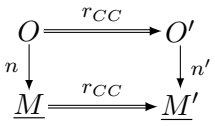
The result is triple graph O that contains all necessary elements to apply shifted rule sr_2 to O. The match is given by $o\colon \underline{L}_2 \to O$. Furthermore, an $\mathcal{M}$-morphism $n\colon O \to \underline{M}$ with $n \in \mathcal{M}$ exists, due to construction of effective pushouts.
Finally, we apply the relevant maximal shifted rule sr_2 as well on model O as on model $\underline{M}$ resulting in triple graphs O' and $\underline{M}'$, respectively.



Case 3: CC extension match (using CC rule) *If no relevant maximal shifted rule can be derived, then a CC rule $r_{CC}\colon L \leftarrow K \rightarrow R$ might be applicable to model $\underline{M}$ via CC extension match. In that case, a match $\underline{m}$ from a CC rule out of the set of all CC rules (cf. Def. 5.6.3) to model $\underline{M}$ is given, i.e., $\underline{m}\colon L \to \underline{M}$. Consequently, rule r_{CC} is applicable to $\underline{M}$. In order to also apply that rule to M, we need to "add" the necessary elements to M, so that r_{CC} is applicable. For that, an effective pushout is constructed resulting in an enriched model O (see the following diagram). Then, a match $o : L \to M$ is available so that r_{CC} can be applied to O. In addition, $\mathcal{M}$-morphism $n : O \to \underline{M}$ follows out of the construction of effective pushouts.*



Finally, CC rule r_{CC} can be applied to $\underline{M}$ and O resulting in new models O' and $\underline{M}'$.

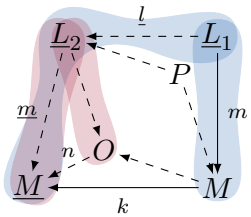


If one of the three cases mentioned above is has been executable, then the Ext *sub-step starts another iteration. For the next iteration, graphs O' and $\underline{M}'$ are provided as input.*

At the end, the Ext *sub-step returns a model $M^F = M'^i \setminus (Att^{\mathbf{T}}_{M'^i} \oplus Att^{\mathbf{F}}_{M'^i})$. i.e., the last model M'^i without markers.* △

Note, the following diagram summarises the three cases of Rem. 5.6.6 in one diagram illustrating the difference between the three kinds of extension matches.

standard extension match (case 1)



guided & CC extension match (case 2 & 3)

Finally, the derived propagation framework returns a model update $u'\colon M \to N$ if each marker in N_{marked} is set to $\mathbf{T}$. N is model N_{marked} but the markers are omitted, i.e., this is the resulting model of the derived propagation framework. If the derived propagation framework is not able to derive a consistent model, i.e., if it is not able to derive a model that only contains $\mathbf{T}$-markers, then it returns update $u'\colon M \to \varnothing$.

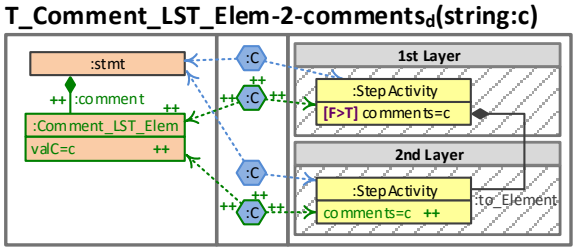


Figure 5.18: Shifted Rule T_Comment_LST_Elem-2-comments$_d$

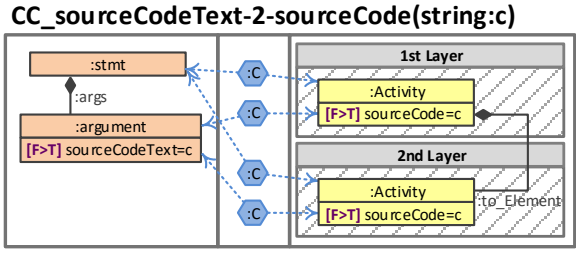


Figure 5.19: CC Rule CC_sourceCodeText-2-sourceCode

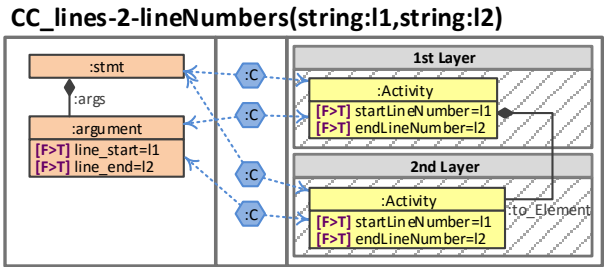


Figure 5.20: CC Rule CC_lines-2-lineNumbers

5.6.1. Running Example: Ext sub-step in great detail

After defining the formal construction of the Ext sub-step, we illustrate the execution of the extension sub-step Ext on our running example in great detail.

Example 5.6.5 (Sub-step Ext of Add in Great Detail)**.** *The* Ext *sub-step uses the marked models from the second sub-step of* Add *as input (cf. Fig. 5.15). This initial situation is also illustrated in Fig. 5.21. As described in Rem. 5.6.6, the* Ext *sub-step successively checks if one of the mentioned cases is applicable, i.e., if a standard extension match, a guided extension match or a CC extension match can be found. We will now show all iterations of this process in detail. Note, if we include matches within the following gigures explicitly, we illustrate them by means of gray dots.*

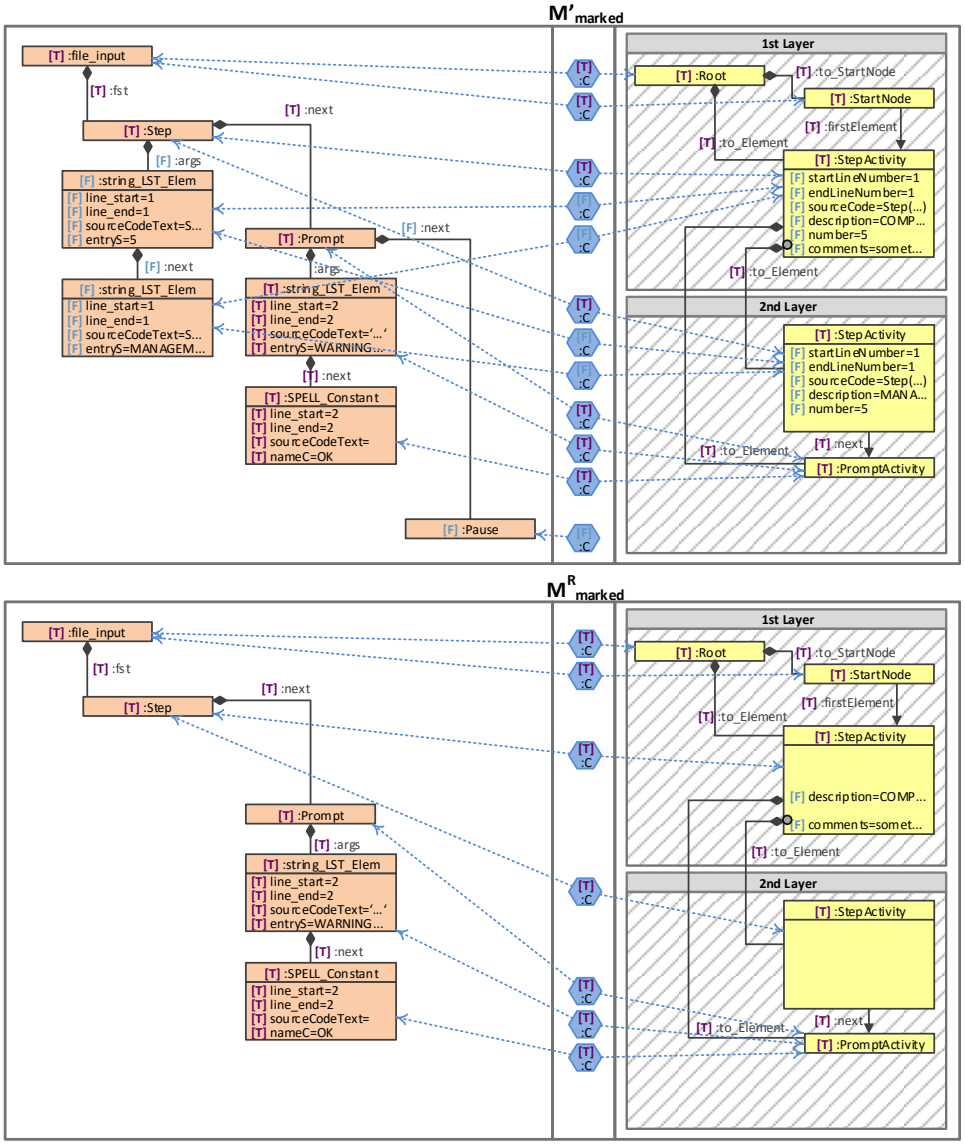


Figure 5.21: Initial Situation, Including Match

1st iteration: *First of all,* Ext *checks, if a shifted rule with* standard match *can be found with regard to the models* M'_{marked} *and* M^R_{marked}*. This is possible w.r.t. to attribute* comments *in the SPELL-Flow domain. If we consider triple rule* T_Comment_LST_Elem-2-comments *in Fig. 4.10 then we can derive the maximal relevant shifted rule* T_Comment_LST_Elem-2-comments$_d$ *(cf. Fig. 5.18) w.r.t. model* M^R_{marked} *which is applicable to both models* M'_{marked} *and* M^R_{marked} *(cf. Rem. 5.6.6). The application of this relevant maximal shifted rule leads to models* M''_{marked} *and* O^R_{marked} *which are illustarted in Fig. 5.22. Finally, both models are extended by the new com-*

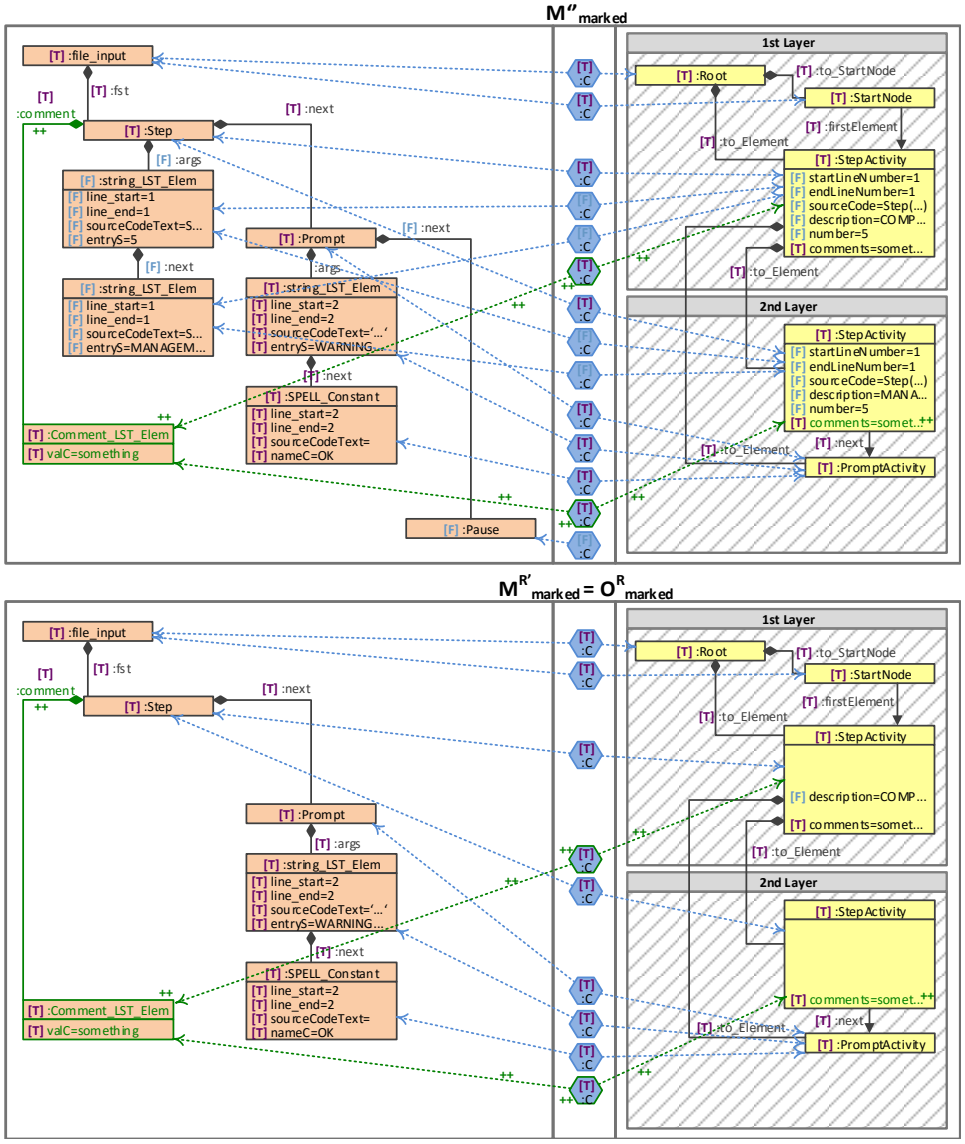


Figure 5.22: Standard Extension Match: Result of Shifted Rule Application

ment. In explicit these are: node : Comment_LST_Elem *with its attribute* valC = something *and with the containment edge* : comment *in the SPELL domain. Furthermore, the* : StepActivity *node on the second layer of the SPELL-Flow domain is enriched by a new attribute* comments = something. *Finally, the necessary correspondence nodes and edges are added, too. The markers of all created elements are set to* **T**. *The marker of the* comments *attribute being part of the* : StepActivitiy *node on the first layer is changed from* **F** *to* **T**.

2nd iteration: *Both models* M''_{marked} *and* O^R_{marked} *are taken as input*

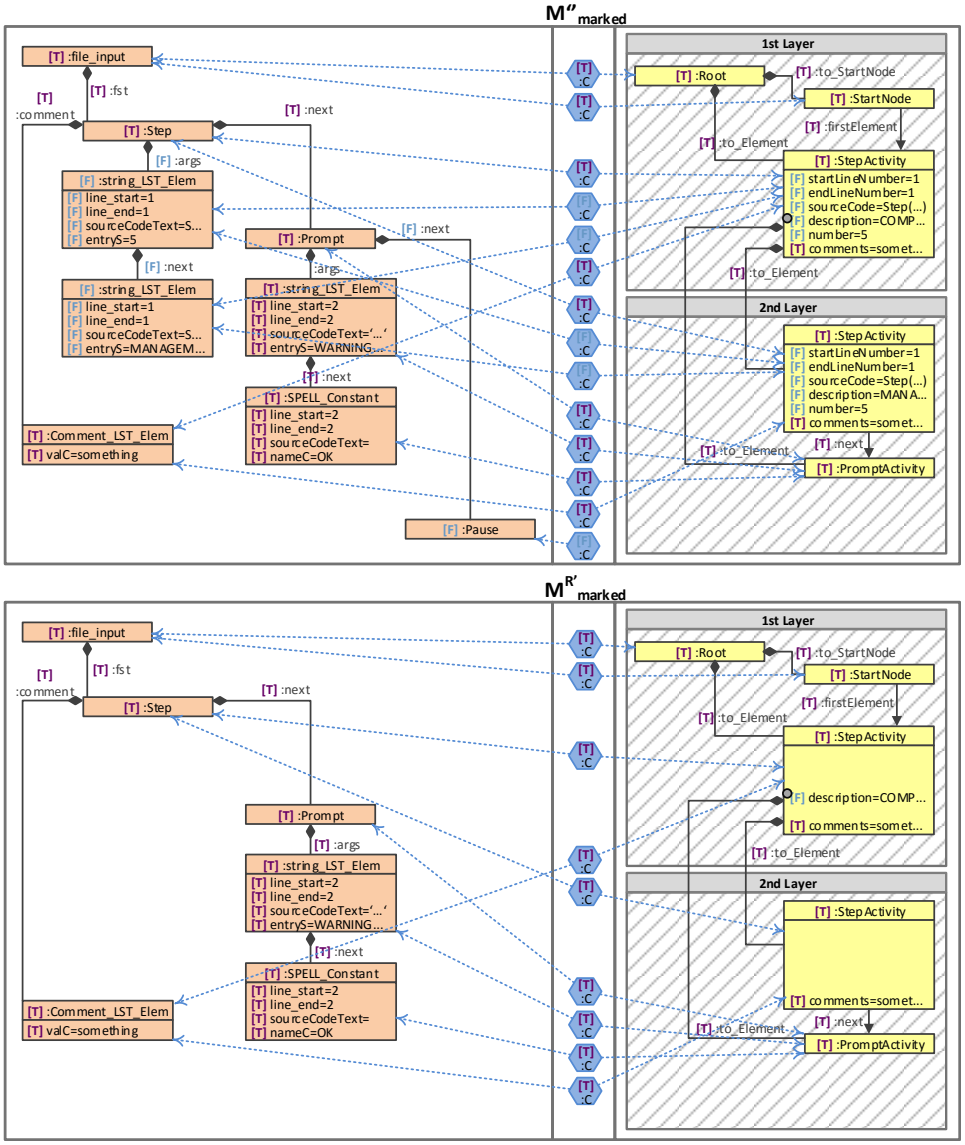


Figure 5.23: New Input to next Ext Iteration with Guided Extension Match

to the next iteration of the Ext *sub-step. Again, it is checked, if a standard extension match can be found. This is not possible. Instead, a* guided extension match *can be found w.r.t. model* $M_m^{R'}$ *arked. This situation is illustrated in Fig. 5.23. The corresponding maximal relevant shifted rule w.r.t.* $M_m^{R'}$ *arked is rule* $\mathsf{T\_Step\_args\text{-}2\text{-}attrs}_{\mathsf{d1}}$ *visualised in Fig. 5.17 (3). According to Rem. 5.6.6, a new intermediate model* $O^{R'}_{marked}$ *is calculated via effective pushout construction which is shown in Fig. 5.24. With regatd to this new triple graph, another relevant maximal shifted rule is derived:* $\mathsf{T\_Step\_args\text{-}2\text{-}attrs}_{\mathsf{d2}}$ *which is shown in Fig. 5.17 (4). This shifted rule is*

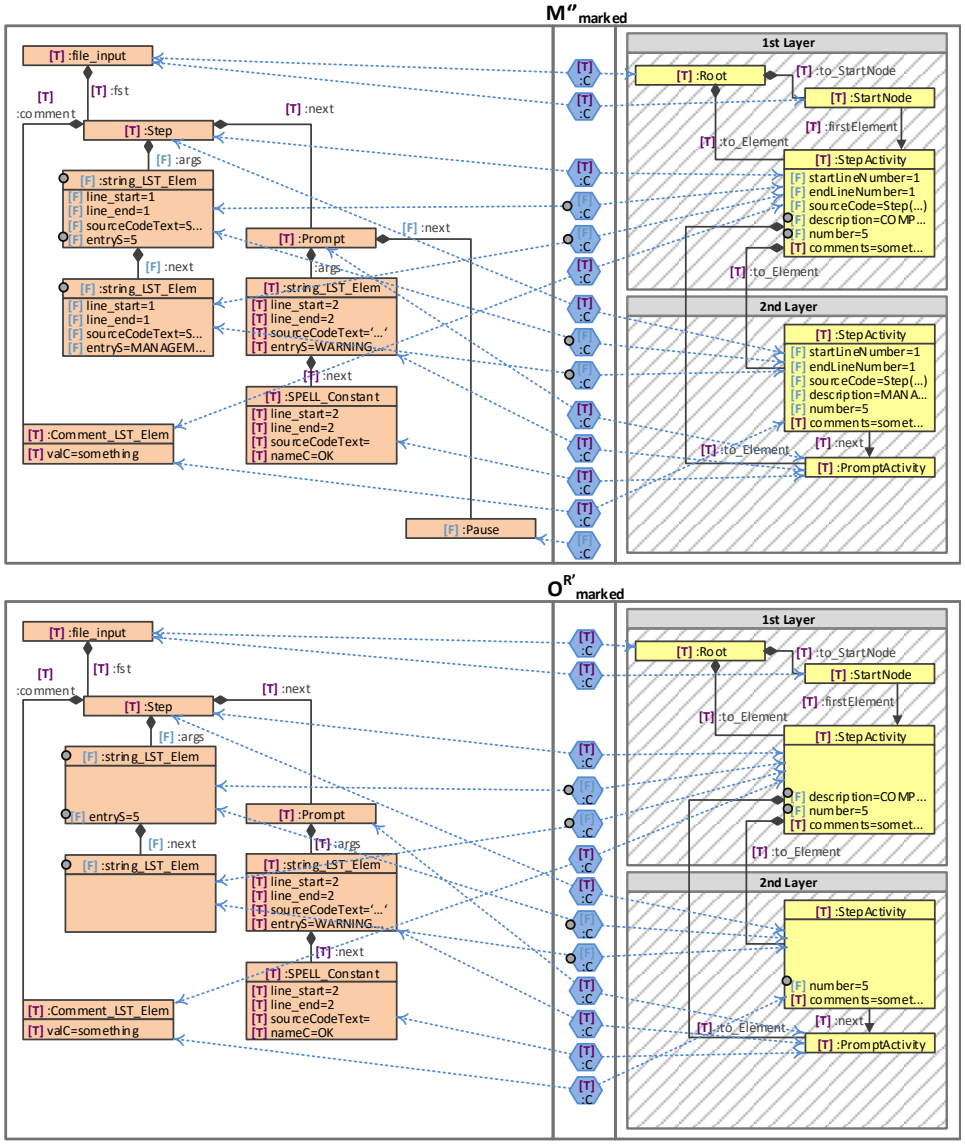


Figure 5.24: Guided Extension Match: Result of Effective Pushout, Including Match

finally applicable to triple graphs $O^{R'}_{marked}$ *and* M''_{marked}*. It guides the reconstruction of elements that got lost during the deletion step (*Del*, cf. Sec. 5.4). Finally, the application of shifted rule* $\mathsf{T\_Step\_args\text{-}2\text{-}attrs}_{d2}$ *leads to models* M'''_{marked} *and* $M^{R''}_{marked}$ *that are given in Fig. 5.25.*

3rd iteration: *In the next iteration step,* Ext *checks again, if a standard extension match or a guided extension match w.r.t. triple graph* $M^{R''}_{marked}$ *can be found, which is not possible. Then, according to Rem. 5.6.6, the* Ext *sub-step searches for a CC extension match w.r.t.* M'''_{marked}*.*

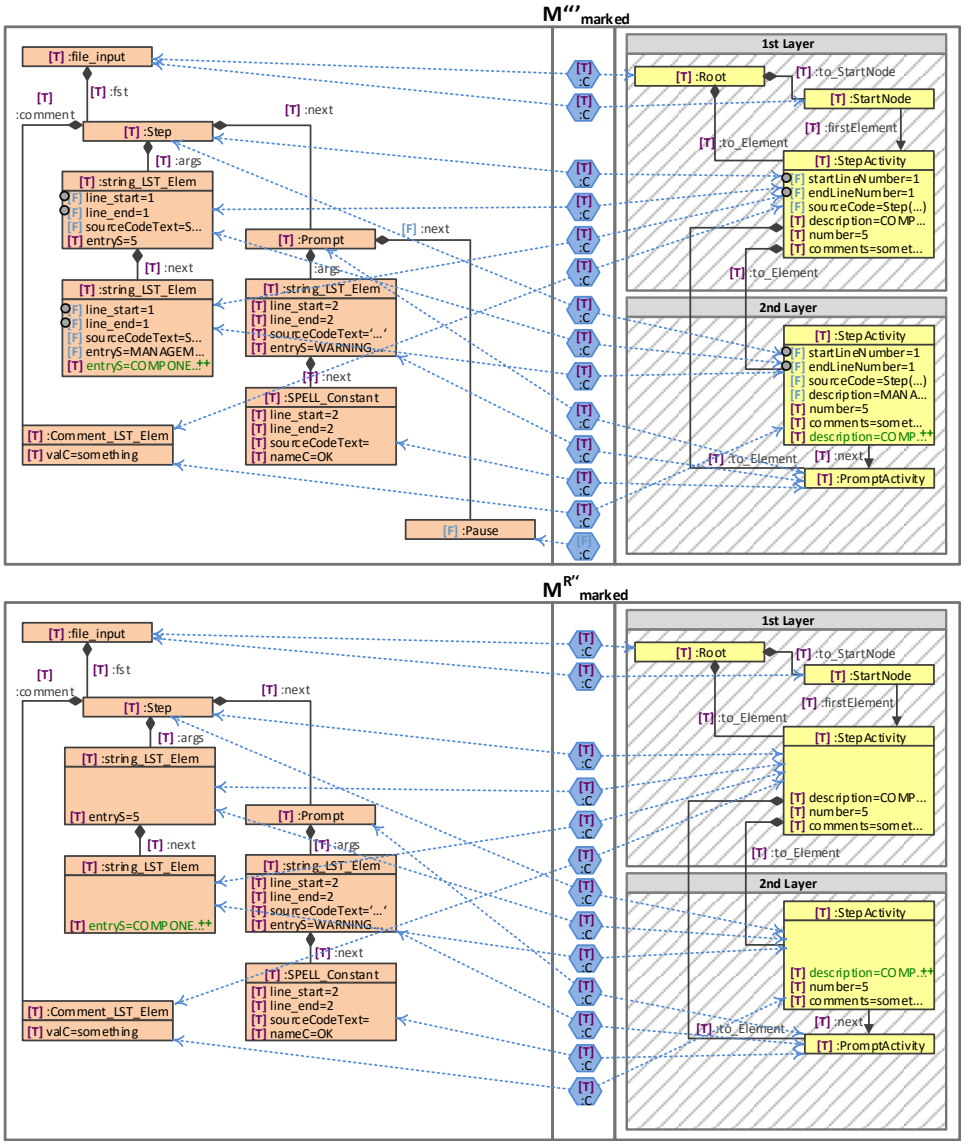


Figure 5.25: Result of Shifted Rule Application, New Input to Next Iteration

This kind of match is possible. In Fig. 5.25 the CC extension match is illustrated, i.e., both line_start *attributes and both* line_end *attributes in the SPELL domain, as well as both* startLineNumber *attributes and both* endLineNumber *attributes in the SPELL-Flow domain. The corresponding CC rule* CC_lines-2-lineNumbers *is illustrated in Fig. 5.20 which is derived out of the triple rule* T_lines-2-lineNumbers *in Fig. 4.13. Then, an effective pushout is constructed in order to add those missing attributes to model* $M^{R''}_{marked}$ *resulting in model* $O^{R''}_{marked}$ *that is given in Fig. 5.26. This picture contains the match also for model* $O^{R''}_{marked}$*. The application of the CC rule*

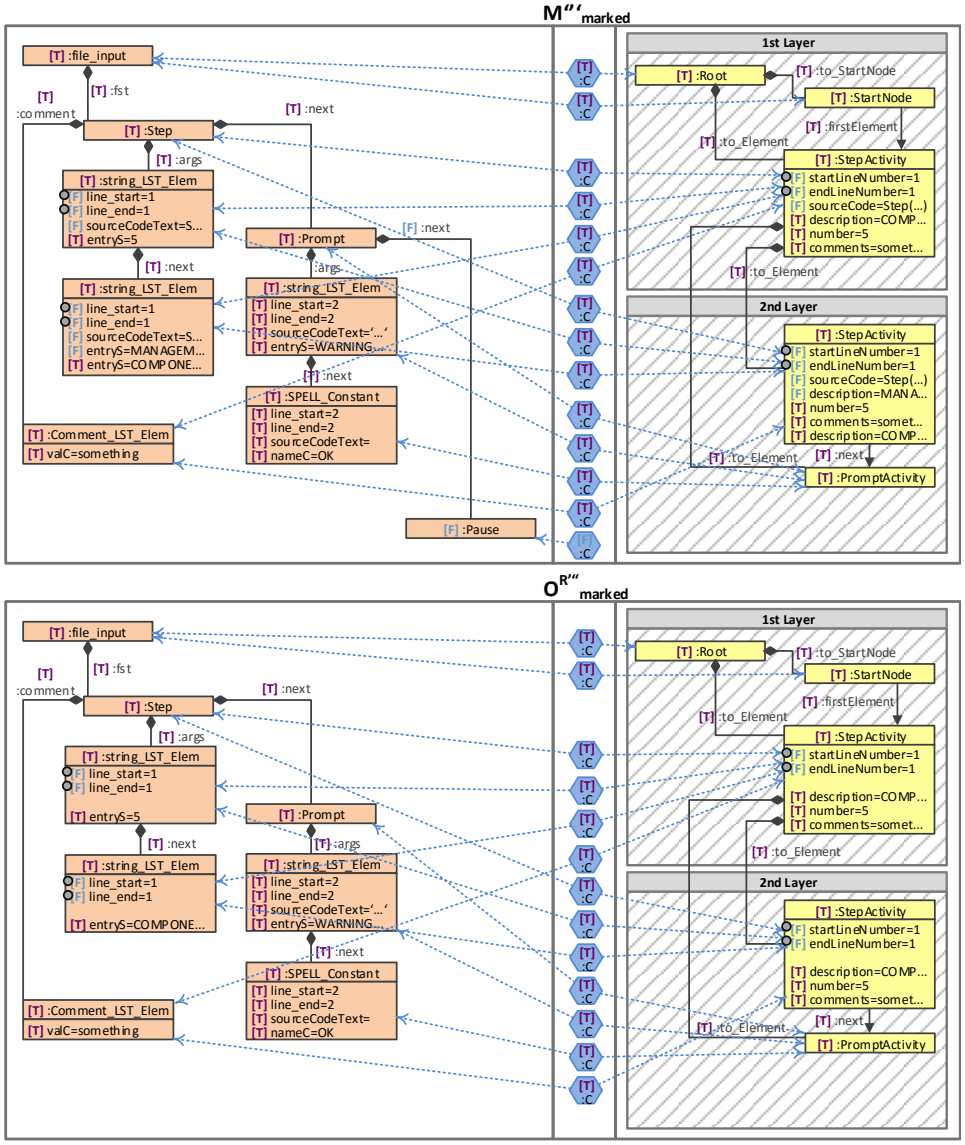


Figure 5.26: CC Extension Match: Result of Effective Pushout, Including Match

CC_lines-2-lineNumbers *sets the* **F** *markes of the mentioned attributes to* **T**. *The resulting triple graphs* M^4_{marked} *and* $M^{R'''}_{marked}$ *are shown in Fig. 5.27.*

4th iteration: *The next iteration step is similar to the third iteration. No standard extension match and no guided extension match can be found. Instead, a CC match w.r.t. model* M^4_{marked} *exists which is also illustrated in Fig. 5.27. Both attribute* sourceCodeText *in the SPELL domain and both* sourceCode *attributes in the SPELL-Flow domain are target of the CC extension match to* M^4_{marked}. *We illustrate the relevant*

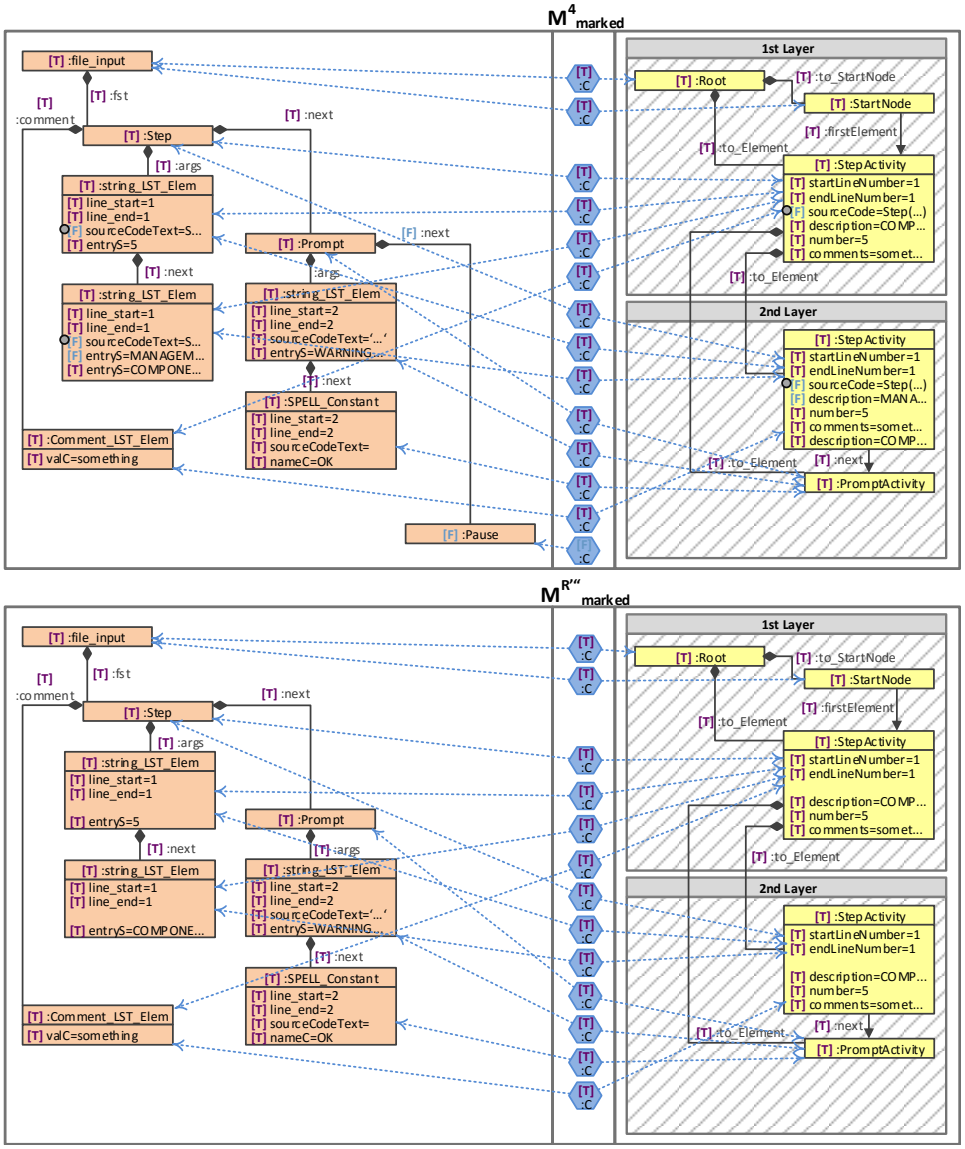


Figure 5.27: Result of CC Rule Application, New Input to Next Iteration

CC rule CC_sourceCodeText-2-sourceCode *that is derived out of triple rule* T_sourceCodeText-2-sourceCode *(cf. Fig. 4.12) in Fig. 5.19. Again, an effective pushout is constructed resulting in model* O'''_{marked} *which is* M'''_{marked} *extended by the aforementioned attributes so that the CC rule is applicable (cf. Fig. 5.28). The application of the CC rule yield in models* M^5_{marked} *and* M^{R4}_{marked}*, where the* **F** *markers of those attributes are changed to* **T**. *Both models are shown in Fig. 5.29.*

Final result: *The* Ext *sub-step finishes successfully with models* M^5_{marked} *and* M^{R4}_{marked}*, because neither a standard extension match, a guided*

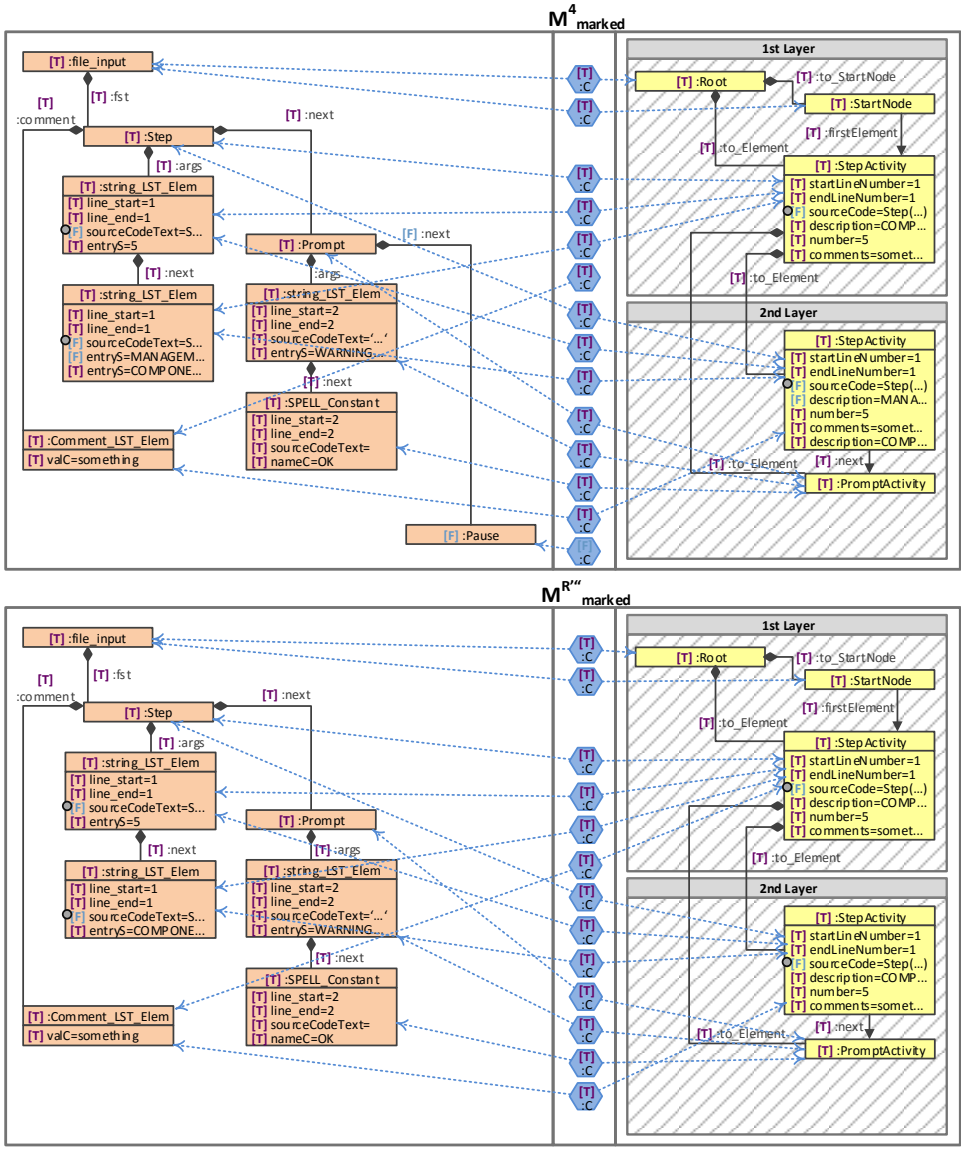


Figure 5.28: CC Extension Match: Result of Effective Pushout, Including Match

extension match, nor a CC extension match can be found according to the given TGG. Besides, the extended model M^{R4}_{marked} *only contains* $\mathbf{T}$ *markers. This model is also the relevant one, i.e., it will be returned as part of an update* $u' : M \rightarrow M^{R4}$ *by the derived propagation framework, where* $M^{R4} = M^{R4}_{marked} \ominus Att^{\mathbf{F}}_{M^{R4}} \ominus Att^{\mathbf{T}}_{M^{R4}}$*, i.e., model* M^{R4}_{marked} *without markers. The second triple graph* M^{5}_{marked} *can be understood as a helper model for the recreation of elements that were deleted during the* Del *step. This model will be discarded at the end of the* Ext *sub-step.* △

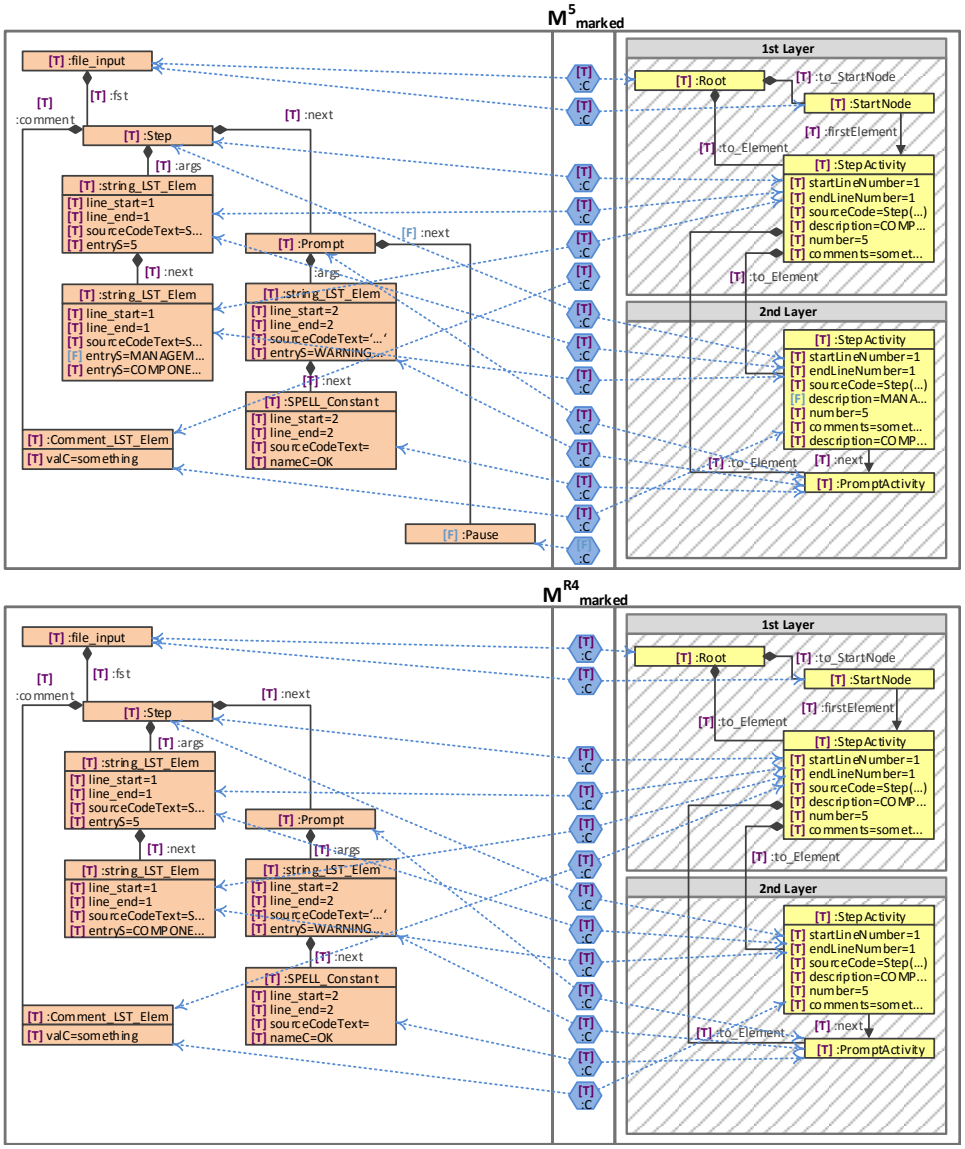


Figure 5.29: Final Result

5.6.2. Derived Propagation Framework

By combining all steps we obtain the derived propagation framework $Ppg(TGG)$ in Def. 5.6.10 with main properties in Thm. 5.6.1 and full proofs in Sec. 5.7.

Definition 5.6.10 (Derived Propagation Framework)**.** *Given a TGG over domains D_1 and D_2, then each propagation operation $(Ppg_{D_i})_{i=1,2}$ of the derived propagation framework $Ppg(TGG)$ applying update u on M and deriving a consistent model M' is defined by the following sequence of steps:*

1. Del *step, cf. Rem. 5.4.1,*
2. Add *step consisting of sub-steps:*
   (a) *sub-step first addition, cf. Rem. 5.5.1,*
   (b) *sub-step marking, cf. Def. 5.5.1,*
   (c) *sub-step extension* Ext *is executed iteratively, cf. Rem. 5.6.6.*

△

The proposed propagation framework is defined formally and fulfills the following properties. The proofs showing that these properties hold, are summarised afterwards. Note, we defined the propagation framework $Ppg(TGG)$ to be within the category of attributed (triple) graphs, i.e., it is within a $\mathcal{M}$-adhesive category.

Theorem 5.6.1 (Properties of $Ppg(TGG)$)**.** *The derived propagation framework $Ppg(TGG)$ fulfills the following properties.*

1. *The propagation preserves* ***identity:*** $Ppg(\_, id, \_) = \{id\}$.
2. *If the propagation finishes successfully, then it always yields* ***consistent*** *results: $u' \in Ppg(\_, u, \_) \Rightarrow u'$ is consistent D_i-update for D_i-update u.*
3. *If the given update $u = (M \xleftarrow{u_1} M_P \xrightarrow{u_2} M')$ is consistent, then the propagation yields a consistent result $u' = (M \xleftarrow{u'_1} N^2 \xrightarrow{u'_2} N^3)$ where the resulting models are isomorphic to each other $M' \cong N^3$ (**C-preservation**).*
4. *The propagation always returns a* ***unique result****: $|Ppg(\_, u, \_)| = 1$ for D_i-update u.*
5. ***Functional behaviour****: The propagation terminates, if models and grammars are finite (i.e., we have a finite number of rules, and the model is finite) and if all triple rules of the grammar are relevant (cf. e.g., Def. 5.6.4). Furthermore, the propagation is deterministic, if triple rules are local confluent (Thm 2.43 in [EEPT06], Sec. 2.2.3), i.e., two parallel dependent rules can be applied, so that they lead to the same result. Termination and determinism lead to functional behaviour, i.e., for the same model M, the propagation will always produce the same result.*
6. *As in both sub-steps of* Add *previously deleted elements may be recreated, the proposed propagation framework* ***prioritises creation over deletion****.*

5.7. Proofs & Proof Ideas of Thm. 5.6.1

In the following, we will present the proofs and proof ideas of the properties of the derived propagation framework listed in Thm. 5.6.1.

5.7.1. Proof: Preservation of Identity

Proof. Given a consistent model M w.r.t. a given TGG and an identical update $u = M \hookleftarrow M \hookrightarrow M$. In explicit $u = (u_1, u_2) = (id, id)$ and $id : M \hookrightarrow M$.

Then, the update delta δ_u and also the creating update delta $\underline{\delta}_u$ are empty, i.e. $\delta_u = \underline{\delta}_u = (\varnothing \hookleftarrow \varnothing \hookrightarrow \varnothing)$. Informally, this means that if update u does not change anything, then δ_u ($\underline{\delta}_u$) is empty, because it only reflect changes (creations) in update u.

In order to show that the propagation framework $Ppg(TGG)$ preserves identity, we will apply all steps and show that each step (and sub-step) preserves identity. According to Chap. 5, we will show the preservation on domain D_2, first. The proof for D_1 is dual.

The first step is the Del step: If we consider diagram Fig. 5.9, then an identical update leads to the following situation:

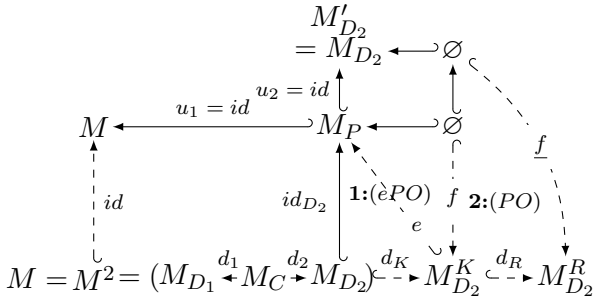
$$\begin{array}{ccccc} M & = & (M_{D_1} & \xleftarrow{d_1} M_C \xrightarrow{d_2} & M_{D_2}) \\ {\scriptstyle id}\uparrow & & {\scriptstyle id}\uparrow \quad {\scriptstyle id_{D_1}}\uparrow & {\scriptstyle id}\uparrow \ \mathbf{1}{:}(PB) & {\scriptstyle id_{D_2}}\uparrow \\ M & & & M_C \xrightarrow{d_2} & M_P \\ & & & {\scriptstyle id_C}\uparrow & {\scriptstyle id_{D_2}}\uparrow \\ M = M^2 & = & (M_{D_1} & \xleftarrow{d_1} M_C \xrightarrow{d_2} & M_{D_2}) \end{array}$$

In the Del step, an identical update u leads to model M_P in domain D_2 which is identic to the original model M_{D_2} in that domain, because if nothing is deleted, then the model stays unchanged. Furthermore, pullback (1) preserves identity, because in general, pushouts and pullbacks in $\mathcal{M}$-adhesive categories preserve identity (cf. Def. 4.2 in [EEGH15]). The resulting model of that sub-step is M.

Afterwards, the consistency creating step (2) will be applied, but due to model M being consistent (see assumption) and the resulting model M^2 of the CC step being equal to M, this CC sub-step will result in the same model M, again. Consequently, the Del step preserves identity.

$$\begin{array}{ccccc} M = M^2 & = & (M_{D_1} & \xleftarrow{d_1} M_C \xrightarrow{d_2} & M_{D_2}) \\ & & & \mathbf{2{:}CC} \Uparrow tr^* & \\ \varnothing & = & (\varnothing & \longleftarrow \varnothing \longrightarrow & \varnothing \ ) \end{array}$$

Now, we will consider Fig. 5.9, again, for the Add step. Note, the resulting model of the Del step is $M^2 = M$.



In the first sub-step of Add, we apply the creating delta $\underline{\delta}_u : \varnothing \hookleftarrow \varnothing \hookrightarrow \varnothing$ of update u to model M_P resulting in model $M'_{D_2} = M_{D_2}$ (top row of the diagram), i.e., the identity, because $M_P = M_{D_2}$ (see above).

The effective pushout construction (1) will lead to $M^K_{D_2}$ which is, again, M_{D_2}. Finally, the pushout (2) leads to $M^R_{D_2}$ being equal to M_{D_2}. For both, we use the property that pushouts and pullbacks in $\mathcal{M}$-adhesive categories preserve identity (cf. Def. 4.2 in [EEGH15]). The resulting triple graphs are $M^R = M$ and $M' = M$.

In the next sub-step of Add, we mark M' and M^R via M^2 according to Def. 5.5.1. All models are identic, i.e., $M = M' = M^R = M^2$. This leads to the marked models $M'_{marked} = M^R_{marked}$, where all elements are marked with **T**. Consequently, the third sub-step Ext of Add is not applicable anymore. Therefore, the propagation framework does not change the original model, but returns the same model.

So, the propagation framework $Ppg(TGG)$ preserves identity: $Ppg(\_, id, \_) = \{id\}$. □ □

5.7.2. Proof Idea: Consistent Results

Proof Idea 5.7.1. *Given a consistent model M w.r.t. a given TGG and D_i-model update $u = (u_1, u_2)$ with $M \xleftarrow{u_1} M_P \xrightarrow{u_2} M'$ in domain D_i ($i \in \{1, 2\}$).*

We will show that, if $u' \in Ppg(\_, u, \_)$ then, u' is a consistent D_i-update for D_i-update u.

The Ext *step extends the model via shifted rules/CC rules. If all markers are set to true, then resulting model M' is automatically consistent, because only shifted rules/CC rules were applied. I.e., $M' \in \mathcal{L}(TGG)$. If no consistent model can be derived, i.e., the application of shifted rules/CC rules will stop, even if some markes are still* **F**, *then the propagation aborts (cf.* Ext*step). That means, we have no result. ⇒ If we get a result, then it is consistent.* □

5.7.3. Proof: C-Preservation

Proof. Given a consistent model M w.r.t. a given TGG and a consistent model update $u = (u_1, u_2)$ with $M \xleftarrow{u_1} M_P \xrightarrow{u_2} M'$. Update $u = (M \xleftarrow{u_1} M_P \xrightarrow{u_2} M')$ being consistent means that $M' \in \mathcal{L}(TGG)$.

In the following, we will show, that if model update $u = (M \xleftarrow{u_1} M_P \xrightarrow{u_2} M')$ is consistent and $u' = (M \xleftarrow{u'_1} N^2 \xrightarrow{u'_2} N^3)$, then $M' \cong N^3$. For that, we will propagate update u on M. We will consider u being an update in domain D_2. The proof for u being an update in domain D_1 is equivalent.

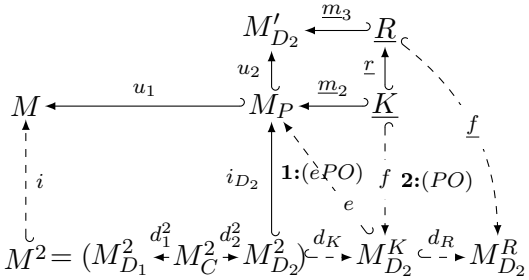
The first step is the Del step: If we consider diagram Fig. 5.9, then applying the deletion part u_1 of the consistent update u lead to the following situation, where M_P is not necessarily consistent.

$$\begin{array}{ccccc} M = (M_{D_1} & \xleftarrow{d_1} & M_C & \xrightarrow{d_2} & M_{D_2}) \\ & & \underline{u}_1 \uparrow & \mathbf{1}{:}(PB) & \uparrow u_1 \\ & & \underline{M}_C & \dashrightarrow & M_P \\ & & & \underline{d}_2 & \end{array}$$

Afterwards, the consistency creating step (2) will be applied leading to consistent model M^2, due to the construction of the $\mathsf{CC}_{\mathsf{Del}}$ sub-step.

$$\begin{array}{ccccc} & & \underline{M}_C & \dashrightarrow & M_P \\ & & i_C \uparrow & \underline{d}_2 & \uparrow i_{D_2} \\ M^2 = (M^2_{D_1} & \xleftarrow{d^2_1} & M^2_C & \xrightarrow{d^2_2} & M^2_{D_2}) \\ & & \mathbf{2{:}CC} \Uparrow & tr^* & \\ \varnothing = (\varnothing & \longleftarrow & \varnothing & \longrightarrow & \varnothing \ ) \end{array}$$

Now, we will consider Fig. 5.9, again, for the Add step.



In applying u_2, model M'_{D_2} is consistent, i.e., ...

In the first sub-step of Add, we apply the creating delta $\underline{\delta}_u : \varnothing \hookleftarrow \varnothing \hookrightarrow \varnothing$ of update u to model M_P resulting in model $M'_{D_2} = M_{D_2}$ (top row of the diagram), i.e., the identity, because $M_P = M_{D_2}$ (see above).

The effective pushout construction (1) will lead to $M^K_{D_2}$ which is, again, M_{D_2}. Finally, the pushout (2) leads to $M^R_{D_2}$ being equal to M_{D_2}. For both, we use the property that pushouts and pullbacks in $\mathcal{M}$-adhesive categories preserve identity (cf. Def. 4.2 in [EEGH15]). The resulting triple graphs are $M^R = M$ and $M' = M$.

In the next sub-step of Add, we mark M' and M^R via M^2 according to Def. 5.5.1. All models are identic, i.e., $M = M' = M^R = M^2$. This leads to the marked models $M'_{marked} = M^R_{marked}$, where all elements are marked

with **T**. Consequently, the third sub-step Ext of Add is not applicable anymore. Therefore, the propagation framework does not change the original model, but returns the same model.

So, the propagation framework $Ppg(TGG)$ preserves identity: $Ppg(\_, id, \_) = \{id\}$.

□ □

5.7.4. Proof Idea: Unique Result

Proof Idea 5.7.2. *Given model M w.r.t. a given TGG and model update $u = (M \xleftarrow{u_1} M_P \xrightarrow{u_2} M')$.*

We will show that the propagation always returns a unique result for an update u: $|Ppg(\_, u, \_)| = 1$ for D_i-update u.

The propagation returns exactly one result: Follows out of the Ext *sub-step: We do not backtrack, i.e., if the* Ext *sub-step finishes successfully, i.e., all markers are set to* **T**, *then it returns this result u' without trying to derive another solution.*

The result is unique: This follows out of the property "propagation returns exactly one result" and out of the property "functional behaviour" Proof Idea 5.7.3. □

5.7.5. Proof Idea: Functional Behaviour

Proof Idea 5.7.3. *Given model M w.r.t. a given TGG and model update $u = (M \xleftarrow{u_1} M_P \xrightarrow{u_2} M')$.*

We will show functional behaviour, similar to the proof in [HEOG10]. As sub-steps for functional behaviour, we need to show:

1. *Termination: If TGG and models are finite and rules are relevant (i.e., $LHS \neq RHS$). Proof is similar to proof of termination of forward translation / backward translation in model synchronisation framework in [HEOG10] and [EEHP09].*
2. *Determinism: Definition of confluence from [EEPT06] need to be applied to construction, similar to [HEOG10] and [EEHP09].*

□

5.7.6. Proof Idea: Priorisation of Creation over Deletion

Proof Idea 5.7.4. *Given model M w.r.t. a given TGG and model update $u = (M \xleftarrow{u_1} M_P \xrightarrow{u_2} M')$.*

We will show that creation has higher priority than deletion.

This property follows directly out of the construction of the derived propagation framework of model updates in Chap. 5. □

Case Study at SES

6

In Chap. 1 we already gave an introduction to the case study with our industrial partner SES and to the running example which was taken from this case study. In this chapter, we will present the outcomes of the practical part of the PhD research project. In short, they are:

- A visual environment for SPELL-Flow as Eclipse plugin [ecl16a].
- A prototype translation from a subset of SPELL statements to SPELL-Flow.
- A plugin in Eclipse for an automated translation, i.e., in executing this functionality, all steps of the translation are performed automatically.

In Sec. 3.3, we gave a detailed introduction in the unidirectional translation from SPELL to SPELL-Flow and the bidirectional translation between SPELL and SPELL-Flow by means of the methodology which was also developed and presented in Chap. 3. The practical implementation of the unidirectional and bidirectional translation between SPELL and SPELL-Flow follows this methodologies directly.

6.1. Implementation

The industrial project in cooperation with SES was part of the PhD project. The tasks of the applied part were to develop an automated prototype translation from SPELL to SPELL-Flow (and vice versa) using TGGs and as supplement to the translation, to develop an Eclipse plugin which performes all translation steps automatically. Moreover, a tool for visualising the resulting SPELL-Flow model was implemented which is based on Eclipse GMF. In the following section, we will present the implementation details of the tasks mentioned above.

6.1.1. Guidelines for the Translation set up by SES

At the beginning of the industrial cooperation, requirements were set up by SES, which need to be fulfilled by the automated translation and also by the desired visualisation. The first requirement was that the resulting visualisation of SPELL source code shall be a flowchart which is adapted to the needs of SES, i.e., some special statements get special shapes in order to attract more attention of the satellite operator. Table 6.1 lists the specification of node shape types and their corresponding SPELL statements that was set up by SES.

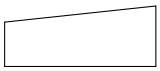
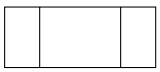
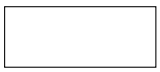
Node Type	**Shape**
start node	
end nodes (finish, abort, end)	
step node	
branching nodes (if, while, for, try)	
elif nodes	
gettm and verify nodes	
prompt node	
goto node	
expression node	
function call node	
other nodes not mentioned explicitly	

Table 6.1: Node types as defined by SES

Another specification which was set by SES is that the SPELL-Flow model shall be divided into different hierarchies. Thus, the (possibly) complex SPELL-Flow model will be divided into different abstraction layers in order to separate the most important information from less important ones and in order to keep the first view of the SPELL-Flow model as concise as possible.

We will now list the rules for the separation of information into different hierarchical layers of the SPELL-Flow model as it was provided by SES in the following enumeration. In addition, some information will be omitted during the translation. The rules for defining which statements in which context will be excluded, are contained in that list, too.

1. The translation from SPELL to SPELL-Flow shall preserve as much information as possible, i.e., on the most detailed level, the SPELL-Flow model shall be nearly identical to the SPELL source code. Still, many SPELL procedures contain a list of initialisations at the beginning apparent from IVARS and ARGS keywords.

2. The SPELL-Flow model shall fullfil the mapping from SPELL statements to SPELL-Flow shapes as given in Table 6.1.

3. Comments shall not be represented by a separate shape but may be included in tooltip of the corresponding statement, as well as the corresponding source code snippet.

4. The different hierarchies shall be built up according to the following rules:

   (a) Each "branching" statement shall be on the first layer but if and only if it is situated on the first indentation level. Branching statements are: if, if/elif conditions, loops (for, while), try/except blocks, and goto statements.

   (b) Each Step statement shall appear on the first layer if and only if it is on the first indentation level.

5. For underlying layers, i.e., layer > 1: If nodes of the same type follow each other, then they shall be merged into one shape. This will lead to a new layer which is below of the layer with the merged shapes which will show all nodes as separate shapes.

6. Function calls will be treated differently, depending on the fact, if it calls build-in code or if the underlying source code is available. If the source code is available, then a link shall be established between the node representing the function call and the block which represents the body of the function. If the function call calls built-in code, then the link to the function body shall be omitted.

7. Expressions shall be converted into "pretty Python" in order to increase the readability, e.g., expressions like A == B shall be replaced by A = B or lists (usually surrounded by many brackets) shall be replaced by an enumeration.

6.1.2. Model Synchronrisation via TGGs and GGs

The unidirectional model transformation from SPELL source code to a SPELL-Flow model as well as the bidirectional model transformation which includes also the backward direction are based on the model synchronisation framework that is based on triple graph grammars Sec. 2.2 and that was introduced formally in Chap. 4. Practically, the unidirectional and bidirectional model transformation is performed using HenshinTGG. For the refactoring steps, we use flat algebraic graph transformation (cf. Sec. 2.1.3) that will be applied in practice using Henshin.

In the following section, we will introduce the Multi-View Henshin-Editor and HenshinTGG, so that we are able to give details on the graph grammars and the triple graph grammar that were developed for realising a prototype transformation. This description includes some explicit example rules taken from all grammars.

Multi-View Henshin-Editor

Henshin is a development environment based on Eclipse EMF and GEF. It supports the visual modelling of EMF-based rules and the execution of model transformations using those EMF rules [Ecl16b]. The integrated graphical interface of the Henshin development environment can be replaced by the *Multi-View Henshin-Editor* [Hen16b]. In the framework of this project, we used this alternative. This visual multi-view editor was developed in 2010 at the Technische Universität Berlin. It is based on the Muvitor framework which provides a multi-view interface, i.e., editors using Muvitor provide a tree-view of the transformation system, as well as different graphical views for rules and graphs of the transformation system. Another special feature of Muvitor is that the graphical views include different views of the rule: one for the LHS and one for the RHS, and also one for the application conditions (ACs) of a rule that are displayed at the same time, i.e., rules are visualised simultaneously by two views or even by three different views, if the rule contains ACs.

The Multi-View Henshin-Editor supports the following formal frameworks and techniques:

- Definition of typed attributed graphs and typed attributed graph rules,
- Typed attributed graph rules can contain (complex) application conditions (cf. Sec. 2.1.5)
- In-place typed attributed graph transformation (cf. Sec. 2.1.3), i.e., the transformation is executed on the same graph directly and changes the graph.

- The graph transformation can be structured by transformation units (cf. Sec. 2.3), whereas the smallest transformation unit is the graph rule. Depending on the type of the transformation unit, it may contain one or more other units, i.e., units can be nested [ABJ+10].

HenshinTGG

HenshinTGG is an extension of the Henshin-Editor [Hen16a] to be able to perform triple graph transformation, an exogenous transformation (or out-place transformation). HenshinTGG is used for unidirectional model transformations from one model to another model, bidirectional model transformations, model integration, concistentcy checks of the triple graph, incremental model synchronisation (delta-based and state-based model synchronisation). It is based on the formal background which we introduced in Sec. 2.2 and Chap. 4.

HenshinTGG supports the following formal frameworks and techniques [EHGB12]:

- Specification of triple rules that may contain negative application conditions (NACs), and defintion of triple graphs.
- Generation of the following types of operational rules, i.e., forward (FT), backward (BT), consistentcy creating (CC) and integration rules (IT).
- Execution of exogenous model transformations, i.e., forward transformations and backward transformations using FT and BT rules, respectively. Execution of model integrations using IT rules and performing concistency checks of a triple graph using CC rules.
- Execution of incremental model synchronisations. HenshinTGG supports delta-based and state-based forward and backward transformations [Kir14].

In the framework of the industrial project in cooperation with SES, HenshinTGG needed to be extended by a basic implementation of sequential units in order to determine execution sequence of the triple rules or operational rules of the triple graph transformation. As illustrated in the pseudo code in Listing 6.1, the existing implementation in HenshinTGG allows the application of all forward rules in one list, which will be executed from top to bottom. Each rule will be applied as long as possible. After the list is processed, the algorithm starts again to iterate through the list. This is done as long as possible, i.e., as long as a forward rule is still applicable.

```
1 terminate = false
2 while(!terminate)
```

```
  terminate = true
  for(r : FT-rules)
    while(r is applicable)
      apply r
      terminate = false
```

Listing 6.1: Existing application of FT-rules in HenshinTGG

We extended the implementation in HenshinTGG by a very limited implementation of *sequential units*. They allow us to structure the application order of all forward rules more precisely. The effect of the *sequential unit* is given in pseudo code Listing 6.2. Sequential units usually contain a list of rules or sub-units. This list will be iterated once and it will be checked if the current rule or sub-unit is applicable. If it is applicable, it will be applied temporarily. Otherwise, the whole previous application sequence of that sequential unit will be rolled back. If the whole sequence was applicable, then, the temporary graph will be returned, i.e., the whole application sequence is reflected in the current graph.

```
temp = current triple graph
for (r : FT-rules or sub-units in sequential unit)
  success = false
  while(r is applicable)
    apply r to temp
    success = true
  // a rule / sub-unit was not applicable: rollback
  if (!success) return current triple graph
// successful application of whole sequence
return temp
```

Listing 6.2: Application of FT-rules in case study

Realisation

First, we will consider Fig. 1.2 from Chap. 1 in which we illustrated a scheme which shows the correlation of an instance, a model and the corresponding (meta-)meta-models. This scheme can be directly applied to the industrial case study, which is shown in Fig. 6.1.

In the SPELL domain, the instance is given by the SPELL source code files. A SPELL source code files are described by the cooresponding SPELL abstract syntax graph (SPELL ASG), which is the model. The SPELL ASG conforms to the SPELL grammar, i.e., it is typed over the SPELL grammar. The SPELL grammar conforms to the EMF specification.

The similar case holds for the SPELL-Flow domain. Each SPELL-Flow visualisation, which is stored in an XMI file, is an instance. Each SPELL-Flow visualisation is represeted by a SPELL-Flow abstract syntax graph

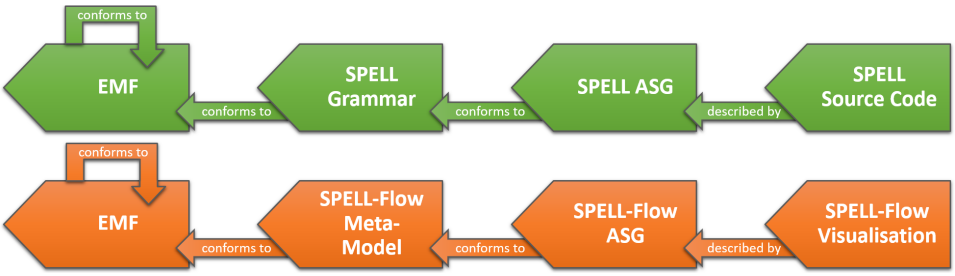


Figure 6.1: Models and Meta-Models in SPELL-2-Flow

(SPELL-Flow ASG). Each SPELL-Flow ASG conforms to the SPELL-Flow meta-model. The latter conforms to the EMF specification. In both cases, the EMF specification conforms to itself, too.

In detail, this means that for SPELL and SPELL-Flow EMF meta-models were specified. For SPELL, we reused the meta-model that was developed for the PIL2SPELL project [HGN+13] and modified it slightly (mainly with regard to NEWLINE nodes, the structure of comments and by internal elements that help to store the corresponding source code and line numbers in the SPELL instance graph (which is the SPELL ASG)). The full meta-model in Xtext syntax as well as the corresponding EMF model are provided in Appendix A.1 and A.2. The SPELL-Flow model was created using EMF. It is much more slimmer than the SPELL grammar. We refer to Sec. 6.2 for details on the complexity of all meta-models that were used with regard to their number of nodes. The full SPELL-Flow grammar is attached to Appendix A.3. Furthermore, two correspondence models were created: the CORR meta-model defines nodes that mediate between the SPELL model and the SPELL-Flow model (cf. Appendix A.4). In contrast, meta-model CORRFlow2Flow can be seen as a helper meta-model for the flat graph grammars: Its source and target meta-model is the SPELL-Flow meta-model (cf. Appendix A.5).

Graph Grammar	**Meta-Model**			
	SPELL	SPELL-Flow	CORR	CF2F[1]
SPELL-2-FlowFlat	x	x	x	-
Refactor_SPELL-Flow	-	x	-	x
SPELL-Flow-2-Hierarchy	-	x	-	x

Table 6.2: Meta-models in graph grammars

For realising a prototype translation from SPELL source code files to SPELL-Flow models, the specification of three graph grammars was necessary:

[1] Abbreviation CF2F stands for CORRFlow2Flow.

1. Triple graph grammar SPELL-2-FlowFlat for executing the model transformation from SPELL ASGs to SPELL-Flow ASGs.

2. Flat graph grammar Refactor_SPELL-Flow for refactorings.

3. Flat graph grammar SPELL-Flow-2-Hierarchy for the transformation into a model containing the demanded hierarchy structure.

Table 6.2 gives an overview of all three grammars and the meta-models that were used in each grammar. The triple graph grammar uses the SPELL meta-model, the SPELL-Flow meta-model and the correspondence model CORR, because it translates from the SPELL AST to a (flat) SPELL-Flow AST. In contrast, both refactoring grammars use the SPELL-Flow meta-model and the helper model CORRFlow2Flow, because both change the SPELL-Flow AST.

In the following paragraphs, we will describe all three graph grammars in detail and illustrate some example rules from all three graph grammars, in order to give an overview of the structure of the translation and also of the complexity of the rules. We will start with a small set of triple rules and the corresponding forward translation that were generated out of this triple rules. In some cases, the generated forward rules needed manual modification, especially in cases where those rules contained filter NACs (cf. Def. 2.2.14). Afterwards, we present some flat graph transformation rules from the graph grammar Refactor_SPELL-Flow. We finish this part with some example graph transformation rules from the flat graph grammar SPELL-Flow-2-Hierarchy.

Translation with SPELL-2-FlowFlat The translation from a SPELL ASG to a SPELL-Flow ASG is executed using the forward translation (FT) rules that are derived out of the triple rules of the triple graph grammar (TGG) SPELL-2-FlowFlat.

Fig. 6.2 shows a screenshot of the FT rules of our TGG in HenshinTGG for the case study. The uppermost folder *FTRuleFolder* is the main folder for all forward translation rules and units. In our case, it contains the sequential unit folder *FT_SequentialFolder* on top. This unit again contains the given transformation units that we described above in Sec. 6.1.2 and especially in Listing 6.1.

In our case study, we divided the application of the forward rules in the following steps, wheras the content of the uppermost unit, i.e., the sequential unit FT_SequentialFolder is only iterated once, i.e., we do not repeat the application of all underlying units.

FT_SequentialFolder.

- *PreProcessing*: This sub-unit contains rules which execute preprocessing steps, e.g., some helper nodes will be created, and the topmost

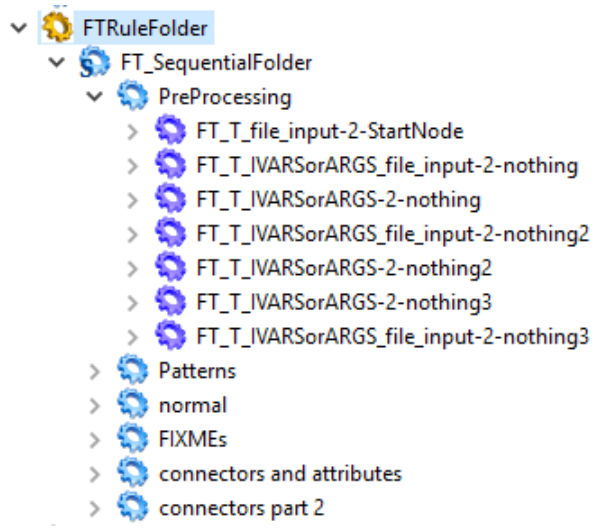


Figure 6.2: Screenshot from HenshinTGG: structuring FT-rules

parent node of the SPELL instance graph (file¡nput) is translated into a StartNode and a RootNode in the SPELL-Flow model. The RootNode is the main container of all nodes in the SPELL-Flow model. It won't be visualised. The StartNode is the starting node of the diagram and will be shown in the SPELL-Flow model. Furthermore, IVARS and ARGS statements that are at the beginning of the source code will be translated into nothing, i.e., the translation markers will be changed from **F** to **T**.

- *Patterns*: Patterns are larger structures. After the pre processing phase is finished patterns will be translated, because they may contain nodes that will be translated separately by the next sub-unit. Due to this sub-unit, we will prevent undesired results.

- *normal*: This is the largest sub-unit containing the main part of the forward translation rules.

- *FIXMEs*: We already mentioned above, that the current state of the case study is only a prototype, i.e., we are not able to translate all existing SPELL statements. In case the SPELL source code which we want to translate contains SPELL statements which do not already have a corresponding rule / set of rules, then we will translate this statement into a FIXME node in order to show, which SPELL statement has no specific triple rule in our TGG.

- *connection and attributes*: Finally, we will restore connectors in the SPELL-Flow domain (target domain) and translate attributes which are not translated, yet.

- *connectors part 2*: For the restoration of connectors, we need a second sub-unit which will try to restore edges that could not be restored at the stage of the previous sub-unit. The last two sub-units contain similar rules, because if we remember the description of sequential units (also cf. Fig. 6.2), sequential units iterate through each sub-unit only once. In our case study, we need to iterate through this set of rules twice, so we "duplicated" the previous sub-unit.

As a result of the translation phase, a SPELL-Flow abstract syntax graph will be returned. The ASG contains the whole corresponding SPELL-Flow model in largest detail, without any abstractions into hierarchies. This will be done in the next phase, the refactoring phase.

Example 6.1.1 (FT Rules (& Triple Rules) from SPELL-2-FlowFlat)**.** *We will now give examples of some forward translation rules and their corresponding triple rules. Note, we will only describe the FT rules in explicit. For triple rules it holds that they are applied in all three domains at the same time, i.e., they are used for building up the triple graph. This means also, that they do not include translation markers, but instead, all elements to be created are marked with* $< ++ >$*, whereas all other elements are unmarked (cf. Sec. 2.2).*

Each SPELL AST has node : file_input *as topmost node, i.e., it is the root node of each SPELL AST. This node is the topmost container of all other nodes of that SPELL AST. The cause of this property can be found within the Xtext source file of SPELL, which is provided in explicit in Appendix A.1. We repeat the line in question in Listing 6.3. It is observable that* : file_input *is defined to be the first node which is addressed in the Xtext grammar. No other node calls* : file_input*, wheras* : file_input *calls two nodes:* : NEWLINE *and* : stmt_LST_Elem*. Starting from the latter node, all other nodes are called.*

```
6 file_input: {file_input} nl=NEWLINE? (fst=stmt_LST_Elem)?;
```

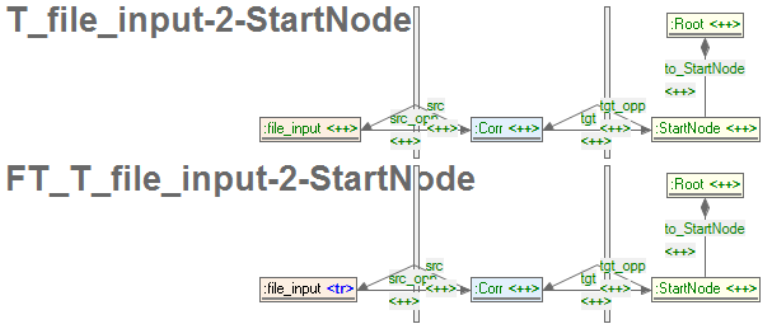


Figure 6.3: Triple rule T_file_input-2-StartNode (top) and its forward rule (bottom)

Listing 6.3: Snippet of line 6 from SPELL Xtext grammar

Due to the fact that : file_input *is always the root node in the SPELL AST and also in considering Fig. 6.2, i.e., the forward rule* FT_T_file_input-2-StartNode *(Fig. 6.3 (bottom)) is the first rule within the first sub-unit of* FT_SequentialFolder*, this forward rule is usually the first FT rule that is executed. It translates a* : file_input *node to a* : Root *node that contains a* : StartNode *in the SPELL-Flow AST (cf. screenshot on bottom of Fig. 6.3). The* : Root *node will be the topmost node in the SPELL-Flow AST which contains all other nodes, i.e., it is the root node.* : StartNode *is the start node of the SPELL-Flow flow chart diagram, which appears only on the first layer. The FT rule also creates a correspondence part which consist of one node* : Corr *and edges* : src, : src_opp, : tgt *and* : tgt_opp*. The corresponding triple rule is also shown in Fig. 6.3 (top).*

The next forward rule is illustrated in Fig. 6.4 (bottom) and is named FT_T_fst_stmt_LST_Elem_file_input_stmt-2-firstElement_StartNode_Activity_Root*. It assumes that nodes* : file_input *and* stmt_LST_Elem *in the SPELL domain are already translated into nodes* : StartNode *that is contained by a* Root *node and an* : Activity *node in the SPELl-Flow domain. Two* : Corr *nodes in the CORR domain and their required edges need to be available, too. Edge* : fst *in the SPELL domain is still untranslated in the SPELL AST. Elements that are already translated as prerequisite are marked with a green* [tr]*. Elements that will be translated with this FT rule are denoted with a blue* < tr > *marker. In applying this FT rule, edge* : fst *will be translated into two edges:* : firstElement *as connection from* : StartNode *to* : Activity *and containment edge* : to_Element *from node* : Root *to node* : Activity*. Note, code* : Activity

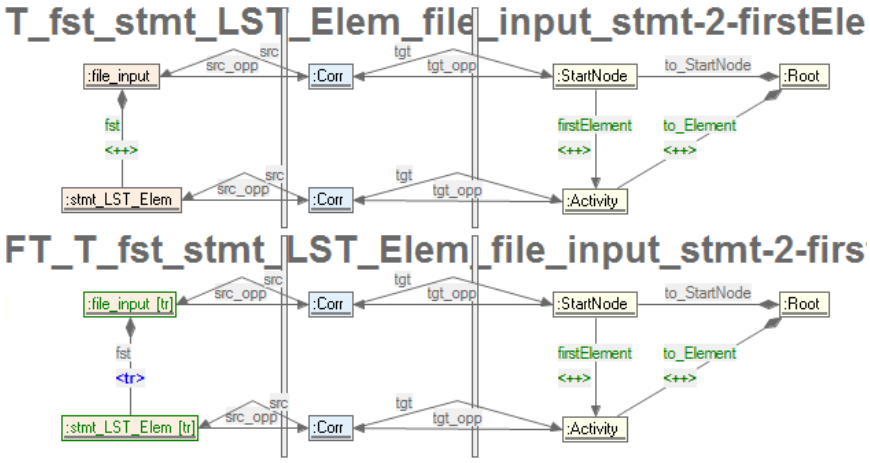


Figure 6.4: Triple rule T_fst_stmt_LST_Elem_file_input_stmt-2-firstElement_StartNode_Activity_Root (top) and its forward rule (bottom)

is the parent node of a variety of children nodes. We refer to the ECORE-model in Appendix A.3 to verify this relationship. This means that the FT rule is able to find a match also to kinds of children nodes of Activity*, i.e., we do not need to write rules for each kind of children node. The triple rule out of which this FT rule is generated, is also provided in Fig. 6.4 (top).*

Forward translation rule FT_T_IVARSorARGS_file_input-2-nothing *in Fig. 6.5 is interesting from several points of view: First, it is an example FT rule for one rule that only changes markers from from* **F** *to* **T** *without cre-*

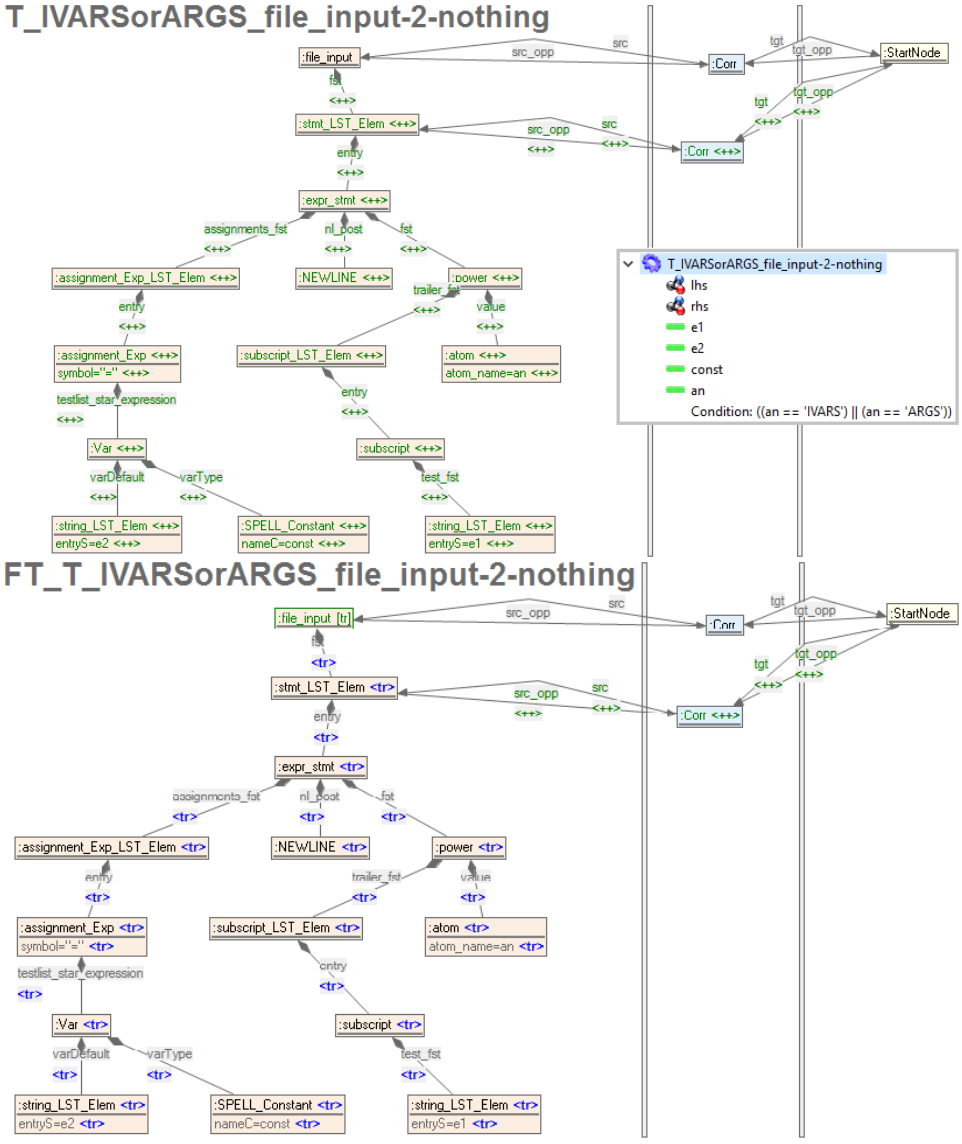


Figure 6.5: Triple rule T_IVARSorARGS_file_input-2-nothing (top) and its forward rule (bottom)

ating anything in the SPELL-Flow domain in the forward translation step, because it is one FT rule that omits the IVARS and ARGS statements that are at the beginning of the source code. Nevertheless, this FT rule creates a : Corr *node in the correspondence part and the necessary edges. Secondly, this FT rule can be also seen as pattern, because it translates a larger block in the SPELL AST. This block will always have the same structure. Different values of attributes of the several nodes in the SPELL domain are handled by means of rule parameters. This is possible for this FT rule because each manifestation of that block shall be processed the same way, i.e., it shall be excluded. Thirdly, this FT rule (and also the corresponding triple rule) contains an attribute condition. The attribute condition for this rule is* $((\mathsf{an} ==' \mathsf{IVARS}')||(\mathsf{an} ==' \mathsf{ARGS}'))$*, wheras* an *as a parameter that will be evaluated in the matching process. If attribute* atom_name *of node* : atom *in the SPELL domain holds value* 'IVARS' *or* 'ARGS'*, then this FT rule is applicable. Otherwise, do not apply this FT rule. The corresponding triple rule is shown on top of Fig. 6.5.*

The next rule which we want to discuss is the forward translation rule FT_T_NEWLINE_expr_stmt-2-empty_POST *which is visualised in Fig. 6.6 (bottom). This FT rule contains a filter NAC (cf. Def. 2.2.14). If we con-*

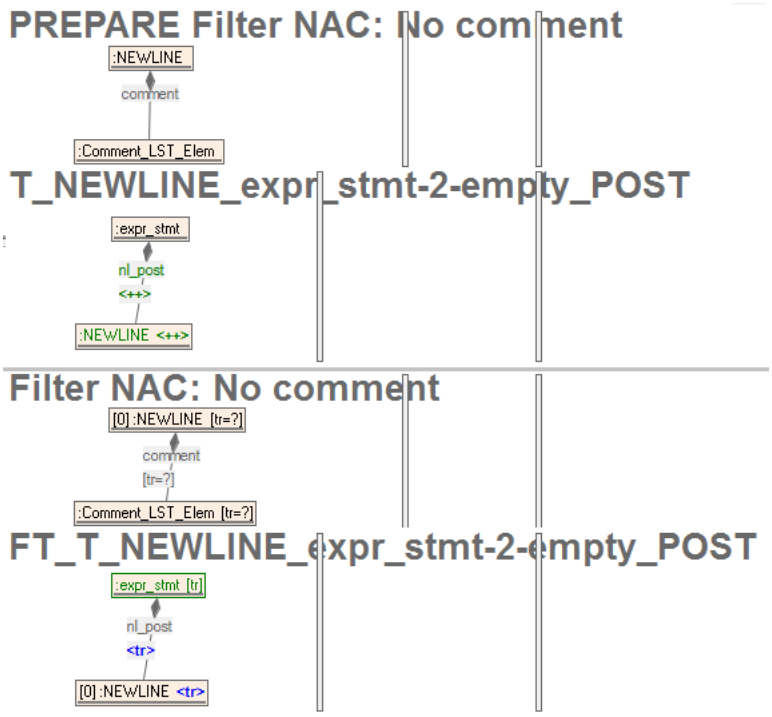


Figure 6.6: Triple rule T_NEWLINE_expr_stmt-2-empty_POST (top) and its forward rule (bottom)

sider the corresponding triple rule T_NEWLINE_expr_stmt-2-empty_POST*, then we observe that the NAC has no mapping. In the the case without any mapping this would mean, that it will be checked, if none of all* : NEWLINE *nodes in the instance graph is connected to a* : Comment_LST_Elem *node via edge* : comment *in the SPELL domain. This is not desired. Instead, we want a NAC for the forward translation that checks, if the* : NEWLINE *node to be translated is not connected to a* : Comment_LST_Elem *node via edge* : comment*. This cannot be checked in the triple graph transformation step, because node* : NEWLINE *is not available in the matching process and this case cannot occur. Consequently, for the triple graph transformation and also for all other operational rules, we do not want to have a NAC. Therefore, after generation of the FT rule out of the triple rule, manual editing is necessary in which the correct mapping is set. For all other operational rules (and also for the triple graph transformation case), this NAC shall be deleted manually. Note also, that this FT rule will only change markers in the SPELL domain, wheras the correspondence and SPELL-Flow domain stay unchanged. The generated backward translation rule (BT rule) will be identical to the triple rule, i.e., it will create elements in the SPELL domain but will do nothing in all other components. For the backward transformation, this BT rule will result into an endless loop, because it is always applicable. So, it needs to be taken care that this BT rule is not part of the backward transformation process.*

The last FT rule which we illustrate explcitly is FIXME_untranslated_Root *in Fig. 6.7. It belongs to transformation unit* FIXMEs *and translates nodes in the SPELL domain that there not translated in that stage into an* : OtherActivity *node that holds the specified value* FIXME : Untranslated Element *for attribute* description*. Furthermore, the line numbers and the corresponding source code text is taken over to the* : OtherActivity *node in the SPELL-Flow domain. Listing 6.4 shows the JavaScript code which is assigned to attribute* sourceCode *in the newly created node* : OtherActivity*. In line 2, a local variable* returntext *is created*

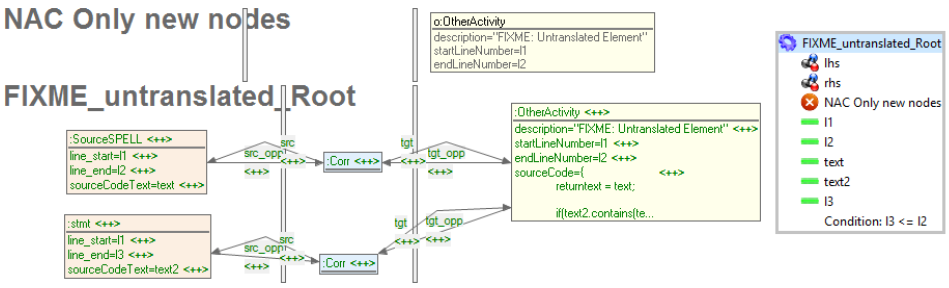


Figure 6.7: Triple rule FIXME_untranslated_Root (top) and its forward rule (bottom)

and the value of attribute sourceCodeText *of node* : SourceSPELL *is assigned to this variable. Afterwards (lines 4-6), it is checked, if attribute* sourceCodeText *of node* : stmt *contains the same or a bigger source code snippet that includes the other source code snippet. If it is true, then the source code snippet of* : stmt *is assigned to variable* returntext. *Otherwise, nothing is done. Finally, the value of* returntext *is returned, so that it can be assigned to attribute* sourceCode *of node* : OtherActivity. △

```
1 {
2   returntext = text;
3
4   if(text2.contains(text)) {
5     returntext = text2;
6   }
7   returntext;
8 }
```

Listing 6.4: JavaScript in sourceCodeText attribute

In Ex. 4.1.8 we already introduced a subset of reduced forward translation rules. The extended version of these FT-rules is also part of the set of FT-rules in the SEPLL-2-FlowFlat triple graph grammar.

Refactoring Phase with Refactor_SPELL-Flow The refactoring phase is defined within the plain graph grammar Refactor_SPELL-Flow using plain graph transformation. The grammar is created with the help of Henshin and will be executed using the Henshin engine for plain graph transformation (cf. Sec. 6.1.2).

In Henshin, it is possible to use different kinds of transformation units (cf. Sec. 2.3). The screenshot in Fig. 6.8 shows the main transformation unit *Main*, which is a sequential unit containing three sequential units: *Pre-Processing* (Fig. 6.9), *Refactoring* (Fig. 6.10) and CleanUp (Fig. 6.11).

During the forward transformation using triple graph grammar SPELL-2-FlowFlat, each : Activity might be equipped with a list of : Argument nodes that reflect the list of arguments that were available in the SPELL ASG. For displaying the SPELL-Flow model using the SPELL-Flow visualisation tool, it is necessary to merge those : Argument nodes into one attribute being part of the parent : Activity node. The first set of transformation units within the pre-processing sub-phase deals with the merging of the list of : Argument nodes of : Step nodes, because they always have the same structure, i.e., exactly two arguments, the step number and step description. In addition, empty : Argument nodes will be simply removed.

The second set of transformation units in the pre-processing unit equip all : Argument, : Activity, and : Expression nodes with : Helper nodes that are necessary to perform the next step, the *Refactoring* sub-phase. The

structures for equipping the SPELL-Flow model with helper nodes is available due to the use of *CORRFlow2Flow* meta-model within this flat graph grammar. The meta-model is visualised explicitly in Appendix A.5.

In the refactoring sub-phase, all remaining : Argument nodes will be merged into one node starting with the last node and adding its value to the value of the predecessor node until the list of nodes is reduced to one : Argument node. In processing the list of arguments from bottom to top, we are able to keep the correct order of arguments. For merging the : Argument nodes, the helper structures from meta-model *CORRFlow2Flow* are used. Furthermore, : Comment nodes as well as nodes describing a "FIXME" structure are merged in a similar way.

In order to draw a connection between the nodes in the final SPELL-

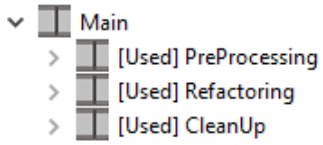


Figure 6.8: Screenshot from Henshin: Main Transformation Unit

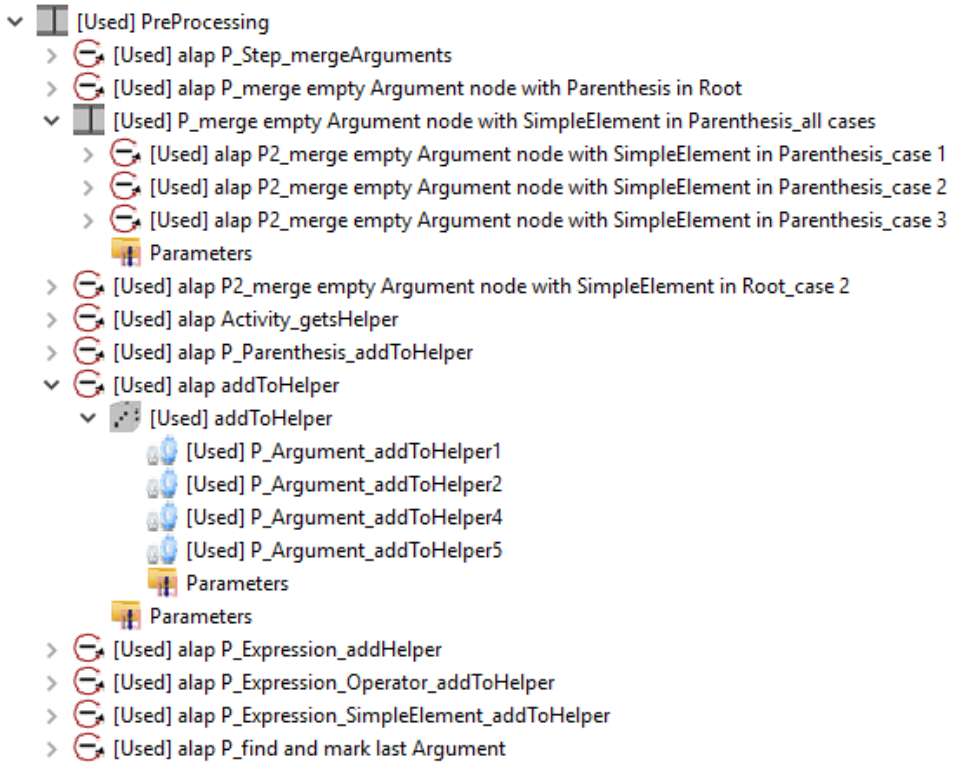


Figure 6.9: Transformation Unit for Preprocessing Sub-Phase

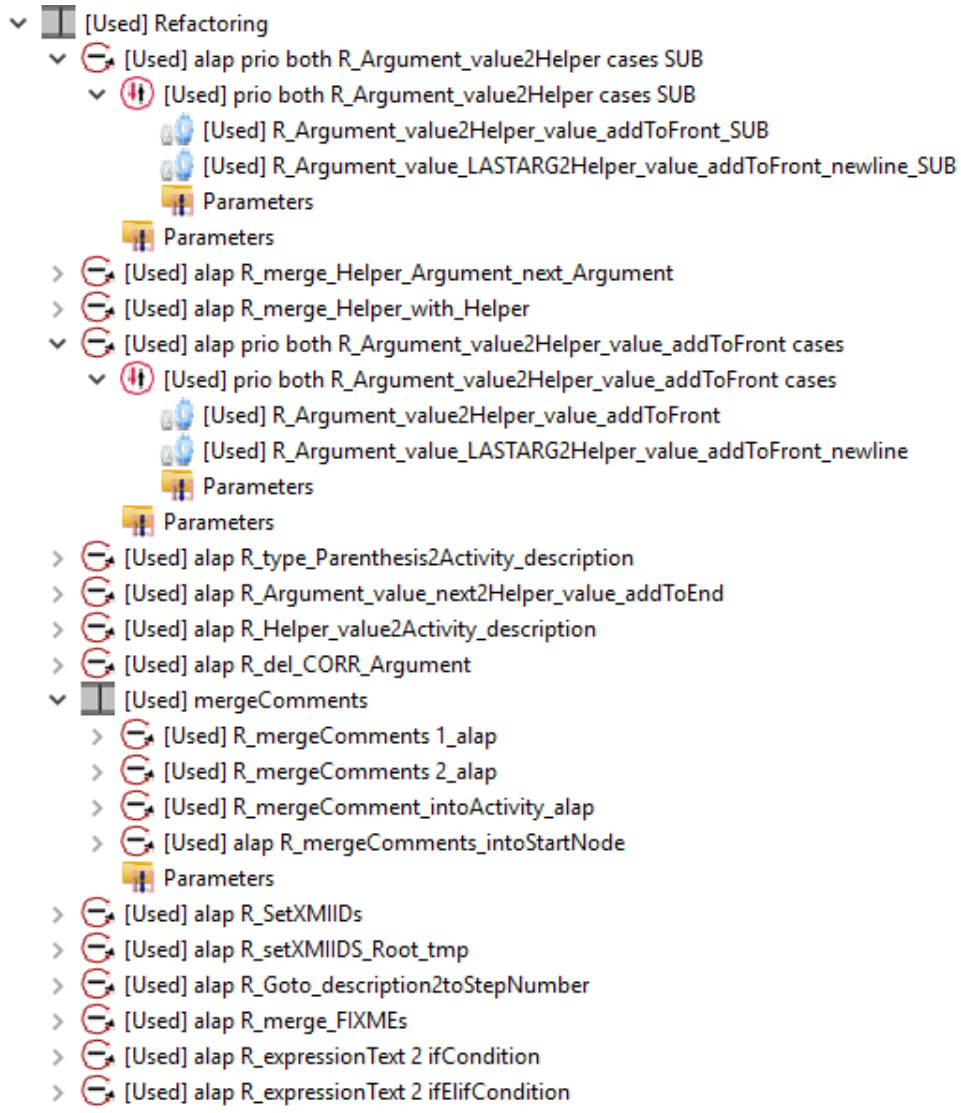


Figure 6.10: Transformation Unit for Refactoring Sub-Phase

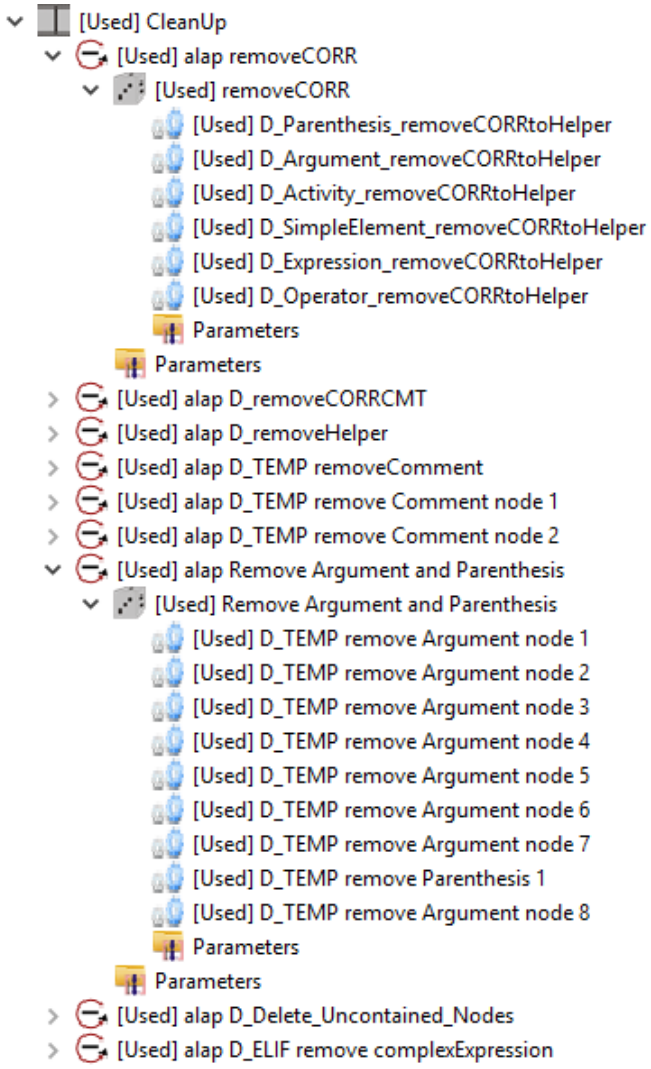


Figure 6.11: Main Transformation for Clean Up Sub-Phase

Flow model and the design elements that will be created in the last step of the whole translation: the enrichment by design elements (cf. Sec. 6.1.4), each node shall be equipped with a unique id. This id is already defined by default as attribute xmiid for each node by EMF. Thus, it is not explicitly defined as attribute within the meta-models (cf. Appendix A.1 and A.3 to A.5). The unique id is also generted within the refactoring sub-phase.

The last sub-phase is called *CleanUp* and removes all helper structures as well as all nodes that are not contained by the root node anymore from the SPELL-Flow model.

Example 6.1.2 (Rules from Refactor_SPELL-Flow)**.** *We will now present some plain graph rules in explicit from the graph grammar* Refactor_SPELL-Flow.

The first graph rule that we want to illustrate explicitly is named P2_merge empty Argument node with SimpleElement in Parenthesis_case 1 *and is given in screenshot Fig. 6.12. It removes an empty* : Argument *which is situated between two* : SimpleElement *nodes being both contained by a* : Parenthesis *node. The check for emptyness of the node is executed using the following attribute condition:*

```
1 if(a.getValue() == null)
```

Listing 6.5: Attribute condition for parameter a

The node : Argument *is assigned to parameter* a. *In calling* a.getValue() *the current value of the attribute named* value *is fetched. It will be checked if it is empty, i.e., if it is* null.

The second graph rule P2_merge empty Argument node with SimpleElement in Parenthesis_case 2 *(Fig. 6.13) is applied, when the previous one is not applicable anymore. It merges a* : SimpleElement *node that follows an* : Argument *node and that are both contained by a* : Parenthesis *node, but only if the* : Argument *node is empty. The emptyness is again checked by an attribute condition that is identical to the one given in Listing 6.5. The negative application condition (NAC) forbids the application of that graph rule, if* : SimpleElement *is followed by another node, i.e., this rule is only applicable, if* : SimpleElement *is a leaf node. The merge is performed in copying the value of attribute* valE *of* : SimpleElement *into the attribute* value *of node* : Argument *and in deleting the* : SimpleElement *node at the same time.*

Plain graph rule P2_merge empty Argument node with SimpleElement in Parenthesis_case 3 *(Fig. 6.14 is applied if the previous two rules are not applicable anymore. It merges a* : SimpleElement *node which is followed by an arbitrary* : Element *node into its predecessor* : Argument *node. The merge is performed, only if the* : Argument *node is empty. Again, this is checked by an attribute condition, which is identical to Listing 6.5.*

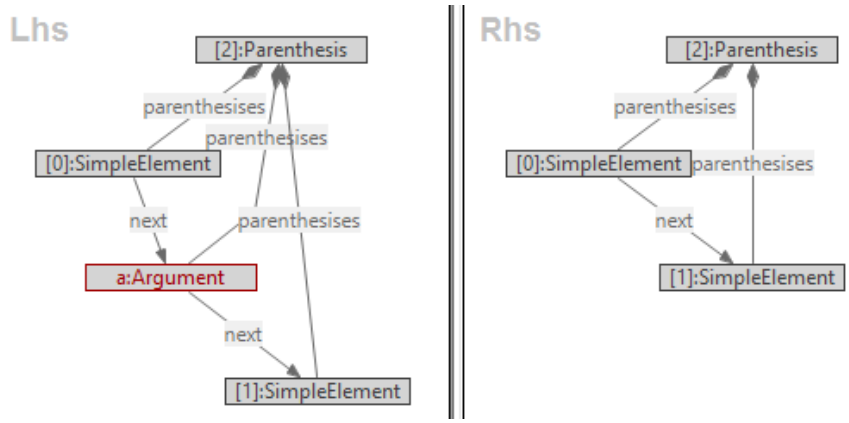


Figure 6.12: Rule P2_merge empty Argument node with SimpleElement in Parenthesis_case 1

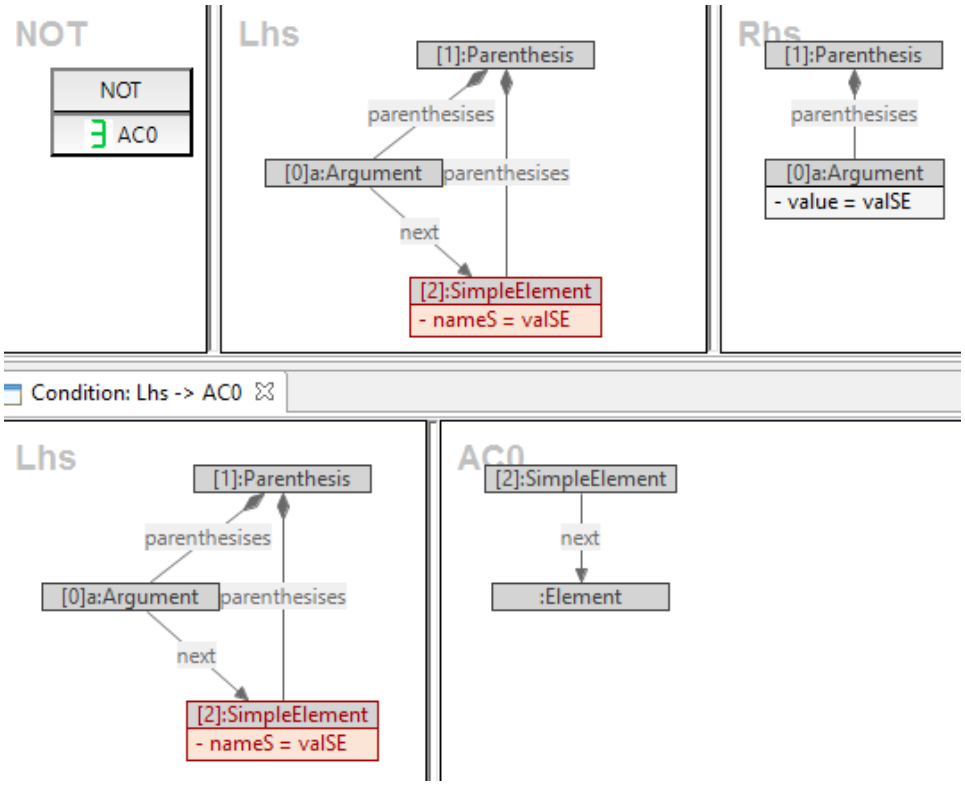


Figure 6.13: Rule P2_merge empty Argument node with SimpleElement in Parenthesis_case 2

The enrichment of helper structures that is also part of the pre-processing sub-step is visualised on the basis of graph rule P_Activity_getsHelper *Fig. 6.15. The helper structure is given by two* :CORR *nodes and one*

: Helper *node that connect two nodes that belong to each other. In the given rule, an* : Argument *node belongs to an* : Activity *node which is indicated by edge* arguments. *The enrichment shall be executed only once for each combination of nodes. Therefore, a NAC is necessary which is equal to the RHS of the rule.*

The next graph rules that we want to discuss are part of the next subphase that is called Refactoring.

Rule R_Argument_value_next2Helper_value_addToEnd *(Fig. 6.16) copies the value of an* : Argument *node into the corresponding* : Helper *node. In addition, it removes the connection between this* : Argument *node and its* : Helper *node in deleting the connecting* : CORR *node with its edges. The copying of attribute* value *from* : Argument *to attribute* value *or node* : Helper *is executed using the JavaScript script assigned to attribute* value *of the* : Helper *node. Due to readability, we croped the script in Fig. 6.16. Instead, we listed and formatted the script in Listing 6.6. Line 2 defines the variable* tmp. *The condition in line 3 checks, if attribute* value *of node* : Helper *(*h.getValue()*) is empty. If this is true, then the value* val *assigned to attribute* value *of node* : Argument *is assigned to* tmp *directly. Otherwise,* h.getValue(), *a linebreak (*\n*) and the content of* val *are assigned to* tmp. *Finally, the content of* tmp *will be returned, i.e., it will be assigned to attribute* value *of node* : Helper. *In short: the current value of attribute* value *of node* : Helper *is extended by the value of attribute* value *of node* : Argument.

```
1 {
2   var tmp;
3   if(h.getValue() == null) {
4     tmp = val;
```

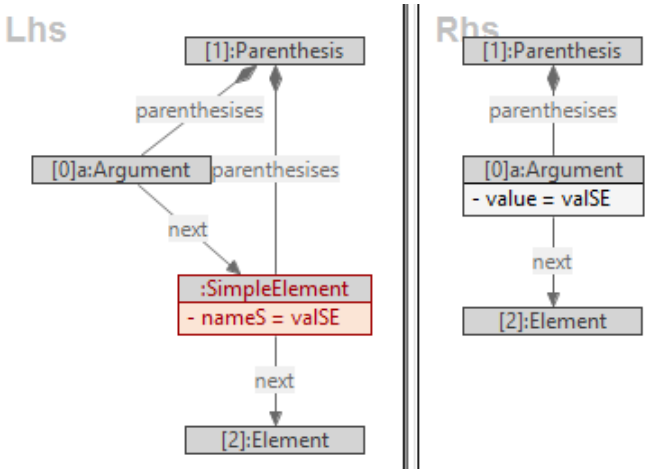


Figure 6.14: Rule P2_merge empty Argument node with SimpleElement in Parenthesis_case 3

```
5     } else {
6       tmp = h.getValue() + "\n" + val;
7     }
8     tmp
9   }
```

Listing 6.6: Script assigned to attribute value

Refactoring rule R_Helper_value2Activity_description *in screenshot* fig : applied : ref6 *shows, how the value of node* : Helper *is copied to the corresponding* : Activity *node. Furthermore, it deletes the necessary helper structure, i.e., nodes* : CORR *and* : Helper *and the corresponding edges. The NAC is necessary in order to apply this rule only if all* : Argument *nodes of the parent activity are handled, before. This is visible, if no* : Argument *node connected via edge* arguments *to the* : Activity *node is connected via* : CORR *node to the corresponding* : Helper *node.*

The last rule which we want to illustrate explcitly out of the set of refac-

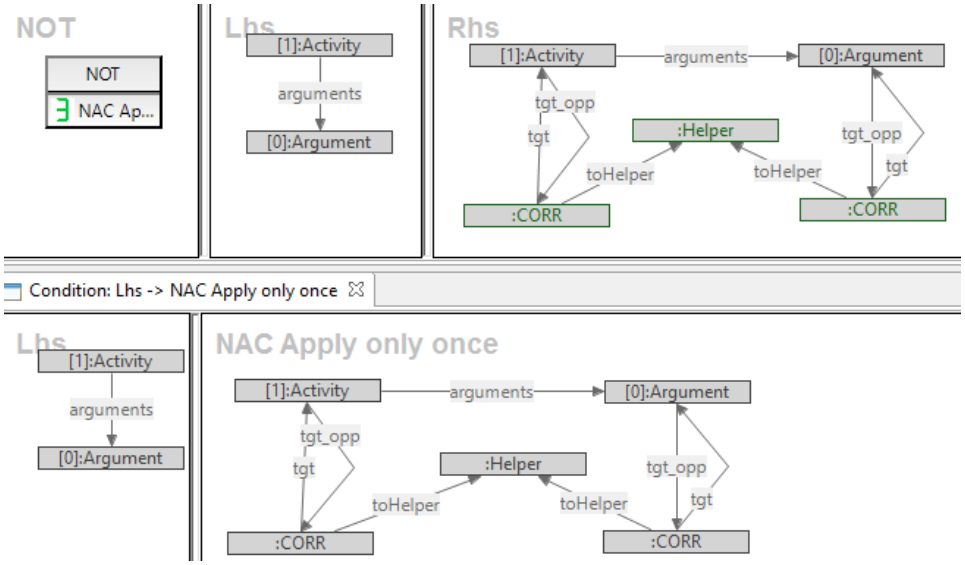


Figure 6.15: Rule P_Activity_getsHelper

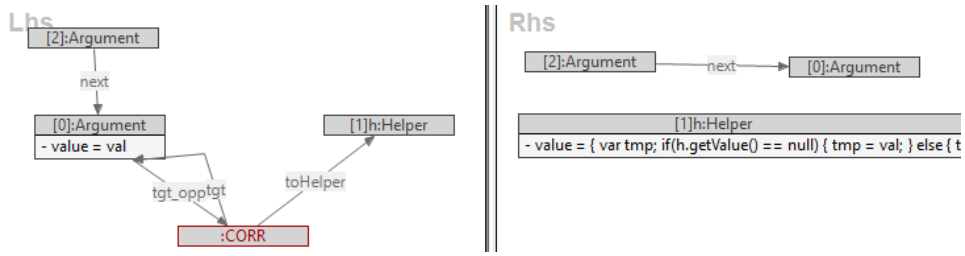


Figure 6.16: Rule R_Argument_value_next2Helper_value_addToEnd

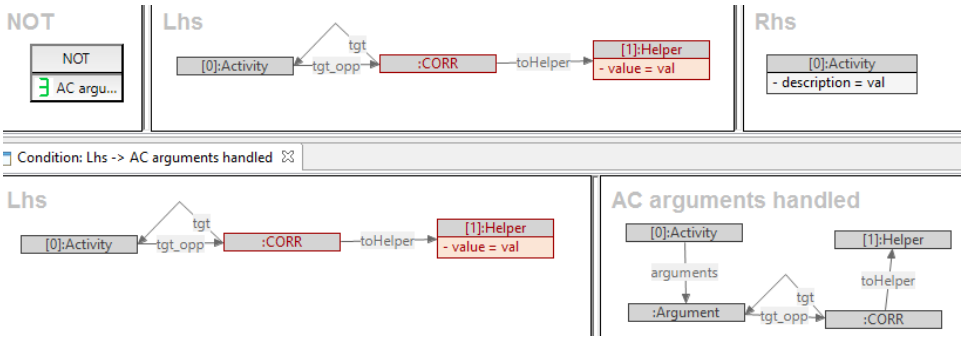


Figure 6.17: Rule R_Helper_value2Activity_description

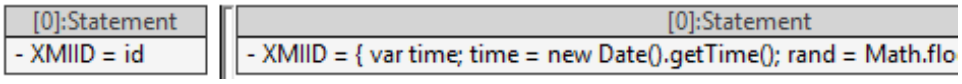


Figure 6.18: Rule R_setXMIID

toring rules is rule R_setXMIID *visualised in Fig. 6.18. It generates a new unique id for each node that is derived from* : Statement *in the SPELL-Flow model, i.e., for each node in the SPELL-Flow model that was generated via triple graph grammar* SPELL-2-FlowFlat. *The script in Listing 6.7 is assigned to attribute* XMIID *of node* : Statement. *The unique id starts with a "_" and is combined with the current timestamp which is concatenated with a random number between 1 and 100000. This rule shall be applied only once for each node. This requirement is fulfilled in using the attribute condition* id == null, *i.e., this rule will only be applied to nodes that do not have any unique id, yet.*

```
{
  var time;
  time = new Date().getTime();
  rand = Math.floor((Math.random() * 100000) + 1);
  var tmp = "_" + time.toString() + rand.toString();
  tmp
}
```

Listing 6.7: Script for generting a unique id

The last two rules, which we want to present, are taken out of the set of rules that are contained by the sequential unit CleanUp. *The first rule* D_RemoveHelper *(Fig. 6.19) simply removes all* : Helper *nodes. The second rule* D_Delete_Uncontained_Nodes *which is illustrated with the aid of screenshot Fig. 6.20, removes all* : Element *nodes that are not contained by any other node. The NAC forbids the deletion of nodes that are contained by another node. Note, this rule will not be applied to all nodes recursively, starting with node* : Root, *because uncontained nodes of type* : Element *will be*

deleted. Node : Root *is not derived from* : Element *(cf. Appendix A.3).* △

Introduce Hierarchies with SPELL-Flow-2-Hierarchy The second plain graph grammar we use for the translation from SPELL source code to a SPELL-Flow viusal model is called *SPELL-Flow-2-Hierarchy*. It gets a flat SPELL-Flow model as input and returns a hierarchical model. The hierarchies are created according to the rules set up by SES which we introduced in Sec. 6.1.1.

Similar to the refactoring grammar *Refactor_SPELL-Flow*, the processing is executed using the sequential transformation unit Main. It is divided into four different sequential sub-units (see screenshot Fig. 6.21).

The first sequential sub-unit contains rules that perform pre-processing operations. During the pre-processing sub-phase, helper structures are created in order to find parent nodes for branchings, e.g., all activities below an if-structure get a link to this if-structure, because if-structures will be on layer n, whereas all activities that follow this if will be later arranged on layer $n + 1$, i.e., on the underlying layer. Similar helper structures are created to goto statements. The helper structures are defined in the meta-model *CORRFlow2Flow* (cf. Appendix A.5). Furthermore, in the pre-processing step, some further merging operations are executed.

The second sub-unit of *Main* is called *Hierarchies*. It introduces the

Figure 6.19: Rule D_RemoveHelper

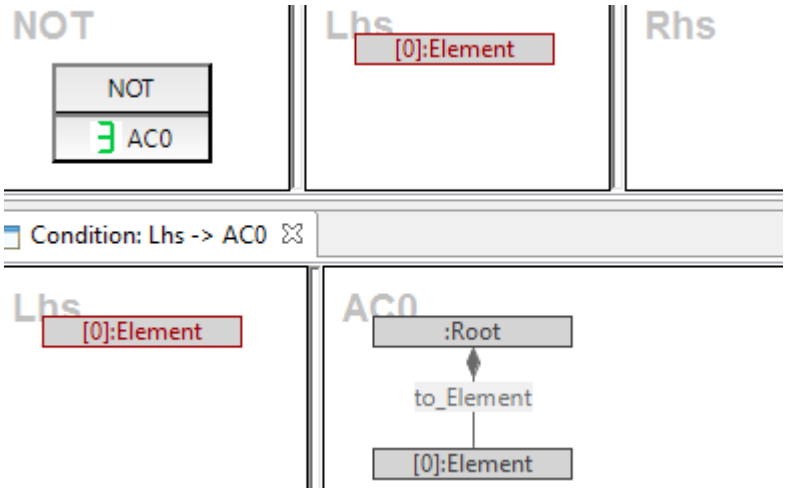


Figure 6.20: Rule D_Delete_Uncontained_Nodes

hierarchical structure as desired by SES. In general, the hierarchical structure is created in copying the node which shall occur on layer n as well as on layer n + 1. Then, the : next edges as well as the first edge from node : StartNode to its first predecessor will be rerouted, so that on layer n, i.e., the overlying and more abstract layer, the correct structure will be reflected, again. In the underlying layer (n + 1), the correct order of statements will be kept, because they will be shifted to the underlying layer in removing the old containment edges and setting new containment edges between the node that stays on layer n and the set of underlying statements. Usually,

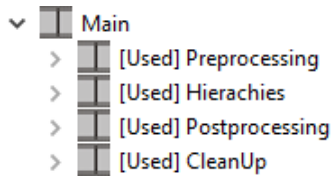


Figure 6.21: Screenshot from Henshin: Main Transformation Unit

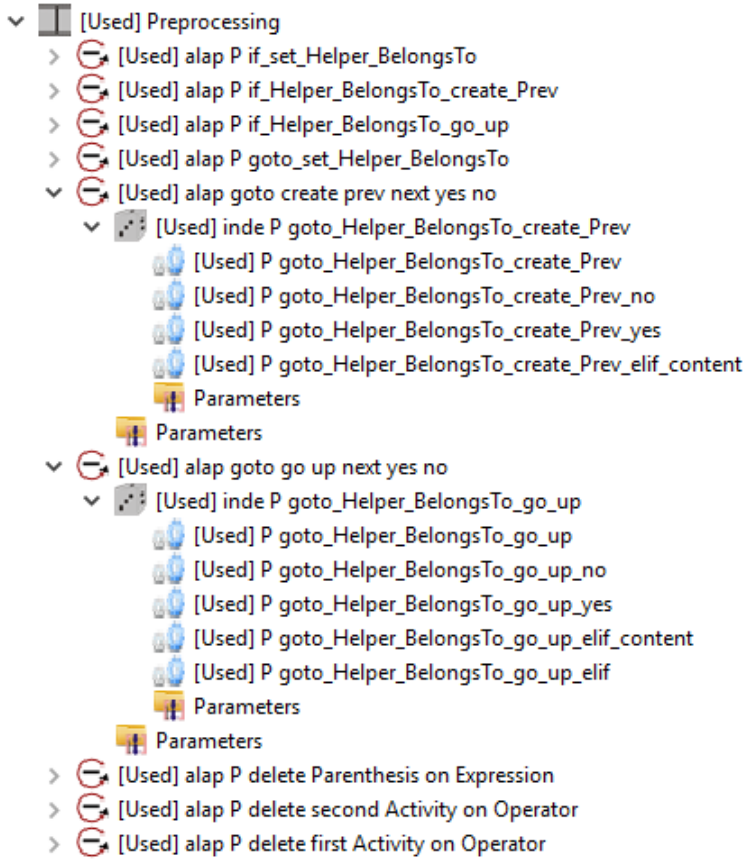


Figure 6.22: Transformation Unit for Preprocessing Phase

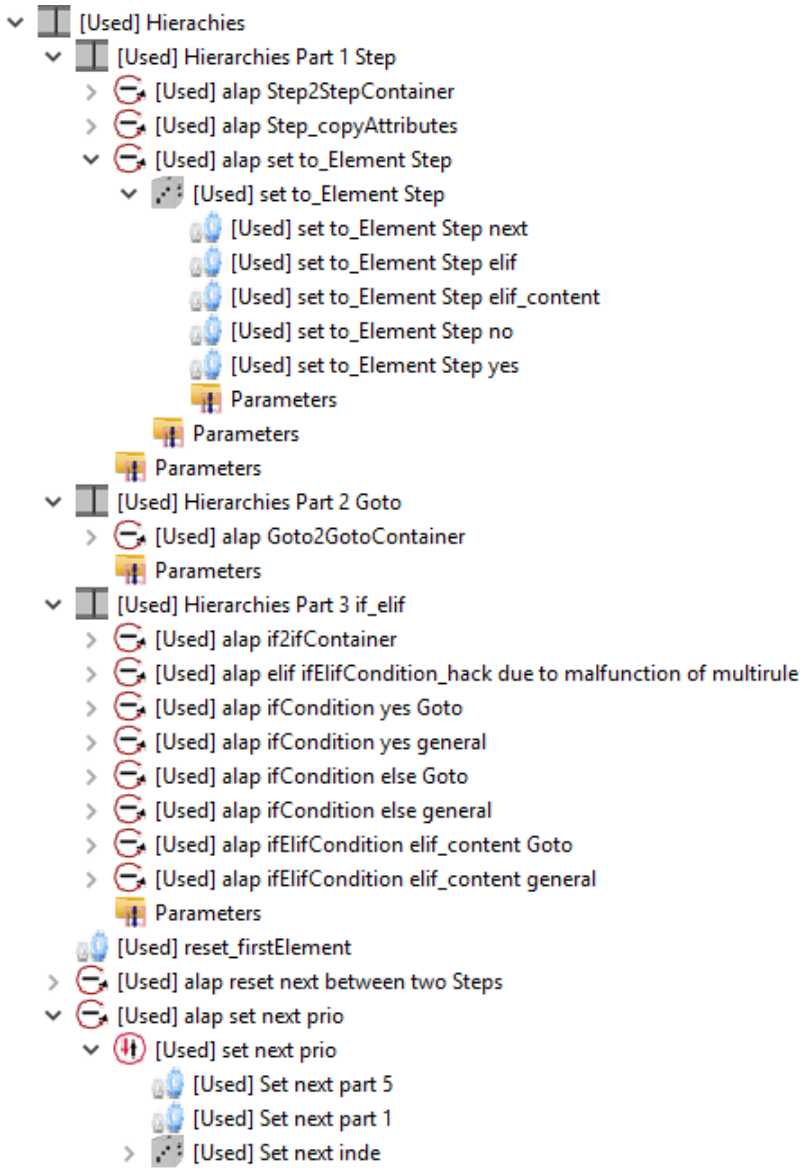


Figure 6.23: Transformation Unit for Introducing Hierarchies

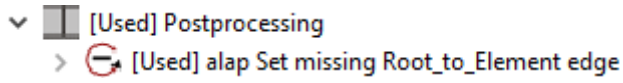


Figure 6.24: Transformation Unit for Postprocessing Phase

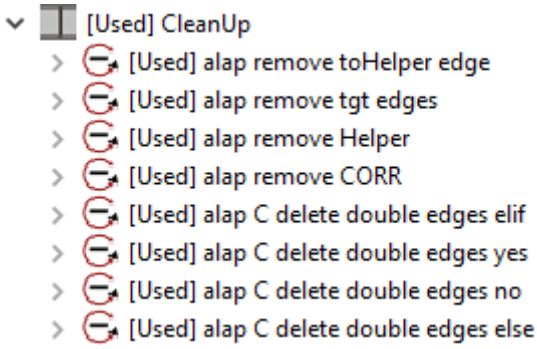


Figure 6.25: Transformation Unit for Clean Up Phase

the : next edges (and similar types of edges that define the correct order of statements) stay unchanged.

The next sub-unit is the post-processing step which contains only one rule that will be applied as long as possible. It is possible, that during the previous operations the containment edges between some elements and the : Root node will be removed. Those edges will be recreated by this post-processing step.

Finally, the helper structures out of meta-model *CORRFlow2Flow* will be removed. Furthermore, more than one identical edges may occur between the same statements. This will be corrected within this last transformation unit.

Example 6.1.3 (Rules from SPELL-Flow-2-Hierarchy)**.** *We will now illustrate and describe some example graph rules out of each sequential transformation sub-unit of graph grammar* SPELL-Flow-2-Hierarchy *that we discussed in the previous part.*

The first graph rule P if_set_Helper_BelongsTo *(Fig. 6.26) creates the helper structure for an* : ifCondition *node, but only if this structure does not exist for this node (cf. NAC). Note,* : ifElifCondition *nodes are derived from* : ifCondition*, therefore this holds also for* : ifElifCondition *nodes. The same structure is created for* : GotoActivity *nodes. This helper structure is used for defining the new order of statements on the first layer, i.e., on the most abstract layer. According to Sec. 6.1.1, this layer only contains* : StepActivity*s, conditions and loops if they are on the first indentation level and gotos.*

The : Helper *node is created with two attributes* value = BelongsTo *and* number = 1*. Within this graph grammar, different kinds of* : Helper *nodes are used. To distinguish those types of nodes, the attribute* value *is used.*

Attribute number *is initially set to one. This number will be increased using rules like* P if_set_Helper_BelongsTo_go_up*, i.e., the rule that we will describe next. The number is used for numbering conditions, gotos and loops on the first indentation level but that still belong to a* : Step *node. In the current case,* number *describes the distance between the* : ifCondition *node and its predecessing* : StepActivity *node. Currently, similar rules exist for gotos (and are planned for loops). Thus, each statement of a* : StepActivity *node that shall occur on the first layer will have a different number.*

The next graph rule P if_set_Helper_BelongsTo_go_up *(Fig. 6.27) illustrates the increase of attribute* number *of node* : Helper*. The helper structure includes correspondences between the* : ifCondition *node and an* : Element *node. The connection between the current* : Element *node will be reset to the previous* : Element *node in order to "walk up" to the right predecessing*

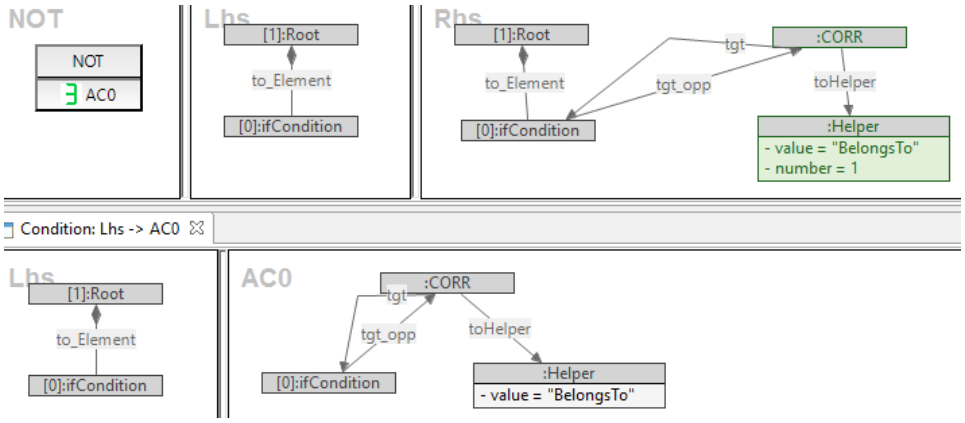


Figure 6.26: Rule P if_set_Helper_BelongsTo

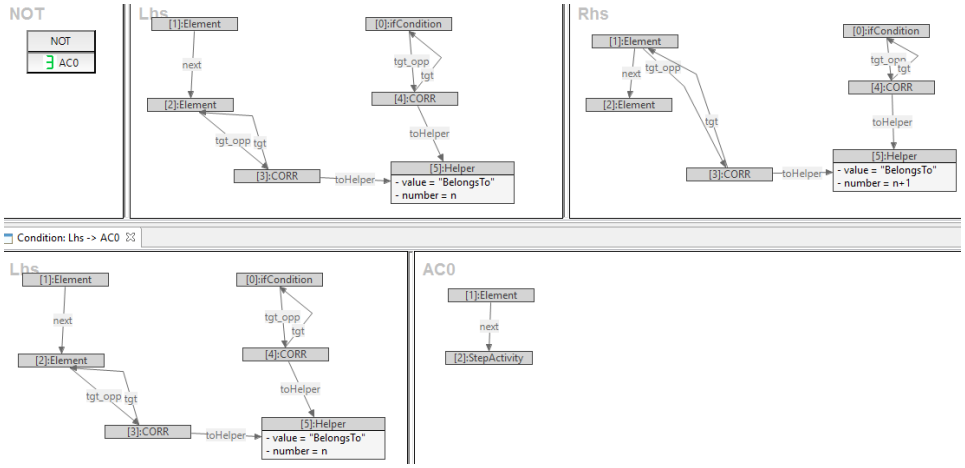


Figure 6.27: Rule P if_set_Helper_BelongsTo_go_up

: StepActivity *node. The NAC forbids to "walk up", if the corresponding* : StepActivity *node is already reached.*

Graph rule Step2StepContainer *in Fig. 6.28 belongs to sequential sub-unit* Hierarchies*, the sub-unit in which the main operations of that grammar are performed. This graph rule copies an existing* : StepActivity *node, creates a containment between the original and the copy and creates the necessary helper structure between both. Note, the corresponding* : Helper *node includes attribute* value = USED*, which defines a second kind of* : Helper *node. In order to have unique xmiids for both* : StepActivities*, the original id is overtaken for the new one and extended by a postfix string "_top". The NAC forbids the application of that rule to the same* : StepActivity *nodes, again.*

Graph rule Step_copyAttributes *(cf. Fig. 6.29) follows graph rule* Step2StepContainer*. It copies all necessary attributes from the original* : StepActivity *node to its copy. The NAC forbids multiple applications on the same nodes.*

The next graph rule is called set to_Element Step next *which is illustrated in Fig. 6.30. It sets a missing containment edge* : to_Element *between a conatiner node of type* : StepActivity *and an* : Element *node which is connected via edge* : next *to an existing node of type* : Element *that is a child node of* : StepActivity*. This rule owns an application condition which forbids the existence of pattern* AC0 *or pattern* AC1 *(cf. Fig. 6.30 (bottom)). In detail it is checked, if the second* : Element *node, i.e., the one that shall be connected by a new container edge, is not already contained by another node.*

The post-processing step contains one graph rule Set missing

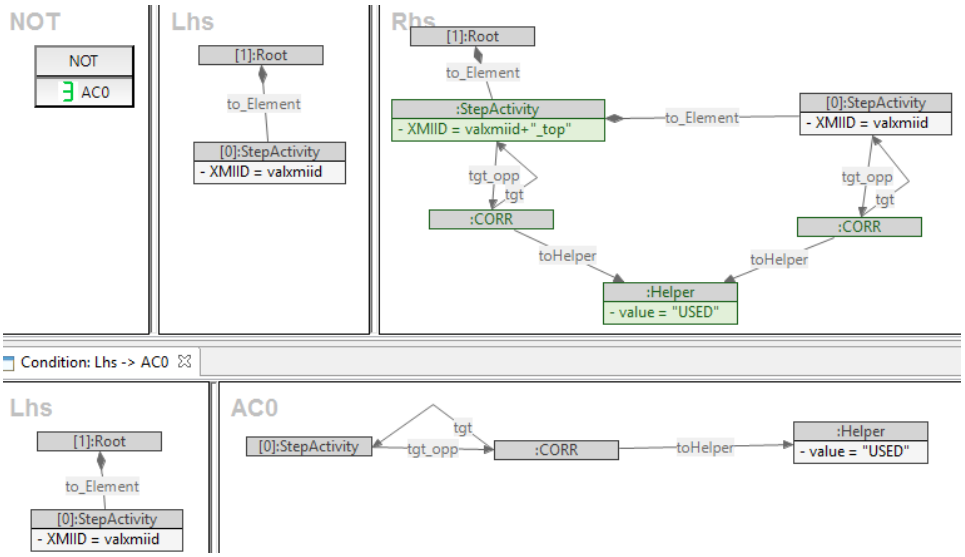


Figure 6.28: Rule Step2StepContainer

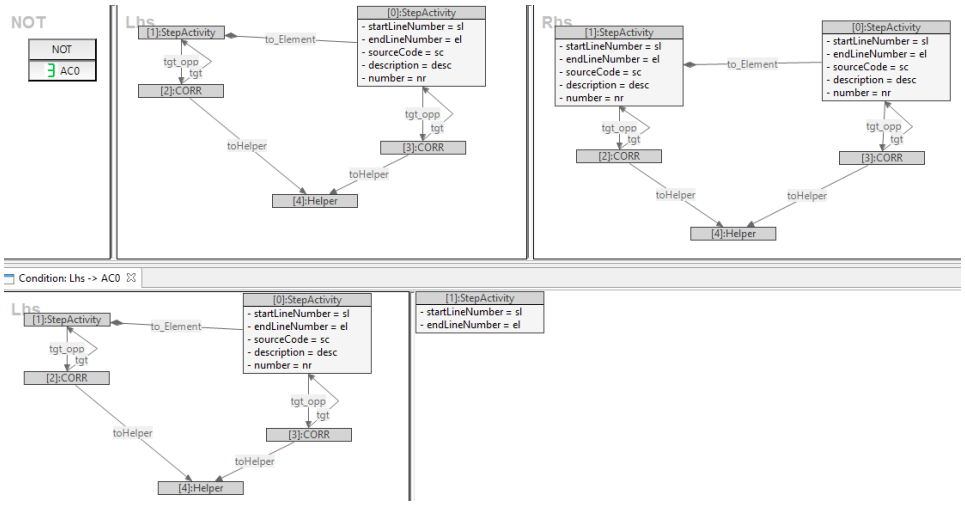


Figure 6.29: Rule Step_copyAttributes

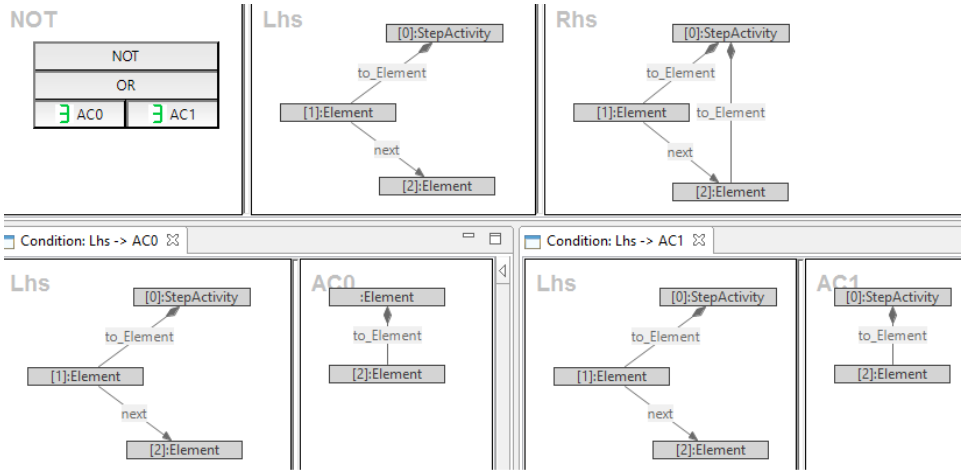


Figure 6.30: Rule set to_Element Step next

Root_to_Element edge *(cf. Fig. 6.31) that will be applied as long as possible. It sets a containment edge* : to_Element *from node* : Root *to an* : Element *node, if the latter is not already contained by a* : Root *node (cf. NAC).*

The last transformation unit is called CleanUp. *It conatins graph rules that remove all helper structures and that tidies up the SPELL-Flow model. We will introduce two rules out of this sequential transformation unit: Graph rule* remove toHelper edge *(cf. Fig. 6.32) deleted the incoming edge of a* : Helper *node. The second rule* remove Helper *(cf. Fig. 6.33) deletes a* : Helper *node. Both rules will be applied as long as possible.* △

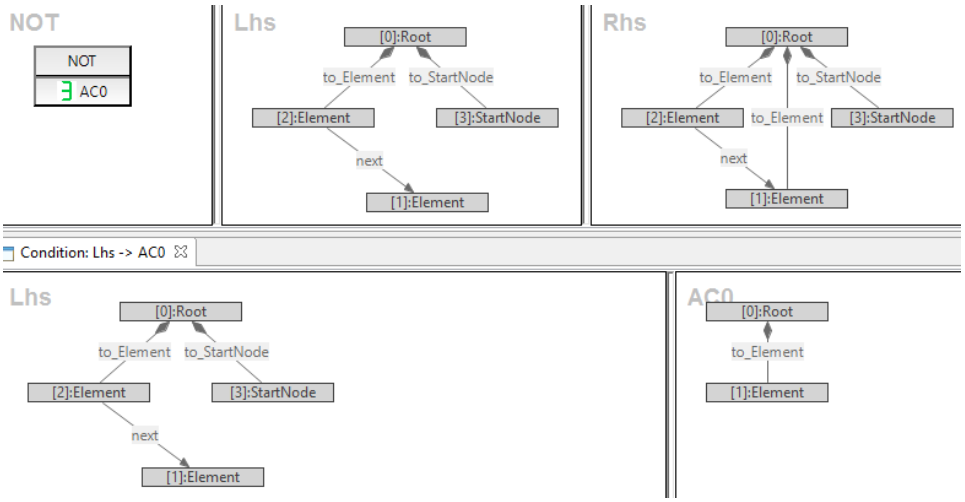


Figure 6.31: Rule Set missing Root_to_Element edge

6.1.3. SPELL-Flow Visualisation Tool

The implementation of a prototype SPELL-Flow visualisation tool was part of the practical project in industrial cooperation with SES. In this subsection, we will present the implementaion details of the SPELL-Flow visualisation tool. First of all, we will give a brief outline of the software development tools we used within the implementation.

Background on Software Development Tools

We will summarise the necessary development techniqes in the following.

Java Java is a object-oriented programming language [jav16] which was developed by Sun Microsystems and which appeared in 1995. It is platform-independent, i.e., the same source code can be run on different machines. It is independent from the underlying hardware. During compilation, Java is

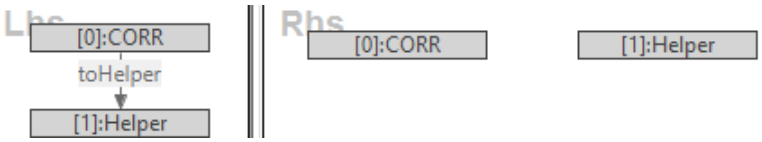


Figure 6.32: Rule remove toHelper edge

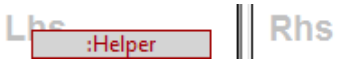


Figure 6.33: Rule remove Helper

converted into platform-independent bytecode which will be interpreted on the target platform.

Eclipse Eclipse is an integrated development environment (IDE) [ecl16a] mainly for the programming language Java (cf. Sec. 6.1.3). It is non-commercial and open-source enabling the development and usage of a wide varity of frameworks and tools. The pre-version of Eclipse was developed by IBM that released the source code of Eclipse in 2001. Since 2004, the Eclipse Foundation was founded (which is led by IBM) being the current developer of Eclipse. Eclipse is based on Java and works on several platforms, e.g., Linux, Mac OS X or Windows.

GMF The Graphical Modeling Framework (GMF) [GMF16] is an Eclipse framework which provides an environment for developing graphical editors. GMF is based on the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF) [EMF16, GEF16, MDG+04]. GMF consists of a generative component and a runtime infrastructure. The generative component provides the possibility for the developer to define a tooling, a graphical and a mapping specification. Based on those definitions, GMF is able to generate a graphical editor in Eclipse. The runtime infrastructure provides pre-defined and complete features which will be automatically integrated into each generated editor, e.g., components for printing, image export, or toolbars, so that the manual implementation is not necessary. The generated editor can be extended by the developer.

EMF is another Eclipse framework for defining a model specification. Out of this specification, EMF is able to generate a set of Java classes that represent the model, a set of adapter classes that support the viewing and editing of the model, and a basic editor for defining instance models that follow the specification. Ecore is the core meta-model of EMF and specifies how models specified via EMF shall be constructed. If follows the Meta Object Facility (MOF) specification that was set up by the Object Management Group (OMG) [MOF16].

GEF is again an Eclipse framework. In using GEF, developers are able to create visual editors for an arbitrary model, e.g., also for a model that was defined using EMF.

Xtext Another open-source framework provided by Eclipse is Xtext [xte16] being part of the EMF project. It is used for defining domain-specific languages (DSLs). The DSL is specified by the developer in a notation which resembles the Extended Backus-Naur form (EBNF) [BBG+60, EBN96]. Out of this specification, Xtext generates a parser, an EMF meta-modell and a text editor which is integrated into Eclipse and provides syntax-highlighting.

Furthermore, Xtext generates the infrastructure that is needed for further developments based on the DSL.

The SPELL-Flow visualisation tool that was developed in the framework of the industrial cooperation. The industrial partner SES demanded the following requirements for the SPELL-Flow visualisation tool:

- The SPELL-Flow visualisation tool shall show the SPELL-Flow model, which have to follow the guidelines from Sec. 6.1.1.
- The SPELL-Flow visualisation tool shall show the model in only one tab, i.e., a special kind of navigation is necessary which opens underlying or overlying layers, respctively, in the same tab.
- Goto statements (and Step statements) shall provide the possibility to jump to the corresponding Step (or Goto statements) by clicking on the shape or via context menue entry.
- The first prototype of the tool shall be read-only.
- The SPELL-Flow model shall be layouted automatically.
- Each statement shall provide a view on the source code via tooltip. The source code shall be syntax-highlighted.
- The SPELL-Flow visualisation tool shall be a plugin which shall provide the possibility to be easily integrated into the SES satellite control application for SPELL (SPELL GUI). At a later stage a SPELL-Flow editor which is based on the SPELL-Flow visusalisation tool, shall provide the possibility to get easily integrated into the SES development environment for SPELL (SPELL DEV).
- Due to the possibly large size of the SPELL-Flow model, zooming in and out shall be possible as well as scrolling.
- An outline view is desired.

The prototype of the SPELL-Flow visualisation tool which was developed during the work on the industrial project fullfils the demanded requirements. It is a non-editable Eclipse plugin which was created using GMF and extended according to the requirements that were set up by SES. Fig. 6.34 shows a screenshot of the SPELL-Flow visualisation plugin. We will elaborate the prototype SPELL-Flow visualisation tool by means of this screenshot.

In the current stage of development, the SPELL-Flow visualisation tool will start with the following views that are marked with green balloons in Fig. 6.34: the project explorer is usually situated on the left top side. There, a ∗sfld file can be selected and opened in order to open a SPELL-Flow

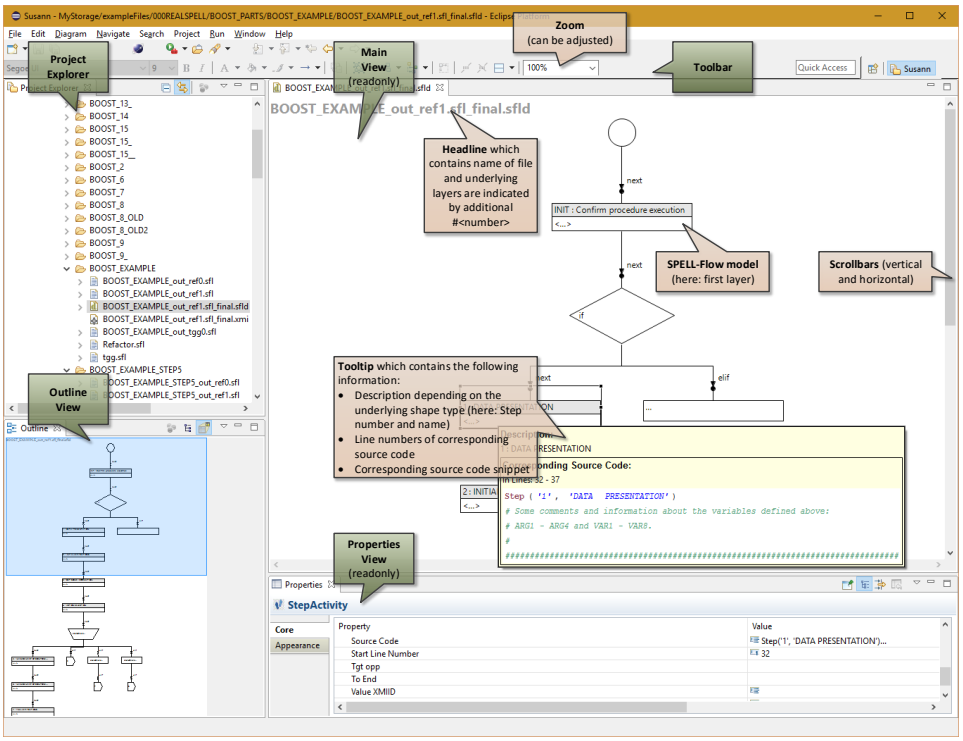


Figure 6.34: Overview SPELL-Flow visualisation tool

model. In addition, a toolbar is available at the top which provides some default functionality that was already generated by GMF, like the possibility to zoom in and out. If a model is opened, the SPELL-Flow visualisation tool shows the main view containing the SPELL-Flow visusalisation (by default the larges view, top right), an outline view, which is below of the project explorer and a properties view, which is below of the main view. SES demanded a read-only tool, therefore, it is not possible to edit any property in the properties view. In addition, it is not possible to add or remove any shapes from the model. It is not allowed to change the layout of the model or the position of any shape on the canvas, respectively. Consequently, the corresponding toolbar options are deactivated.

The main view displays the SPELL-Flow model and a headline on top, which currently shows the name of the model and, if an underlying layer is opened, the ID of that layer (indicated by # < id >). When the SPELL-Flow model is loaded, an auto-layout action will be performed. The result is a clean layouted picture which is displayed in the main view. If the model is larger than the canvas, then scrollbars will be displayed.

If the user hovers the mouse over a node, a tooltip will be diplayed which contains the following information, if it is available:

- A more detailed description will be shown, e.g., for Step statements

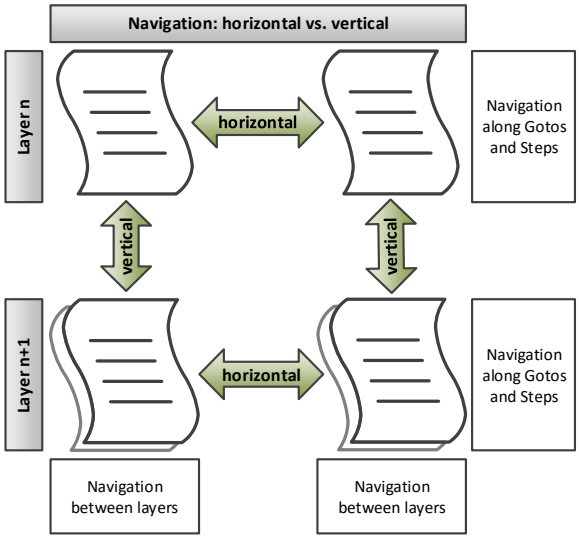


Figure 6.35: Navigation

both attribute values because, if both values are too long for the displayed node, they will be abbreviated. There, we do not overtake the source code formatting. Instead a "pretty Python" version is displayed (cf. item 7).

- The line numbers of the corresponding statement in the SPELL source code file is given.

- The corresponding SPELL source code snippet of that statement is displayed. The SPELL source code is syntax-highlighted.

The navigation in the SPELL-Flow visualisation is performed in two directions, as is is also visualised in Fig. 6.35. The horizontal navigation does not change the layer, i.e., the main view jumps to another position of the model, but within the same layer. This case occurs when we navigate between Step and Goto statements. The vertical navigation is performed between different layers, i.e., the main view opens an underlying or the overlying layer, repectively, of the current one.

Within the master's thesis in [Kha15], a basic version of the vertical navigation was implemented. The SPELL-Flow visualisation tool uses slighly more complex vertical navigation and also it contains an implementation of the horizontal navigation. Both kinds of navigation options are available via context menu entries of the nodes which allow this kinds of navigation. The horizontal navigation option is only available for Step and Goto statements. Note that not all nodes allow vertical navigation, too, because they do not contain an underlying (or overlying) layer, e.g., Pause statements do not

contain any children nodes, therfore they have no context menu entry for navigating to an underlying layer.

Example 6.1.4 (Horizontal vs. vertical navigation)**.** *Fig. 6.36 includes screenshots of the SPELL-Flow visualisation tool illustrating both kinds of navigations. In the upper left screenshot, the user clicks on the* Goto 5 *node. Then, the tool jumps to the corresponding* Step *statement, which is* Step 5 *(upper right screenshot). The tool also marks the corresponding node in blue, which is not visualised, here.*

If the user selects option Go deeper *in the context menu of a node (here: of node* Step 5*), then, the tool opens the underlying layer (screenshot on the bottom right). In the underlying layer, the user is able to select option* Go up *in any node. Then, the main view opens the overlying layer (screenshot on the bottom left).*

The last screenshot also illustrates the case where the user selects one Goto *statement that refers to this* Step *node from the list in the context menu. Note, the list might contain more than one* Goto *references. After selecting one reference, then the main view jumps to the corresponding* Goto *statement and highlights it in blue (upper left screenshot).* △

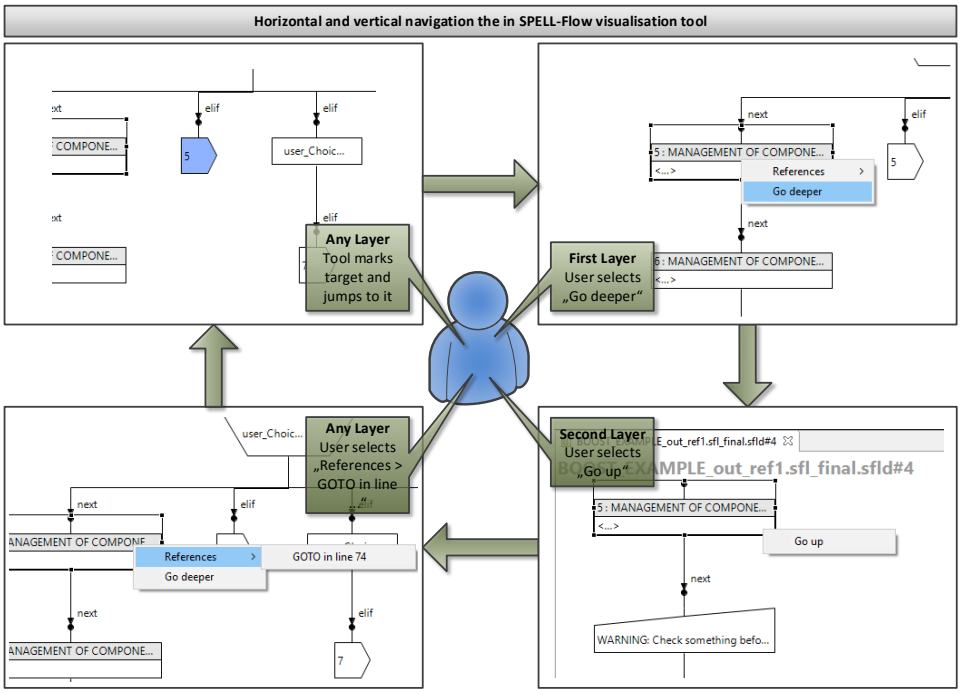


Figure 6.36: Navigation in SPELL-Flow visualisation tool

6.1.4. Automated Translation ("Button")

The task of implementing an automated translation is realised by an Eclipse-plugin using Java, EMF, Xtext and the Henshin engine.

In practice, the Eclipse plugin is executed by the user as follows: The corresponding Eclipse instance needs to be active. After selecting one or more SPELL source code files,the user needs to click the button with icon  . Afterwards the automated translation is performed resulting in a new folder containing the intermediate files and also the resulting ∗.sfld file which can be opened with the SPELL-Flow visualisation tool (cf. Sec. 6.1.3).

Example 6.1.5. *In the following screenshot we show the resulting files for an automated translation of the example SPELL source code file* BOOST_EXAMPLE.py. *After performing the automated translation, a new*

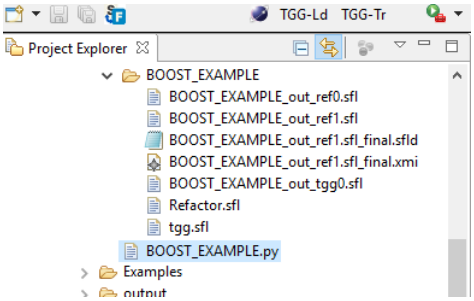


Figure 6.37: Project view in Eclipse shoing newly created files after translating BOOST_EXAMPLE.py

folder with the name of the file to be translated is created. This folder contains three intermediate files which have the file extension ∗.sfl, *one final file which can be opened with the SPELL-Flow visualisation tool* ∗.sfld *and one* ∗.xmi *file which is identical to the* ∗.sfld *file and which can be regarded as backup file.* △

Technically, the automated translation executes the follwing steps that we also illustrate by means of Fig. 6.38.

After selecting one or more files and clicking the corresponding button, the SPELL source code file [filename].py will be parsed into the SPELL abstract syntax graph (ASG) using Xtext. Xtext uses the SPELL grammar for parsing (cf. Listing A.1).

The SPELL ASG will be provided as input to the next step: the translation via triple graph grammars using the Henshin engine. Currently, it uses TGG SPELL2FlowFlat.henshin for the first translation step. The result of the TGG translation is a flat SPELL-Flow ASG. Flat means, that no hierarhical structures are available. It's XMI representation is also stored within the file [filename]_out_tgg0.sfl.

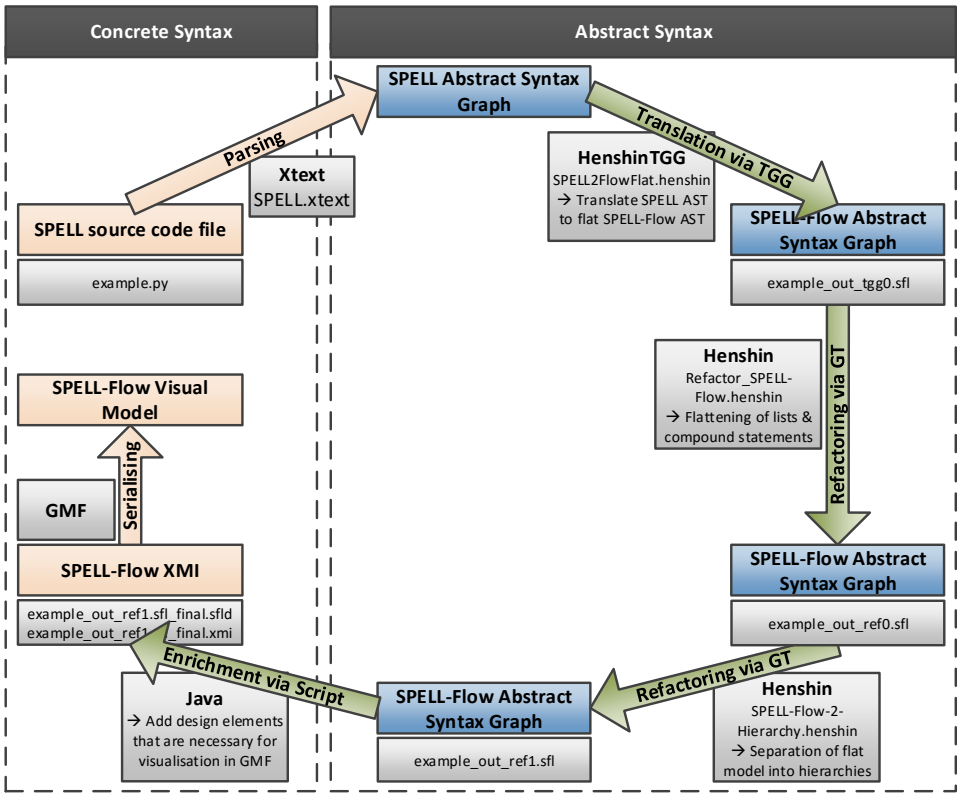


Figure 6.38: Automated Translation via Button

The flat SPELL-Flow ASG is taken as input to the second step, the flat graph transformation using the Henshin engine. In the current version, the software executes two flat graph grammars after each other, the output of the first one is the input of the second one. The first graph grammar is called Refactor_SPELL-Flow.henshin. It is improving the flat SPELL-Flow ASG in the sense that it merges lists and other compound statements consisting of several nodes to one node. It returns a flat SPELL-Flow ASG, which is "tidied up". The second flat graph grammar is called SPELL − Flow-2-Hierarchy.henshin. It introduces the hierarchical layers of the flat SPELL-Flow ASG according to the rules we summarised in item 4 of Sec. 6.1.1. Both steps also output the intermediate result in separate files [filename]_out_ref[i].sfl, where $i \in \{0, 1\}$.

The last result, i.e., the hierarchical SPELL-Flow ASG is provided as input to the last step: the enrichment of design elements. In order to be able to open the SPELL-Flow model, the GMF-based SPELL-Flow visualisation tool needs design elements for each node of the final SPELL-Flow model. This property is demanded by GMF. The enrichment is executed by means of a Java script. This work was started in cooperation with a master's thesis

in [Gon15]. For further details on the implementation during the master's thesis and for design decisions of that solution, we refer to [Gon15]. The full solution, which also considers hierarchical layers, was extended during the work on this industrial project. The enrichment process finishes with two new files that contain the same content: [filename]_out_ref1.sfl_final.sfld and [filename]_out_ref1.sfl_final.xmi. The latter is just stored as backup file for the first one. The ∗.sfld file can be finally imported into the Spell-Flow visualisation tool.

The screenshot given in Fig. 6.39 illustrates the structure and the source code files of the Eclipse project SPELL-2-SPELL-Flow-Button, which is the project containing the automated translation plugin. The first

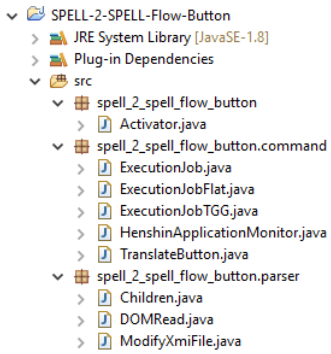


Figure 6.39: Project containing source code files for automated translation

package spell_2_spell_flow_button contains one source code which is responsible for activating the button as plugin. The second package spell_2_spell_flow_button.command contains the main part of the plugin, i.e., all Java files that contain the source code for the automated translation. File TranslateButton.java can be seen as the "main" source code file which calls each of the steps we mentioned in the preceding part and in Fig. 6.38 in the right order. The third package spell_2_spell_flow_button.parser includes all source code files that are necessary for the enrichment of the final SPELL-Flow model with design elements.

6.2. Evaluation

In this section, we will evaluate our implementation with regard to two aspects: Firstly we will take a closer look to the implementation of the unidirectional translation, i.e., we will analyse the size of the meta-models and of the three graph grammars that were applied. Secondly, we performed example translations and measured the duration of the full translation as well as of the single steps of the translation. Finally, we will discuss the unidirectional approach, point out drawbacks of the translation and argue

about possible improvements. Finally, we review the possibilities of realising a backward translation which would lead to a functioning bidirectional solution.

6.2.1. Numbers on Meta-Models and Grammars

The translation from SPELL source code to SPELL-Flow models and vice versa is based triple graph transformations and plain graph transformations using typed attributed (triple) graphs. Therefore, our model transformations need at least two meta-models that can be seen as type graphs for the relevant instance: The first meta-model that is necessary shall describe the structure of SPELL abstract syntax graphs (ASG) that will be parsed out of SPELL source code. The second meta-model defines the SPELL-Flow language. As already mentioned in Sec. 6.1.2, we also defined two correspondence meta-models: *CORR* for characterising the mapping between SPELL and SPELL-Flow models used in the triple graph transformation step, and *CORRFlow2Flow* used as helper structure in the flat graph transformations.

The diagram given in Fig. 6.40 compares the sizes of all four meta-models based on the number of nodes. The SPELL meta-model is by far the largest meta-model containing 175 nodes. The SPELL-Flow model includes 42 nodes. Both correspondence models are much smaller: meta-model *CORR* contains 10 nodes and meta-model has only 3 nodes.

As elaborated in Sec. 6.1.2, the translation from SPELL source code to a visual SPELL-Flow model, uses three graph grammars: The triple graph grammar *SPELL-2-FlowFlat* which is followed by two flat graph grammars *Refactor_SPELL-Flow* and *SPELL-Flow-2-Hierarchies*. Fig. 6.41 compares the size of all three grammars in a diagram. The triple graph grammar is by

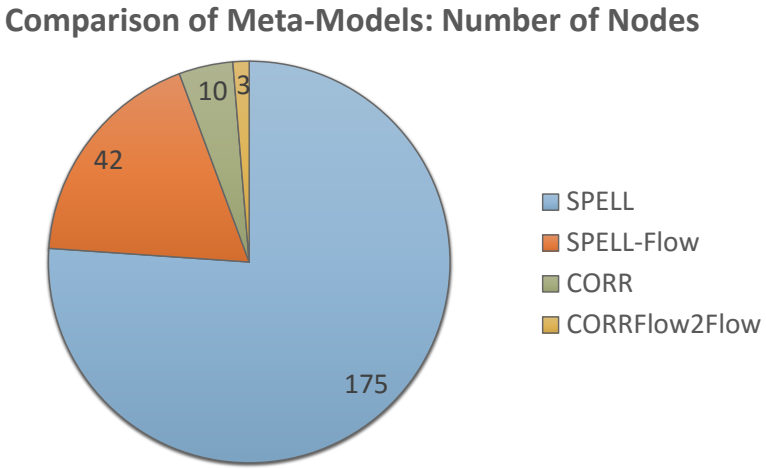


Figure 6.40: Meta-Models: Number of Nodes

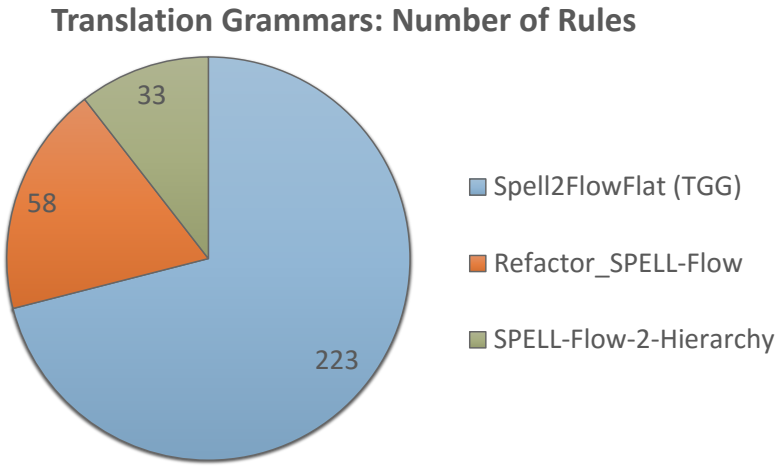


Figure 6.41: Translation Grammars: Number of Rules

far the largest grammar consisting of 223 triple rules and additionally the same number of forward rules, i.e., 446 rules in sum, for the translation from SPELL source code to a SPELL-Flow model. The diagram only includes the number of triple rules, because the operational rules are derived out of the triple rules. Graph grammar *Refactor_SPELL-Flow* includes 58 graph rules whereas graph grammar *SPELL-Flow-2-Hierarchies* is the smallest with 33 graph rules.

6.2.2. Unidirectional Translation of similar examples but with different sizes

For the example translations in this sub-section, we used twelve SPELL source code files that were derived out of our running example (Listing 1.1 introduced in Chap. 1). In order to achieve the corresponding lines of code that we wanted to analyse, we deleted parts from Listing 1.1 or duplicated parts. That means, that all example files are very similar to each other, except their size with regard to the number of source code lines (LOC). We analysed examples from 1 LOC to 350 LOC. A similar evaluation with different files will produce different results, because the translation time heavily depends on the complexity of the source code, and also on the diversity of statements: the more different all statements are, the faster is the translation process, because the matching phase, especially during the application of both flat graph grammars will speed up.

The time measurments were performed on a machine with the following specifications:

- CPU: Intel Core i5-4200H with 2.80 GHz, 2 cores
- RAM: 8GB

- OS: Microsoft Windows 10
- Development Environment:
  - Eclipse Mars, Release 4.5.1
  - Java Version 8
  - Xtext 2.7.0
  - GMF version 3.3.1
  - GEF 3.10.1
  - EMF 2.11.1
  - Eclipse Modeling Tools 4.5.1

Table 6.3 lists size of the SPELL source code file with regard to the number of nodes and the number of edges. Note, that the final SPELL-Flow model is a hierarchical model, i.e., duplications of different parts exist in the model. Fig. 6.42 compares the number of nodes and edges of the SPELL ASG and of the final SPELL-Flow ASG. The first column of the table identifies each dataset by means of the LOC of the original SPELL source code file. This property for identifying the corresponding datasets are used in all tables of this section.

LOC SPELL	**# Nodes SPELL ASG**	**# Edges SPELL ASG**	**# Nodes SPELL-Flow ASG**	**# Edges SPELL-Flow ASG**
1	9	8	4	4
10	104	103	8	12
20	292	291	23	42
50	309	308	27	49
100	601	600	72	135
130	749	748	94	177
150	861	860	113	214
200	1112	1111	152	290
250	1446	1445	193	370
300	1712	1711	218	416
350	1939	1938	254	487
500	2823	2822	380	729

Table 6.3: Comparison SPELL ASG with SPELL-Flow ASG: # of nodes and edges in source and target model

The diagram in Fig. 6.42 visualises the data of Table 6.3 in a diagram. It is obvious, that the final SPELL-Flow model (i.e., SPELL-Flow ASG) is much more smaller than the original SPELL model. Note that the resulting

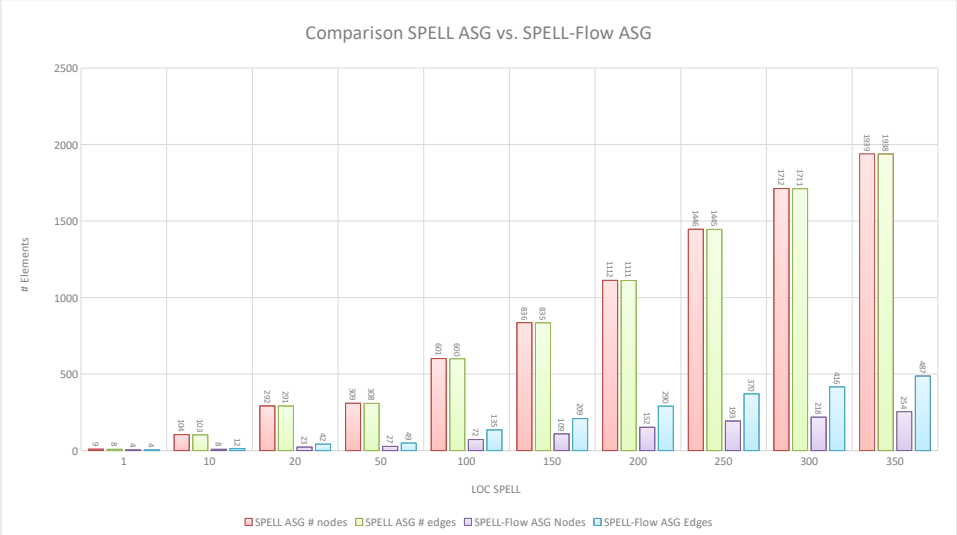


Figure 6.42: Comparison of examples: # of elements in SPELL ASG file vs. # of elements in final SPELL-Flow model

SPELL-Flow ASG even contains duplications of elements because it is a hierarchical model representing different layers of abstractions.

Table 6.4 presents the results of the time measurements of this evaluation. The second row lists the complete translation time for all example files in ms. The third row lists the same time in minutes. Those results are also visualised in Fig. 6.42.

LOC SPELL	**Time for translation in ms**	**Time for translation in min**
1	5643	0.09
10	5644	0.09
20	6586	0.11
50	9183	0.15
100	22236	0.37
150	125517	2.09
200	381843	6.36
250	738429	12.31
300	1510998	25.18
350	2964126	49.4

Table 6.4: Time for unidirectional translation from SPELL to SPELL-Flow

If we take a closer look at the translation times for all three graph grammars, as listed in Table 6.5 and visualised in Fig. 6.44, we observe that the refactoring step is by far the most time-consuming step. The triple

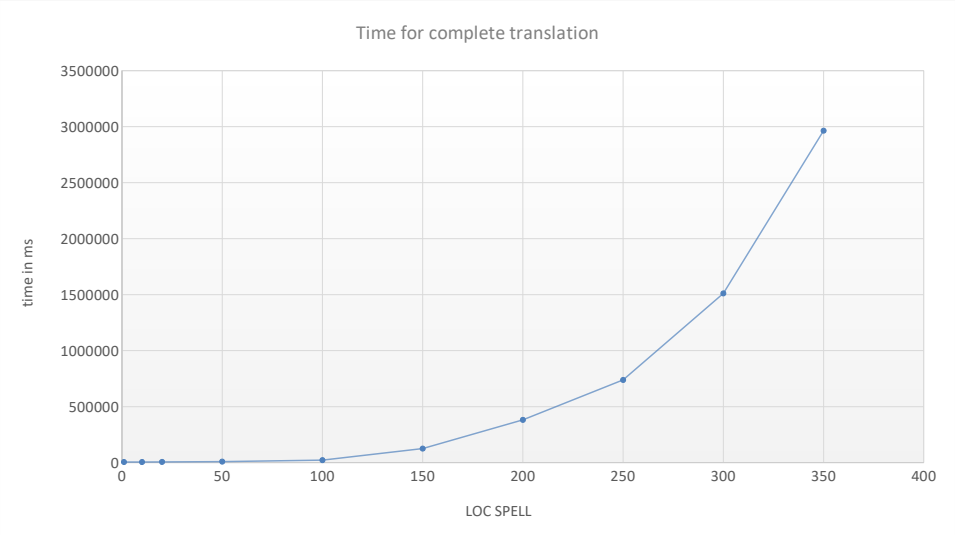


Figure 6.43: Diagram according to Table 6.3: Time for whole translation

LOC SPELL	Time for TGG translation in ms	Time for refactorings in ms	Time for hierarchies in ms
1	2484	1241	1815
10	2569	1492	1471
20	3276	1028	2195
50	3464	3991	1600
100	3956	15098	3043
150	4351	117879	3140
200	4962	352227	24265
250	5383	661745	71137
300	5635	1338094	166785
350	6864	2753045	203993

Table 6.5: Detailed time for unidirectional translation from SPELL to SPELL-Flow

graph transformation is the fastest transformation part. This is achieved because the implementation of the triple graph transformation in HenshinTGG is optimised with regard to the use of translation attributes. The refactoring step requires so much time, because the matching is very time-consuming. First of all, the plain graph transformation uses no translation attributes. Therefore, the implementation of plain graph transformation in the Henshin engine (used in the Henshin-Editor, cf. Sec. 6.1.2) differs from the implementation for triple graph grammars in the Henshin engine (used in HenshinTGG, cf. Sec. 6.1.2). For triple graph grammars, nodes that

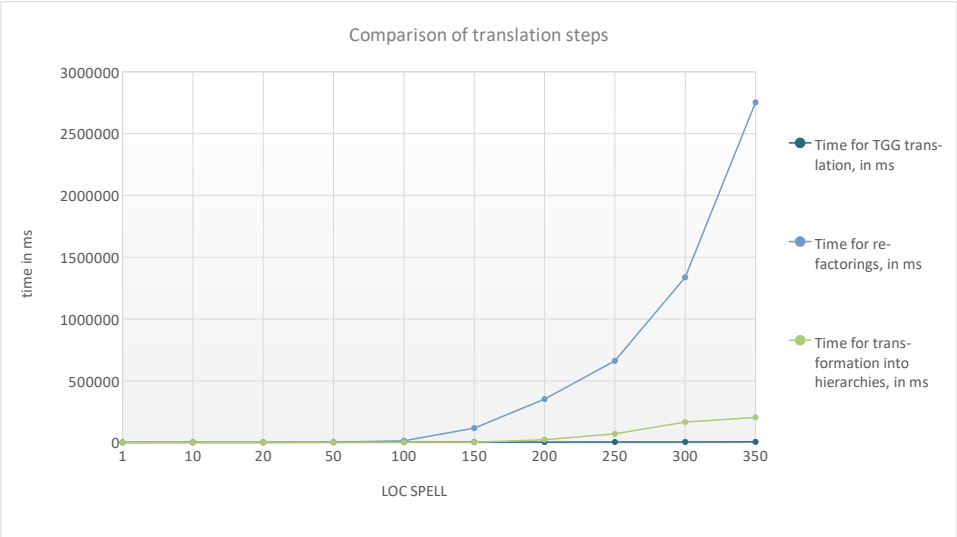


Figure 6.44: Comparison of translation steps (via TGG and GGs)

have a translation marker which are set to **T**, are not conisdered anymore in the matching phase. In contrast, the Henshin engine needs to consider and check every node in the graph again for each match during plain graph transformation.

For the refactoring phase, the input graph of this step is quite large for the matching process for plain graphs. In the following examples, we illustrate this fact.

Example 6.2.1 (Refactoring phase)**.** *For our example consisting of 100 LOCs, the triple graph transformation returns a flat SPELL-Flow model with 122 nodes and 233 edges that will be used as input to the refactoring phase. After the refactoring phase, the flat SPELL-Flow model is reduced to a graph consisting of 63 nodes and 117 edges. The refactoring phase took 396 rule applications. Note, the final SPELL-Flow model consists of 72 nodes and 135 edges.*

In the example, which consists of 200 LOCs, the triple graph transformation returns a flat SPELL-Flow model with 263 nodes and 510 edges that will be used as input to the refactoring phase. After the refactoring phase, the flat SPELL-Flow model is reduced to a graph consisting of 135 nodes and 257 edges. The refactoring phase took 874 rule applications. Note, the final SPELL-Flow model consists of 152 nodes and 290 edges.

In the example that had 300 LOCs, the triple graph transformation returns a flat SPELL-Flow model with 377 nodes and 734 edges that will be used as input to the refactoring phase. After the refactoring phase, the flat SPELL-Flow model is reduced to a graph consisting of 196 nodes and 377 edges. The refactoring phase took 1212 rule applications. Note, the final

SPELL-Flow model consists of 218 nodes and 416 edges. △

6.2.3. Discussion on Bidirectional Approach

As depicted in this chapter, a prototype implementation for the unidirectional transformation from SPELL source code to a SPELL-Flow visual model exists. It follows the methodology that we presented in Sec. 3.1. It is up to future work to implement the backward direction in order to realise a bidirectional solution. In general, triple graph grammars provide the possibility to derive backward transformation rules directly out of the triple rules in order to receive a solution for the backward transformation. If we consider the existing translation from SPELL source code to a SPELL-Flow visual model, we observe that the translation consist of four steps, i.e., it is much more complex (for details compare with Fig. 6.38).

We will now answer the question: *How can a backward transformation be implemented?* The input of the backward transformation is the SPELL-Flow visual model provided as XMI file. In principle, we have to revert all four steps, so that we are able to derive the corresponding SPELL source code. Still, it follows the methodology that we introduced in Sec. 3.2.

The first step of the backward transformation is the rolling back of the "enrichment of design elements". There, the XMI tags that describe all design elements simply needs to be deleted out of the XMI file. No further adaptions are necessary.

In the second step of the backward transformation we have to revert the changes that were perfromed by graph grammar *SPELL-Flow-2-Hierarchy*. For that step, a new graph grammar needs to be written because in plain graph transformations, we are not able to derive the set of "opposite graph rules" out of the existing set of graph rules. In this new reversion graph grammar the hierarchical model will be flattened, i.e., all elements from the most detailed levels will be kept and connected correctly with each other in order to reflect the proper execution sequence. All elements that reflect the more abstract layers will be deleted, because duplicates of them will be still available in the most detailed level. The resulting model is a flat SPELL-Flow model.

It is hardly possible to undo the refactoring step that was executed using graph grammar *Refactor_SPELL-Flow*, because in the refactoring step, elements were merged with each other, e.g., a list of arguments or comments. To derive the correct (original) structure is very difficult or even impossible. So, we would propose to omit the reversion of this step in the backward transformation.

The last step is the backward translation using backward rules that can easily be derived out of the existing triple rules of TGG *SPELL-2-FlowFlat*. The TGG already includes filter rules for the backward direction that need manual adaption after the generation of backward rules. Fur-

thermore, the developer needs to be aware, that some backward rules will cause problems in the backward translation process and therefore, they need to be excluded. Examples are the backward rules that were derived out of triple rules T_IVARSorARGS_file_input-2-nothing (cf. Fig. 6.3) or T_NEWLINE_expr_stmt-2-empty_POST (cf. Fig. 6.6). The backward translation rule BT_T_NEWLINE_expr_stmt-2-empty_POST would be empty in the SPELL-Flow domain, i.e., it will be applied endlessly during the backward translation. Backward rule BT_T_IVARSorARGS_file_input-2-nothing causes two problems: In the SPELL-Flow domain, no translation markers are set, so that this rule can be applied endlessly, too. Furthermore, it would create a tree of 13 elements in the SPELL-domain. Some nodes also contain attributes. For all attributes, a value needs to be derived, but the translation process is not able to "guess" valid values for all attributes. Therefore, the developer needs to exclude those rules or implement some complex algorithm for guessing valid values. Still, the "origial" IVARS and ARGS statements cannot be generated. In contrast, the backward rule BT_T_file_input-2-StartNode that corresponds to triple rule T_file_input-2-StartNode (cf. Fig. 6.3) is applicable in the backward translation step without any manual adaptions.

The reader may guess already that the final SPELL ASG from which a valid SPELL source code file can be serialised using Xtext, still differs from the original SPELL source code file, even if no changes were performed on the corresponding SPELL-Flow visual model. The reason for this is obvious: During the translation process from SPELL source code to a SPELL-Flow model, information is omitted (e.g. IVARS, ARGS) that cannot be reconstructed. Therefore, the SPELL source code that is created by the backward translation can be seen as skelleton of the real SPELL source code. It needs

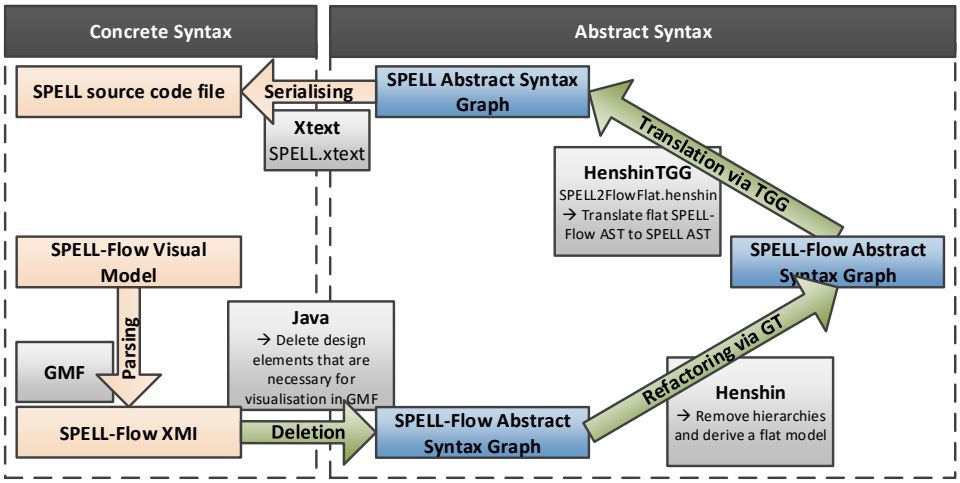


Figure 6.45: Automated Backward Translation

manual checks and adaptions by a SPELL developer.

In oder to achieve an automated backward translation, we need a second implementation similar to Sec. 6.1.4 which calls the relevant steps for the backward translation as described in this subsection. For all single steps, the existing technologies (Xtext, Henshin, HenshinTGG) as well as the metamodels (cf. Sec. 6.1.2) can be reused without any modifications. Similar to Fig. 6.38, the backward translation will follow the following steps as illustrated in Fig. 6.45.

For implementing bidirectional model synchronisation, a suitable method must be selected first, e.g., real-time synchronisation system vs. transaction-based system (cf. discussion in Chap. 7). Then, depending on the method, the implementation of the bidirectional solution stongly depends on the software environment where it shall be integrated at SES, i.e., the SPELL development environment, which is an Eclipse-based IDE for developing SPELL procedures. HenshinTGG and its engine are also Eclipse-based, so that the new implementaion may reuse the model synchronisation implementaion from HenshinTGG (cf. Sec. 6.1.2).

7 Related Work

In this chapter we will review related work with regard to the following two aspects: we will present approaches considering the translation between domain specific languages, especially approaches that are more practical and that are already applied in industrial projects. Then, we will discuss related work in the area of model transformations and synchronisations from a theoretical point of view.

7.1. Related Work with Regard to Practical Applications

In industrial projects, there is a wide acceptance to include concepts of MDE in the development life cycle [HWRK11], especially, in the form of agile MDE where models and source code evolve in parallel in an incremental and iterative form [Mat11].

Model transformations define how to transform a model into another model. In Chap. 1, we introduced the distinction of model transformation approaches based on [CH06]. We will now present greater details of the different approaches and give a short introduction in some tool support that is used in practice. Fig. 7.1 illustrates a detailed distinction of different model transformation approaches and is an extended version of Fig. 1.1. It is based on [CH06] and [DREP12]. In the following, we will give a short summary of the sub-categories of the model-2-text and the model-2-model approaches. Model-2-text approaches encompass the *template-based approach* as well as the *visitor-based approach*.

Visitor-Based Approach: In this easy approach to code generation the internal structure of the model will be traversed and code will be directly derived and written to a text. It is an unidirectional approach. Tool support is avaialable, e.g., the Jamda framework [Boo16]. Jamda is based on Java. It takes an UML model as input, enriches this model

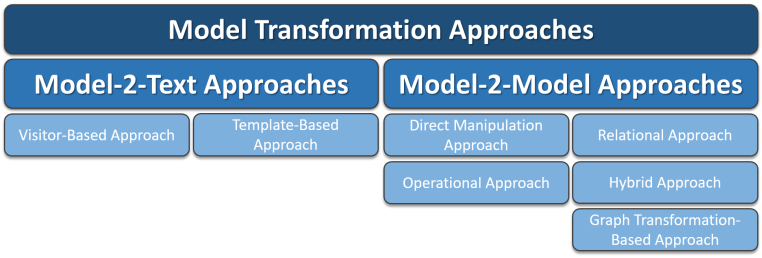


Figure 7.1: Model Transformation Approaches (cf. [CH06])

with necessary classes in order to generate executable Java code out of it.

Template-Based Approach: The template-based approach is the most popular approach in the field of model-driven engineering (MDE). It uses templates (text patterns) that include meta-tags. If a template can be found in the source text or model, it will be replaced by a defined target source code snippet. During this process, the meta-tags will be replaced by elements of the source model. This approach offers the integration of some execution logic, e.g., loops, conditions, etc., in order to guide the translation. Tools using this approach are: AndroMDA [and16, and14], JET [Ecl16c], CodaGen [cod16], FUUT-je [fuu16], MetaEdit+ [Met16], OptimalJ [Opt16], Rational Rose XDE [IBM16], Visual Paradigm for UML [vis16], and many more. We will now take a closer look at AndroMDA and JET.

AndroMDA is an open source framework which takes one or more models as input and generates source code and is also able to setup a new project for it, e.g., a J2EE project. The model is a UML model which is usually stored in XMI format. It is platform-independent. The model will be created by modelling tools that are not part of AndroMDA. The generated code is platform-specific and it depends on the plugins and configurations the developer has chosen. Currently, plugins for generating Java, .Net, HTML, PHP exist. Those plugins can be adapted by the developer or it is even possible to define own plugins for the translation.

JET generates source code or text, respectively, out of EMF models. JET is able to generate Java, SQL, XML, HTML, text and many more using templates. Templates are defined by the developer of the translation.

Model-2-model approaches that we want to introduce in the following are the *direct manipulation approach*, the *operational approach*, the *relational approach*, the *hybrid approach*, and the *graph transformation-based approach*.

Direct Manipulation Approach: The direct manipulation approach is similar to the visitor-based approach of model-2-text. Frameworks usually provida a minimal infrastructure in which the developer defines the transformation rules and the execution logic completely by herself. The Jamda framework [Boo16] supports the direct manipulation approach, too.

Operational Approach: The operational approach is similar to the direct manipulation approach, but extended by certain facilities that enhance the model transformations, e.g., an extension for expressing computations in the meta-model. Tool support is provided by Kermeta [ker16], or specified by QVT-Operational [qvt16a].

The latest version of Kermeta (K3) is built on top of Xtend. It is a framework that supports the developer to extend Ecore meta-models by new features and operations, i.e., the meta-model is equipped with semantics. Kermeta also provides the possibility to define model transformations and constraints on those models that will be executed.

Relational Approach: The main idea behind the relational approach is to define relationships between source and target models using constraints. In general, constraints are non-executable, but they can be equipped with executable semantics. Most relational approaches separate source and target models, i.e., in-place updates are not allowed. Advantages of the relational approach are that they allow the specification of bidirectional transformations, they preserve non-determinism while backtracking is possible. The relational approach is specified or implemented, respectively, in QVT-Relations [QVT16b], Tefkat [tef16] or, AMW [amw16].

Tefkat is based on EMF models. It supports the specification and execution of mappings between source and target models using transformation rules, transformation patterns and transformation templates. Patterns are like rules that can only match elements, in contrast, templates are like rules that can only create or set elements. Rules, patterns and templates are based on F-logic, an ontology language which has a simple SQL-like syntax with logic-based semantics.

Hybrid Approach: Hybrid approaches combine different model-2-model approaches. A prominent example tool support is ATL [atl16].

Graph Transformation-Based Approach: The graph transformation-based approach summarises all kinds of transformation techniques based on (typed) (attributed) graphs (with labels) Chap. 2. It is sometimes also referred to as graph rewriting. The source model have to be a graph and the resulting target model will be a graph, too. The developer specifies the transformation in terms of graph rules.

Triple graph grammars (TGGs) belong to the graph transformation-based approach. Tool support for graph transformation-based approaches is given by GReAT [gre16], VIATRA [Lc16, VBH+16], Henshin [Hen16b, ?, ABJ+10] (cf. Sec. 6.1.2), AGG [agg16, Tae04], and ATOM3 [ato16]. An implementation of graph transformation based on TGG is realised in HenshinTGG [Hen16a] (cf. Sec. 6.1.2), Fujaba [Pad16, NNZ00], and MOFLON [DP16, LAS14]. For a survey on TGG tool support, we refer to [HLG+13, KS06].

Based on the discussion in paper [CH06] we will give a short summary of the applicability of the above-mentioned approaches. The model-2-text template-based approach is the most popular approach in MDE. It is used for code generation. The direct manipulation approach is the most low-level approach. No guidance for the specification of the transformation is available for the developer. In our industrial case study, we introduced the enrichment of design elements (cf. Sec. 6.1.4). This implementation can be seen as direct manipulation approach. The operational approach is an extension of the direct manipulation approach by some computation facilities. With the relational approach it is possible to implement bidirectional transformations. Due to the use of constraints and the underlying constraint solving techniques, the performance of the model transformation strongly depends on the complexity of constraints. Hybrid approaches combine different approaches, so that it is possible to exploit the advantages of the other approaches. Usually, most practical projects are likely to be hybrid approaches. In our industrial case study, we applied mainly the graph transformation-based approach (cf. Sec. 6.1, i.e., the triple graph grammar approach, which is a well-studied area in theoretical computer science. In general, graph transformation is based on graphs and in our application, the source language (SPELL), as well as the visual target language (SPELL-Flow) are represented by abstract syntax trees (ASTs). The SPELL language is parsed to an AST using Xtext. In contrast, the SPELL-Flow AST will be directly stored to an XMI file, i.e., no additional transformation is necessary. Therefore, the source and target instances are represented in ASTs, so that the translation form an AST to a visual diagram (e.g., UML model) can be easily done. Using the ASTs forms the basis for defining (triple) graph transformation rules. According to [CH06], graph transformation and also triple graph transformation is an intuitive approach due to the set of visual transformation rules and due to the declarative character of this approach. The disadvantages, which were mentioned in [CH06], e.g., termination and non-determinism, and also analysis methods using triple graph grammars were discussed in past works in the meantime [KW07, HEO+11a, HEGO14, HEOG10, HEGO10a, EEHP09] and also in this thesis solutions and optimisations for dealing with those problems were presented (cf. Sec. 4.4). Furthermore, extensions for structuring and guid-

ing the rule application process were developed and implemented in different tools, so that this approach is getting more and more feasible for practical projects like our industrial case study [KKS97, LAST15].

7.2. Related Work with Regard to Formal Framework

The formal foundations of graph transformations and triple graph transformations are introduced in Chap. 2. Furthermore, we presented the model ((non-deterministic) concurrent) synchronisation approach based on TGGs which we use as basis in this work, in Chap. 4. This model synchronisation approach is inspired by the lens framework [FGM+07]. The basic idea of the lens framework is that one model is the view model of the other model, i.e., the view can be seen as a subset of the other model or as a different abstracion. In general, a view in our understanding describes a different abstraction level of a model, while the same information is possibly available in the model and its corresponding view(s) [EEEP10]. As a consequence, we require some explicit mapping which defines the dependency of elements in the view(s) and elements in the corresponding model. In the lens framework, model synchronisation is performed asymmetrically in state-based manner, i.e., the framework takes the models before and after the model update as input and calculates the output models. Another approach is the synchronous case of the lens framework. There, one model cannot be a view of the other model [HPW11]. An extension is the delta-lens framework [DXC+11] in which the deltas, i.e., the changes of the model update, are taken as input and output. The ((non-deterministic) concurrent) synchronisation approach based on TGGs [HEO+15, HEEO12, GHN+13a] is inspired by the delta-lens framework and can be seen as a simplified solution with regard to the delta-lens approach.

Several works discuss multi-view models. Another approach are view triple graph grammars (ViewTGGs) [JKS06, JS08, ARDS14]. It is imaginable to interpret the multi-view model in one domain as ViewTGG. In our running example, each abstraction layer can be considered as separate views. In order to apply and compare the theory of ViewTGGs to examples which were treated by the proposed derived propagation framework (cf. Chap. 5), e.g., the industrial case study. The multi-view model needs to be split into separate TGGs: each abstraction (view) belongs to a new domain model, i.e., the modest detailed layer and each view will be reflected by separate TGGs. Then, it is possible to apply the model synchronisation techniques from [ARDS14]. However, the approach we propose in this thesis is more flexible and is also able to deal with models, where the concept of ViewTGGs cannot be applied, but where elements in one domain are still strongly interweaved with each other.

In [TA15, TA16] the authors introduce a new approach for handling multiple models that also include multi-view models, called the *graph diagrams approach*. The graph diagrams approach is a generalisation of triple graph grammars to an arbitrary number of models and relations. Relations can connect more than two models with each other. A diagram base specifies which kind of models and relations are available and a graph diagram is an instance. Thus, graph diagrams are able to reflect n:m relationships. In addition the authors define translation rules and lift the concept of the ((non-deterministic) concurrent) synchronisation approach based on TGGs to graph diagrams in [TA15, TA16]. The propsed derived propagation framework (cf. Chap. 5) is equivalent to the graph diagrams approach in cases where no elements will be recreated during the propagation, because the graph diagrams approach does not recreate elements, but still, they discuss the necessity of recreations. It is very interesting to combine both approaches, because the graph diagrams approach enebales to propagate changes in multiple models which are in a kind of n:m relation with each other. Furthermore, both approaches are based on the same model synchronisation framework, i.e., the ((non-deterministic) concurrent) synchronisation approach based on TGGs that we introduced and extended in Chap. 4.

Conclusion & Future Work

8

In the framework of the industrial case study, as introduced in Sec. 1.1, we developed a prototype of an automated translation of satellite procedures into their visual representations (visualisation) and discussed the possibility of also applying the backward direction in order to generate code out of visual models.

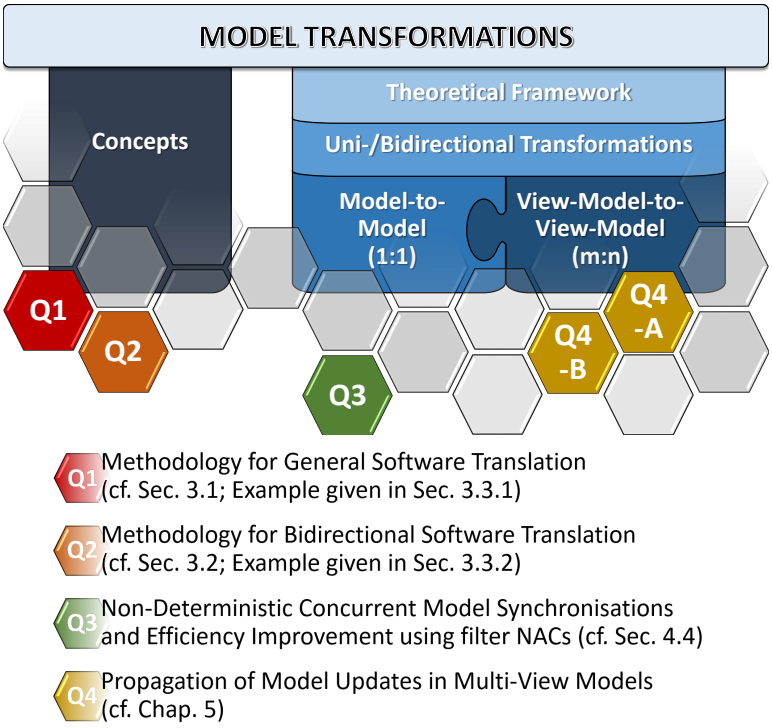


Figure 8.1: Research Questions: Summary, Solutions & Classification of Research Areas

The (bidirectional) translation is based on the formal framework of model transformations using triple graph grammars (TGGs) [Sch95]. During the development of the prototype, several theoretical research questions evolved (cf. Sec. 1.2) which we answered within this thesis. Fig. 8.1 summarises the research questions and the chapters and sections in which we discussed their solutions. Research questions *Q1* and *Q2* ask for a general methodology for applying uni- and bidirectional model transformations, i.e., for translating concrete syntax (e.g., source code) to another concrete syntax and vice versa. In explicit, both questions are:

According to which concept is it possible to transform model $\mathcal{L}1$ *to model* $\mathcal{L}2$*? (Q1).*

According to which concept is it possible to transform model $\mathcal{L}1$ *to model* $\mathcal{L}2$ *and vice versa? (Q2).*

We presented general concepts in Chap. 3 for the translation of concrete syntax or instances of language $\mathcal{L}1$, respectively, to concrete syntax or instances of language $\mathcal{L}2$ (unidirectional, Q1) and also vice versa (bidirectional, Q2). We have shown that both concepts are applicable in practice. In Sec. 3.3 we applied the concept of unidirectional translation to the industrial case study and discussed how the bidirectional approach will be applied to the case study in order to achieve a bidirectional translation. In future work, it would be interesting to extend the industrial case study so that the backward direction (from visualisation to source code) is realised in a prototype, too. Then, it is possible to show that the methodology for the bidirectional translation is applicable, too.

Both research questions ask for concepts in the area of unidirectional as well as bidirectional model transformations, which is illustrated in Fig. 8.1 by arranging both hexagons *Q1* and *Q2* in the close neighbourhood of the *Concepts* box.

Research question *Q3* deals with an extension of the theoretical framework in the domain of concurrent model synchronisations. In Fig. 8.1 we categorised this research question to the box of uni- and bidirectional 1:1 model transformations, i.e., currently, the concurrent model synchronisation framework deals with default triple models, and does not consider recurring elements in the same domain, as it is the case in layered models (cf. research question *Q4*). In explicit, *Q3* is formulated as follows:

In which way is it possible to treat a non-deterministic set of rules in concurrent model synchronisations? (Q3)

After introducing the model synchronisation framework and the extension to the concurrent model synchronisation framework in Sects. 4.2 and 4.3, we are able to present our approach of a generalised concurrent model synchronisation framework in Sec. 4.4. It is based on the concurrent model synchronisation framework, but extended by filter NACs (cf. Def. 2.2.14) in order to reduce backtracking steps for conflicting rule applications and therefore improve the efficiency of the model synchronisation

framework.

The last research question *Q4* which we discussed in this thesis is divided in two parts:

If a model update in one view is performed, then how is it possible to consistently propagate this model update to all other views (Q4-A) and also to the other domain? (Q4-B).

In Chap. 5, we present our new solution for both parts of the question, i.e., we introduced our derived propagation framework which is able to propagate a model update in one domain to the other domain, and also to other elements in the same domain which repeat themselves in the same domain or which are strongly interweaved with elements that are modified by the domain model update and therefore need to be updated, too. In the thesis, we use the terms *views* or *layers* to describe depending elements in the same domain.

In Fig. 8.1, we classified *Q4* to the theoretical framework of m:n model transformations.

We are confident that the derived propagation framework can be easily applied to concurrent model updates in both domains. Necessary extensions that treat conflicting domain model updates in the derived propagation framework will be similar to the extensions that were necessary for the concurrent model synchronisation framework. It is up to future work to investigate concurrent model updates in the conctext of the derived propagation framework. Furthermore, different approaches exists that discuss view-based model transformations, e.g. [TA15] and also view triple graph grammars [JKS06, ARDS14]. It is also desired as future work to check the compatibility of our approach with those concepts and to extend the derived propagation framework so that those works will be covered, too.

Another solution for propagating model updates in multi view models is based on the existing theory of model synchronisations Chap. 4 [EEPT06, HEO+15]. If we assume a separate model and TGG for each view, then, a domain model update in one view will be propagated to the source model and also to all other view(s), i.e., the separate TGGs can be understood as composition. This solution considers different models for each view as a separate model which is the main difference with regard to the derived propagation framework which we propose in Chap. 5. The split into separate TGGs rises questions concerning the conflict resolution of simoultaneous updates in several views. Another question to be discussed would be, how to deal with model updates that consider only views and are not reflected by the other domain model. Then, the propagation of the model update may get lost due to the missing dependency. The development and discussion of this solution and also the comparison with the derived propagation framework is topic of future work.

In general, it is also desired to implement the theoretical findings that resulted out of *Q3* and *Q4* in suitable triple graph transformation tools,

like *HenshinTGG*, which already includes an implementation of the model synchronisation framework [HEO+15]. Then, it is possible to apply those frameworks in industrial case studies and real-world scenarios in order to close the gap between theoretical concepts and their realisations, especially in industrial contexts.

Bibliography

[ABJ+10] Arendt, Thorsten ; Biermann, Enrico ; Jurack, Stefan ; Krause, Christian ; Taentzer, Gabriele ; Petriu, Dorina C. (Hrsg.) ; Rouquette, Nicolas (Hrsg.) ; Haugen, Øystein (Hrsg.): *Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations*. http://dx.doi.org/10.1007/978-3-642-16145-2_9. Version: 2010

[agg16] *AGG*. http://www.user.tu-berlin.de/o.runge/agg/. Version: 2016. – Visited 2016

[AHS90] Adámek, Jiří ; Herrlich, Horst ; Strecker, George: *Abstract and Concrete Categories: The Joy of Cats*. New York : John Wiley and Sons, 1990 http://katmat.math.uni-bremen.de/acc/acc.pdf. – Republished in: Reprints in Theory and Applications of Categories, No. 17 (2006) pp. 1-507

[ALS15] Anjorin, Anthony ; Leblebici, Erhan ; Schürr, Andy: 20 Years of Triple Graph Grammars: A Roadmap for Future Research. In: *ECEASST* 73 (2015). http://journal.ub.tu-berlin.de/eceasst/article/view/1031

[amw16] *Atlas Model Weaver (AMW)*. http://www.eclipse.org/gmt/amw/. Version: 2016. – Visited 2016

[and14] *AndroMDA*. http://andromda.sourceforge.net/. Version: 2014. – Visited 2016

[and16] *AndroMDA*. http://www.andromda.org/. Version: 2016. – Visited 2016

[ARDS14] Anjorin, Anthony ; Rose, Sebastian ; Deckwerth, Frederik ; Schürr, Andy: Efficient Model Synchronization with View Triple Graph Grammars. Version: 2014. http:

//dx.doi.org/{10.1007/978-3-319-09195-2_1}. In: Cabot, Jordi (Hrsg.) ; Rubin, Julia (Hrsg.): *Modelling Foundations and Applications* Bd. 8569. Springer International Publishing, 2014. – DOI 10.1007/978–3–319–09195–2$_1$. – – *ISBN*978 – –3 – –319 – –09194 – –5, 1 – 17

[atl16] *ATL*. http://www.eclipse.org/atl/. Version: 2016. – Visited 2016

[ato16] *Atom3*. http://atom3.cs.mcgill.ca/. Version: 2016. – Visited 2016

[BBG+60] Backus, John W. ; Bauer, Friedrich L. ; Green, Julian ; Katz, Charles ; McCarthy, John ; Perlis, Alan J. ; Rutishauser, Heinz ; Samelson, Klaus ; Vauquois, Bernard ; Wegstein, Joseph H. ; Wijngaarden, Adriaan van ; Woodger, Mike: Report on the Algorithmic Language ALGOL 60. In: *Commun. ACM* 3 (1960), Mai, Nr. 5, 299–314. http://dx.doi.org/10.1145/367236.367262. – DOI 10.1145/367236.367262. – ISSN 0001–0782

[BEL+03] Bardohl, Roswitha ; Ehrig, Hartmut ; Lara, Juan de ; Runge, Olge ; Taentzer, Gabriele ; Weinhold, Ingo: Node Type Inheritance Concept for Typed Graph Transformation / Technische Universität Berlin. Version: 2003. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.4.9257&rep=rep1&type=pdf. 2003 (2003-19). – Forschungsbericht. – ISSN 1436–9915

[BET08] Biermann, Enrico ; Ermel, Claudia ; Taentzer, Gabriele: Precise Semantics of EMF Model Transformations by Graph Transformation. In: Czarnecki, Krzysztof (Hrsg.) ; Ober, Ileana (Hrsg.) ; Bruel, Jean-Michel (Hrsg.) ; Uhl, Axel (Hrsg.) ; Völter, Markus (Hrsg.): *Model Driven Engineering Languages and Systems: 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008. – ISBN 978–3–540–87875–9, 53–67

[BET12] Biermann, Enrico ; Ermel, Claudia ; Taentzer, Gabriele: Formal foundation of consistent EMF model transformations by algebraic graph transformation. In: *Software & Systems Modeling* 11 (2012), Nr. 2, 227–250. http://dx.doi.org/10.1007/s10270-011-0199-7. – DOI 10.1007/s10270–011–0199–7. – ISSN 1619–1374

[BH02] Baresi, Luciano ; Heckel, Reiko ; Corradini, Andrea (Hrsg.) ; Ehrig, Hartmut (Hrsg.) ; Kreowski, Hans J. (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Tutorial Introduction to Graph Transformation: A Software Engineering Perspective*. http://dx.doi.org/10.1007/3-540-45832-8_30. Version: 2002

[Boo16] Boocock, Paul: *Jamda*. http://jamda.sourceforge.net/. Version: 2016. – Visited 2016

[CH06] Czarnecki, Krzysztof ; Helsen, Simon: Feature-based Survey of Model Transformation Approaches. Riverton, NJ, USA : IBM Corp., July 2006. – ISSN 0018–8670, 621–645

[Cho59] Chomsky, Noam: On certain formal properties of grammars. In: *Information and Control* 2 (1959), Nr. 2, 137 - 167. http://dx.doi.org/http://dx.doi.org/10.1016/S0019-9958(59)90362-6. – DOI http://dx.doi.org/10.1016/S0019–9958(59)90362–6. – ISSN 0019–9958

[CNB15] Chinchilla, Rafael ; Noguero, Javier ; Bouleau, Fabien: *SPELL - Language Reference - 2.4.4*. https://sourceforge.net/projects/spell-sat/files/Documentation/. Version: 2015

[cod16] *CodeGen*. http://www.omg.org/mda/mda_files/codagen_technologies.htm. Version: 2016. – Visited 2016

[Dis11] Diskin, Zinovy: Model Synchronization: Mappings, Tiles, and Categories. In: Fernandes, Joã. (Hrsg.) ; Lämmel, Ralf (Hrsg.) ; Visser, Joost (Hrsg.) ; Saraiva, João (Hrsg.): *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. – ISBN 978–3–642–18023–1, 92–165

[DP16] Darmstadt, TU ; Paderborn, University of: *eMoflon*. https://emoflon.github.io/. Version: 2016. – Visited 2016

[DREP12] Di Ruscio, Davide ; Eramo, Romina ; Pierantonio, Alfonso: Model Transformations. In: Bernardo, Marco (Hrsg.) ; Cortellessa, Vittorio (Hrsg.) ; Pierantonio, Alfonso (Hrsg.): *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. – ISBN 978–3–642–30982–3, 91–136

[DXC+11] Diskin, Zinovy ; Xiong, Yingfei ; Czarnecki, Krzysztof ; Ehrig, Hartmut ; Hermann, Frank ; Orejas, Fernando: From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In: *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, 2011, 304–318

[EBN96] *ISO/IEC 14977:1996*. http://www.iso.org/iso/catalogue_detail?csnumber=26153. Version: 1996

[ecl16a] *Eclipse*. https://eclipse.org/. Version: 2016

[Ecl16b] *Henshin*. http://www.eclipse.org/henshin/. Version: 2016

[Ecl16c] Eclipse.org: *Java Emitter Template (JET)*. http://www.eclipse.org/modeling/m2t/?project=jet#jet. Version: 2016. – Visited 2016

[EEE+07] Ehrig, Hartmut ; Ehrig, Karsten ; Ermel, Claudia ; Hermann, Frank ; Taentzer, Gabriele ; Dwyer, Matthew B. (Hrsg.) ; Lopes, Antónia (Hrsg.): *Information Preserving Bidirectional Model Transformations*. http://dx.doi.org/10.1007/978-3-540-71289-3_7. Version: 2007

[EEEP10] Ehrig, Hartmut ; Ehrig, Karsten ; Ermel, Claudia ; Prange, Ulrike: Consistent integration of models based on views of meta models. In: *Formal Aspects of Computing* 22 (2010), Nr. 3, 327–344. http://dx.doi.org/10.1007/s00165-009-0127-6. – DOI 10.1007/s00165–009–0127–6. – ISSN 1433–299X

[EEGH15] Ehrig, Hartmut ; Ermel, Claudia ; Golas, Ulrike ; Hermann, Frank: *Graph and Model Transformation: General Framework and Applications*. Springer Berlin Heidelberg, 2015. http://dx.doi.org/10.1007/978-3-662-47980-3. http://dx.doi.org/10.1007/978-3-662-47980-3. – ISBN 978–3–662–47979–7

[EEH08] Ehrig, Hartmut ; Ehrig, Karsten ; Hermann, Frank: From Model Transformation to Model Integration based on the Algebraic Approach to Triple Graph Grammars. In: *ECEASST* 10 (2008). http://dx.doi.org/10.14279/tuj.eceasst.10.154. – DOI 10.14279/tuj.eceasst.10.154. – ISSN 1863–2122

[EEHP06] Ehrig, Hartmut ; Ehrig, Karsten ; Habel, Annegret ; Pennemann, Karl-Heinz: Theory of Constraints and Application

Conditions: From Graphs to High-Level Structures. In: *Fundam. Inf.* 74 (2006), Oktober, Nr. 1, 135–166. http://dl.acm.org/citation.cfm?id=1231199.1231206. – ISSN 0169–2968

[EEHP09] Ehrig, Hartmut ; Ermel, Claudia ; Hermann, Frank ; Prange, Ulrike ; Schürr, Andy (Hrsg.) ; Selic, Bran (Hrsg.): *On-the-Fly Construction, Correctness and Completeness of Model Transformations Based on Triple Graph Grammars*. http://dx.doi.org/10.1007/978-3-642-04425-0_18. Version: 2009

[EEKR99] Ehrig, Hartmut (Hrsg.) ; Engels, Gregor (Hrsg.) ; Kreowski, Hans-Jörg (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation: Volume II. Applications, Languages and Tools*. River Edge, NJ, USA : World Scientific Publishing Co., Inc., 1999. – ISBN 98–102402–01

[EEPT06] Ehrig, Hartmut ; Ehrig, Karsten ; Prange, Ulrike ; Taentzer, Gabriele: *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer Berlin Heidelberg, 2006. http://dx.doi.org/10.1007/3-540-31188-2. http://dx.doi.org/10.1007/3-540-31188-2. – ISBN 978–3–540–31187–4

[EET11a] Ehrig, Hartmut ; Ermel, Claudia ; Taentzer, Gabriele: A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications. In: *Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, 2011, 202–216

[EET11b] Ehrig, Hartmut ; Ermel, Claudia ; Taentzer, Gabriele: A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications / Technische Universität Berlin. Version: 2011. http://www.eecs.tu-berlin.de/fileadmin/f4/TechReports/2011/tr_2011-01.pdf. 2011 (2011-01). – Forschungsbericht. – 33 S.. – ISSN 1436–9915

[EGRW96] Ehrig, H. ; Grosse-Rhode, M. ; Wolter, U.: On the role of category theory in the area of algebraic specifications. In: Haveraaen, Magne (Hrsg.) ; Owe, Olaf (Hrsg.) ; Dahl, Ole-Johan (Hrsg.): *Recent Trends in Data Type Specification: 11th Workshop on Specification of Abstract Data Types Joint with*

the 8th COMPASS Workshop Oslo, Norway, September 19–23, 1995 Selected Papers. Berlin, Heidelberg : Springer Berlin Heidelberg, 1996. – ISBN 978–3–540–70642–7, 17–48

[EH86] Ehrig, Hartmut ; Habel, Annegret: *Graph Grammars with Application Conditions*. http://dx.doi.org/10.1007/978-3-642-95486-3_7. Version: 1986

[EHGB12] Ermel, Claudia ; Hermann, Frank ; Gall, Jürgen ; Binanzer, Daniel: Visual Modeling and Analysis of EMF Model Transformations Based on Triple Graph Grammars. In: *ECEASST* 54 (2012), 1–14. http://journal.ub.tu-berlin.de/eceasst/article/view/771. ISBN ISSN 1863–2122

[EHK+97] Ehrig, Hartmut ; Heckel, Reiko ; Korff, Martin ; Löwe, Michael ; Ribeiro, Leila ; Wagner, Annika ; Corradini, Andrea: Handbook of Graph Grammars and Computing by Graph Transformation. Version: 1997. http://dl.acm.org/citation.cfm?id=278918.278930. River Edge, NJ, USA : World Scientific Publishing Co., Inc., 1997. – ISBN 98–102288–48, Kapitel Algebraic Approaches to Graph Transformation. Part II: Single Pushout Approach and Comparison with Double Pushout Approach

[Ehr79] Ehrig, Hartmut: Introduction to the algebraic theory of graph grammars (a survey). In: Claus, Volker (Hrsg.) ; Ehrig, Hartmut (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Graph-Grammars and Their Application to Computer Science and Biology: International Workshop Bad Honnef, October 30 – November 3, 1978*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1979. – ISBN 978–3–540–35091–0, 1–69

[EKMR99] Ehrig, Hartmut ; Kreowski, Hans-Jörg ; Montanari, Ugo ; Rozenberg, Grzegorz: *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism, and Distribution*. River Edge, NJ, USA : World Scientific Publishing Co., Inc., 1999 http://dl.acm.org/citation.cfm?id=320647. – ISBN 9–810240–21–X

[EM45] Eilenberg, Samuel ; MacLane, Saunders: General theory of natural equivalences. In: *Trans. Amer. Math. Soc.* 58 (1945), 231–291. http://dx.doi.org/10.1090/S0002-9947-1945-0013131-6. – DOI 10.1090/S0002–9947–1945–0013131–6

[EM85] Ehrig, Hartmut ; Mahr, Bernd: *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*.

Bd. 6. Springer Berlin Heidelberg, 1985. http://dx.doi.org/10.1007/978-3-642-69962-7. http://dx.doi.org/10.1007/978-3-642-69962-7. – ISBN 978–3–642–69964–1

[EMC+01] Ehrig, Hartmut ; Mahr, Bernd ; Cornelius, Felix ; Grosse-Rhode, Martin ; Zeitz, Philip: *Springer-Lehrbuch*. Bd. 2: *Mathematisch-strukturelle Grundlagen der Informatik*. Springer Berlin Heidelberg, 2001. http://dx.doi.org/10.1007/978-3-642-56792-6. http://dx.doi.org/10.1007/978-3-642-56792-6. – ISBN 3–540–41923–3

[EMF16] *Eclipse EMF*. https://eclipse.org/modeling/emf/. Version: 2016

[EPS73] Ehrig, Hartmut ; Pfender, Michael ; Schneider, Hans J.: Graph-grammars: An algebraic approach. In: *Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on*, 1973. – ISSN 0272–4847, S. 167–180

[EPT04] Ehrig, Hartmut ; Prange, Ulrike ; Taentzer, Gabriele ; Ehrig, Hartmut (Hrsg.) ; Engels, Gregor (Hrsg.) ; Parisi-Presicce, Francesco (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Fundamental Theory for Typed Attributed Graph Transformation*. http://dx.doi.org/10.1007/978-3-540-30203-2_13. Version: 2004

[FC169] *Flowcharting Techniques*. http://www.eah-jena.de/~kleine/history/software/IBM-FlowchartingTechniques-GC20-8152-1.pdf. Version: 1969

[FGM+07] Foster, J. N. ; Greenwald, Michael B. ; Moore, Jonathan T. ; Pierce, Benjamin C. ; Schmitt, Alan: Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. In: *ACM Trans. Program. Lang. Syst.* 29 (2007), Mai, Nr. 3. http://dx.doi.org/10.1145/1232420.1232424. – DOI 10.1145/1232420.1232424. – ISSN 0164–0925

[fuu16] *FUUT-je*. http://www.program-transformation.org/Gmt/FuutjeModeler. Version: 2016. – Visited 2016

[GEF16] *GEF*. http://www.eclipse.org/gef/. Version: 2016

[GEH11] Golas, Ulrike ; Ehrig, Hartmut ; Hermann, Frank: Formal Specification of Model Transformations by Triple Graph Grammars with Application Conditions. In: *ECEASST* 39 (2011), 1 – 26. http://journal.ub.tu-berlin.de/eceasst/article/view/646

[GHE+13] Gottmann, Susann ; Hermann, Frank ; Ermel, Claudia ; Engel, Thomas ; Morelli, Gianluigi: Towards Bidirectional Engineering of Satellite Control Procedures Using Triple Graph Grammars. In: *Proceedings of the 7th Workshop on Multi-Paradigm Modeling co-located with the 16th International Conference on Model Driven Engineering Languages and Systems, MPM@MoDELS 2013, Miami, Florida, September 30, 2013.*, 2013, 67–76

[GHN+13a] Gottmann, Susann ; Hermann, Frank ; Nachtigall, Nico ; Braatz, Benjamin ; Ermel, Claudia ; Ehrig, Hartmut ; Engel, Thomas: Correctness and Completeness of Generalised Concurrent Model Synchronisation Based on Triple Graph Grammars. In: *Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, September 29, 2013*, 2013

[GHN+13b] Gottmann, Susann ; Hermann, Frank ; Nachtigall, Nico ; Braatz, Benjamin ; Ermel, Claudia ; Ehrig, Hartmut ; Engel, Thomas: Correctness of Generalisation and Customisation of Concurrent Model Synchronisation Based on Triple Graph Grammars / Technische Universität Berlin, Fak. IV. Version: 2013. http://www.eecs.tu-berlin.de/fileadmin/f4/TechReports/2013/tr_2013-08.pdf. 2013 (2013-08). – Forschungsbericht. – ISSN 1436–9915

[GL06] Guerra, Esther ; Lara, Juan de: Attributed Typed Triple Graph Transformation with Inheritance in the Double Pushout Approach / Universidad Carlos III de Madrid. Version: 2006. http://arantxa.ii.uam.es/~jlara/investigacion/techRep_UC3M.pdf. 2006 (UC3M-TR-CS-06-01). – Forschungsbericht. – 61 S.

[GLEO12a] Golas, Ulrike ; Lambers, Leen ; Ehrig, Hartmut ; Orejas, Fernando: Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs. In: *Theor. Comput. Sci.* 424 (2012), 46–68. http://dx.doi.org/{10.1016/j.tcs.2012.01.032}. – DOI 10.1016/j.tcs.2012.01.032

[GLEO12b] Golas, Ulrike ; Lambers, Leen ; Ehrig, Hartmut ; Orejas, Fernando: Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs. In: *Theoretical Computer Science* 424 (2012), 46 - 68. http://dx.

doi.org/http://dx.doi.org/10.1016/j.tcs.2012.01.032. – DOI http://dx.doi.org/10.1016/j.tcs.2012.01.032. – ISSN 0304–3975

[GMF16] *Eclipse GMF*. http://www.eclipse.org/modeling/gmp/. Version: 2016

[GNE+16a] Gottmann, Susann ; Nachtigall, Nico ; Engel, Thomas ; Ermel, Claudia ; Hermann, Frank: Propagation of Model Updates along different Views in Multi-View Models - Extended Version / SnT. Version: 2016. TODO. 2016 (TODO). – Forschungsbericht. – To appear

[GNE+16b] Gottmann, Susann ; Nachtigall, Nico ; Engel, Thomas ; Ermel, Claudia ; Hermann, Frank: Towards the Propagation of Model Updates along different Views in Multi-View Models. In: *Proceedings of the 5th International Workshop on Bidirectional Transformations* 1571 (2016), S. 45–60

[Gol11] Golas, Ulrike: *Analysis and Correctness of Algebraic Graph and Model Transformations*. Vieweg+Teubner, 2011. http://dx.doi.org/10.1007/978-3-8348-9934-7. http://dx.doi.org/10.1007/978-3-8348-9934-7. – ISBN 978–3–8348–1493–7

[Gon15] Gonçalves, Bruno P.: *Generation of layout information for a visualization of satellite control procedures in SPELL based on graph transformation and Eclipse*. Luxembourg, Université du Luxembourg, Master's Thesis, 2015

[gre16] *Graph Rewriting and Transformation (GReAT)*. http://www.isis.vanderbilt.edu/tools/great. Version: 2016. – Visited 2016

[Hec06] Heckel, Reiko: Graph Transformation in a Nutshell, 2006. – ISSN 1571–0661, 187 – 198

[HEEO12] Hermann, Frank ; Ehrig, Hartmut ; Ermel, Claudia ; Orejas, Fernando: Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars. In: *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, 2012, 178–193

[HEGO10a] Hermann, Frank ; Ehrig, Hartmut ; Golas, Ulrike ; Orejas, Fernando: 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*. New York, NY, USA : ACM, 2010 (MDI '10). – ISBN 978–1–4503–0292–0, 22–31

[HEGO10b] Hermann, Frank ; Ehrig, Hartmut ; Golas, Ulrike ; Orejas, Fernando: Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars / Technische Universität Berlin. Version: 2010. http://www.eecs.tu-berlin.de/fileadmin/f4/TechReports/2010/tr_2010-13.pdf. 2010 (2010-13). – Forschungsbericht. – 28 S.. – ISSN 1436–9915

[HEGO14] Hermann, Frank ; Ehrig, Hartmut ; Golas, Ulrike ; Orejas, Fernando: Formal Analysis of Model Transformations Based on Triple Graph Grammars. In: *Mathematical Structures in Computer Science* 24 (2014), Nr. 4. http://dx.doi.org/{10.1017/S0960129512000370}. – DOI 10.1017/S0960129512000370

[Hen16a] *HenshinTGG*. http://de-tu-berlin-tfs.github.io/Henshin-Editor/. Version: 2016

[Hen16b] *Visual Multi-View Henshin-Editor*. http://www.user.tu-berlin.de/lieske/tfs/projekte/henshin/. Version: 2016

[HEO+11a] Hermann, Frank ; Ehrig, Hartmut ; Orejas, Fernando ; Czarnecki, Krzysztof ; Diskin, Zinovy ; Xiong, Yingfei: Correctness of Model Synchronization Based on Triple Graph Grammars. In: Whittle, Jon (Hrsg.) ; Clark, Tony (Hrsg.) ; Kühne, Thomas (Hrsg.): *Model Driven Engineering Languages and Systems: 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. – ISBN 978–3–642–24485–8, 668–682

[HEO+11b] Hermann, Frank ; Ehrig, Hartmut ; Orejas, Fernando ; Czarnecki, Krzysztof ; Diskin, Zinovy ; Xiong, Yingfei: Correctness of Model Synchronization Based on Triple Graph Grammars - Extended Version / TU Berlin, Fak. IV. Version: 2011. https://www.eecs.tu-berlin.de/fileadmin/f4/TechReports/2011/tr_2011-07.pdf. 2011 (TR 2011-07). – Forschungsbericht

[HEO+15] Hermann, Frank ; Ehrig, Hartmut ; Orejas, Fernando ; Czarnecki, Krzysztof ; Diskin, Zinovy ; Xiong, Yingfei ; Gottmann, Susann ; Engel, Thomas: Model synchronization based on triple graph grammars: correctness, completeness and

invertibility. In: *Software and System Modeling* 14 (2015), Nr. 1, 241–269. http://dx.doi.org/{10.1007/s10270-012-0309-1}. – DOI 10.1007/s10270–012–0309–1

[HEOG10] Hermann, Frank ; Ehrig, Hartmut ; Orejas, Fernando ; Golas, Ulrike ; Ehrig, Hartmut (Hrsg.) ; Rensink, Arend (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.) ; Schürr, Andy (Hrsg.): *Formal Analysis of Functional Behaviour for Model Transformations Based on Triple Graph Grammars*. http://dx.doi.org/10.1007/978-3-642-15928-2_11. Version: 2010

[HGN+13] Hermann, Frank ; Gottmann, Susann ; Nachtigall, Nico ; Braatz, Benjamin ; Morelli, Gianluigi ; Pierre, Alain ; Engel, Thomas: On an Automated Translation of Satellite Procedures Using Triple Graph Grammars. In: Duddy, Keith (Hrsg.) ; Kappel, Gerti (Hrsg.): *Theory and Practice of Model Transformations* Bd. 7909, Springer Berlin Heidelberg, 2013 (Lecture Notes in Computer Science). – ISBN 978–3–642–38882–8, 50-51

[HGN+14a] Hermann, Frank ; Gottmann, Susann ; Nachtigall, Nico ; Ehrig, Hartmut ; Braatz, Benjamin ; Morelli, Gianluigi ; Pierre, Alain ; Engel, Thomas ; Ermel, Claudia: Triple Graph Grammars in the Large for Translating Satellite Procedures. Version: 2014. http://dx.doi.org/{10.1007/978-3-319-08789-4_9}. In: Di Ruscio, Davide (Hrsg.) ; Varró, Dániel (Hrsg.): *Theory and Practice of Model Transformations* Bd. 8568. Springer International Publishing, 2014. – DOI 10.1007/978–3–319–08789–4$_9$. – $-ISBN978--3--319--08788--7, 122-137$

[HGN+14b] Hermann, Frank ; Gottmann, Susann ; Nachtigall, Nico ; Ehrig, Hartmut ; Braatz, Benjamin ; Morelli, Gianluigi ; Pierre, Alain ; Engel, Thomas ; Ermel, Claudia: Triple Graph Grammars in the Large for Translating Satellite Procedures - Extended Version / SnT. Version: 2014. http://orbilu.uni.lu/handle/10993/16887. 2014 (TR-SnT-2014-7). – Forschungsbericht. – ISBN 978–2–87971–128–7

[HHT96] Habel, Annegret ; Heckel, Reiko ; Taentzer, Gabriele: Graph Grammars with Negative Application Conditions. In: *Fundam. Inf.* 26 (1996), Dezember, Nr. 3,4, 287–313. http://dl.acm.org/citation.cfm?id=2379538.2379542. – ISSN 0169–2968

[HLG+13] Hildebrandt, Stephan ; Lambers, Leen ; Giese, Holger ; Rieke, Jan ; Greenyer, Joel ; Schäfer, Wilhelm ; Lauder, Marius ; Anjorin, Anthony ; Schürr, Andy: A Survey of Triple Graph Grammar Tools. In: *ECEASST* 57 (2013). http://journal.ub.tu-berlin.de/eceasst/article/view/865

[HP05] Habel, Annegret ; Pennemann, Karl-Heinz ; Kreowski, Hans-Jörg (Hrsg.) ; Montanari, Ugo (Hrsg.) ; Orejas, Fernando (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.) ; Taentzer, Gabriele (Hrsg.): *Nested Constraints and Application Conditions for High-Level Structures*. http://dx.doi.org/10.1007/978-3-540-31847-7_17. Version: 2005

[HPW11] Hofmann, Martin ; Pierce, Benjamin ; Wagner, Daniel: Symmetric Lenses. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 2011 (POPL '11). – ISBN 978–1–4503–0490–0, 371–384

[HWRK11] Hutchinson, John ; Whittle, Jon ; Rouncefield, Mark ; Kristoffersen, Steinar: Empirical Assessment of MDE in Industry. In: *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA : ACM, 2011 (ICSE '11). – ISBN 978–1–4503–0445–0, 471–480

[IBM16] IBM: *IBM Rational Rose XDE: eXtended Development Environment*. http://www-03.ibm.com/software/products/en/enterprise/. Version: 2016. – Visited 2016

[jav16] *Java*. https://www.java.com/. Version: 2016

[JKS06] Jakob, Johannes ; Königs, Alexander ; Schürr, Andy: Non-materialized Model View Specification with Triple Graph Grammars. Version: 2006. http://dx.doi.org/{10.1007/11841883_23}. In: Corradini, Andrea (Hrsg.) ; Ehrig, Hartmut (Hrsg.) ; Montanari, Ugo (Hrsg.) ; Ribeiro, Leila (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Graph Transformations* Bd. 4178. Springer Berlin Heidelberg, 2006. – DOI $10.1007/11841883_23$. – $-ISBN 978--3--540--38870--8, 321--335$

[JS08] Jakob, Johannes ; Schürr, Andy: View Creation of Meta Models by Using Modified Triple Graph Grammars . In: *Electronic Notes in Theoretical Computer Science* 211 (2008), Nr. 0, 181 - 190. http://dx.doi.org/{http://dx.doi.org/10.1016/j.entcs.2008.04.040}. – DOI

http://dx.doi.org/10.1016/j.entcs.2008.04.040. – ISSN 1571–0661. – Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)

[ker16] *Kermeta 3 (K3).* http://diverse-project.github.io/k3/. Version: 2016. – Visited 2016

[Kha15] Khan, Nida: *Implementation of an Extension of a Visualisation Tool for the Flowchart-based Visual Satellite Control Language SPELL-Flow.* Luxembourg, Université du Luxembourg, Master's Thesis, 2015

[Kir14] Kirpach, Gérard: *Concept and Implementation of Formal Integration and Synchronization Techniques for Automated Language Translation in Eclipse Henshin.* Luxembourg, Université du Luxembourg, Master's Thesis, 2014

[KK96] Kreowski, Hans-Jörg ; Kuske, Sabine: On the interleaving semantics of transformation units — A step into GRACE. In: Cuny, Janice (Hrsg.) ; Ehrig, Hartmut (Hrsg.) ; Engels, Gregor (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Graph Grammars and Their Application to Computer Science: 5th International Workshop Williamsburg, VA, USA, November 13–18, 1994 Selected Papers.* Berlin, Heidelberg : Springer Berlin Heidelberg, 1996. – ISBN 978–3–540–68388–9, 89–106

[KKR08] Kreowski, Hans-Jörg ; Kuske, Sabine ; Rozenberg, Grzegorz ; Degano, Pierpaolo (Hrsg.) ; De Nicola, Rocco (Hrsg.) ; Meseguer, José (Hrsg.): *Graph Transformation Units – An Overview.* http://dx.doi.org/10.1007/978-3-540-68679-8_5. Version: 2008

[KKS97] Kreowski, Hans-Jörg ; Kuske, Sabine ; Schürr, Andy: Nested Graph Transformation Units. In: *International Journal of Software Engineering and Knowledge Engineering* 07 (1997), Nr. 04, 479-502. http://dx.doi.org/10.1142/S0218194097000278. – DOI 10.1142/S0218194097000278

[KS06] Königs, Alexander ; Schürr, Andy: Tool Integration with Triple Graph Grammars - A Survey. In: *Electronic Notes in Theoretical Computer Science* 148 (2006), Nr. 1, 113 - 150. http://dx.doi.org/http://dx.doi.org/10.1016/j.entcs.2005.12.015. – DOI http://dx.doi.org/10.1016/j.entcs.2005.12.015. – ISSN 1571–0661

[Kus00] Kuske, Sabine: *Transformation Units - A Structuring Principle for Graph Transformation Systems*, University of Bremen, Diss., 2000

[KW07] Kindler, Ekkart ; Wagner, Robert: Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios / Universität Paderborn. Version: 2007. http://www.wagner-gt.de/fileadmin/user_upload/publications/KW07.pdf. 2007 (tr-ri-07-284). – Forschungsbericht. – 75 S.

[Lac05] Lack, Sobociski P. Stephen: Adhesive and quasiadhesive categories. In: *RAIRO - Theoretical Informatics and Applications - Informatique Thorique et Applications* 39 (2005), Nr. 3, 511-545. http://eudml.org/doc/245558

[LAS14] In: Leblebici, Erhan ; Anjorin, Anthony ; Schürr, Andy: *Developing eMoflon with eMoflon*. Cham : Springer International Publishing, 2014. – ISBN 978–3–319–08789–4, 138–145

[LAST15] Leblebici, Erhan ; Anjorin, Anthony ; Schürr, Andy ; Taentzer, Gabriele ; Parisi-Presicce, Francesco (Hrsg.) ; Westfechtel, Bernhard (Hrsg.): *Multi-amalgamated Triple Graph Grammars*. http://dx.doi.org/10.1007/978-3-319-21145-9_6. Version: 2015

[LBE+07] Lara, Juan de ; Bardohl, Roswitha ; Ehrig, Hartmut ; Ehrig, Karsten ; Prange, Ulrike ; Taentzer, Gabriele: Attributed graph transformation with node type inheritance. In: *Theoretical Computer Science, Fundamental Aspects of Software Engineering* 376 (2007), Nr. 3, 139 - 163. http://dx.doi.org/http://dx.doi.org/10.1016/j.tcs.2007.02.001. – DOI http://dx.doi.org/10.1016/j.tcs.2007.02.001. – ISSN 0304–3975

[Lc16] Ltd., IncQuery L. ; community, Eclipse: *Viatra*. http://www.eclipse.org/viatra/. Version: 2016. – Visited 2016

[Mah09] Mahr, Bernd: Information science and the logic of models, 2009. – ISSN 1619–1374, 365–383

[Mat11] Matinnejad, Reza: Agile Model Driven Development: An Intelligent Compromise. In: *Proceedings of the 2011 Ninth International Conference on Software Engineering Research, Management and Applications*. Washington, DC, USA : IEEE Computer Society, 2011 (SERA '11). – ISBN 978–0–7695–4490–8, 197–202

[MDG+04] Moore, Bill ; Dean, David ; Gerber, Anna ; Wagenknecht, Gunnar ; Vanderheyden, Philippe: *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks, 2004 http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf

[Met16] *MetaEdit+*. http://www.metacase.com/de/products.html. Version: 2016. – Visited 2016

[MOF16] *MOF*. http://www.omg.org/mof/. Version: 2016

[NNZ00] Nickel, Ulrich A. ; Niere, Jörg ; Zündorf, Albert: Tool demonstration: The FUJABA environment. In: *Proc. of the 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland*, ACM Press, 2000, S. 742–745

[OMG16] *OMG - Object Management Group*. http://www.omg.org/. Version: 2016

[Opt16] *OptimalJ*. http://www.compuware.com/. Version: 2016. – Visited 2016

[Pad16] Paderborn, University of: *Fujaba: From UML to Java and back again*. http://www.fujaba.de. Version: 2016. – Visited 2016

[PIL12] *Space travel with a new language in tow*. http://wwwen.uni.lu/snt/news_events/space_travel_with_a_new_language_in_tow. Version: 2012

[PIL14] *Sprachenchaos im All: Übersetzer für Fernsehsatelliten gebraucht*. http://science.lu/de/content/sprachenchaos-im-all-%C3%BCbersetzer-f%C3%BCr-fernsehsatelliten-gebraucht. Version: 2014

[PR69] Pfaltz, John L. ; Rosenfeld, Azriel: Web Grammars. In: *Proceedings of the 1st International Joint Conference on Artificial Intelligence*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1969 (IJCAI'69), 609–619

[qvt16a] *QVT-Operational*. https://projects.eclipse.org/projects/modeling.mmt.qvt-oml. Version: 2016. – Visited 2016

[QVT16b] *QVT-Relations*. http://projects.eclipse.org/projects/modeling.mmt.qvtd. Version: 2016. – Visited 2016

[Roz97] Rozenberg, Grzegorz (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. River Edge, NJ, USA : World Scientific Publishing Co., Inc., 1997 http://www.worldscientific.com/worldscibooks/10.1142/3303. – ISBN 98–102288–48

[RS97] Rozenberg, Grzegorz ; Salomaa, Arto: *Handbook of Formal Languages*. 1st. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 1997. – ISBN 3540614869

[Sch95] Schürr, Andy ; Mayr, Ernst W. (Hrsg.) ; Schmidt, Gunther (Hrsg.) ; Tinhofer, Gottfried (Hrsg.): *Specification of graph translators with triple graph grammars*. http://dx.doi.org/10.1007/3-540-59071-4_45. Version: 1995

[SE77] Schneider, H. J. ; Erlangen ; Karpiński, Marek (Hrsg.): *Graph grammars*. http://dx.doi.org/10.1007/3-540-08442-8_99. Version: 1977

[Sel12] Selic, Bran: The Less Well Known UML: A Short User Guide. In: *Proceedings of the 12th International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-driven Engineering*. Berlin, Heidelberg : Springer-Verlag, 2012 (SFM'12). – ISBN 978–3–642–30981–6, 1–20

[SEM+12] Schölzel, Hanna ; Ehrig, Hartmut ; Maximova, Maria ; Gabriel, Karsten ; Hermann, Frank: Satisfaction, Restriction and Amalgamation of Constraints in the Framework of $\mathcal{M}$-Adhesive Categories. In: *Proceedings Seventh ACCAT Workshop on Applied and Computational Category Theory, ACCAT 2012, Tallinn, Estonia, 1 April 2012.*, 2012, 83–104

[SES16] *SES*. http://www.ses.com. Version: 2016

[SK08] Schürr, Andy ; Klar, Felix: 15 Years of Triple Graph Grammars. In: *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*, 2008, 411–425

[SPE15] *SPELL - Satellite Procedure Execution Language and Library*. https://sourceforge.net/projects/spell-sat/. Version: 2015

[SVC06] Stahl, Thomas ; Voelter, Markus ; Czarnecki, Krzysztof (.: *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. – ISBN 0–470–02570–0

[TA15] Trollmann, Frank ; Albayrak, Sahin: Extending Model to Model Transformation Results from Triple Graph Grammars to Multiple Models. Version: 2015. http://dx.doi.org/{10.1007/978-3-319-21155-8_16}. In: Kolovos, Dimitris (Hrsg.) ; Wimmer, Manuel (Hrsg.): *Theory and Practice of Model Transformations: 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings* Bd. 9152. Springer International Publishing, 2015. – DOI 10.1007/978–3–319–21155–8$_1$6. – –$ISBN$978 – –3 – –319 – –21155 – –8, 214 – –229

[TA16] Trollmann, Frank ; Albayrak, Sahin: Extending Model Synchronization Results from Triple Graph Grammars to Multiple Models. In: *Theory and Practice of Model Transformations: 9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings*, 2016, 91–106

[Tae04] In: Taentzer, Gabriele: *AGG: A Graph Transformation Environment for Modeling and Validation of Software*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004. – ISBN 978–3–540–25959–6, 446–453

[tef16] *Tefkat*. http://tefkat.sourceforge.net/. Version: 2016. – Visited 2016

[TR05] Taentzer, Gabriele ; Rensink, Arend: Ensuring Structural Constraints in Graph-Based Models with Type Inheritance. In: Cerioli, Maura (Hrsg.): *Fundamental Approaches to Software Engineering: 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2005. – ISBN 978–3–540–31984–9, 64–79

[UML16] *UML - Unified Modeling Language*. http://www.uml.org/. Version: 2016

[VBH+16] Varró, Dániel ; Bergmann, Gábor ; Hegedüs, Ábel ; Horváth, Ákos ; Ráth, István ; Ujhelyi, Zoltán: Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. In: *Software & Systems Modeling* 15 (2016), Nr. 3, 609–629. http://dx.doi.org/10.1007/s10270-016-0530-4. – DOI 10.1007/s10270–016–0530–4. – ISSN 1619–1374

[vis16] *Visual Paradigm for UML (VP-UML).* https://www.visual-paradigm.com/. Version: 2016. – Visited 2016

[Wag95] Wagner, Annika: On the Expressive Power of Algebraic Graph Grammars with Application Conditions. In: *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*. London, UK, UK : Springer-Verlag, 1995 (TAPSOFT '95). – ISBN 3–540–59293–8, 409–423

[xte16] *Xtext.* https://eclipse.org/Xtext/. Version: 2016

Appendix A

A.1. SPELL Xtext grammar

```
grammar lu.uni.snt.spell.SPELL

hidden(WS)
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate sPELL "http://www.uni.lu/snt/spell/SPELL"

/* according to Python v3.2 documentation */

file_input          : {file_input} nl=NEWLINE? (fst=stmt_LST_Elem)?;
stmt_LST_Elem       : entry=stmt (next=stmt_LST_Elem)?;
decorator           : '@' dotted_name=dotted_name ( '(' (arglist=arglist) ')' )? nl=NEWLINE;
decorator_LST_Elem      : entry=decorator (next=decorator_LST_Elem)?;
decorated           : decorators=decorator_LST_Elem (classdef=classdef | funcdef=funcdef);
funcdef             : 'def' def_name=NAME parameters=parameters ('->' test=test)? ':' suite=suite
                      '#ENDDEF';
parameters          : {parameters} '(' (typedargslist=typedargslist)? ')';
typedargslist       : {typedargslist}
                      ( fst=typedarg_LST_Elem
                        (',' ('*' (tfpdef_one_star=tfpdef)? (',' one_star_list=typedarg_LST_Elem)?
                        (',' '**' tfpdef_two_stars=tfpdef)? | '**' tfpdef_two_stars=tfpdef)? )?
                        | '*' (tfpdef_one_star=tfpdef)? (',' one_star_list=typedarg_LST_Elem)?
                          (',' '**' tfpdef_two_stars=tfpdef)?
```

```
23                  | '**' tfpdef_two_stars=tfpdef );
24 typedarg_LST_Elem    : entry=typedarg  (',' next=typedarg_
       LST_Elem)?;
25 typedarg          : tfpdef=tfpdef ('=' asg_test=test)?;
26 tfpdef            : tfpdef_name=NAME (':' test=test)?;
27 varargslist        : {varargslist} ( fst=vararg_LST_Elem (',
       ' ('*' (vfpdef_one_star=vfpdef)? (',' one_star_list=
       vararg_LST_Elem)?
28                                (',' '**' vfpdef_two_stars=
                                      vfpdef)? | '**' vfpdef_two_
                                      stars=vfpdef)? )?
29                          | '*' (vfpdef_one_star=vfpdef)? (','
                               one_star_list=vararg_LST_Elem)?
30                               (',' '**' vfpdef_two_stars=vfpdef
                                    )?
31                            | '**' vfpdef_two_stars=vfpdef );
32 vararg_LST_Elem       : entry=vararg (',' next=vararg_LST_
       Elem)?;
33 vararg             : vfpdef=vfpdef ('=' test=test)?;
34 vfpdef             : NAME;
35 stmt             : (simple_stmt | compound_stmt) (nl_post=
       NEWLINE);
36 simple_stmt         : small_stmt ( {simple_stmt.fst=current}
       next=small_stmt_LST_Elem )?
37                  (';')? ;
38 small_stmt_LST_Elem    : ';' entry=small_stmt next=small_
       stmt_LST_Elem?;
39 small_stmt          : (expr_stmt | del_stmt | pass_stmt |
       flow_stmt | import_stmt | global_stmt | nonlocal_stmt |
       assert_stmt );
40 expr_stmt         : testlist_star_expr  ( {expr_stmt.fst=
       current} (  augassign_symbol=augassign (snd=yield_expr|
       snd=testlist)
41                                                  | (assignments_
                                                        fst=
                                                        assignment_
                                                        Exp_LST_Elem
                                                        ) ) )?;
42 assignment_Exp_LST_Elem : entry=assignment_Exp (next=
       assignment_Exp_LST_Elem)?;
43 assignment_Exp       : symbol='=' (yield_expression=yield_
       expr |testlist_star_expression=testlist_star_expr );
44 testlist_star_expr     : test_Or_star_exp ( {testlist_star_
       expr.fst=current} ',' next=test_Or_star_exp_LST_Elem )?
       //(',')?
45                  ;
46 test_Or_star_exp_LST_Elem:  (nl=NEWLINE)? entry=test_Or_
       star_exp (','  next=test_Or_star_exp_LST_Elem)?
47 ;
```

```
48 test_Or_star_exp     : (test |star_expr );
49 augassign            : ('+=' | '-=' | '*=' | '/=' | '%=' | '&='
       | '|=' | '^=' | '<<=' | '>>=' | '**=' | '//=');
50 del_stmt             : 'del' exprlist=exprlist
51   ;
52 pass_stmt            : {pass_stmt} 'pass';
53 flow_stmt            : break_stmt | continue_stmt | return_stmt
       | raise_stmt | yield_stmt;
54 break_stmt           : {break_stmt} 'break';
55 continue_stmt        : {continue_stmt} 'continue';
56 return_stmt          : {return_stmt} 'return'   (testlist=
       testlist)?;
57 yield_stmt           : yield_expr=yield_expr;
58 raise_stmt           : {raise_stmt} 'raise' (raise_test=test (
       'from' from_test=test)?)?;
59 import_stmt          : import_name | import_from;
60 import_name          : 'import' dotted_as_names=dotted_as_
       names;
61 import_from          : ('from' ((dots+=dot_or_dots)* name_from
       =dotted_name | (dots+=dot_or_dots)+)
62                              'import' (import_star='*' | ('('
                                    import_as_names=import_as_names
                                    ')') | import_as_names=import_as
                                    _names));
63 dot_or_dots          : '.' | '...';
64 import_as_name       : name_import=NAME ('as' name_as=NAME)
       ?;
65 dotted_as_name       : dotted_name=dotted_name ('as' name_as
       =NAME)?;
66 import_as_names      : fst=import_as_name_LST_Elem (',')?;
67 import_as_name_LST_Elem : entry=import_as_name (',' next=
       import_as_name_LST_Elem)?;
68 dotted_as_names      : fst=dotted_as_name_LST_Elem;
69 dotted_as_name_LST_Elem : entry=dotted_as_name (',' next=
       dotted_as_name_LST_Elem)?;
70 dotted_name          : name_fst=dotted_name_LST_Elem;
71 dotted_name_LST_Elem   : entry=NAME ('.' next=dotted_name_
       LST_Elem)?;
72 global_stmt          : 'global'  fst=name_LST_Elem;
73 name_LST_Elem        : entry=NAME (',' next=name_LST_Elem)?;
74 nonlocal_stmt        : 'nonlocal' fst=name_LST_Elem;
75 assert_stmt          : 'assert' test_fst=test (',' test_snd=
       test)?;
76 compound_stmt          : (if_stmt | while_stmt | for_stmt |
       try_stmt | with_stmt | funcdef | classdef | decorated) ;
77 if_stmt             : 'if' if_test=test ':' if_suite=suite (
       else_if_exp_List=else_if_exp_LST_Elem)?
78                  ('else' ':' last_else_suite=suite)?
79                  '#ENDIF';
```

```
80 else_if_exp_LST_Elem  : entry=else_if_exp (next=else_if_exp
       _LST_Elem)? ;
81 else_if_exp       : 'elif' else_test=test ':' else_suite=
       suite;
82 while_stmt        : 'while' test=test ':' while_suite=suite
        ('else' ':' else_suite=suite)?
83                   '#ENDWHILE';
84 for_stmt          : 'for' for_exprlist=exprlist 'in' in_
       testlist=testlist ':' in_suite=suite ('else' ':' else_
       suite=suite)?
85                   '#ENDFOR';
86 try_stmt          : ('try' ':' try_suite=suite
87                        ( except_list_fst=except_exp_LST_Elem
88                        ( 'else' ':' else_suite=suite)?
89                        ( 'finally' ':' finally_suite=suite)?
90                        | 'finally' ':' finally_suite=suite)
91                          )
                      '#ENDTRY';
92 except_exp_LST_Elem   : entry=except_exp (next=except_exp_
       LST_Elem)?;
93 except_exp          : except_clause=except_clause ':' suite=
       suite;
94 with_stmt          : 'with' fst=with_item_LST_Elem   ':' with_
       suite=suite
95                   '#ENDWITH';
96 with_item_LST_Elem    : entry=with_item (',' next=with_item
       _LST_Elem)?;
97 with_item          : test=test ('as' as_expression=expr)?;
98 except_clause      : {except_clause} 'except' (except_test=
       test ('as' as_name=NAME)?)?;
99 suite             : simple_stmt | (nl=NEWLINE fst=stmt_LST_Elem
       ) ;
100 test             : (or_test ({test.fst=current} 'if' if_test=
       or_test 'else' else_test=test)?)
101                    | lambdef=lambdef;
102 test_nocond       : or_test | lambdef_nocond;
103 lambdef           : 'lambda' ( varargslist=varargslist)? ':'
       test=test;
104 lambdef_nocond      : 'lambda' ( varargslist=varargslist)?
       ':' test_nocond=test_nocond;
105 or_test           : and_test ({or_test.fst=current}  'or'
       next=or_operand_LST_Elem )?;
106 or_operand_LST_Elem   : entry=and_test ('or' next=or_
       operand_LST_Elem)?;
107 and_test          : not_test ({and_test.fst=current} 'and'
       next=and_operand_LST_Elem )?;
108 and_operand_LST_Elem  : entry=not_test ('and' next=and_
       operand_LST_Elem)?;
```

```
109 not_test          : ((negations=negation_LST_Elem) entry=
        comparison) | comparison;
110 negation_LST_Elem    : {negation_LST_Elem} 'not' (next=
        negation_LST_Elem)?;
111 comparison          : expr ( {comparison.fst=current} next=
        comparison_Cond_LST_Elem)?;
112 comparison_Cond_LST_Elem: comp_op=comp_op snd=expr (next=
        comparison_Cond_LST_Elem)?;
113 comp_op            : op=('<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'
        |'is') | notIn?=('not') 'in' | 'is' isNot?=('not');
114 star_expr          : '*' expression=expr;
115 expr              : xor_expr ({expr.fst=current} '|' next_xor=
        xor_expr_LST_Elem )?;
116 xor_expr_LST_Elem    : entry=xor_expr ('|' next=xor_expr_LST
        _Elem)?;
117 xor_expr           : and_expr ({xor_expr.fst=current} '^' next
        _and=and_expr_LST_Elem )?;
118 and_expr_LST_Elem    : entry=and_expr ('^' next=and_expr_LST
        _Elem)?;
119 and_expr           : shift_expr ({and_expr.fst=current} '&'
        next_shift=shift_expr_LST_Elem )?;
120 shift_expr_LST_Elem    : entry=shift_expr ('&' next=shift_
        expr_LST_Elem)?;
121 shift_expr          : arith_expr ({shift_expr.fst=current}
        next_arith_expr=arith_expr_LST_Elem)?;
122 arith_expr_LST_Elem    : shift_Symbol=('<<'|'>>') operand=
        arith_expr (next=arith_expr_LST_Elem)?;
123 arith_expr          : term ({arith_expr.fst=current} arith_
        exp_Symbol=('+'|'-') (nl=NEWLINE)? next_term=term_LST_
        Elem)?;
124 term_LST_Elem       : operand=term ( arith_exp_Symbol=('+'|'-
        ') (nl=NEWLINE)? next=term_LST_Elem)?;
125 term              : factor ({term.fst=current} next_factor=
        factor_LST_Elem )?;
126 factor_LST_Elem       : term_Symbol=('*'|'/'|'%'|'//')
        operand=factor (next=factor_LST_Elem)?;
127 factor             : (factor_symbols=factor_symbol_LST_Elem
        base=powerOrSPELL) | powerOrSPELL;
128 factor_symbol_LST_Elem  : entry=('+'|'-'|'~') (next=factor_
        symbol_LST_Elem)? ;
129 powerOrSPELL        : power;
130 power             : atomOrSPELL ({power.value=current}
131                       ( (trailer_fst=trailer_LST_Elem) ('**'
                              factor=factor)? )
132                       | ('**' factor=factor)     )?;
133 atomOrSPELL         : atom|SPELL_Exp;
134 SPELL_Exp         : SPELL_Constant | SPELL_Modifier | SPELL_
        Function_Name | SPELL_Fun;
```

```
135 SPELL_Fun        : Abort | BuildTC | CreateDictionary |
        ChangeLanguageConfig | DisableAlarm
136                  | DisableUserAction | DismissUserAction |
                       Display | DisplayStep | EnableAlarm
137                  | EnableUserAction | Event | Finish |
                       GetResource | GetTM | GetLimits
138                  | IsAlarmed | OpenDisplay | Pause |
                       PrintDisplay
139                  | Prompt | Var | SaveDictionary | Send |
                       SetGroundParameter | SetUserAction
140                  | SetResource | SetLimits | StartProc |
                       Verify | WaitFor | WaitForComplex
141                  | Goto | Step | Paragraph;
142 SPELL_Function_Name    : nameF=('LoadDictionary');
143 Prompt           : 'Prompt'          '(' args=arglist ')';
144 GetTM            : 'GetTM'           '(' args=arglist ')';
145 Verify           : 'Verify' '(' '[' verifyL=verifyEntry_LST_
        Elem ']' (',' (nl=NEWLINE)? args=arglist)? ')';
146 verifyEntry_LST_Elem     :    entry=atom (',' nl=NEWLINE
147                               next=verifyEntry_LST_Elem)?;
148 Var              : 'Var' '(' 'Type' '=' varType=SPELL_Constant
149                      (',' 'Range' '=' varRange=test)?
150                      (',' 'Default' '=' varDefault=test)?
151                      (',' 'Confirm' '=' varConfirm=test)?
152                      (',' 'Expected' '=' varExpected=test)? ')
                           ';
153 Display          : 'Display'         '(' args=arglist ')';
154 DisplayStep      : 'DisplayStep'     '(' args=arglist ')';
155 BuildTC          : 'BuildTC'         '(' args=arglist ')';
156 Send             : 'Send'            '(' 'command' '=' command
        =atom (',' (nl=NEWLINE)? args=arglist)? ')';
157 WaitFor          : 'WaitFor'         '(' args=arglist ')';
158 WaitForComplex       : 'WaitFor_TimeOut' '(' 'condition' '='
        '"' condition=test '"' ( ')' | (',' nl=NEWLINE
159                               'ForDelay' '=' delay=test ')')
                                    );
160 SetGroundParameter     : 'SetGroundParameter'  '(' args=
        arglist ')';
161 GetLimits        : 'GetLimits'       '(' args=arglist ')';
162 SetLimits        : 'SetLimits'       '(' args=arglist ')';
163 EnableAlarm       : 'EnableAlarm'     '(' args=arglist ')';
164 DisableAlarm      : 'DisableAlarm'      '(' args=arglist ')
        ';
165 IsAlarmed        : 'IsAlarmed'       '(' args=arglist ')';
166 Event            : 'Event'          '(' args=arglist ')';
167 SetResource      : 'SetResource'     '(' args=arglist ')';
168 GetResource      : 'GetResource'     '(' args=arglist ')';
169 OpenDisplay      : 'OpenDisplay'     '(' args=arglist ')';
```

```
170 PrintDisplay      : 'PrintDisplay'      '(' args=arglist ')'
    ';
171 Pause             : 'Pause'             '(' args=arglist ')';
172 Abort             : 'Abort'             '(' args=arglist ')';
173 Finish            : 'Finish'            '(' args=arglist ')';
174 SetUserAction     : 'SetUserAction'     '(' args=arglist ')'
    ';
175 EnableUserAction    : 'EnableUserAction'    '(' args=
    arglist ')';
176 DisableUserAction   : 'DisableUserAction'   '(' args=
    arglist ')';
177 DismissUserAction   : 'DismissUserAction'   '(' args=
    arglist ')';
178 CreateDictionary    : 'CreateDictionary'    '(' args=
    arglist ')';
179 SaveDictionary      : 'SaveDictionary'    '(' args=arglist
    ')';
180 StartProc       : 'StartProc'       '(' args=arglist ')';
181 ChangeLanguageConfig  : 'ChangeLanguageConfig'  '(' args=
    arglist ')';
182 Goto            : 'Goto'            '(' args=arglist ')';
183 Step            : 'Step'            '(' args=arglist ')';
184 Paragraph       : 'Paragraph'         '(' args=arglist ')';
185
186 SPELL_Constant       : nameC=('NOW' | 'TODAY' | 'TOMORROW' |
     'YESTERDAY' | 'HOUR' | 'MINUTE' | 'SECOND' | 'DATE'
187                | 'DATETIME' | 'RELTIME' | 'ABSTIME'
188                | 'RAW' | 'ENG' | 'SKIP' | 'LONG' | 'STRING'
                     | 'BOOLEAN'
189                | 'TIME' | 'FLOAT' | 'DEC' | 'HEX' | 'OCT' |
                     'BIN' | 'VALUE' | 'ALL'
190                | 'ACTIVE' | 'LIST' | 'NUM' | 'OK' | 'ALPHA'
                     | 'OK_CANCEL'
191                | 'CANCEL' | 'YES' | 'NO' | 'YES_NO' | '
                     INFORMATION' | 'WARNING'
192                | 'ERROR' | 'NOACTION' | 'ABORT' | 'REPEAT' |
                     'RESEND' | 'RECHECK'
193                | 'FIXME_unknownConstant');
194
195 SPELL_Modifier       : nameM=( 'AdjLimits' | 'Automatic' | '
    Block' | 'Blocking' | 'Confirm' | 'Default'
196                | 'Delay' | 'HandleError' | 'HiBoth' | 'HiRed
                     '
197                | 'HiYel' | 'Host' | 'IgnoreCase' | 'Interval
                     ' | 'LoadOnly' | 'LoBoth'
198                | 'LoRed' | 'LoYel' | 'Message' | 'Midpoint'
                     | 'Notify' | 'OnFailure'
199                | 'OnTrue' | 'OnFalse' | 'Printer' | '
                     PromptUser' | 'Radix' | 'Retries'
```

```
200                 | 'SendDelay' | 'Severity' | 'Time' | '
                      Timeout' | 'Tolerance'
201                 | 'Type' | 'Units' | 'Until' | 'ValueFormat'
                      | 'ValueType' | 'Visible'
202                 | 'Wait'
203                 | 'Extended' | 'args' | 'command' | '
                      ReleaseTime'
204                 | 'ConfirmCritical' | 'sequence' | 'group' |
                      'Group' | 'addInfo' | 'verify'
205                 | 'Select' | 'Nominal' | 'Warning' | 'Error'
                      | 'Ignore'
206                 | 'Delta' | 'Format' | 'FIXME_
                      ModifierUnknown'
207                 );
208
209 yield_expr_OR_testlist_comp : yield_expr | testlist_comp;
210
211 atom            : (round_brackets?='(' (entry = yield_expr_OR
    _testlist_comp)? ')'
212                 | square_brackets?='[' (testlist_comp=
                      testlist_comp)? ']'
213                 | curly_brackets?='{' (dictorsetmaker=
                      dictorsetmaker)? '}'
214                 | atom_name=NAME | number=NUMBER
215                 | string_LST_Elem | basic='...' | basic='None
                      ' | basic='True' | basic='False' );
216 string_LST_Elem      : entryS=STRING  (NL nextS=string_LST_
    Elem)? ;
217 testlist_comp      : test_Or_star_exp=test_Or_star_exp (
218                   ( comp_for=comp_for | (',' next_exp =test_
                        Or_star_exp_LST_Elem  (',')? )  )  )?;
219 trailer_LST_Elem     : trailer ({trailer_LST_Elem.entry=
    current}  next=trailer_LST_Elem)?;
220 trailer          : ( '(' arglist ')' ) | ( '[' subscript_LST
    _Elem ']' ) | ( '.' Name_Node );
221 Name_Node        : nameN=NAME;
222 subscript_LST_Elem     : entry=subscript (',' (nl=NEWLINE)?
    next=subscript_LST_Elem)?;
223 subscript        : {subscript} ( ( test_fst=test (colon=':'
     (test_snd=test)? (sliceop=sliceop)? )? )
224                       | ( colon=':' (test_snd=test)? (
                            sliceop=sliceop)?  ) );
225 sliceop          : {sliceop} ':' (test=test)?;
226 exprlist         : exp_Or_star_exp ( {exprlist.fstexp=
    current} ',' next_expr=exp_Or_star_exp_LST_Elem  )?;
227 exp_Or_star_exp_LST_Elem: entry=exp_Or_star_exp (',' next=
    exp_Or_star_exp_LST_Elem )?;
228 exp_Or_star_exp      : (expr|star_expr);
```

```
testlist          : test ({testlist.fst_test=current} ','
    next_test=test_LST_Elem )?;
test_LST_Elem     : entry=test (',' next=test_LST_Elem)?;
dictorsetmaker      : fst=test ( ( ':' test_to=test ( (comp
    _for=comp_for | (',' (nl=NEWLINE)? next_range=test_
    range_LST_Elem))  )? )  // removed (',')? from end
                  | (comp_for=comp_for | (',' next_test=
                      test_LST_Elem) ) );
test_range        : test_from=test ':' test_to=test;
test_range_LST_Elem   : entry=test_range (',' (nl=NEWLINE)?
     next=test_range_LST_Elem)?;
classdef          : 'class' name_class=NAME ('(' (arglist=
    arglist) ')')? ':' suite=suite  '#ENDCLASS';
arglist           : (argument_list ( {arglist.argument_list=
    current}
               ',' star_test_arg_list=star_test_arg_list  )?
                 | star_test_arg_list=star_test_arg_list  );
star_test_arg_list    : '*' test_one_star=test (',' one_
    star_arguments_fst=argument_LST_Elem)?
                          (',' '**' test_two_stars=test)? |
                              '**' test_two_stars=test;
argument_list     : {argument_list} (fst=argument_LST_Elem)
    ? ;
argument_LST_Elem   : entry=argument (',' (nl=NEWLINE)?
    next=argument_LST_Elem)?;
argument          : test ( {argument.test=current}  ( arg_
    comp_for=comp_for | assigment=Value_Assignment)  )?;
Value_Assignment    : symbol= '=' value=test;
comp_iter         : comp_for | comp_if;
comp_for          : 'for' for=exprlist 'in' in=or_test (comp_
    iter=comp_iter)?;
comp_if           : 'if' if=test_nocond (comp_iter=comp_iter)
    ;
yield_expr          : {yield_expr} 'yield' (testlist=testlist
    )? ;
NUMBER            : integer | real |octal|hex;
octal           : val=_SCAL_OCTAL_;
hex             : val=_SCAL_HEX_;
integer           : value=_SCAL_INT_;
real            : (value=decimal (exponent=Exponent)?)
                | value=integer exponent=Exponent;
decimal           : value=_SCAL_REAL_;
Exponent          : ('E') (sign=('+'|'-'))? value=_SCAL_INT_
    ;
NAME            : ID;
Comment_LST_Elem    :   valC=(SL_COMMENT | ML_COMMENT) (NL+
     nextC=Comment_LST_Elem)?;
NEWLINE           :   {NEWLINE}  (NL+ (comment=Comment_LST_
    Elem NL+)?)
```

```
260                                          | (comment=Comment_LST_
                                               Elem NL+);
261 terminal ID          : ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|
        '_'|'#'|'0'..'9')*;
262 terminal _SCAL_INT_   : ('0'..'9')+;
263 terminal _SCAL_REAL_  : (_SCAL_INT_)? '.' _SCAL_INT_;
264 terminal _SCAL_HEX_   : '0' ('x'|'X') ('0'..'9'|'A'..'F'|'a
        '..'f')+;
265 terminal _SCAL_OCTAL_ : '0' 'o' ('0'..'9')+;
266 terminal STRING       :   "'" ( '\"' ('b'|'t'|'n'|'f'|'r'|'"'
        |"'"|'\"') | !('\"'|"'") )* "'"
267                     | '"' ( '\"' ('b'|'t'|'n'|'f'|'r'|'"'|"'"|'\"
                          ') | !('\"'|'"') )* '"'
268                     ;
269 terminal SL_COMMENT     : ('#' // every comment not
        starting with "#END" and not containing a separator "<<"
270                       ( ( (!('E'|'\n'|'\r'| '<<') | 'E'!('N'|'\n
                            '|'\r'| '<<') | 'EN'!('D'|'\n'|'\r'| '<
                            <')
271                            ) !('\n'|'\r' | '<<')*
272                            )
273                         | 'E'
274                       )? ); // ('\r'? '\n')?; NL is used for
                             parsing - no white space
275 terminal ML_COMMENT   : "'''" ->  "'''";
276 //// Only for Parser
277 terminal WS             :  ' '|'\t'|'\\\r'|'\\\n'|'\\\r\n'|'
        \\\n\r';
278 terminal ANY_OTHER     : .;
279 terminal NL            :   '\n'|'\r'|'\r\n'|'\n\r';
280 // For TGG: super type for target language
281 // rules that inherit from Target already transitively do
        not occur here
282 SourceSPELL: file_input
283 |stmt_LST_Elem
284 |decorator|decorator_LST_Elem
285 |parameters|typedargslist|typedarg_LST_Elem|typedarg|tfpdef
        |varargslist|vararg_LST_Elem
286 |vararg|stmt
287 |small_stmt_LST_Elem
288 |assignment_Exp_LST_Elem|assignment_Exp
289 |test_Or_star_exp_LST_Elem
290 |import_as_name
291 |dotted_as_name|import_as_names|import_as_name_LST_Elem|
        dotted_as_names|dotted_as_name_LST_Elem
292 |dotted_name|dotted_name_LST_Elem
293 |name_LST_Elem
294 |else_if_exp_LST_Elem|else_if_exp
295 |except_exp_LST_Elem|except_exp
```

```
296 | with_item_LST_Elem | with_item | except_clause
297 | suite
298 | test_nocond
299 | lambdef
300 | or_operand_LST_Elem
301 | and_operand_LST_Elem
302 | negation_LST_Elem
303 | comparison_Cond_LST_Elem | comp_op
304 | xor_expr_LST_Elem
305 | and_expr_LST_Elem
306 | shift_expr_LST_Elem
307 | arith_expr_LST_Elem
308 | term_LST_Elem
309 | factor_LST_Elem
310 | factor_symbol_LST_Elem
311 |  verifyEntry_LST_Elem
312 | yield_expr_OR_testlist_comp
313 | trailer_LST_Elem | trailer
314 | subscript | sliceop | exprlist
315 | exp_Or_star_exp_LST_Elem | testlist | test_LST_Elem |
      dictorsetmaker
316 | test_range | test_range_LST_Elem
317 | star_test_arg_list
318 | argument_LST_Elem
319 | argument | Value_Assignment | comp_iter
320 |NUMBER;
```

Listing A.1: SPELL Xtext grammar

A.2. SPELL meta-model

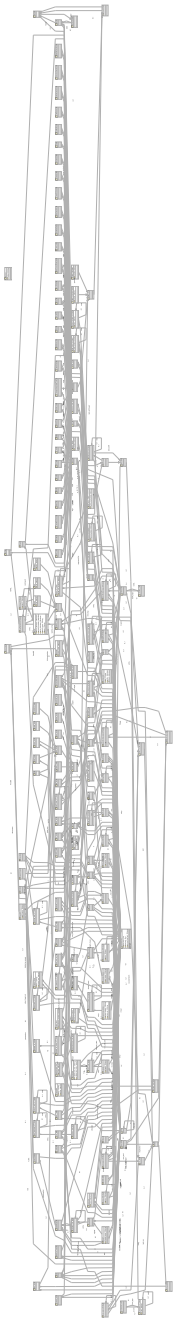


Figure A.1: SPELL meta-model, screenshot of Eclipse ecore diagram file

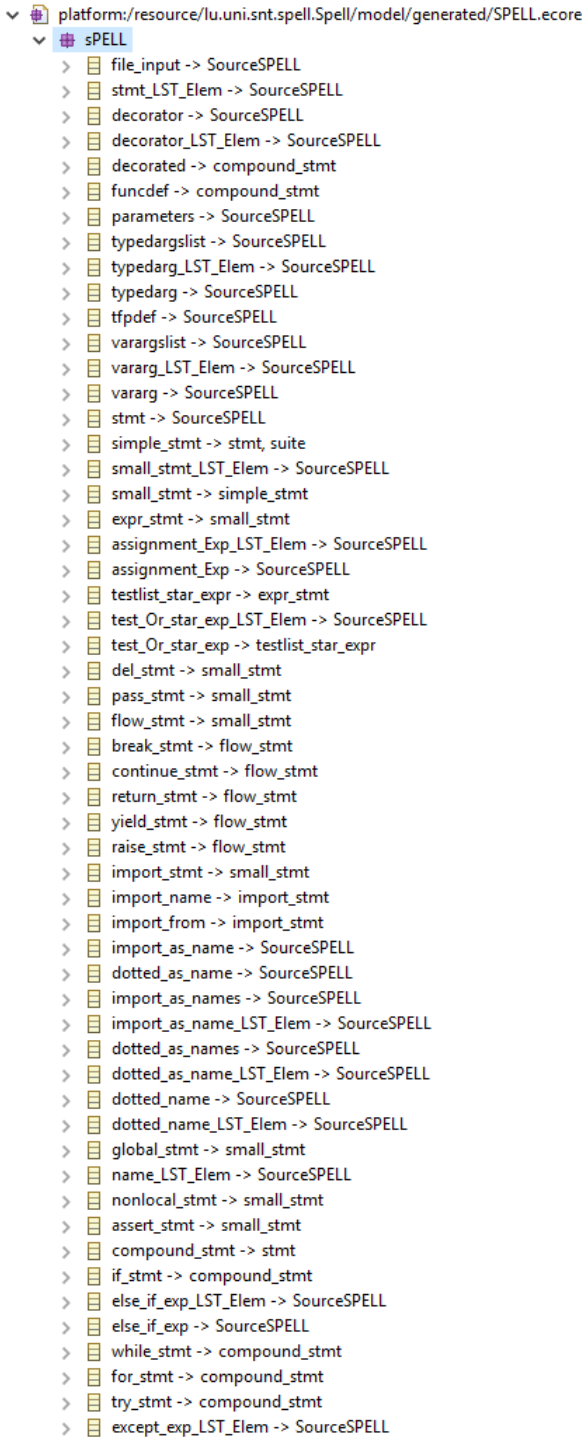


Figure A.2: SPELL meta-model, screenshot of Eclipse ecore file, part 1

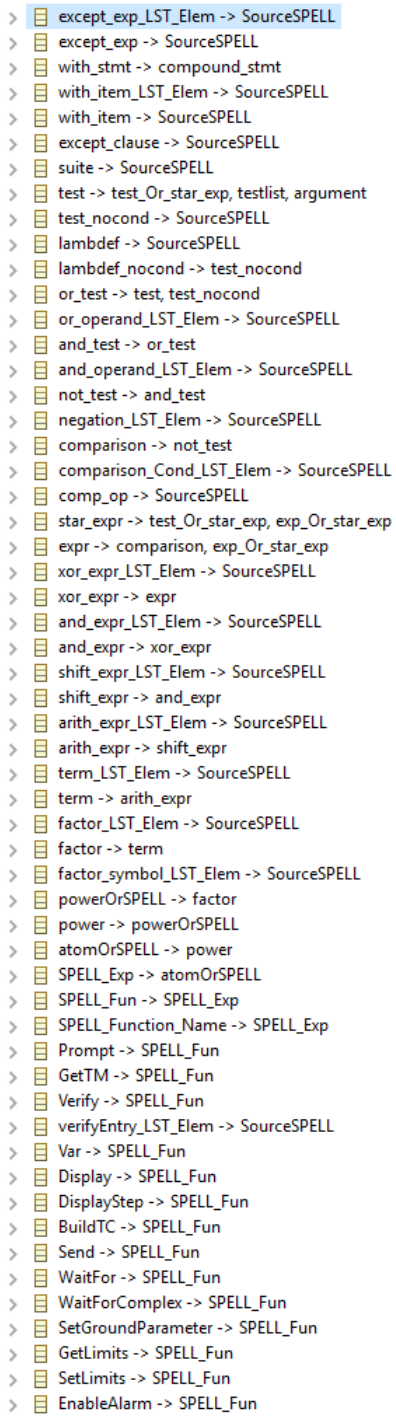


Figure A.3: SPELL meta-model, screenshot of Eclipse ecore file, part 2

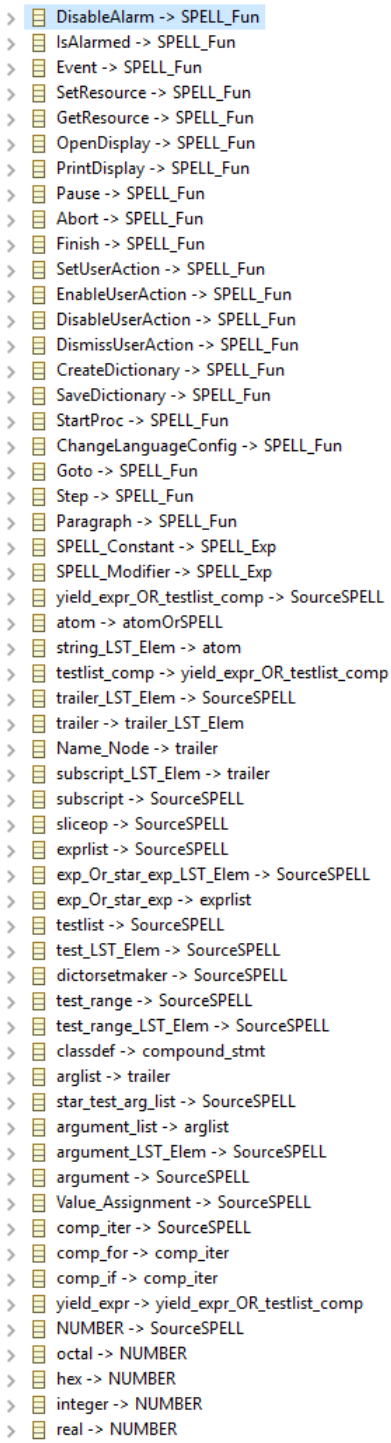


Figure A.4: SPELL meta-model, screenshot of Eclipse ecore file, part 3

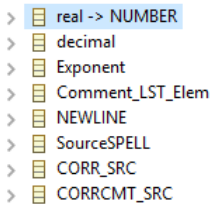


Figure A.5: SPELL meta-model, screenshot of Eclipse ecore file, part 4

A.3. SPELL-Flow meta-model

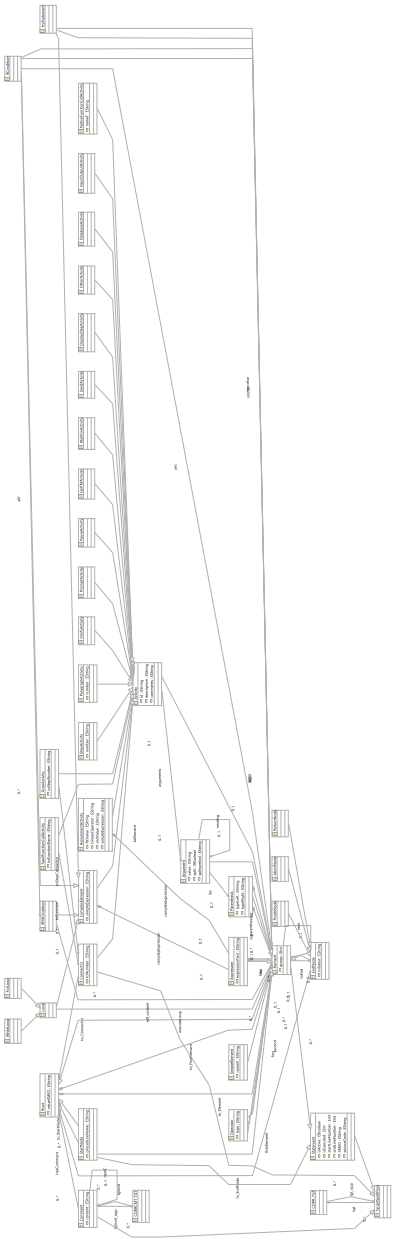


Figure A.6: SPELL-Flow meta-model, screenshot of Eclipse ecore diagram file

Figure A.7: SPELL-Flow meta-model, screenshot of expanded Eclipse ecore file, part 1

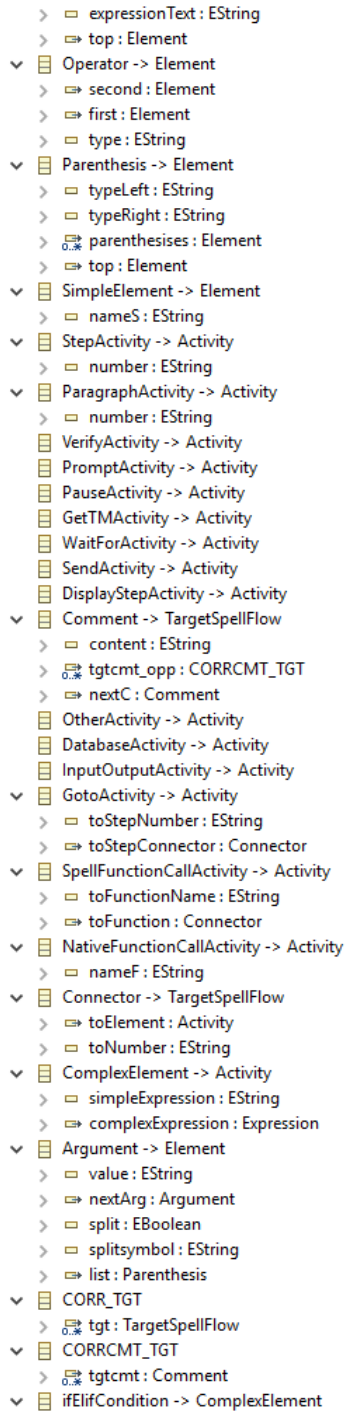


Figure A.8: SPELL-Flow meta-model, screenshot of expanded Eclipse ecore file, part 2

A.4. Correspondence meta-model

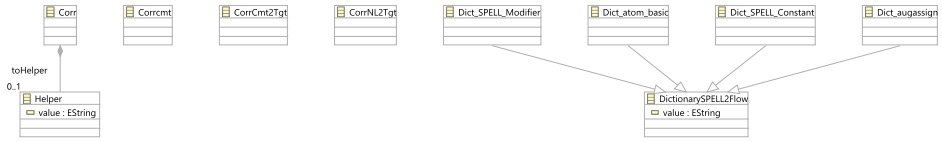


Figure A.9: Correspondence meta-model, screenshot of Eclipse ecore diagram file

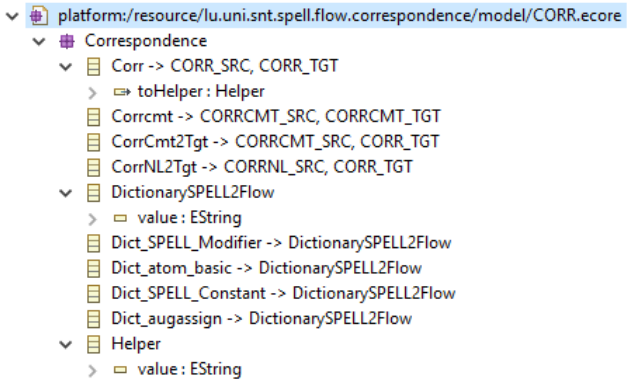


Figure A.10: Correspondence meta-model, screenshot of expanded Eclipse ecore file

A.5. Correspondence Flow2Flow meta-model

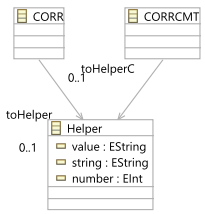


Figure A.11: Helper correspondence meta-model for SPELL-Flow to SPELL-Flow, screenshot of Eclipse ecore diagram file

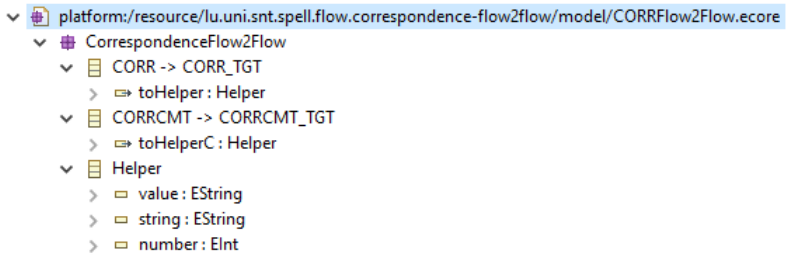


Figure A.12: Helper correspondence Flow2Flow meta-model, screenshot of expanded Eclipse ecore file

A.6. Triple Graph of Running Example in Chap. 5

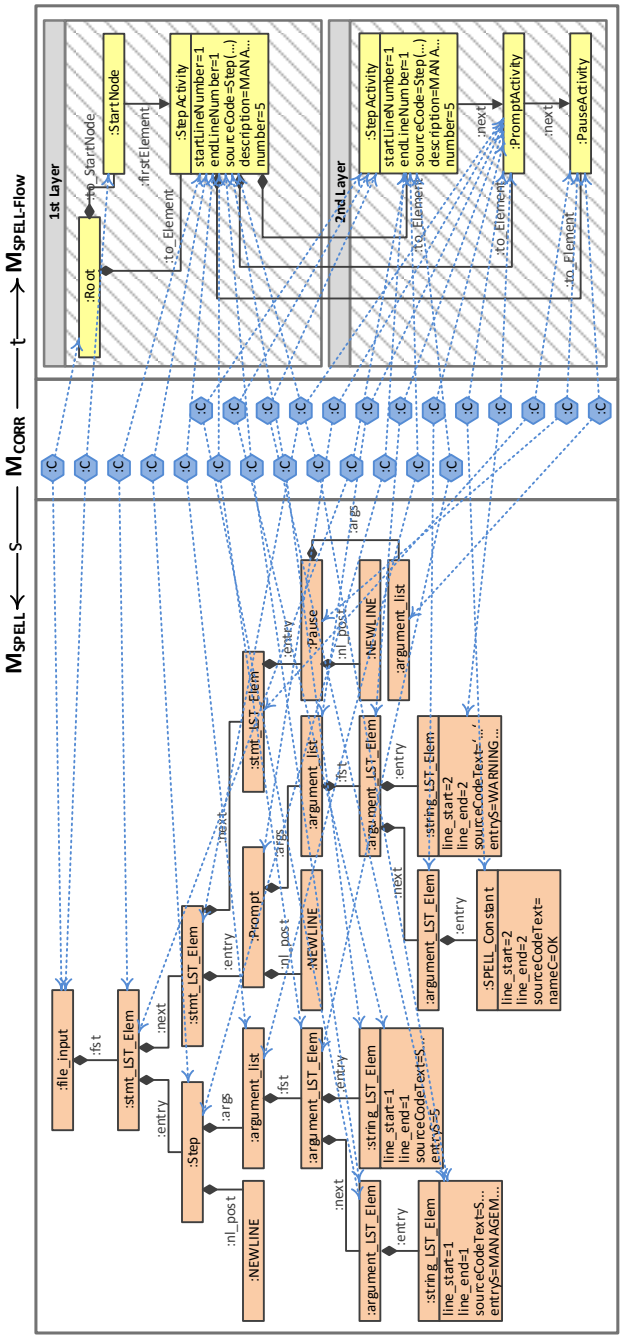


Figure A.13: Running example: triple graph that corresponds to Fig. 5.5, some attributes are omitted

A.7. List of Publications

Year	Articles / Reports
2012	*On Modelling Communication in Ubiquitous Computing Systems using Algebraic Higher Order Nets* by Susann Gottmann, Nico Nachtigall, Katrin Hoffmann, Proceedings of the 5th International Workshop on Petri Nets, Graph Transformation and other Concurrency Formalisms in Electronic Communications of the EASST (2012), http://journal.ub.tu-berlin.de/eceasst/article/view/778/768
	On the Concurrent Semantics of Transformation Systems with Negative Application Conditions by Andrea Corradini, Reiko Heckel, Frank Hermann, Susann Gottmann, Nico Nachtigall, in Universidad Complutense de Madrid Departamento de Sistemas Informáticos y Computación, TR-08/12, http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-8-12.pdf
2013	*Transformation Systems with Incremental Negative Application Conditions* by Andrea Corradini, Reiko Heckel, Frank Hermann, Susann Gottmann, Nico Nachtigall, in Recent Trends in Algebraic Development Techniques (2013), http://link.springer.com/chapter/10.1007%2F978-3-642-37635-1_8
	Towards Bidirectional Engineering of Satellite Control Procedures Using Triple Graph Grammars by Susann Gottmann, Frank Hermann, Claudia Ermel, Thomas Engel, Gianluigi Morelli, in Christophe Jacquet, Daniel Balasubramanian, Edward Jones, Tamás Mészáros (Eds.), Proc. Int. Workshop on Multi-Paradigm Modeling 2013 (MPM'13) (2013), http://ceur-ws.org/Vol-1112/
	Correctness and Completeness of Generalised Concurrent Model Synchronisation Based on Triple Graph Grammars by Susann Gottmann, Frank Hermann, Nico Nachtigall, Benjamin Braatz, Claudia Ermel, Hartmut Ehrig, Thomas Engel, in Benoit Baudry, Juergen Dingel, Levi Lucio, Hans Vangheluwe (Eds.), Proc. Int. Workshop on Analysis of Model Transformations 2013 (AMT'13) (2013) http://ceur-ws.org/Vol-1077/

Year	Articles / Reports
	Model synchronization based on triple graph grammars: correctness, completeness and invertibility by Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, Yingfei Xiong; Susann Gottmann, Thomas Engel, in Software & Systems Modeling (2013), http://link.springer.com/article/10.1007/s10270-012-0309-1
	On an Automated Translation of Satellite Procedures Using Triple Graph Grammars by Frank Hermann, Susann Gottmann, Nico Nachtigall, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, Thomas Engel, in Duddy, Keith; Kappel, Gerti (Eds.) Theory and Practice of Model Transformations (2013), http://link.springer.com/chapter/10.1007/978-3-642-38883-5_4
2014	*Triple Graph Grammars in the Large for Translating Satellite Procedures* by Frank Hermann, Susann Gottmann, Nico Nachtigall, Hartmut Ehrig, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, Thomas Engel, Claudia Ermel, in Theory and Practice of Model Transformations (2014, July), http://link.springer.com/chapter/10.1007%2F978-3-319-08789-4_9
	Triple Graph Grammars in the Large for Translating Satellite Procedures - Extended Version by Frank Hermann, Susann Gottmann, Nico Nachtigall, Hartmut Ehrig, Benjamin Braatz, Gianluigi Morelli, Alain Pierre, Thomas Engel, Claudia Ermel, TR-SnT-2014-7, http://orbilu.uni.lu/handle/10993/16887
	Solving the FIXML2Code-case Study with HenshinTGG by Frank Hermann, Nico Nachtigall, Banjamin Braatz, Thomas Engel, Susann Gottmann, in Louis M. Rose, Christian Krause, Tassilo Horn (Eds.), Proceedings of the 7th Transformation Tool Contest - part of the Software Technologies: Applications and Foundations (STAF 2014) federation of conferences (2014), http://ceur-ws.org/Vol-1305/
2016	*Towards the Propagation of Model Updates along different Views in Multi-View Models* by Susann Gottmann, Nico Nachtigall, Claudia Ermel, Frank Hermann, Thomas Engel, in Proceedings of the 5th International Workshop on Bidirectional Transformations (2016), http://ceur-ws.org/Vol-1571/

A.8. Formal Details

In the following, we cite interesting formal details, to which we referred in previous chapters.

Remark A.8.1 (Execution of Non-Deterministic Forward Propagation (cf. Rem. 3.16 in [GHN+13b]))**.** *In the first step of operation* fPpg *in Fig. 4.19, the dangling correspondences are removed by the forward alignment operation* fAln *leading to a new integrated model* $D^S \leftrightarrow G^T$*. This first step is performed via a pullback construction according to Fig. 4.18, which visualises the details of the construction of the auxilary operations* fAln, Del *and* fAdd*. The second step via operation* Del *marks all elements of the integrated model that are still consistent prepares for the deletion of the remaining inconsistent elements. This leads to a corresponding triple sequence* $(\varnothing \overset{tr^*}{\Longrightarrow} G_k)$ *with consistent integrated model* G_k*. The construction of this sequence is performed by taking the current integrated model* $D^S \leftrightarrow G^T$*, marking all elements with translation markers* $tr = \mathbf{F}$ *and applying the rules in* TR_{CC} *as long as possible. Since we do not require that the set* TR_{CC} *is deterministic, the derived transformation sequence is in general not unique. Due to the composition and decomposition result for triple graph grammars [EEE+07, HEO+11a, HEO+15], there is a corresponding forward sequence* $G_0 \overset{tr^*_F}{\Longrightarrow} G_k$ *with* $G_0 = (G_k^S \leftarrow \varnothing \rightarrow \varnothing)$*. This sequence is extended in the third step via operation* fAdd *to* $G'_0 \overset{tr^*_{F,1}}{\Longrightarrow} G'_k \overset{tr^*_{F,2}}{\Longrightarrow} G'_n$*, where* $G'_0 = (G'^S \leftarrow G_k^C \rightarrow G_k^T)$*. Due to non-determinism of the operational rules, this sequence is not always source consistent. In that case, it does not specify a valid forward model transformation sequence. Therefore, we have to apply backtracking. This backtracking is successful, because the completeness result for model transformations based on forward rules ensures that there is a source consistent forward sequence* $s_2 = (G'_0 \overset{tr^*_{F,3}}{\Longrightarrow} G'_n)$*. Note that this means that we may also have to backtrack steps of sequence* $s_1 = (G'_0 \overset{tr^*_{F,1}}{\Longrightarrow} G'_k)$ *obtained from step 2 via operation* Del*. This means that we derive from* s_1 *a separation into two sub-sequences* $s_{1a} = (G'_0 \overset{tr^*_{F,1a}}{\Longrightarrow} G'_i)$ *and* $s_{1b} = (G'_0 \overset{tr^*_{F,1b}}{\Longrightarrow} G'_k)$*, where* s_{1a} *is the part that is preserved in* s_2 *and* s_{1b} *is the part that was reverted due to possible backtracking. From these sequences, we directly derive the required modifications* $a_f \colon G_k^S \rightarrow G'^S$ *and* $b_f \colon G_k^T \rightarrow G'^T$*.* △

A.9. More details on evaluation results

In this section, we provide more details of the time measurments we presented in Sec. 6.2.2. In Fig. 6.44, we illustrated the time measurements of the three translation step via (triple) graph grammars in one diagram. The following diagrams provide the same data but separated in three diagrams.

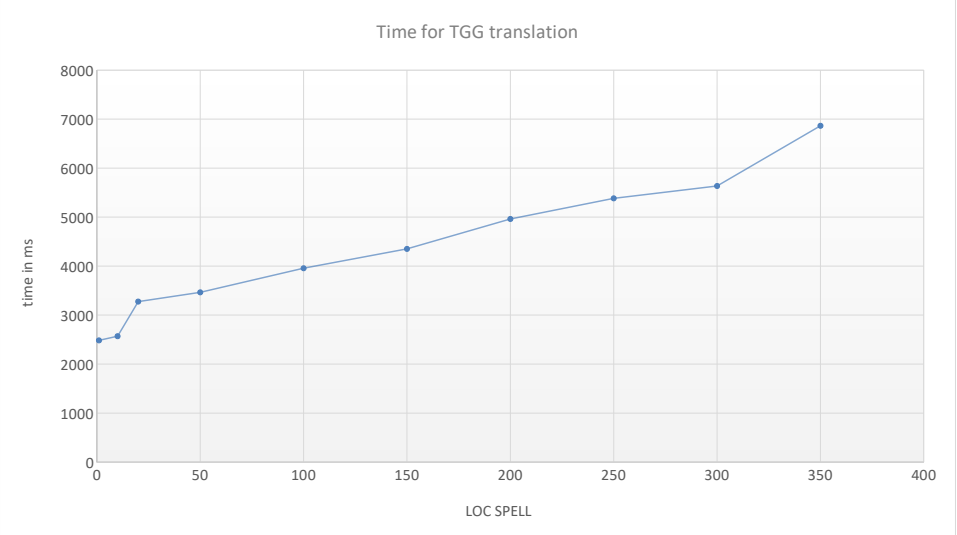


Figure A.14: Translation step 1: TGG

Figure A.15: Translation step 2: Refactoring via flat GG

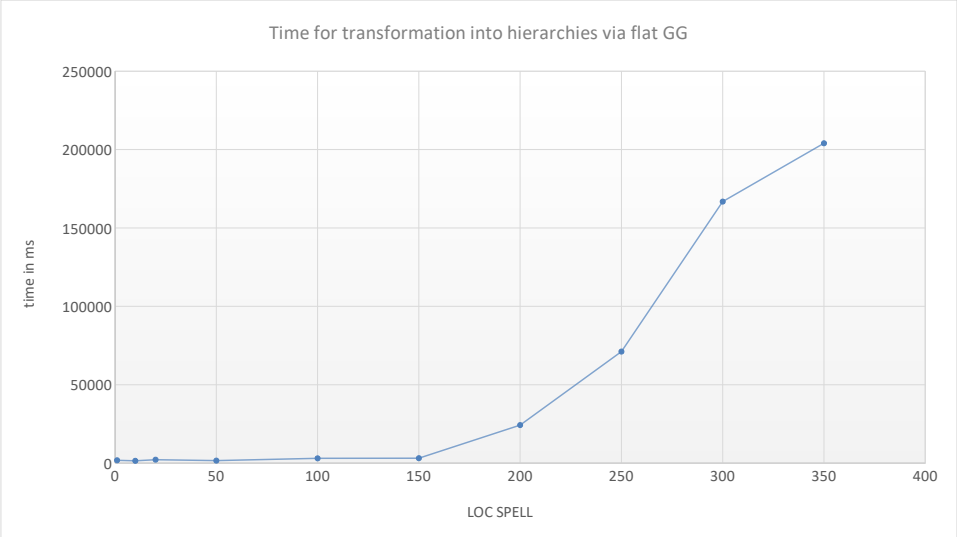


Figure A.16: Translation step 3: Hierarchies via flat GG

A.10. Numbers With Regard to Implementation

The following diagrams and tables summarise numbers with regard to the implementation of:

- Meta-Models: SPELL, SPELL-Flow, the correspondence model (CORR) which mediates between SPELL and SPELL-Flow and the correspondence model (CORRFlow2Flow) which includes helper nodes for both refactoring phases
- SPELL-Flow visualisation tool
- Eclipse-plugin for automated translation from SPELL source code files to SPELL-Flow models (i.e., XMI files)

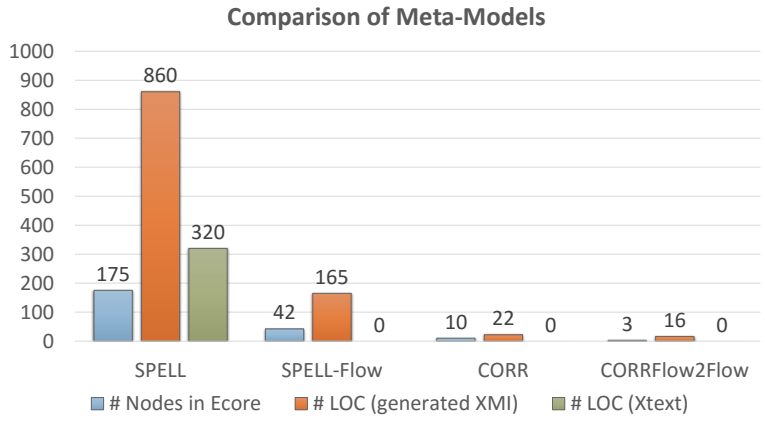


Figure A.17: Comparison of meta-models: number of nodes, number of lines of code in corresponding XMI file, number of lines of code in corresponding Xtext file

Meta-Model	# Nodes in Ecore	# LOC (XMI)	# LOC (Xtext)
SPELL	175	860	320
SPELL-Flow	42	165	0
CORR	10	22	0
CORRFlow2Flow	3	16	0
In sum	230	1063	320

Table A.2: Comparison of meta-models: numbers

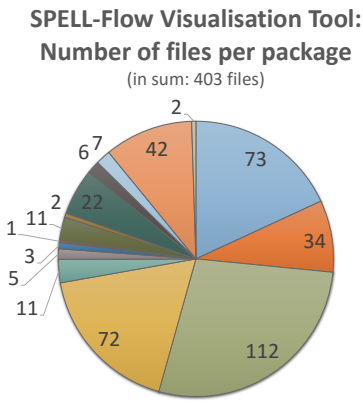


Figure A.18: Number of files per package in SPELL-Flow Visualisation Tool

Package	**# files in package**
SpellFlowLanguage.diagram.edit.commands	73
SpellFlowLanguage.diagram.edit.helpers	34
SpellFlowLanguage.diagram.edit.parts	112
SpellFlowLanguage.diagram.edit.policies	72
SpellFlowLanguage.diagram.helperClasses	11
SpellFlowLanguage.diagram.helperClasses.actions	5
SpellFlowLanguage.diagram.helperClasses.edit.parts	3
SpellFlowLanguage.diagram.helperClasses.requests	1
SpellFlowLanguage.diagram.navigator	11
SpellFlowLanguage.diagram.parsers	2
SpellFlowLanguage.diagram.part	22
SpellFlowLanguage.diagram.preferences	6
SpellFlowLanguage.diagram.providers	7
SpellFlowLanguage.diagram.providers.assistants	42
SpellFlowLanguage.diagram.sheet	2
In sum	403

Table A.3: Number of files per package

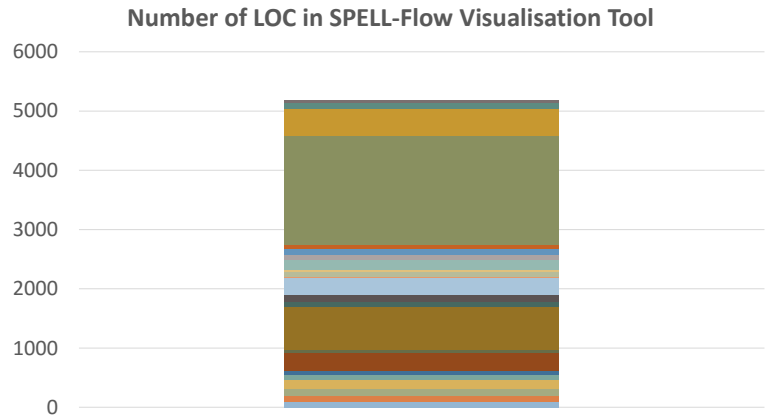


Figure A.19: Lines of code written for SPELL-Flow visualisation tool

File	**#LOC**
ActivityHelper.java	93
FancyTooltip.java	110
GoDeeperContextMenuProvider.java	103
GotoButton.java	164
GoUpContextMenuProvider.java	71
ISpellFlowOutlinePage.java	13
NavigationHistory.java	70
ParseSpell2SyntaxHighlightingLabel.java	303
SpellFlowContentOutline.java	44
SpellFlowThubnailEx.java	725
StepContextMenuProvider.java	88
GoDeeperAction.java	119
GotoAction.java	281
GotoReferences.java	12
GoUpAction.java	92
Task_.java	35
ChangedAnchorEditPart.java	161
DiamondAnchorEditPart.java	90
SpellFlowNodeToolBar.java	102
OpenDiagramSelectionRequest.java	64
EditParts	1840
SpellFlowModelDiagramEditor.java	450
OpenDiagramEditPolicy.java	115
other files	35
In sum	5180

Table A.4: #LOC written for SPELL-Flow visualisation tool

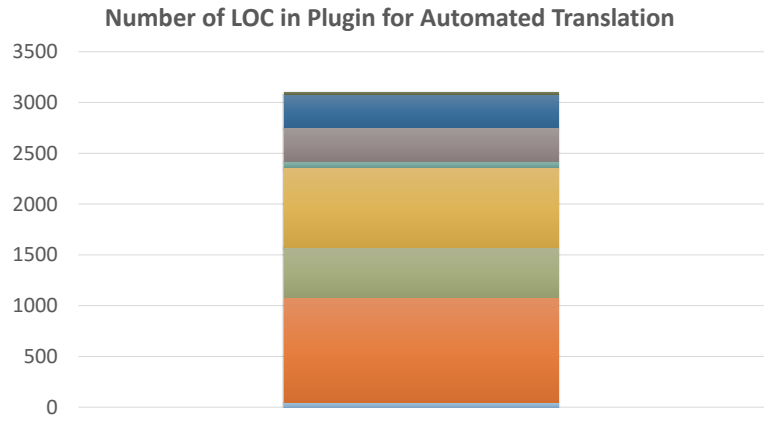


Figure A.20: Lines of code written for plugin for automated translation from SPELL to SPELL-Flow

Package	**# files in package**
Activator.java	50
ExecutionJob.java	1028
ExecutionJobFlat.java	495
ExecutionJobTGG.java	790
HenshinApplicationMonitor.java	58
TranslateButton.java	334
ModifyXMIFile.java	324
DOMRead.java	3
Children.java	18
In sum	3100

Table A.5: #LOC written for automated translation plugin