Textual model-based software/system architecture documentation using MPS

Vincent Aravantinos, Kenji Miyamoto, Zaur Molotnikov, Nikolaus Regnat, Bernhard Schätz
{aravantinos,miyamoto,molotnikov,schaetz}@fortiss.org
fortiss GmbH
nikolaus.regnat@siemens.com
Siemens Corporate Technology Research

Abstract: A system architecture provides essential communication means between the architect and the user of the system diminishing thus the risk of misunderstanding during the development phase. In many domains, the tools used for documentation are word processors, typically Microsoft Word. However such tools allow to manipulate text with little structure only thus potentially yielding many *inconsistencies* in the document. In this paper, we report on a case study making use of textual models and model-based editors in order to develop such a documentation system for Siemens AG. We propose here to transfer these techniques to a new domain: (non-necessarily software) system documentation. The use of models allows to introduce more semantics into the editor, with the concrete effect of imposing more structure on the document to ensure the consistency of the document and thus prevent issues earlier in the development. We implemented the above process using the model-based programming environment MPS.

1 Introduction

Developing and documenting a system architecture is a necessary task which is essential to ensure the quality of the system both in a short- and a long-term perspective: at first it enforces activities which help to detect issues early, next the developed architecture serves as a material, based on which exchange and collaboration on the system-under-development might happen. This exchange typically happens between the system architect and the development team. The early artifact, the system architecture, can help to structure the exchange and avoid some of the misunderstandings.

In many domains, the tools used for documentation are office software applications, typically Microsoft Word or Powerpoint. However such tools only allow to manipulate text with a structure whose semantics are not clearly identified, e.g.:

- Word allows to organize a document in sections and subsections, but the meaning of these sections w.r.t. the system itself only lies in the eye of the beholder. For instance, when the section-structure of the document is supposed to mimic the compositional structure of a system (i.e., one section documents one sub-system): the architect and the user are aware of this intention, but *Word is not.*
- Powerpoint allows to draw diagrams, but the meaning of, e.g., a square inside an-

other square is defined only by common, implicit, understanding between the collaborators. For instance, when this is used to represent the structure of a system: the architect and the users know that a square in another square denotes a subsystem, or that a line between two squares represents a cable, but *Powerpoint does not*.

This potentially yields many *inconsistencies* in the document:

- In the case where the sections of the document are intended to mimic the structure of the system, then a document with, say, three sections documenting a system with four subsystems is a completely valid document for Word. Of course, it actually does not make sense w.r.t. the implicit semantics of the document.
- In the case where a diagram in Powerpoint is intended to represent a system, then a diagram connecting two subsystems that are not connectable (e.g., a USB port to an Ethernet port) is again valid for Powerpoint. Of course such a diagram actually does not make sense for the architects and users.

Based on these considerations, we developed a tool that is more aware of the intent of the document being edited, in order to avoid the above-mentioned problems. To do so, we use in this paper model-based approach to structure and bring more semantics to the documents. More precisely, we use a textual-model-based approach via the projectional editor created in JetBrains MPS[1]. This approach allows to ensure the consistency of the document and thus prevent issues earlier in the development.

The rest of the paper is organized as follows: Section 2 introduces the domain and the related encountered problems in more details. Section 3 recalls briefly notions about Model-Based Development (MBD). Section 4 presents our solution, i.e., the use of MBD in the documentation domain. Section 5 concludes the paper.

2 Domain and problem

As mentioned in the Introduction, the domain we focus on in this paper is the system documentation. Depending on the system to be documented, this domain can be pretty broad. We characterized some features of a documentation based on the analysis of various architecture documentation approaches within Siemens – from building technologies to healthcare systems.

Based on this analysis, we came up with an example document describing a system architecture for a complex software-intensive system. The system architecture involves multiple hardware subsystems as well as various interfaces and connections between those elements. This document is just an example and is specific to the domain of system architectures. However, the overall similarity with different domains at Siemens gives us confidence that our approach could be extended to the other domains.

Architectural documentation as described above contains usually textual descriptions of the relevant structural and behavioral aspects, sometimes down to a very detailed level,

[1] `http://jetbrains.com/mps`

e.g., when describing hardware or software interfaces. System architects often add pictures or diagrams in various styles to enhance the documentation. However, in many of the documents we encountered, these are only a small part of the overall documentation, often less than 10%. Note that documents with similar structure and features can be found in all Siemens divisions.

With the rise in complexity during the last decade the architecture documents also grew larger and it is not uncommon to encounter documents with several hundred or even thousands of pages nowadays within Siemens. As a consequence, this documentation is often split into several smaller documents, and different sections or documents often describe the same elements more than once.

This increase in size, partitioning and redundancy gives rise to several problems:

- inconsistencies between the textual and graphical descriptions,
- inconsistencies in the graphical parts themselves,
- non-matching interfaces between two connected elements,
- elements that are referred to (either on diagrams or in text) without being documented.

The list of possible problems that results from these documentation approaches is substantial. These problems can be the reason why the established review processes take a lot of time. Everything that slips through the review has to be compensated by the knowledge of the involved engineers and is therefore shifted to a later development phase.

3 Models

Models and model-based development have been widely adopted in several domains. Many different variations can however be encountered: some understand the notion of a model as the use of MATLAB to perform simulation and generate code, some identify the notion with UML (or SysML), when some try to capture some broader characteristics (see, e.g., [SPHP02]). In this paper, we consider the following simple but key features of a model, and put them in parallel with the domain we consider here:

1. A model is an abstraction of (some) reality, i.e., it "forgets" some details that are present in reality; e.g., in our domain, knowing the brand of a USB cable is (generally) not meaningful for the documentation of the system, so this information is absent of the model.
2. The considered level of abstraction (i.e., how much we forget) is adapted to the task which is targeted; e.g., in our domain, knowing whether a given cable is USB or Ethernet *does* matter for documentation purposes, therefore such an information should not be removed.

In this respect, Word can be seen as a model-editor which abstracts too much away by representing every element of reality by a sequence of characters.

Pragmatically, an abstraction consists in mapping some elements of reality *and their relationships* to some *meta-model*. In our case, we use a meta-model which has the usual notions of class with attributes, aggregation, composition, and inheritance [BRJ05]. These concepts have their own particularities in the context of JetBrains MPS, but these do not matter for the contents of this paper.

In our context, devising an adequate model according to item 2 above allows to solve the problems identified in the previous sections, e.g.:

- Having a class for documents, which aggregates several subsystems, each coming with their own documentation text, disallows the existence of a document with a non-matching number of subsystems and documentations.
- Having a composition relation with multiplicity exactly 1 for description text allows to impose that a given element *must* have a description.
- Having classes for different port types (e.g., USB and Ethernet) allows to express, in the meta-model, that one cannot create a cable between two ports of different types.

To implement this solution, we used a "projectional editor": such an editor displays like a usual text editor; however the manipulated object is not a sequence of characters but directly the abstract syntax tree in the input language. For instance when the user types in the keyword `subsystem`, the editor creates a node labelled `subsystem` in the abstract syntax tree with the corresponding children (e.g., for a subsystem, its name, its interfaces, among others). It then prints (or *projects*) the new abstract syntax tree onto the text which is displayed to the user.

Compared to a usual textual editor, a projectional editor has the advantage that it is model-based: it works directly with models (the abstract syntax tree structure is described by a meta-model) instead of mere strings; it is therefore impossible to build content which is not an instance of the internal meta-model. Compared to a usual graphical model-based editor, a projectional editor has the advantage of still being text-oriented. In the domain at concern in this paper, the documents usually encountered are, as mentioned above, heavily text-oriented, with only few images. Therefore, a projectional editor is a tool of choice for a textual and model-based editor. In this case study, we make use of JetBrains MPS, which has the advantage to be one of the most complete solutions when it comes to projectional editors.

4 Description of the solution

The description of our solution focuses essentially on the developed meta-model. The reader should note that an important part of the development also deals with the projection of this meta-model (of which a snapshot is shown in Figure 1), and with the generation of a LaTeX document in order to provide, from the model, a user-friendly PDF document.

We now give an overview of the main structure of our meta-model. Its (simplified) structure is summarized in Figure 2. A first observation was that every documentation encountered in our case-study consists of a front matter (title, authors, change log, and table of

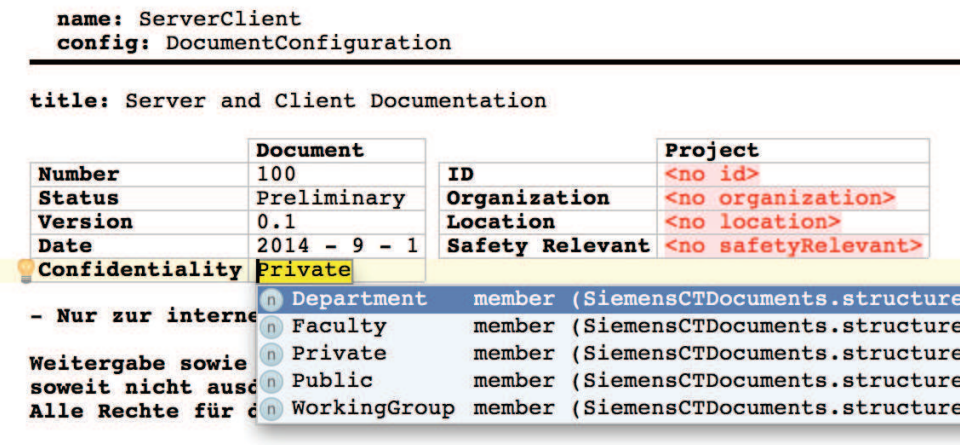

Figure 1: Part of the example documentation, partly edited in our editor

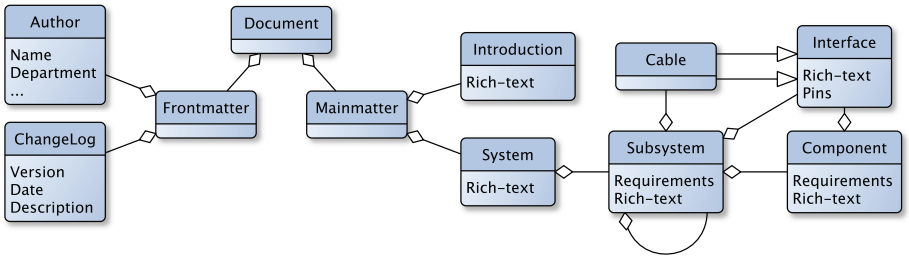

Figure 2: Documentation meta-model

contents) and a main matter containing the system architecture description. Therefore, the first class (called *document*) in our meta-model is composed of a *front matter* and a *main matter*. In practice, the main matter generally consists of four parts: Introduction, System Overview, Subsystems, and Components. The main matter class therefore composes four classes corresponding to those. The *introduction* and *system overview* are simply containers for a rich text field (i.e., a text which can have some styles, e.g., being italic, bold, or others). A *component* consists of *interfaces* and *requirements*. An interface has a specification of its physical pins (e.g., the ones of a USB plug) and a requirement is simply described textually (i.e., by a rich-text string). An interface can be connected with another by a cable. Finally, a *subsystem* is a container for components and subsystems. For each system, subsystem, component, and interface a rich-text description must be assigned, which corresponds to the necessary description of the corresponding element.

Instead of presenting our meta-model in more detail we now emphasize on the benefits that are concretely obtained, by inspection of the motivating problems mentioned in the

previous sections:

- *inconsistencies between the textual and graphical descriptions:* in our solution, the graphical descriptions are generated to LaTeX *from the model itself*, therefore ensuring the consistency.
- *non-matching interfaces between two connected elements:* the meta-model includes type-checking constraints ensuring that connected ports must have the same interface; therefore if a system contains cables connecting incompatible interface, a type error is immediately displayed, thus allowing the detection of such inconsistencies as early as possible.
- *elements that are referred to (either on diagrams or in text) without being documented:* as mentioned above, the meta-model imposes that every element *must* contain a documentation text, which cannot be empty, thus enforcing the presence of a documentation.

These improvements as well as others confirmed the validity of our approach.

5 Conclusion

In this paper, we proposed a technology transfer from techniques coming from computer engineering research (namely, model-based development) to the domain of system documentation (more particularly in our case study, of architecture documentation). The ideas and perspectives developed in research about model-based development allow to propose concrete solutions to problems encountered in the very practical domain of documentation. The use of projectional editing allowed furthermore to realize these ideas without decreasing (too much) the degree of acceptance among users. These solutions were assessed on a synthesized case study of Siemens AG, but, for a full validation, further assessments by the industrial users are needed. The satisfactory solution to the practically encountered problems is very promising and follow-ups to this work would be to develop the meta-model even more in order to fix possibly more inconsistencies, as well as to develop our solutions for more domains. Finally, we currently explore the use of graphics in projectional editing, which would bring even more benefits to documentation editing.

References

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[SPHP02] B. Schätz, A. Pretschner, F. Huber, and J. Philipps. Model-Based Development of Embedded Systems. In J.-M. Bruel and Z. Bellahsene, editors, *Advances in Object-Oriented Information Systems, OOIS 2002 Workshops, Proceedings*, volume 2426 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2002.