In Proceedings of the 5th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)

An Integrating Approach for Developing Distributed Software Systems –Combining Formal Methods, Software Reuse, and the Experience Base–

Raimund L. Feldmann
University of Kaiserslautern
Dept. of Computer Science
P. O. Box 3049
67653 Kaiserslautern, Germany
r.feldmann@computer.org

Birgit Geppert *
University of Kaiserslautern
Dept. of Computer Science
P. O. Box 3049
67653 Kaiserslautern, Germany
b.geppert@computer.org

Frank Rößler *
University of Kaiserslautern
Dept. of Computer Science
P. O. Box 3049
67653 Kaiserslautern, Germany
f.roessler@computer.org

Abstract

The development of complex software systems is driven by many diverse and sometimes contradictory requirements such as correctness and maintainability of resulting products, development costs, and time-to-market. To alleviate these difficulties, we propose a development method for distributed systems that integrates different basic approaches. First, it combines the use of the formal description technique SDL with software reuse concepts. This results in the definition of a use-case driven, incremental development method with SDL-patterns as the main reusable artifacts. Experience with this approach has shown that there are several other factors of influence, such as the quality of reuse artifacts or the experience of the development team. Therefore, we further combined our SDL-pattern approach with an improvement methodology known from the area of experimental software engineering. In order to demonstrate the validity of this integrating approach, we sketch some representative outcomings of a case study.

1. Introduction

The development of complex software systems is driven by many diverse and sometimes contradictory requirements, for instance, correctness, performance, and maintainability of resulting products as well as development costs or time-to-market. Existing software engineering concepts such as software reuse, formal methods, code inspection, or software measurement, each address some of the existing requirements. Thereby, other requirements may be influenced in a negative way. For instance, the use of formal methods increases correctness and maintainability of a product, but may have a bad influence on time-to-market. This paper introduces a development method for distributed systems that integrates formal methods and software reuse as well as the experience base concept to alleviate some of the difficulties.

For developing distributed systems, formal methods are often applied to ensure high quality of the resulting products. A well-tried combination is UML (Unified Modeling Language [10]) together with MSC (Message Sequence Charts [13]) for object-oriented system analysis and SDL (Specification and Description Language [12]) for detailed design. Analysis of current SDL-based reuse technology yields that component-based reuse (in the form of SDL procedures and classes) is well supported. High-level reuse as in the case of frameworks [18] or patterns, however, has just recently become a topic of interest in the SDL community. In [9] we present the SDL pattern approach that extends pattern-based reuse [8] for formal SDL design. It comprises a product model for reusable SDL patterns together with an iterative, use-case driven design process.

Experience with our SDL-pattern approach made apparent that reusing SDL patterns is a promising way for transferring existing design knowledge into new projects. However, as a matter of fact, developing such artifacts is quite expensive and time consuming. Additionally, the latest state of the art is always expected, as reuse benefits strongly depend on the quality of reuse artifacts. Another influencing factor is the experience of the development team with the applied methods as well as its familiarity with the application domain. This all calls for a systematic and efficient instrument for detecting and capturing improvement potential for our approach as well as offering existing knowledge to other developers. We therefore combine our SDL-pattern approach with an approach introduced by Basili et. al. [2] that defines a basic improvement

* Now working at: PaceLine Technologies, A Lucent New Ventures Company, 263 Shuman Blvd., Naperville, IL 60566, USA

process supported by a reuse repository, namely the experience base, and, measures this process with the help of goal-oriented measurement programs [3]. In accordance with our pattern-based reuse process, an improvement cycle is established, which is triggered upon the demand of application engineers and directly exploits practical experience with the reuse artifacts, the reuse process, and the application domain. Thereby, gained knowledge is systematically and directly captured through a tailored measurement program, which is executed in parallel to the development projects. The approach is organized by a reuse repository, an instantiation of the experience base, which stores the reuse artifacts and interrelates them with further experience elements that are essential to the improvement cycle.

The remainder of this paper is organized as follows: Section 2 describes the SDL-pattern approach that combines pattern-based reuse with SDL as design language. In Section 3, we integrate Basili's approach. This results in the definition of an improvement process as well as an advanced repository for comprehensive reuse. Example measurement data from a case study that illustrates how to benefit from this integrating approach is presented in Section 4. Finally, we discuss related work in Section 5 and conclude with a summary in Section 6.

2. The SDL-pattern approach: combining pattern-based reuse with SDL

2.1. The specification and description language SDL

SDL is a graphical, object-oriented description technique with a formal syntax and semantics. It is under standardization by the International Telecommunication Union (ITU). An SDL system specification is hierarchically structured into subsystems (SDL blocks) and active objects with data attributes (SDL processes), resulting in a tree-like architecture with the system as root, blocks as intermediate nodes, and processes as leaves. SDL processes are defined in terms of extended finite state machines. They run concurrently and communicate asynchronously by signal exchange. SDL adopts the object-oriented paradigm. Classes of objects are specified by so-called SDL types, which can be specialized by certain inheritance mechanisms. Associations that can be modeled with SDL are block aggregation and object communication. However, multiple inheritance and dynamic binding are not supported.

Table 1. SDL pattern description template

Name: the name of the pattern, which should intuitively describe its purpose.
Intent: a short informal description of the particular design problem and its solution.
Motivation: a short description of one or more examples where the design problem arises. This is appropriate for illustrating the relevance and need of the pattern.
Structure: a graphical representation of the structural aspects of the design solution using an UML object model. This defines the involved design elements and their relations.
Message scenario: example scenarios illustrating typical interactions between the objects involved in the design solution are specified by using *MSC diagrams.*
SDL fragment: the mere syntactical part of the design solution is defined by a generic SDL fragment representing the context invariant parts of the solution. If more than one SDL version of the design solution is possible (e.g., interaction by message passing or shared variables), fragments for the most frequent versions are included. The fragments must be adapted and textually embedded into the context specification when applying the pattern. For each fragment, this is prescribed in terms of *syntactical embedding rules* [9].
Semantic properties: the formal basis provided by SDL allows to specify semantic properties of a pattern expressed in an assumption/commitment style. That means, if the *assumptions* on the embedding context hold, applying the pattern results in the specified *commitments*. Normally, for verifying the assumptions, a tool supported verification technique based on state space exploration such as model checking is needed. In order to simplify verification, an assumption can be complemented with one or more *sufficient conditions* that imply the considered assumption and that can be analyzed statically
Refinement: an embedded pattern instance can be further refined, e.g., by the embedding of another pattern instance in subsequent development steps. Care has to be taken not to destroy a pattern's intent by such refinement steps. Therefore, refinements compatible with the pattern's intent are specified.
Cooperative usage: possible usage with other patterns is described. This links the pattern with other patterns of the repository and distinguishes our SDL pattern repository from a mere pattern catalogue where the patterns are unrelated or only loosely related.

2.2. SDL patterns

Scattered parts of a given SDL specification may collaborate and thereby offer a certain functionality. By analysis, abstraction, and documentation, such a design solution can be reused whenever the design problem arises again. This is the main idea of software patterns in general and of SDL patterns in particular. Roughly speaking, an SDL pattern is defined as a reusable software artifact representing a generic solution for a recurring design problem with SDL as applied design language. SDL patterns describe design

expertise and experience gained in prior projects and allow to pass this knowledge on to other developers.

Design reuse as in the case of patterns is more flexible than code reuse, but the learning curve required before a pattern can be reused could be very high. It is therefore important to keep the specification of SDL patterns precise but intelligible. For this purpose, a standard description template is defined (Table 1).

2.3. A reuse-driven SDL process

An SDL system development process [14, 17] generally encompasses the following development phases: object-oriented analysis, SDL-based design, formal validation, and, finally, code generation. In [9] we present an iterative, use-case driven design process tailored to SDL patterns (Figure 2). First, system requirements are decomposed into single protocol functions, which are further analyzed and designed one after the other in single development steps. Analysis includes developing (or improving) an outline system architecture, finding and exploring use cases, and breaking them down into single collaborations between the design elements. Detailed design is done incrementally by incorporating the discovered collaborations step by step. In each design step we analyze whether existing SDL patterns can be applied, or an ad hoc solution must be developed. If a pattern has been selected, it must be adapted and finally composed with the embedding context specification according to its application rules. The result of each development step is an executable SDL design specification, which is validated before entering the next step. A comprehensive case study is described in [15]. Tool support including automatic code generation is offered by several SDL development environments together with SDL-pattern specific tool components [4].

3. Extending the SDL-pattern approach to allow continuous improvement

The initial driving factor for this integration was the desire to systematically evaluate and improve our SDL reuse artifacts [7]. However, the resulting approach also takes into account other products, such as training material for unexperienced developers, as well as applied processes and tools, and hands them over to new projects when requested.

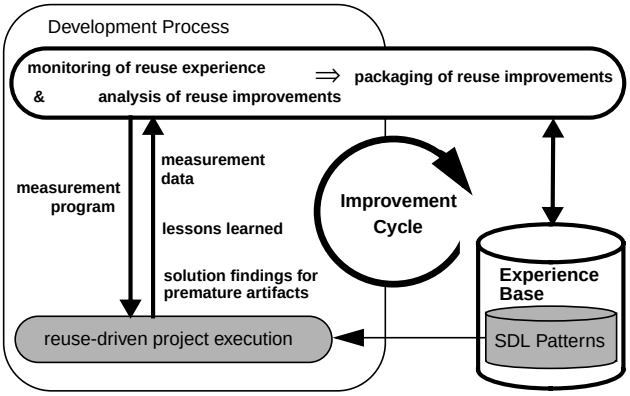


Figure 1. Interaction of Experience Base and improvement cycle

3.1. The experience base concept for comprehensive reuse

In [2] Basili et. al. suggest a logical and/or physical organization that supports the continuous improvement of an organization by: a) transferring experience from former projects into new projects, b) systematically analyzing each project to gain new or validate existing experience, and c) package, i. e., (re-)structure and store, the results from the analysis, so that they can be used as valuable input for future projects. Every single project is consequently seen as an experiment from which the organization can gain new experience to improve its competence. Therefore, each project is conducted according to the six steps of the Quality Improvement Paradigm (QIP) [2] to ensure capturing of the projects' experience for future projects. In addition, each project is monitored by carefully planned, goal-oriented measurement programs. They are used to provide quantitative data for the analyzing and packaging steps of the QIP. During these two steps, the newly gained experience is fixed and transferred into the experience base, a comprehensive reuse repository that, on demand, offers the existing experience to new projects that are being planned or executed.

3.2. An improvement cycle for the SDL-pattern approach

Our integrating approach supports an incremental improvement of the SDL-pattern approach that is essentially driven by immediate practical experience and triggered upon the request of application engineers. Figure 1 shows a graphical representation of the suggested improvement cycle (bold lines) and its interaction with the reuse process (thin lines). It is shown that reuse activities during the execution phase of a project are monitored by means of a measurement program, which, for instance, gives feedback concerning the quality of the artifacts or the processes

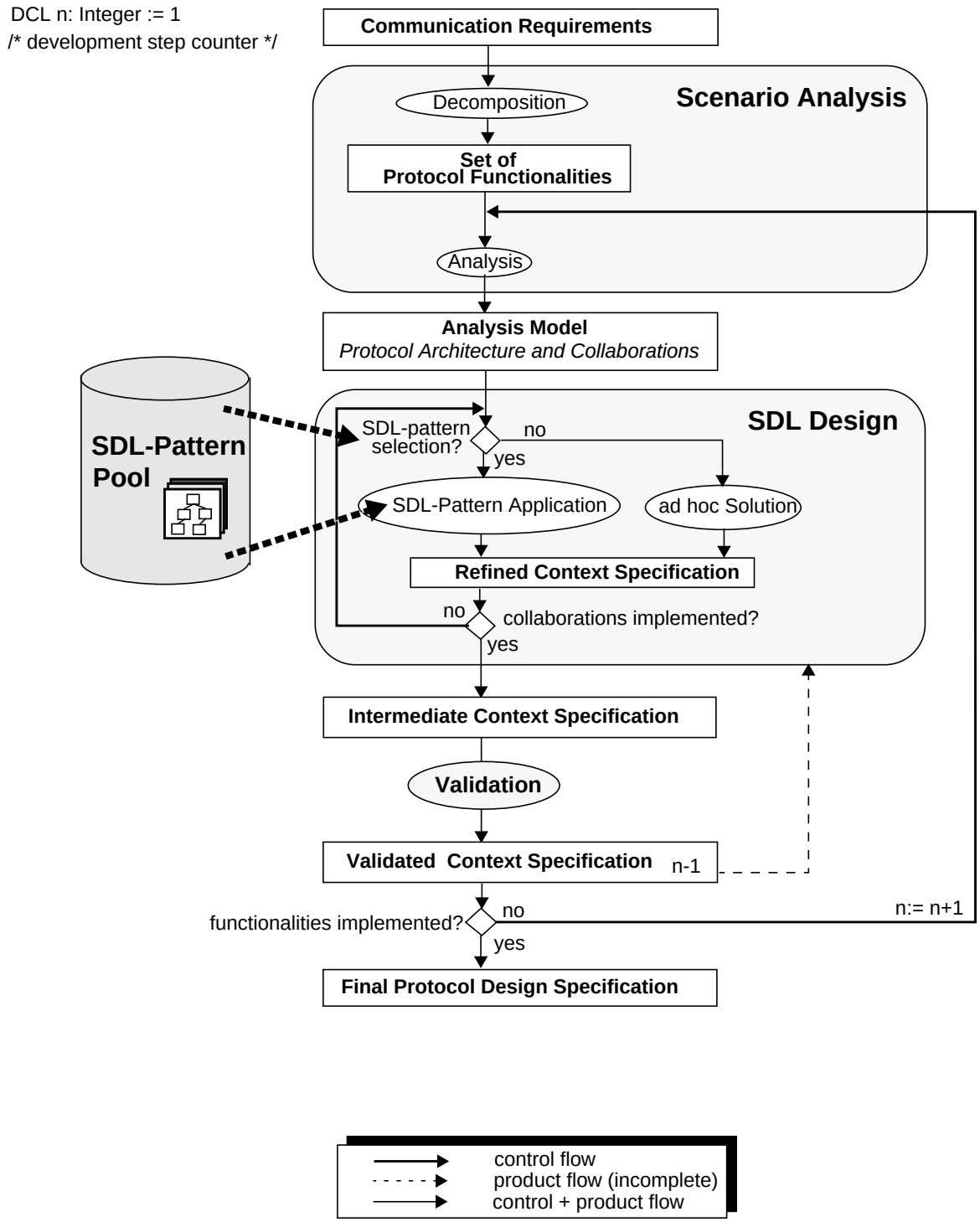


Figure 2. Reuse process model (partial)

in use. The achievements flow directly into a proposal for improvements. The suggested improvements need to be related to experience from other application projects. This is done in a subsequent packaging step, so that the improved artifact or process can finally be stored in the reuse repository, a specialized instantiation of the experience base. The reuse repository plays a central role in the improvement cycle, because it enables the transfer of reuse experience between different project contexts, which is decisive for a successful packaging activity.

Measurement program

The measurement program is based on the description models of the artifacts and the applied processes. Through the use of measurement, we aim at evaluating the current reuse practice as well as systematically identifying the potential of improvements. Therefore, the process model is supplemented with certain measurement points (see MP1 - MP6 in Figure 3). They indicate where metrics can be collected with the help of some questionnaires that must be filled out by members of the development team during project execution. Typical measurement points are the entry or exit points of a (sub-)process, or the occurrence of a special event or exception. According to the GQM paradigm [3], we defined a set of evaluation and improvement goals and subsequently refined them into more detailed questions and metrics. As an example, we show an excerpt of the GQM plan for our '*errors and faults*' model in the upper left corner of Figure 3. Within a GQM plan, all metrics for a certain measurement goal are listed. For each new project a measurement plan is set up to list all selected measurement goals and define the needed questionnaires. A measurement plan is basically a table that comprises information about all metrics that need to be collected, in order to meet selected measurement goals for the project. One column of the measurement plan denotes the point-in-time when data must be collected. Each point-in-time is matched to the measurement point(s) of the process model. Roughly speaking, a measurement plan integrates the different information from the GQM plans and the process model. Once the measurement plan is fixed, the table is resorted according to the different point-in-time entries. The number of different points-in-time defines the needed questionnaires for the data acquisition of the project. As an example, we illustrate an excerpt of a measurement plan and parts of two of the resulting questionnaires in Figure 3. Note that each measurement program for a certain project can be easily composed from directly reusable artifacts stored in our reuse repository (i .e, predefined standard measurement goals or questionnaire skeletons).

We have, for instance, developed measurement programs for monitoring design errors and reuse effort [5] that enable the evaluation of benefits when compared to historical data. Evaluation plans are directly related to the classical SE goals of reducing development costs and increasing product quality. However, these direct measures give only few hints for improving a specific SDL pattern. With the assumption that the SDL-pattern approach works in principle, we can derive measurement programs from the description model that better help to continuously improve the approach. We defined different maturity levels for SDL patterns that allow the developers to participate in the improvement cycle while reusing premature artifacts. The maturity levels are presented in [7]. It can generally be stated that the effort in reusing an artifact decreases with higher maturity levels. In other words, the higher the previous investment in reuse, the higher the benefits. As long as the quality is low, the developer is mainly forced to develop a solution from scratch, but she may add value to the reuse artifact in two ways. First, by collecting experience with the premature artifact in practice and providing suggestions for improvement and, second, she justifies the necessity and adequacy of the artifact to some extent, since it has just turned out to be valuable. If the pattern maturity level is automatic and the static analyzer [4] was not able to verify some assumption on the pattern application, the developer is asked for a (statically checkable) sufficient condition that might be used. As the developer must validate the pattern application anyway, this does not cause any extra effort. As another example, the time to understand certain items of a pattern is measured, in order to get an idea of the intelligibility. Furthermore, deviations from the UML structure or the SDL fragment during pattern application are recorded for (possible) incorporation into new pattern releases.

Analysis and packaging

Due to the fact that the description model of SDL patterns (and the accompanying reuse process model) is rather detailed, project-specific analysis proceeds straightforward. As illustrated above, the outputs from the measurement program capture many areas of possible improvements, so that "ad hoc" analysis is often avoided. In addition to the quantitative measurement data, lessons learned are also fixed textually. They may trigger an extension of the measurement program and eventually result in an improvement of processes or products. Before the analysis results can be deposited in the reuse repository, they must be compared and related to experience from other projects. Only if the suggested improvements turn out to be general enough, can they be stored in the repository as common experience. Otherwise, they keep their preliminary status. How the packaging activity is supported by the repository itself is further discussed in Section 3.3.

Model: errors and faults

Failure: Departure of observed behavior from expected behavior;
Fault: Inconsistency in product that causes failure(s);
Error: Human action resulting in software with fault(s);

Model Definition

:

Q_2: For each detected fault: What type of errors caused the fault and how are these error types distributed?

MQ_2.1: type of error
[design error / selection error / decomposition error / analysis error]

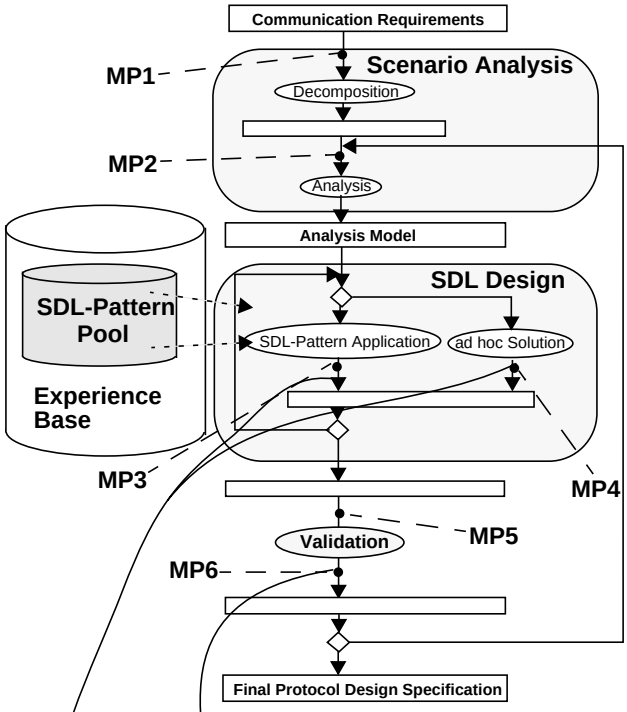
MQ_2.2: errors that caused the fault
[list of errors]

:

Q_4: How are the errors distributed to ad hoc design steps and SDL-pattern supported design steps?

MQ_4.1: design step in which errors occurred
[name of design step / number of errors]
Note: identical with MQ_3.1!

MQ_4.2: type of design step
[pattern design step / ad hoc design step]



metric ID	metric description	collected by	point-in-time	questionnaire name	question(s) on questionnaire
MQ_4.2	type of design step	developer	after composition / after elaborate ad hoc solution	number MP 3/4	Q2
:	:	:	:	:	:
MQ_2.1	type of error	developer	after validation	number MP 6	Q2 + Q2.1
MQ_3.1 / MQ_4.1	design step in which errors occurred	developer	after validation	number MP 6	Q2 + Q2.1 + Header

Questionnaire MP 3/4

Name of design step: _____
Date: __.__.____ Person:_____

:

Q2: Did you use an SDL pattern for the design step?

❒ yes ❒ no

If answered with "*yes*":

Q2.1: Which one? _____
(please provide the name of the pattern)
:
:

Questionnaire MP 6

Name of design step: _____
Date: __.__.____ Person:_____

:

Q2: Were there any observable failures? ❒ yes ❒ no
If answered with "*yes*":

Q2.1: Which type of faults caused the failure(s) and which errors were the reason for these faults? Please also name the type of each error that was made:

failure: _____ fault: _____ type of error: _____
failure: _____ fault: _____ type of error: _____

: : : : : :

Figure 3. Excerpts from the GQM-based measurement program artifacts

improvement cycle and the reuse cycle of reuse-driven SDL system development

4. Some outcomes of the integrating approach

As already indicated before, we developed measurement programs for monitoring design errors and reuse effort [5,6]. Figure 4 shows the resulting measurement data for one of the questions of this measurement program that deals with the error distribution to ad hoc and SDL pattern supported design steps (compare to question Q4 of the GQM plan in Figure 3). As can be seen, all SDL-pattern supported design steps turned out to be error free. This holds for some of the applied ad hoc design steps, too. Comparing the number of error free ad hoc design steps with the number of error prone ad hoc design steps, however, showed that about 40% of the ad hoc design steps are afflicted by errors. This is a typical number that had already been observed in former conventional student projects. For the most serious error types (e. g., deadlocks), we had fixed some design rationales in the form of SDL patterns to guide developers in future projects. The measured results indicate that the SDL patterns, indeed, help prevent these design errors [6]. However, as can be seen in Figure 5, we underestimated the analysis errors, which were not directly supported by our SDL pattern approach (compare to question Q2 of the GQM plan in Figure 3). This offers potential for further improvement of the collection of reuse artifacts and of the reuse process.

It turned out, furthermore, that in more than 40% of all design steps the developer freely decided to use the offered SDL patterns solution, instead of trying an ad hoc one. To set this number into a relation, we calculated the maximum possible number of SDL pattern supported design steps at the end of the project. It turned out that at this time we were able to provide SDL pattern solutions for up to 57% of all design steps. In other words, developers currently recognize the applicability of SDL patterns in seven out of ten cases without spending much training effort. This again indicates that our current description of SDL patterns is quite intelligible, but also that some of them should be improved.

After the project we took a closer look at the error prone ad hoc design steps together with the accompanying lessons learned. It turned out that in three of these error prone ad hoc design steps, there was a similar reason for the error to occur. A closer analysis showed that it is possible to extract an SDL pattern that, in the future, can help to prevent errors of this special type. Hence, we detected another potentially useful SDL pattern.

5. Related work

5.1. Developing reusable (SDL) artifacts

In [14] the ***methodology framework SDL+*** is described, which combines SDL and MSC for system development and suggests to reuse "*material from previous designs stored in a re-use library*". It defines an activity for archiving documents that may be useful in later SDL projects. This activity corresponds to the analysis and packaging step of our approach. However, SDL+ does not define what kind of reusable artifacts can be stored and how they are identified or designed. Neither is the internal structure of the reuse library (that also holds current project documentations) determined. This must be put in concrete form when instantiating the methodology framework for a certain project or company. In this sense, our approach could be seen as an advanced instantiation of SDL+.

TIMe (The Integrated Method) [18] is a methodology that supports the definition and reuse of SDL frameworks. Given a system specification with an infrastructure that seems to be common to a family of systems, TIMe advocates the redesign of that system into an SDL framework.

It is defined what a framework design should look like and how an SDL framework is instantiated. However, no further support is offered concerning the initial discovery or continuous improvement of SDL frameworks, nor is the administration of a reuse repository discussed.

The ***BACKDOOR*** approach [16] proposes a discovery cycle for OO design patterns that takes completed, successful designs from the development organization, and in return, delivers workable patterns as input to the development process. That is, pattern development and the rest of the software development life cycle are separated. The basic idea is to conduct reverse engineering in order to identify design structures that are pattern candidates. However, as structural analysis is not sufficient, the approach additionally relies on semantical input from the responsible designers and implementors. Different from our approach, though, pattern improvement is not considered. Furthermore, we came to a different decision regarding the separation of pattern development activities from normal development activities. We actually trigger pattern engineering when the pattern event takes place and therefore, avoid expensive reverse architecting of patterns. Nevertheless, we definitely agree that pattern development must not be an "*ad hoc process spread throughout the normal life cycle activities*".

5.2. Reuse repositories

In [11] Henninger discusses a ***repository for reusable software components***. The paper focuses on the indexing structure of such a repository. A method is suggested of how the index of a repository can be implemented "*with a minimal up-front structuring effort*" and be incrementally refined while the components are reused. The evolutionary construction of such repositories starts with the *Repository Seeding*. That is, a rudimental set of reusable components is stored in the repository with a simple, basic index structure to get the reuse activities started. Then, while the components are being reused, the index structure, which is used for finding components in the repository, is incrementally improved, according to the practical needs of the repository users. - The idea of starting the reuse process in a state of incompleteness to gain practical results as the basis for incremental improvement is similar to our approach. But we mainly apply this idea to the reuse artifacts themselves (e. g, SDL patterns), rather than to the structure of our repository. Nevertheless, the information stored in our repository becomes the more detailed, the more experience elements are being reused, caused by the growing number of relations (e. g., the `used_in/uses` relation) between the experience elements. However, Henninger offers no predefined relation structure between the components, which is a key functionality of our repository to support the improvement cycle.

Finally, the ***ASSET Reuse library WSRD*** [1] is an example of a web-based implementation of an object repository. Like our repository, it can be easily accessed via the World Wide Web (WWW). The WSRD reuse library is a domain-oriented reuse repository that contains more than 1,000 experience elements, dealing with topics such as software reuse practice or the Y2K problem. The repository is organized according to certain domains and collections. WSRD offers cross-references that interrelate the entries. These references can be compared to our relations. But they are much more general and unstructured than the relations defined in our repository.

6. Conclusion

We have presented an integrating approach for developing distributed systems that combines formal methods and software reuse with an approach for continuous improvement. The described approach extends the previously introduced SDL-pattern approach that combines pattern-based reuse and SDL with a repository supported improvement cycle. We captured practical reuse experience during project execution with the help of a customized goal-oriented measurement program in order to systematically identify possible improvements. To transfer gained knowledge between development projects, an extended instantiation of the experience base that stores and interrelates experience elements that go far beyond the reusable SDL artifacts is implemented. First experience with our integrating approach showed the following results:

- When comparing to former conventional SDL projects, it turned out that using SDL patterns results in less design errors (e. g., deadlocks) when specifying distributed systems.
- The quality of products increases, as good artifacts depend on practical and long-term experience and several engineers contribute to quality improvement over time.
- Our approach triggers the development of new reuse artifacts by clearly identifying error prone design steps that can then be analyzed to isolate the underlying design problem. The solution that is finally applied can be used as input to a first version of a new reuse artifact that, in the future, might help to avoid this particular error.

We did not describe implementation details of our reuse repository so far. However, we have developed an Internet-based implementation for distributed, collaborative usage of the repository.

Acknowledgements

Our gratitude is extended to our project leaders Prof. Dr. R. Gotzhein and Prof. Dr. H. D. Rombach. Part of this work has been conducted in the context of the Sonderforschungsbereich 501 "Development of Large Systems with Generic Methods" (SFB 501) funded by the Deutsche Forschungsgemeinschaft (DFG). Last but not least, we would like to thank Sonnhild Namingha from the Fraunhofer Institute for Experimental Software Engineering (IESE) for reviewing the first versions of this paper.

References

[1] Assets in Domain: The WSRD Reuse Library. On-line at <http://www.asset.com/WSRD/indices/domains/REUSE_LIBRARY.html>.

[2] V. R. Basili, G. Caldiera, and H. D. Rombach. Experience Factory. J.J. Marciniak (ed), *Encyclopedia of Software Engineering, Volume 1*, John Wiley & Sons, 1994, 469-476.

[3] V. R. Basili, G. Caldiera, and H. D. Rombach. Goal Question Metric Paradigm. J.J. Marciniak (ed), *Encyclopedia of Software Engineering, Volume 1*, John Wiley & Sons, 1994, 528–532.

[4] D. Cisowski, B. Geppert, F. Rößler, and M. Schwaiger. Tool Support for SDL Patterns. *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (SAM'98)*, Berlin, Germany, 1998, 107-115.

[5] R. L. Feldmann, B. Geppert, and F. Rößler. Towards an Experimental Evaluation of SDL-Pattern based Protocol Design. *SFB 501 Report 04/98*, Computer Science Department, University of Kaiserslautern, Germany, 1998, On-line at <http://www.sfb501.uni-kl.de/sfb/reports/>.

[6] R. L. Feldmann, B. Geppert, and F. Rößler. First Results from an Experimental Evaluation of SDL-Pattern based Protocol Design. *SFB 501 Report 03/99*, Computer Science Department, University of Kaiserslautern, Germany, 1999, in preparation.

[7] R. L. Feldmann, B. Geppert, and F. Rößler. Continuous Improvement of Reuse-Driven SDL System Development. *Proc. of the 11th International Conference on Software Engineering and Knowledge Engineering (SEKE'99)*, Kaiserslautern, Germany, June 17-19, 1999, 320-326.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[9] B. Geppert, and F. Rößler, Generic Engineering of Communication Protocols - Current Experience and Future Issues. *Proc. of the 1st IEEE Int. Conference on Formal Engineering Methods (ICFEM'97)*, Hiroshima, Japan, 1997.

[10] J. Rumbaugh, I. Jacobson, and G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.

[11] S. Henninger. Supporting the Construction and Evolution of Component Repositories. *Proc. of the 18th Intenational Conference on Software Engineering (ICSE'18)*. (Berlin, Germany, March 1996), IEEE Computer Society Press, Los Alamitos, California, USA, 279-288.

[12] ITU-T Recommendation Z.100 (03/93) *CCITT Specification and Description Language (SDL)*. International Telecommunication Union (ITU), 1994.

[13] ITU-T Recommendation Z.120 (10/96) *Message Sequence Chart (MSC)*. International Telecommunication Union (ITU), 1996.

[14] R. Reed. Methodology for Real Time Systems. *Computer Networks and ISDN Systems 28* (1996), 1685-1701.

[15] F. Rößler, B. Geppert, and P. Schaible. Re-Engineering of the Internet Stream Protocol ST2+ with Formalized Design Patterns. *Proc. of the 5th IEEE Int. Conference on Software Reuse (ICSR5)*, Victoria, British Columbia, Canada, 1998.

[16] F. Shull, W. L. Melo, and V. Basili. An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems. *Technical Report UMIACS-TR-96-10*, University of Maryland, 1996.

[17] Telelogic. Tau 3.4 SDT Methodology Guidelines – Part1: The SOMT Method. Telelogic, Sweden, 1998.

[18] TIMe : The Integrated Method - Web Site. On-line at <http://manon.informatics.sintef.no/time/>.