



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
Carpeq 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.

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 performs 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.



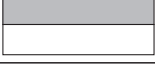





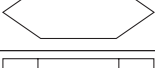
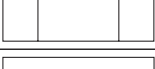

Node Type	Shape
start node	
end nodes (finish, abort, end)	
step node	
branching nodes (if, while, for, try)	
elif nodes	
gettm and verify nodes	
prompt node	
goto node	
expression node	
function call node	
other nodes not mentioned explicitly	

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 fulfil 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.

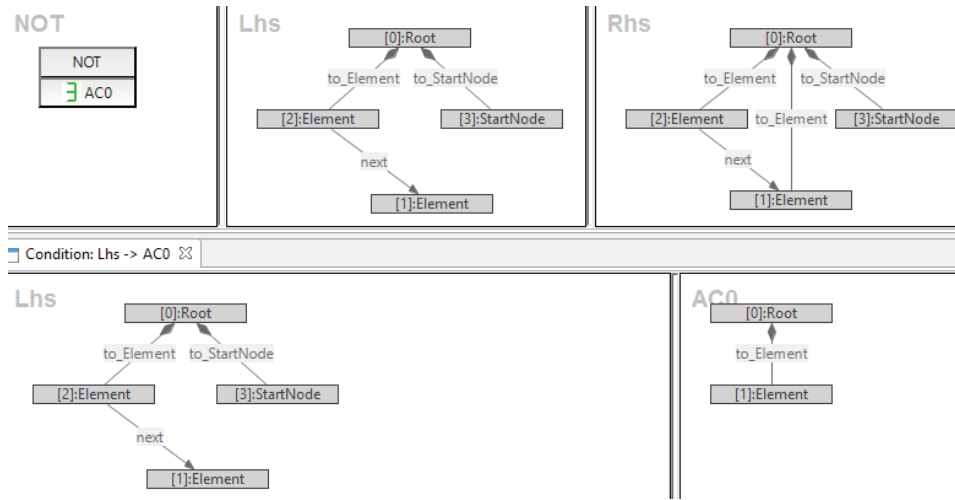


Figure 6.31: Rule Set missing Root_to_Element edge

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 implementation 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 techniques 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

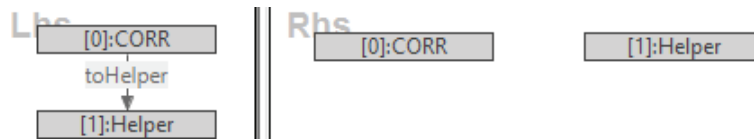


Figure 6.32: Rule remove toHelper edge

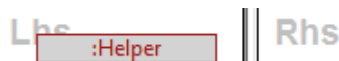


Figure 6.33: Rule remove Helper

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 variety 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. It 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-model 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, respectively, 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 menu 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 visualisation 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 fulfils 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 `*sfl` file can be selected and opened in order to open a SPELL-Flow

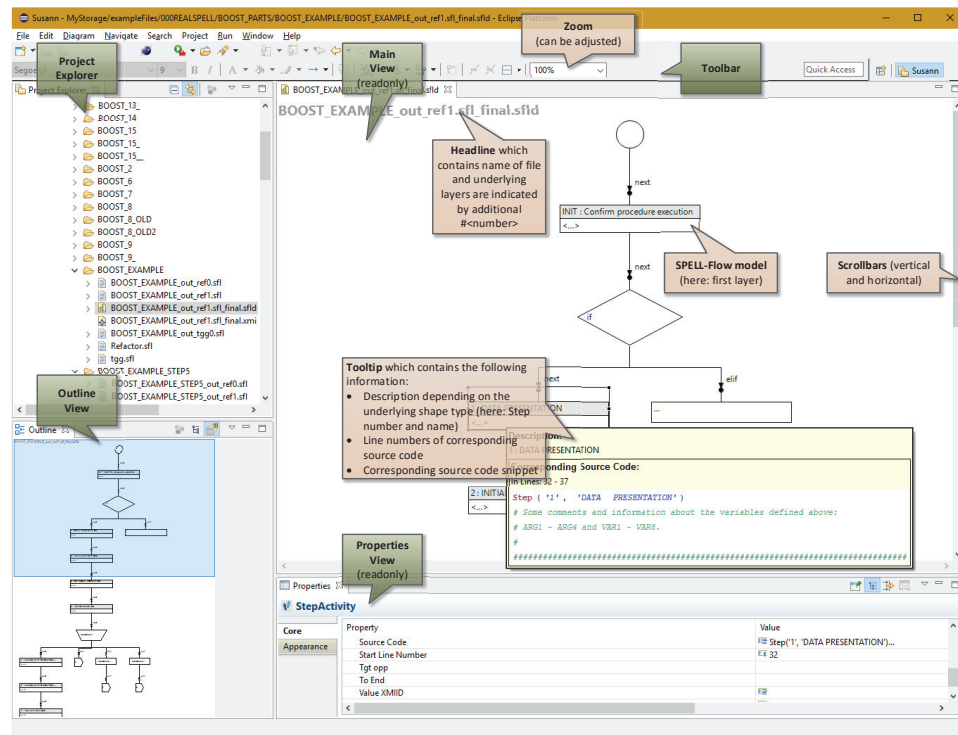


Figure 6.34: Overview SPELL-Flow visualisation tool

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 visualisation (by default the largest 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 displayed which contains the following information, if it is available:

- A more detailed description will be shown, e.g., for Step statements

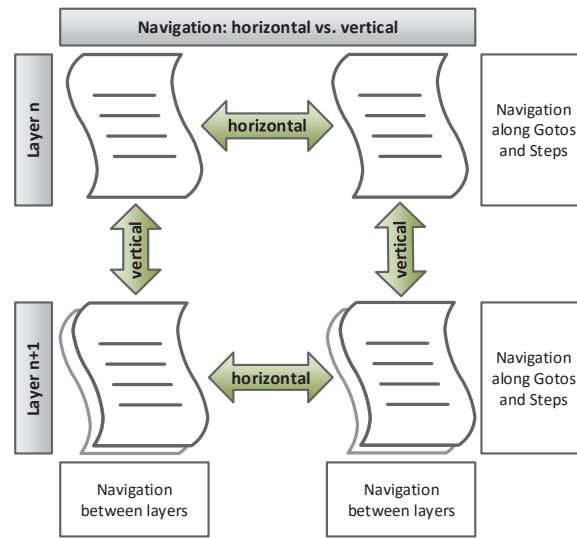


Figure 6.35: Navigation

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 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, respectively, 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 slightly 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, therefore 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). \triangle

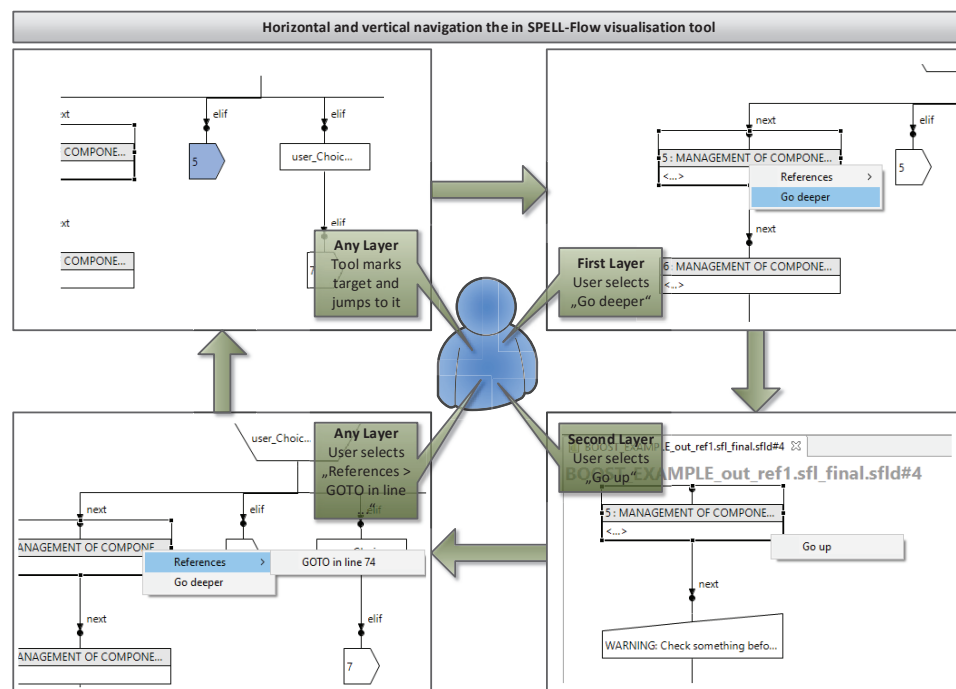



Figure 6.36: Navigation in SPELL-Flow visualisation tool

■ 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 *.sfl 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*

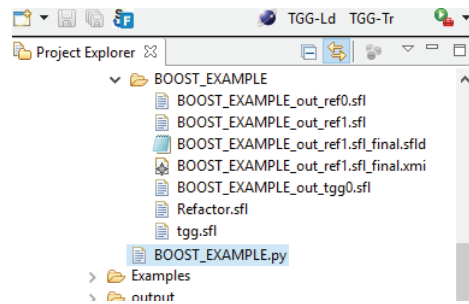


Figure 6.37: Project view in Eclipse showing newly created files after translating BOOST_EXAMPLE.py

*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 following 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 hierarchical structures are available. Its XMI representation is also stored within the file [filename]_out_tgg0.sfl.

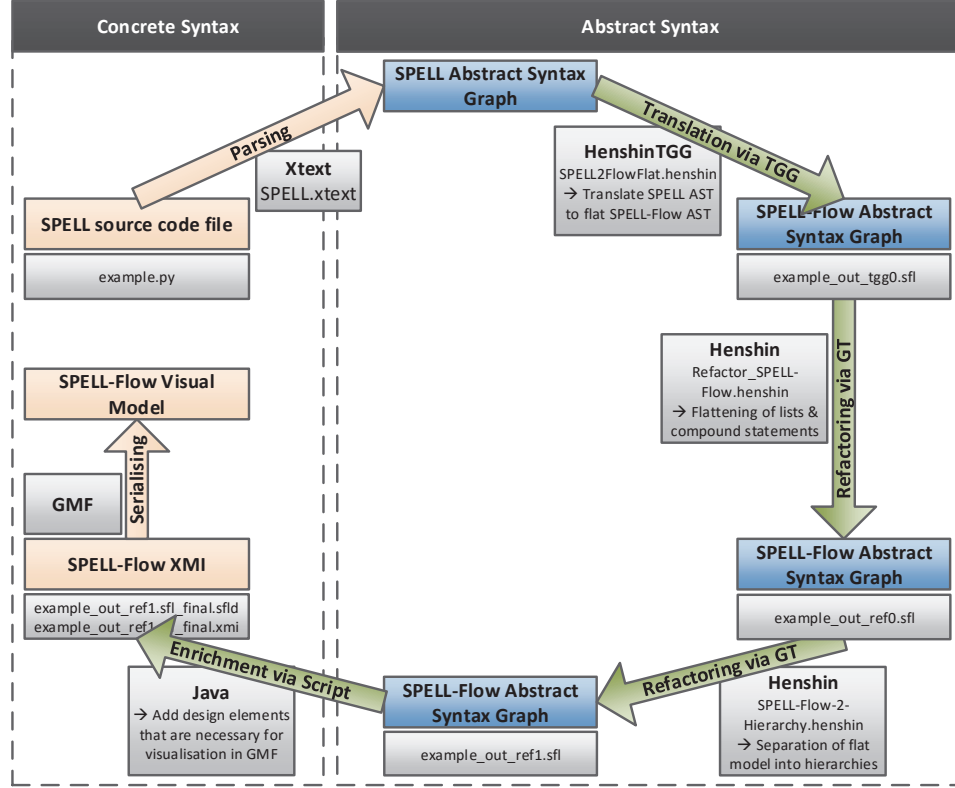


Figure 6.38: Automated Translation via Button

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.sflfinal.sfld and [filename].out_ref1.sflfinal.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

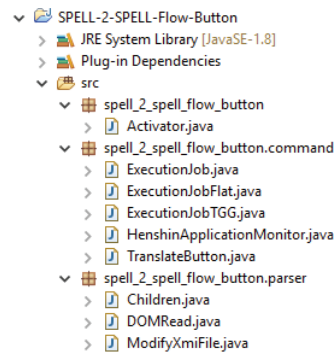


Figure 6.39: Project containing source code files for automated translation

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