MORPHOSIS:
A Lightweight Method Facilitating Sustainable Software Architectures

Heiko Koziolek*, Dominik Domis*, Thomas Goldschmidt*, Philipp Vorst*, Roland J. Weiss†
*Industrial Software Systems Program
ABB Corporate Research, Ladenburg, Germany
†ABB Power Generation, Genova, Italy
heiko.koziolek@de.abb.com

***Abstract*—Managing the cost-effective evolution of industrial software systems is a challenging task because of their complexity and long lifetimes. Limited pro-active evolution planning and software architecture erosion often lead to huge maintenance costs in such systems. However, formerly researched approaches for evolution scenario analysis and architecture enforcement are only reluctantly applied by practitioners due to their perceived overhead and high costs. We have applied several recent sustainability evaluation and improvement approaches in a case study to the software architecture of a large industrial software system currently under development at ABB. We combined our selection of approaches in a lightweight method called MORPHOSIS, for which this paper presents experiences and lessons learned. We found that reasonable sustainability evaluation and improvement is possible already with limited efforts.**

I. Introduction

Software systems in the industrial automation domain often have a lifetime of more than ten years after being deployed to a customer. Distributed control systems feature millions of lines of source code and challenging extra-functional requirements. During their lifetime, they have to be adapted to new platforms with typically shorter life-cycle. They have to incorporate new requirements to stay competitive. To cost-effectively evolve such software systems during their long lifetimes is an enormously challenging task [1].

Researchers have proposed a variety of approaches to evaluate and improve the sustainability of software architectures [2]. Recent approaches are for example scenario-based evaluation methods [3], architecture enforcement methods [4], and architecture-level code metrics frameworks [5]. These methods, however, have gained only limited adoption in practice [6]. Many practitioners are reluctant to apply them because the return on investment is unknown. Some methods are perceived as heavy-weight and intrusive, thus delaying the development process without providing immediate benefits.

At ABB we were in charge with evaluating the architectural design and initial implementation of a novel system from the industrial automation domain. The system is based on a considerable re-design of a former system, which grew to several million lines of code during its lifetime. At the end, the former system suffered from significantly growing maintenance costs and the difficulty to add new features.

To assure the sustainability of the new architecture, we applied several state-of-the-art approaches, which had not been used in such a setting before. We combined them in a holistic method called MORPHOSIS. The included approaches concerned the following activities: We analyzed the architecture against a number of future evolution scenarios [7]. Further, we integrated architecture enforcement into the build process of the system. Finally, we set up a reporting framework for several novel architecture-level code metrics from recent literature [5].

The contributions of this paper are experiences and lessons learned from applying the approaches underlying MORPHOSIS. Contrary to the perceived heavy overhead of the included approaches, we argue that our method needs limited resources and thus is lightweight. Although we can currently not provide a quantified return on investment of the method, we found initial evidence of the promised benefits.

II. MORPHOSIS

Our analysis team was involved in consulting an ABB software development unit in improving the sustainability of their product. The product had undergone an extensive architecting phase and the implementation had started for several months. The architectural design featured a better modularization than the former version and had already been prepared for several potential changes. Still, the development unit did not systematically plan the evolution of the system and was concerned about how to keep the implementation compliant to the architecture design.

After an extensive literature search [2] we followed an approach from different perspectives to achieve sustainability improvement on different levels (Fig. 1). To deal with the challenge of technology changes and unexpected re-designs, we conducted an *evolution scenario analysis* based on the software architecture design (Section II-A).

To avoid architecture erosion, we set up tools for *architecture enforcement*, which ensure compliance of the source code to the designed architecture (Section II-B). Finally, we installed an *architecture-level metric tracking*

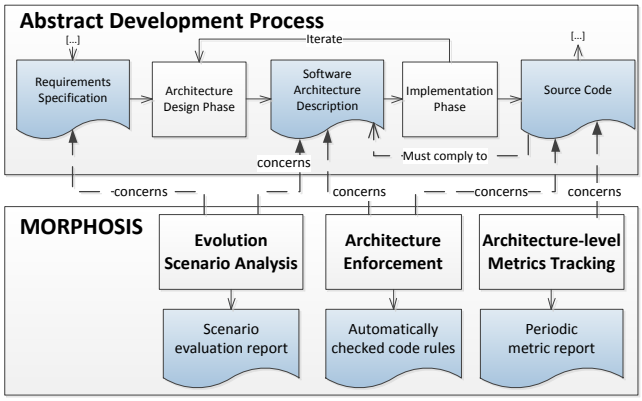

Figure 1. Overview of the MORPHOSIS activities: different phases and artifacts are covered in a holistic approach

framework based on state-of-the-art architectural metrics, which helps to assess trends of sustainability over the course of development and evolution (Section II-C).

A. Evolution Scenario Analysis

The analysis of exploratory scenarios is especially difficult because future changes are difficult to predict [8]. Existing case studies often identify a high number of scenarios but do not evaluate each of them in depth [7]. In our study, we focused on a limited number of scenarios, which we evaluated on multiple levels (e.g., requirements, architecture, code).

We conducted an evolution scenario analysis according to an extended version of the ALMA method [7]. As suggested by ALMA, we performed a combined top-down, bottom-up scenario elicitation. After an initial literature search (*step 1*), we interviewed eight experts from the business domain of the system under study for emerging business and technology trends (*step 2*, top-down elicitation). This investigation resulted in a list of 31 categorized evolution scenarios for generic systems in the business domain. The list contained a subjective evaluation of the importance of each scenario w.r.t. maintainability, which helped in prioritization.

Next, for bottom-up elicitation, one of our team members visited the target ABB development unit (*step 3*). We analyzed the most important development artifacts, such as the requirements specification, the architecture documentation, the initial version of the source code, and the process documentation. We discussed trends and anticipated changes to the system with selected members of the development team.

From the list of generic scenarios and the knowledge gained from bottom-up elicitation, we then selected a number of scenarios most important for the system under study and analyzed their impact based on the available development artifacts (*step 4*). This included tracing the impact of each evolution scenario to the requirements, to the architectural design, and to the source code. We identified the most affected components in the architectural design. Then, we applied dependency management tools, such as NDepend[1] and CppDepend[2] to assess the amount of affected source code as well as potential ripple effects to other components.

Seven evolution scenarios were selected to be analyzed in detail. They concern different changes, such as technical changes (e.g., changing or updating the operating system), business-motivated changes (e.g., increasing the capacity of the system) and industry-standard related changes (e.g., implementation of a new protocol).

Our scenario description template consists of the following 13 items: Besides describing *(1) business goals* of a scenario, *(2) impacted requirements*, and *(3) impacted components*, we also analyzed the *(4) change history* of a similar former system to better understand and motivate each scenario. The extrapolation of trends shows that some scenarios are simply instances or next steps of more long-term underlying trends. We also identified *(5) sensitivity points* in the architecture (i.e., especially critical elements) as suggested by ATAM. We discussed *(6) alternatives* for dealing with the scenario and then evaluated the scenario according to the following criteria: *(7) 5-year occurrence probability*, *(8) abstract cost estimate* (scale 1-10) for implementing the scenario, *(9) risks for ABB* and *(10) customers*, *(11) sales benefits*, and *(12) mitigation effort*. Each scenario was accompanied by four to five *(13) concrete recommendations* on how to improve the architecture to cope with the change.

Figure 2 shows a summarizing high-level diagram which provides a condensed view on the evolution scenarios. The size of each bubble indicates the impact of the scenario on the architecture and thus correlates with the estimated costs.

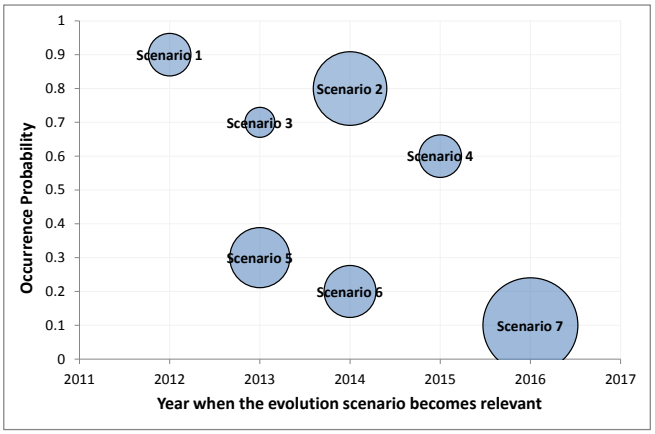

Figure 2. Evolution scenario ranking (the size of a bubble indicates the expected impact of the scenario on the system)

[1]http://www.ndepend.com
[2]http://www.cppdepend.com

B. Architecture Enforcement

Module layers define allowed dependencies between a system's modules. Over time this structure may erode and layering rules may be broken. Violated layering rules severely impact maintenance costs, as they negate the benefits of modularization and complicate module compilability, extensibility, and testability [5].

Researchers have proposed several tools for architecture conformance checking and enforcement (e.g., SARTool, Bauhaus, DiscoTect, Symphony, Lattix) [4]. They are often used on legacy systems for refactoring.

Often the architectural documentation contains the information necessary for architecture enforcement in a module architecture view. In our case, the ABB software development unit supplied us corresponding architecture models specified in UML using Enterprise Architect.

For most cases, modules are allowed to refer only to elements in lower layers (non-strict layering) or even only the directly adjacent lower layer (strict layering). Additionally, some more explicit dependency relations may be defined.

The tools NDepend (for C#) and CppDepend (for C++) check generic dependency rules out-of-the-box, such as disallowed dependency cycles between .NET assemblies. Based on fact databases extracted from the source code, they produce dependency structure matrices (DSMs) for easily analyzing module dependencies and identifying cyclic dependencies. Additionally, developers can specify custom queries to the fact databases using the declarative Code Query Language (CQL). We used this extension mechanism to define CQL queries, checking allowed dependencies from the architecture model.

For each module (i.e., a .NET assembly), we created a specific CQL rule. Listing 1 shows an example with the rule for the `User Interface` module from the Presentation Layer. According to the layer diagram, modules from that layer are allowed to access modules from the business logic layer as well as from the portability layer.

We integrated NDepend/CppDepend runs including our custom CQL queries (approx. 15 queries per subsystem) into the weekly builds of the development unit's build server (Microsoft Team Foundation Server). Hence, dependency rules are checked regularly. Developers can then immediately identify illegal dependencies. This may provoke adapting the source code, the architectural model, or even the CQL rules to comply with a new version.

During the first build runs we already identified multiple violations of the layering structure, although only a fraction of the system had been implemented up to that point. One violation in a subsystem was an unwanted dependency from a lower layer to an upper layer. This issue originated from classes that had been assigned to the wrong module. Mitigation measures could be triggered to move the classes and restructure them into the correct modules. Another violation was found in a different subsystem, where

```
1  // <Name>Presentation Layer Dependencies for module:
2  // User Interface</Name>
3  WARN IF Count > 0 IN
4  SELECT ASSEMBLIES WHERE IsDirectlyUsedBy
5  "ASSEMBLY:User Interface"
6  AND !(
7    //all modules contained in the same layer:
8    (NameIs "UI Controller")
9  OR (
10       //all modules contained in the Business Logic layer:
11       NameIs "Client Services"
12 )
13 OR (
14       //all modules contained in the Portability layer:
15       NameIs ".Net Framework WPF" OR
16       NameIs ".Net Framework WCF" OR
17       NameIs ".Net Framework Core"
18 )
19 )
```

Listing 1. Example CQL rule for layer dependency checking.

dependencies from an upper layer directly accessed platform modules, although the architecture prescribed routing all calls through an intermediate layer. Identifying these issues proved the necessity and validity of our approach. The early identification likely avoided future maintenance costs.

We also found that parts of the architectural model, which were used to define the CQL rules, were out of date considering the design that was implemented in the source code. Thus corresponding dependency rules showed violations. This led the developers to update the architectural model.

C. Architecture-level Metric Tracking

Software metrics [9] have been introduced in many software development organizations to assure code quality.

We found more than 40 architecture-level code metrics in literature [2]. They measure different aspects of sustainability, but most of them are related to the modularization quality of the system. Such modularization metrics concern, for example, similarity of purpose within modules, encapsulation, compilability, extensibility, testability, and module size.

In the following, we describe how we selected a reasonable number of state-of-the-art metrics and integrated them into a fully automated metric tracking framework.

We started by systematically deriving suitable metrics based on our goal to improve the maintainability of the system on the architectural level. We used the Goal Structuring Notation (GSN) [10] to break down maintainability into subcharacteristics according to ISO/IEC 25010.

We emphasize that the selected metrics concern the higher level structures of the source code. These metrics have been introduced recently in literature [5] and have not been widely used in practice yet. For example, the "Module Interaction Stability Index" is normalized between zero and one and promotes the use of stable modules in lower architecture layers. It is calculated for each module based on its fan-in

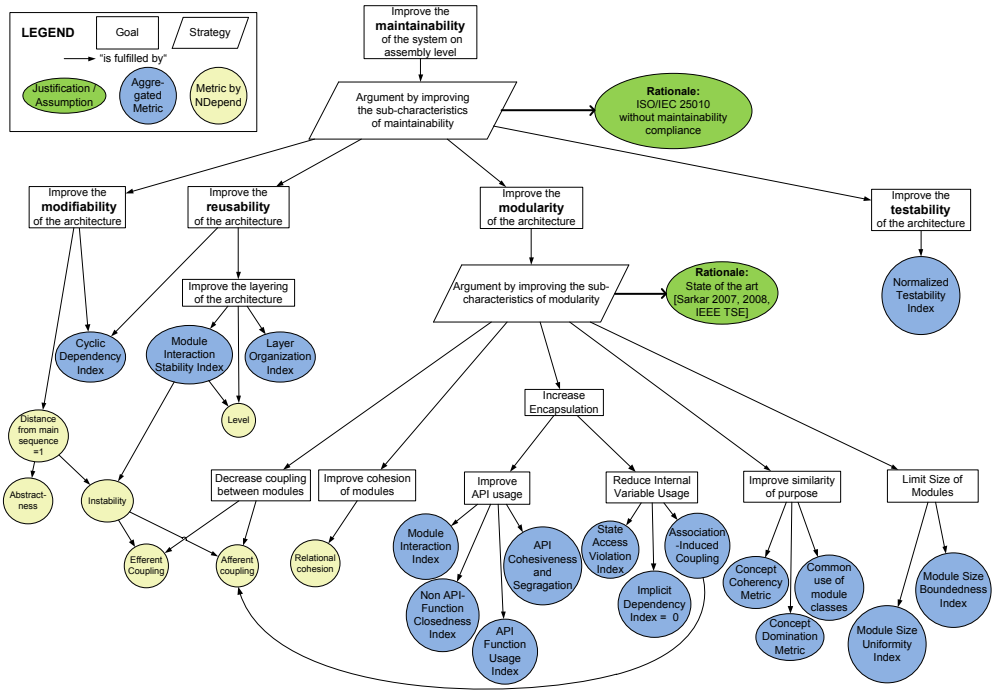

Figure 3. Derivation of complementing, assembly-level code metrics for improving maintainability (Goal Structuring Notation)

and fan-out to other modules and then aggregated for the whole system [5].

Some of the metrics conflict with each other, i.e., blindly optimizing for one metric easily leads to the reduction of the other metric. As a simple example, the ”Module Size Uniformity Index” favors similarly sized modules to ensure a clean distribution of concepts in the system structure. This metric can be optimized by having only two large modules of similar size with poor inner structures. Therefore, the ”Module Size Boundness Index” checks whether modules are larger than a given threshold in order to avoid too large modules.

NDepend and CppDepend analyze the source code and the compiled assemblies of the system at each weekly build. They generate a number of basic statistics, such as LOC per assemblies, types, and methods as well as dependencies, stability, and abstractness indices of modules.

The analysis results in a detailed HTML report and a number of XML files, which are selectively imported into an Excel workbook. Using a spreadsheet provides the advantages of user familiarity and the reuse of data importing (e.g., from Team Foundation Server via OLAP) and reporting facilities (e.g., Kiviat diagrams). The Excel workbook also calculates the architecture-level metrics, which are not directly provided by NDepend/CppDepend.

We applied our metrics tracking tool to the system under study to analyze a period of five months early in the implementation phase. This allowed us to provide first trends of the metrics to the development team. As an example, more than 96 % of types had correctly been assigned to namespaces. In addition, the graph of the *module interaction stability index* [5] shows that over time the subsystem had gained stability of participating modules (assemblies). An opposite trend would indicate that an architect might revise the distribution of responsibilities among modules of the subsystem.

Some of our early reports triggered the developers to schedule respective refactoring sessions. This significantly improved the values for the affected metrics. Regular refactoring activities are currently being integrated in the agile development process of the development unit.

III. Lessons Learned

- **Cost/Benefit:** We were able to analyze each evolution scenario within two to four person days. For the architecture enforcement, deriving CQL rules from the architectural model took another two person days per subsystem. Considering that refactoring projects to restore layering structures of similar-sized systems took several person years [5], this initial effort is easily justifiable. We plan to conduct a longitudinal study to better evaluate the benefits of these methods over a longer period of time.
  Initially setting up the architecture-level tracking consumed a considerable amount of time (approx. 4-5 person months). Main efforts were spent for metric selection, integrating the tools, and calculating the architecture-level metrics. Based on former maintenance efforts, we expect

that the cost for setting up the tools will pay off in 1-2 years.

- **Quantifying the financial impact of an evolution scenario is hard:** Initially, we tried to estimate the financial impact of each scenario based on the lines of code of the affected subsystems. However, we did not have enough data to make reasonable estimations. Therefore we decided to revert to a more abstract cost estimate on a scale from 1 to 10. While this estimate is not useful for business calculations, it still helps to compare the scenarios and prioritize them.
- **Extrapolating former trends is helpful in defining evolution scenarios:** Reviewing the evolution history of former systems in the same domain was instrumental in defining some of the evolution scenarios. There are patterns of scenarios, such as replacing the user interface technology or increasing the capacity of the system. These trends are likely to continue in the future.
- **Externalizing and prioritizing evolution scenarios provides focus:** The participating architects had already informally considered some of the evolution scenarios, but the scenarios were only indirectly made explicit. The ability to focus on the most critical and immediate scenarios as well as to prepare corresponding mitigation measures early was viewed as valuable.
- **Architecture enforcement raises developer awareness:** Installing tools for architecture enforcement led developers to put more emphasis on the architectural model. Whereas formerly some developers disregarded the architecture description, the reports of architectural violations helped to raise the awareness for architectural issues.
- **High developer interest in metrics:** Although the metrics could be misused to assess developer performance, there was a high interest from the developers in the results for the metrics. We had not expected this initially. The developers had a high interest in improving the quality of their code and were thankful for an instrument revealing potential deficiencies.

IV. Discussion

To better put our method and experiences into perspective, this section critically reflects on our approach. In our case, a sophisticated architectural model was already available. It was possible to directly derive layering rules and information necessary for the architectural metrics. Other projects, especially involving grown legacy applications, might need to reconstruct the architectural design and the layering structures beforehand.

Furthermore, MORPHOSIS will in many cases rather lead to a refinement of an architecture than a substantial redesign. Because of the required artifacts (i.e., requirements, architecture, parts of the implementation), major changes to a system at that stage are costly and hard to justify.

A *limitation* of our approach is that some effects can be judged at earliest in several years. Most of the recommended measures and thresholds for the architectural metrics are based on experience with former systems, where they could have avoided maintenance costs.

V. Conclusion

We have applied several state-of-the-art methods for different kinds of architecture sustainability evaluations and improvements to a large-scale industrial software system by ABB. We found that evolution scenario analysis is possible with limited effort. We also found that architecture enforcement can be quickly set up based on an architectural model, but that more automation support would be desirable.

As short-term future work, we will refine our metrics tracking framework and apply it to other systems. As long-term future work, we plan to conduct a longitudinal study correlating the values of architecture metrics to maintenance costs.

References

[1] T. Kettu, E. Kruse, M. Larsson, and G. Mustapic, "Using architecture analysis to evolve complex industrial systems," in *Proc. Workshop on Architecting Dependable Systems V*, ser. LNCS, vol. 5135. Springer, 2008, pp. 326–341.

[2] H. Koziolek, "Sustainability evaluation of software architectures: A systematic review," in *Proc. 7th Int. ACM/SIGSOFT Conf. on the Quality of Software Architectures (QoSA'11)*. ACM, June 2011, pp. 3–12.

[3] P. Clements, R. Kazman, and M. Klein, *Evaluating software architectures: methods and case studies*. Addison-Wesley Reading, MA, 2002.

[4] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Trans. Softw. Eng.*, vol. 35, pp. 573–591, July 2009.

[5] S. Sarkar, G. M. Rama, and A. C. Kak, "API-based and information-theoretic metrics for measuring the quality of software modularization," *IEEE Trans. Softw. Eng.*, vol. 33, pp. 14–32, January 2007.

[6] M. A. Babar and I. Gorton, "Software architecture review: The state of practice," *Computer*, vol. 42, pp. 26–32, July 2009.

[7] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet, "Architecture-level modifiability analysis (ALMA)," *J. Syst. Softw.*, vol. 69, no. 1-2, pp. 129–147, 2004.

[8] N. Lassing, D. Rijsenbrij, and H. van Vliet, "How well can we predict changes at architecture design time?" *J. Syst. Softw.*, vol. 65, pp. 141–153, February 2003.

[9] N. E. Fenton and S. L. Pfleeger, *Software metrics - a practical and rigorous approach (2nd ed.)*. International Thomson, 1996.

[10] T. Kelly and R. Weaver, "The goal structuring notation - a safety argument notation," in *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.