A Repository-Based Improvement Cycle for Reuse-Driven SDL System Development

Raimund L. Feldmann, Birgit Geppert, Frank Rößler

SFB 501 Bericht 11/98

A Repository-Based Improvement Cycle for Reuse-Driven SDL System Development

Raimund L. Feldmann*, Birgit Geppert+, Frank Rößler+
{feldmann, geppert, roessler} @informatik.uni-kl.de

Sonderforschungbereich 501
Technical Report 11/1998

**Software Engineering Group*
+*Computer Networks Group*

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
67653 Kaiserslautern
Germany

ABSTRACT

Reusing software artifacts for system development has become a promising way for transferring existing knowledge into new projects. Yet, the benefits that can be achieved by reusing such artifacts strongly depend on their quality. Thus, the artifacts always have to represent the latest state of the art.

This paper introduces an approach for systematically detecting and capturing the improvement potential of reuse artifacts in the context of SDL-based system development. In accordance with the reuse process of the SDL artifacts, an improvement cycle is established, which profits from the application of the artifacts in practice. The experience gained trigger the improvement of the artifacts. A reuse repository is defined, which stores the artifacts and interrelates them with further context information that is needed throughout our improvement cycle. Examples from several case studies are given to prove the validity of the approach.

Keywords

software reuse, comprehensive reuse, process improvement, Specification and Description Language (SDL), formal methods

Table of Contents

List of Figures

List of Tables

1 INTRODUCTION

Reusing software artifacts for system development has become a promising way for transferring existing knowledge into new projects. Experience in software reuse made it apparent that - beyond the more traditional reuse of code - high-level reuse of architecture and design (as in the case of software patterns and frameworks) helps to achieve systematic and widespread reuse. However, in reality, 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 these artifacts. All this calls for a systematic and efficient method for developing and continuously improving reuse artifacts.

In this paper, we present an approach for systematically detecting and capturing the improvement potential of reuse artifacts in the context of SDL-based system development [15]. For our approach we assume that a reuse process is already established within the organization's development process. In accordance with this reuse process, an improvement cycle is established, which is triggered on the request of application engineers and directly exploits practical experience with the reuse artifacts. Thereby, reuse knowledge is systematically and directly captured through a measurement program, which is executed during the development projects. The approach is organized by a reuse repository, 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 motivates the use of SDL as a design language for distributed systems and covers the state of the art in SDL-based reuse technology. In Section 3, we present our approach that extends reuse-driven SDL system development for continuously improving reusable SDL artifacts. A repository for comprehensive reuse of SDL artifacts and context information needed throughout our improvement cycle is introduced in Section 4. Finally, we discuss related work in Section 5 and summarize the results in Section 6.

2 REUSE-DRIVEN SDL SYSTEM DEVELOPMENT

For developing distributed systems, formal methods are often applied to increase productivity and ensure the quality of resulting products. A well-tried combination is UML (Unified Modeling Language [5]) together with MSC (Message Sequence Charts [16]) for object-oriented system analysis and SDL for detailed system design. In the following, we briefly introduce SDL and discuss how reuse across project boundaries is supported.

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). SDL is in widespread use in industry as design language for reactive, distributed systems, in particular, communicating systems.

Describing System Architecture with SDL

An SDL system specification is hierarchically structured into SDL blocks representing subsystems. An SDL block is either further structured into subsystems or it is decomposed into a set of SDL processes describing system behavior. This results in a tree-like architecture with the system as root, SDL blocks as intermediate nodes, and SDL processes as leaves.

Describing System Behavior with SDL

System behavior is modeled as a set of communicating objects (SDL processes) with their own flow of control. Behavior is described in terms of extended finite state machines, so that an SDL process may contain data objects. SDL processes run concurrently and communicate asynchronously by signal exchange.

Describing Data Objects with SDL

For defining data objects, SDL offers the well-known concepts of conventional programming languages. Furthermore, the abstract data type concept is applied for defining new data types. The operators can be specified axiomatically or algorithmically.

Object-Oriented Concepts of SDL

SDL adopts the object-oriented paradigm. Active objects are represented by SDL processes, passive objects by SDL data objects. An SDL block is an aggregation of subblocks or SDL processes. An SDL system is an aggregation of one or more SDL blocks. Objects can be clustered into classes which are defined by so-called SDL types. SDL types (such as block type or process type) may be parameterized and can also be specialized by adding attributes or redefining virtual types. However, associations between objects are only supported rudimentarily. Besides block aggregation, object communication is the only association that can be modeled in SDL. It is represented by special links connecting SDL processes and/or blocks. Not supported are multiple inheritance and dynamic binding.

Advantages of SDL

One main advantage of designing a distributed system in SDL is that an SDL design specification is already executable and can be used for simulation, validation against test cases[1], and validation of general properties (such as freedom from deadlock or unspecified reception). The possibility of simulating or formally analyzing a distributed system (especially its flow of control) before implementation is of great importance in order to detect design errors in early stages of development. Even code can be automatically derived from an SDL design. Furthermore, SDL system development is well-supported by existing methodologies [18, 22, 23] and tool environments [20].

2.2 Reusable SDL Artifacts

There are several kinds of reusable artifacts which differ in their abstraction level and flexibility: library components, software patterns, and frameworks. Component-based reuse is well supported by SDL. Reuse of architecture and design, as in the case of SDL patterns and SDL frameworks, however, has just recently become a topic of interest. In order to support reuse across project boundaries, a repository for those reusable SDL artifacts must be provided.

SDL Library Components

SDL supports the specification of SDL libraries where reusable procedures and type definitions (such as block types, process types, or data types) can be collected. An SDL procedure encapsulates part of the behavior description of a process. When called within a transition during system interpretation, the control flow changes to the procedure start point and returns when the procedure has finished its execution.

The library components are written in SDL without further documentation. For collecting SDL procedures and SDL types in libraries and offering them to other development projects, parameterization is especially important. Parameterization makes a library component independent from the context in which it is defined, so that instantiation is more flexible. For example, without using signal context parameters, a process type could only be instantiated in a context where the signal definitions of the context exactly match the applied signals of the process type. It turns out that the degree of parameterization is a measure for the potential of reuse of an SDL library component.

SDL Patterns

Scattered parts of a given SDL specification together may produce 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 [12]. 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

1 Test cases describe interaction behavior between active objects (SDL processes) and are specified using MSC (Message Sequence Charts).

intelligible. For this purpose, a standard *description template* is defined [10]. For later reference, details of the pattern description template are presented in Table 2.1 on the next page.

SDL Frameworks

A framework defines a domain-specific architecture common to a family of systems. It is represented as a semi-complete application that is reused and specialized to produce custom applications, i.e., family members. A framework comprises a generic system specification with several virtual SDL types and procedures which must be redefined when instantiating the framework. Concrete types that are common to all (or some) family members can also be included. Besides these SDL written parts, additional documentation is prepared which describes the purpose of the framework, how to instantiate and customize it, and its detailed design. Frameworks are often combined with design patterns [14] which enable their documentation and support their customization, in particular. The existing functionality of the framework is reused and extended by applying certain patterns which define how the framework is to be adapted.

2.3 A Reuse-Driven SDL Process

An SDL system development process [6, 18, 22] generally encompasses the following development phases: object-oriented analysis (see, e.g., [17]), SDL-based design, formal validation, and finally code generation. Each of them can be further refined. It is suggested to provide a possibility for reusing artifacts and other material from previous designs that are stored in a reuse repository [18].

In [12] we present an iterative, use-case driven design process tailored to SDL patterns. First, the requirements are decomposed. If a suitable framework is found in the repository, it is instantiated and serves as a starting point for subsequent design steps. Otherwise, SDL design starts from scratch. In each design step we analyze if an existing SDL pattern could be applied or if an ad hoc solution must be developed for the current design problem. A comprehensive case study is described in [19].

Current SDL methodologies are focused on *design with reuse.* This also holds for our pattern-based design process. A systematic method to support *design for reuse* will be elaborated in the rest of the paper.

3 THE IMPROVEMENT CYCLE

Subsequently, we present our approach to continuously improve reusable SDL artifacts. It will turn out that the approach must be operationalized individually for each type of reuse artifact. An application to SDL patterns will be presented in more detail.

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*, which, e.g., guide renaming of generic identifiers or specialization of embedding design elements. Note that an SDL fragment is generally not a syntactically complete SDL specification. From a semantic point of view, SDL fragments therefore only become formally meaningful when they are part of a complete SDL specification.
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* which imply the considered assumption and which can be analyzed statically. For instance, instead of claiming that a signal should never be consumed by another transition of the same process, it is sufficient to ensure that the applied signal is unique within that process. This can be verified statically when applying the pattern.
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.

Tab. 2.1: SDL pattern description template

3.1 Rationale and Concepts for Continuous Improvement of Reusable SDL Artifacts

At least some informal improvement process is, by necessity, common practice for each kind of reuse artifact. This is due to the fact that artifacts should fit many application contexts and be reused by different developers. Usually, we do not capture all possible achievements of a reuse artifact at the first go. Thus, ideas for a high-quality artifact come in portions while experience accumulates. As an example, we highlight the typical life-cycle of a conventional design pattern [9]: the pattern author triggers pattern development as she witnesses the application of a (potential) pattern for the first time. With the initial pattern idea in mind, she looks for similar occurrences of the pattern and builds an appropriate abstraction that is now documented. This pattern description is then submitted to colleagues for review[2]. Based upon the expert feedback, the pattern author may then revise her pattern and start

2 It is worth mentioning that certain international workshops are periodically organized with the aim to review promising pattern proposals.

another review cycle until she is satisfied with the resulting pattern quality. If pattern development is finished, the pattern is usually published or otherwise offered to software practitioners, who apply it to their software projects. Thereby, a variety of outcomes may be achieved that can influence the pattern author through a feedback loop. If the pattern turns out to be not successful, it is necessary to revise the pattern again or even reject it in the long run. Note that the development process from the first idea of a pattern until the final description that is fit for publication may last one year or more.

Because similar life-cycles are employed for the development of software components or frameworks, traditional development and improvement processes have the following characteristics in common:

- The first application of a reuse artifact requires a relatively high initial investment before a payoff can be expected on future reuse projects. That is, one must be prepared to take a risk w.r.t. reuse artifacts that have not already been validated through practical experience.
- The project contexts, where the instances of a reusable solution actually occur, are not explicitly related to the development activity of the corresponding reuse artifact. Thus, valuable information might be lost and multiple superfluous development activities may also be initiated.
- The quality of the artifacts is only assessed by the developers of the artifacts themselves. That is, we have no objective evidence for the necessity or adequacy of an artifact. In the end, this may lead to a situation that also lacks objective evaluation of reuse benefits for the organization as a whole.

These observations also hold for our former development practice of SDL patterns and SDL frameworks. We therefore suggest an improvement cycle, which is embedded into the normal software development process, so that reusable knowledge can be systematically collected and continuously packaged into reusable SDL artifacts.

For our approach we assume that a reuse process is already established within the organization's software development process and that standard description models for the reuse artifacts exist. Basically, the approach introduces an incremental development and improvement of reuse artifacts that is essentially driven by immediate practical experience and triggered on the request of application engineers. In order to guide the engineer's improvement activities and to generally reduce effort, a measurement program is developed which is executed in parallel to the development projects. *By using measurement we objectively evaluate reuse benefits and systematically identify improvement potential.*

We introduce different maturity levels for SDL artifacts that shall allow the developers to participate in the improvement cycle while reusing premature artifacts. The maturity level informs the developer about the "value" of

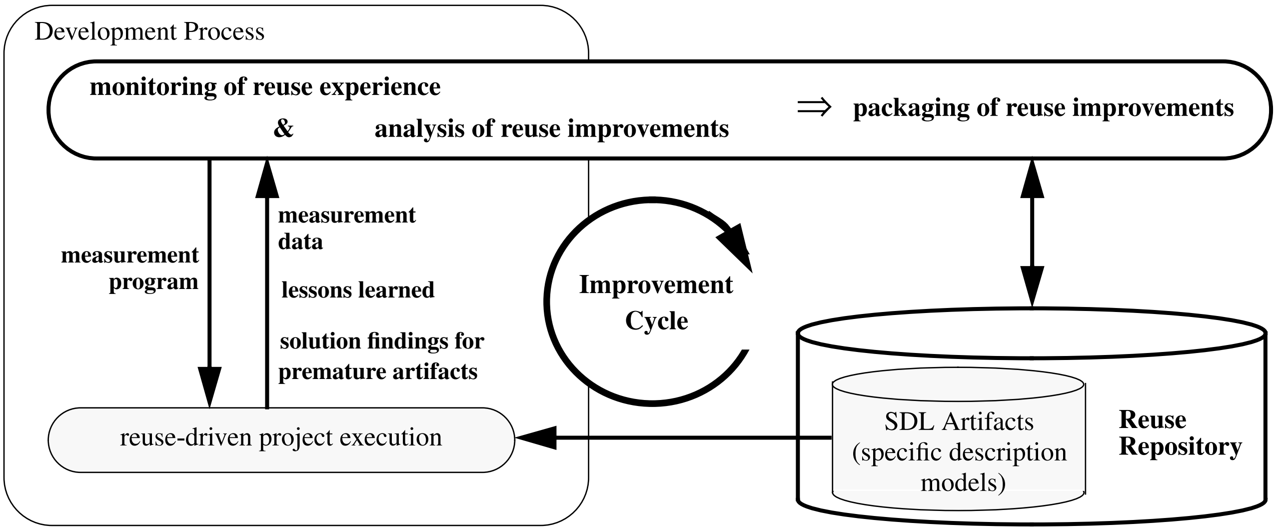

Fig. 3.1: Interaction of reuse process and improvement cycle

the artifact. The definition of different maturity levels depends strongly on the individual type of reuse artifact and will be derived from the corresponding description model. Validation status or completeness of the description are examples of maturity measures. 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.

The measurement program is based on the description models of the artifacts and also takes maturity levels into account. We illustrate a specific design of a measurement program for SDL patterns in the following section. Note that lessons learned not captured by measurement should also be documented.

Figure 3.1 shows a graphical representation of the suggested improvement cycle (bold lines) and its interaction with a given 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 gives feedback concerning the quality of the SDL artifact in use. The lower the maturity level of the artifact, the higher will be the effort for the developer to produce an adequate solution for the current design problem. However, the achievements flow directly into a proposal for improvements of the artifact, if the developer observes the description models that are given for each type of reusable artifact. 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 can finally be stored in the reuse repository. The reuse repository plays a central role in the improvement cycle, because it enables the transfer of reuse experience

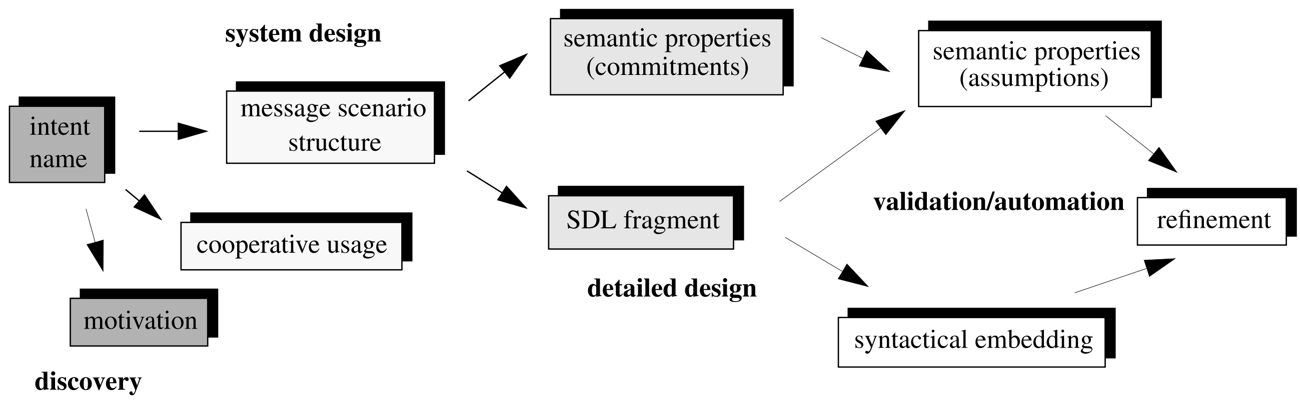

Fig. 3.2: Development steps of an SDL pattern

between different project contexts, which is decisive for a successful packaging activity. The repository and its internal structure is further discussed in Section 4.

3.2 Operationalization for SDL Patterns

In the last section it turned out that the application of the improvement cycle can in some way (several aspects depend on the individual description models of the artifacts) be highly artifact-specific. Subsequently, an instantiation for SDL patterns (Section 2) is presented. From this discussion and the explanations of Section 2, the reader shall get an idea on how to proceed in the case of SDL components and SDL frameworks.

Maturity Levels

For the definition of maturity levels of SDL patterns, we take into account validation status and completeness of description.

From Section 2 we have learned that SDL patterns in general describe and implement collaborations between active objects of a distributed design. Thus the description model comprises items for the specification of class and object architecture (including the embedding context) as well as object behavior. Furthermore, items are given that informally describe the pattern and ensure adequate pattern application (Table 2.1). In Figure 3.2 we classify the description items according to the development phases they support. It turns out that *structure*, *message scenario*, and *cooperative usage* provide the elements for system design, while *semantic properties (commitments)* and *SDL fragment* belong to the detailed design phase. Furthermore, it is shown that *syntactical embedding*, *semantic properties (assumptions)*, and *refinement* support the validation of pattern application and are prerequisites for tool support. *Name*, *intent*, and *motivation* describe a pattern informally and are therefore somehow universal in use. From these observations we derive dependences on completing/developing an SDL pattern specification. The resulting partial order is also illustrated in Figure 3.2, where an arrow indicates that a description item is completed before its corresponding successor(s). As a consequence, we have four maturity levels, which are enriched by further validation criteria in the following definitions:

- Informal: The items *name*, *intent*, and *motivation* are completed. The pattern event (some developer has witnessed the application of the potential pattern) has occurred at least three times.
- Architectural: The items *structure*, *message scenario*, and *cooperative usage* are completed. The relation (new, specialization, cooperation, change, ...) of the pattern to other members of the pattern pool has been determined, so that the necessity of the pattern is confirmed. It has been validated that the pattern architecture meets the pattern's intent. The architecture conforms to the pattern events that have occurred so far.
- Behavioral: The items *SDL fragment* and *semantic properties (commitments)* are completed. The correspondence between UML diagram and SDL specification is checked. It has been validated that the specified pattern behavior implements the message scenarios and the pattern's intent. The behavior conforms to the pattern events that have occurred so far.
- Automatic: The items *syntactical embedding*, *semantic properties (assumptions)*, and *refinement* are completed. They are incorporated into the SDL pattern editor [7] and tested for the pattern events that have occurred so far.

Note that the maturity levels do not prescribe a strict development order of the different items. Rather, the order of completion is expressed. As the witnesses of a pattern event must develop a solution for the corresponding design problem anyway, it is advantageous to partially elaborate the items beyond the current maturity level (Figure 3.2). Thus, experience from former projects is already provided according to the description model of SDL patterns and packaging is made easier when the maturity level rises later. We face this situation, for instance, if a pattern is improved on a lower than the current level. As a consequence, the current maturity level is lowered, while the item specifications stay intact (though they probably must be changed and also validated when the level rises again). Consider, for instance, an SDL pattern on the behavioral level that is improved concerning its architecture. In this case, the new maturity level is architectural, but the old pattern version will be kept in the reuse repository for later reference.

Measurement Program

By measurement we aim at evaluating the current reuse practice as well as systematically identifying the potential of improvements. One measurement approach that allows to realize these goals is the GQM paradigm [3]. We applied this approach to define several evaluation and improvement goals and subsequently refine them into more detailed questions and metrics. For instance, we developed measurement plans for monitoring design errors and reuse effort [8] 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 on principle, we can derive measurement plans from the description model that better help to continuously improve the approach. For instance, if the pattern maturity level is automatic and the static analyzer [7] 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. Nevertheless, the developer has a chance to provide further lessons learned that may finally result in an improvement of the measurement program. 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, they can 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 4.

3.3 Some Outcomes of the Improvement Cycle

We have applied the improvement cycle to several SDL projects and summarize some representative results subsequently. All projects were conducted by different teams of about 2-4 members, consisting of students and researchers of the Computer Science Department at the University of Kaiserslautern.

(Re-)Engineering of SDL Patterns

An SDL pattern, called *BlockingRequestReply* [11], was originally discovered before we applied the improvement cycle. According to our taxonomy it had to be classified as „architectural”. After applying the pattern in two further projects, the improvement cycle identified three syntactical variants of the SDL fragment and discovered two specializations of the whole pattern. As the pattern could finally be viewed as stabilized, it has reached level “behavioral” and is now being implemented with our SDL pattern editor.

Application of the improvement cycle to another SDL pattern, called *DynamicEntitySet* [11], even revealed that the pattern had only reached level "informal" before. Through practical experience, some generalizations turned out that made it a very powerful pattern to expect (it is now at level "architectural").

Evaluation of the SDL Pattern Approach

It turned out that an adequate knowledge of pattern-based design in general is of great importance to the success of a project. This made us reconsider the quality of our training material. Furthermore, experience has shown that decomposition and analysis (these are specific activities in the accompanying reuse process of SDL patterns) have greater impact on pattern-based development than expected. With the current pool of SDL patterns, a bad decomposition or analysis model may result in poor coverage of SDL patterns in the resulting specification. However, further experiments will have to clarify the exact reason.

Regarding the number and distribution of design errors, one case study [8] was performed. It turned out that design errors are reduced by the application of SDL patterns, because the only errors uncovered resulted from ad hoc design steps that did not apply any patterns.

4 THE REUSE REPOSITORY

The described improvement cycle for SDL-based system development not only suggests the systematic improvement and finding of SDL artifacts, but also comprehensive reuse of other artifacts involved in the cycle. Hence, a repository, which systematically stores the different artifacts and offers them to new projects on demand, strongly supports the improvement cycle. This section describes the structure of such a reuse repository. Since all artifacts represent experience, we use the term "*experience elements*" in the following description.

project phase		experience elements
reuse-driven	planning	process models, standardized measurement goals, measurement plans, questionnaires, lessons learned
	execution	process models, SDL library components, SDL patterns, SDL frameworks, lessons learned
improvement-driven	analysis	all experience elements used in the planning and execution phase of the project
	packaging	predefined templates / style guides to store newly gained experience, all experience elements that have been used in the project and need to be updated

Tab. 4.1: Experience elements of different project phases

4.1 The Different Experience Elements

From the last two sections it becomes obvious that there are many different experience elements involved in the improvement of reuse-driven SDL system development. Table 4.1 lists them according to the project phase in which they are involved. While the main concern with the planning and execution phases is reuse of experience elements (i.e., search and retrieval for experience elements in the repository), the analysis and packaging phases are mainly concerned about the improvement of experience in the repository (i.e., adding and updating experience elements to the repository). To allow effective storage, search, and retrieval of the experience elements in the repository, a unique representation of similar experience is regarded. Predefined templates and / or style guides have to be offered when adding new experience into the repository. Those templates are regarded as experience elements themselves, since they represent an approved way of representing special kinds of experience in the repository.

For example, all SDL patterns are represented according to a predefined structure (SDL pattern description template, Table 2.1). Nevertheless, it can be necessary to require different representation forms of an experience element in the repository. Questionnaires, for example, should be stored as a PDF file to support easy viewing and printing when searching the repository, and as a source file of a word processor (e.g., FrameMaker) so that they can be easily adapted / changed when being reused in a new project.

4.2 The Organizational Structure of the Repository

To allow for an efficient search and retrieval for the experience elements, the repository has to be clearly structured. Therefore, we subdivide the repository into so-called "*areas*" that are utterly disjunct. Each area can be seen as a class that holds instances of the same type of experience elements represented according to common styles.

The Component Repositories Area

This area contains the actual experience elements such as SDL patterns and SDL frameworks that are being applied and improved in our approach.

The Process Modeling Area

Process models describing how to conduct a project or apply a technique are offered in this area. For example, our reuse-driven design process (Section 2), the SOMT process model [22], or the Bræk and Haugen model [6] for developing real-time systems with SDL are provided in our repository.

The Measurement Area

This area stores all experience elements relevant for defining measurement programs, for capturing quantitative knowledge, and for validating or rejecting some hypothesis. In the repository that we use for our experiments, we store standardized measurement goals, measurement plan fragments, and standardized questionnaires for evaluating the quality of SDL patterns in this area.

The Qualitative Experience Area

Experience that is gained spontaneously or event-driven is recorded as a lesson learned in this area. A lesson learned can be a problem solution, a guideline, an improvement suggestion, or a result/success report. A systematic description of these categories for representing lessons learned can be found in [4]. For example, experience about adequate decomposition of communication requirements, gained in projects with students at the University of Kaiserslautern, is deposited as guidelines in this area of our repository.

All the experience elements listed in Table 4.1 can be stored exactly within one of the areas described so far. With the help of this simple structure, a person wanting to add or retrieve a certain kind of experience element gets an initial guidance on where to search or place it in the reuse repository. But this alone is not sufficient to support the improvement cycle. Furthermore, it would be useful to, additionally, capture the characteristics of a project, and support easy access to its whole documentation in the analysis and packaging phases. Therefore, our reuse repository is extended by an area for project databases.

The Project Area

In this area, all information concerning single projects is stored with the help of a predefined template. For each project a new copy of the template is created. The structure of the template is defined in accordance with the first phases of our improvement and reuse cycle and filled out while the project is conducted (i.e., planned, and executed). Among others, the project plan, the measurement plan, guidelines for the development (e.g., in form of lessons learned), and the model of the SDL development process used are stored in the entries of the template that capture the planning phase. The most important entries of the execution phase list the reused SDL artifacts, the complete system documentation, review and simulation protocols, the methodically collected measurement data, process traces, and newly gained lessons learned.

With the help of the project documentations in the project area we are now able to support the analyzing and packaging process in a better way. That is, we can offer a precise characterization and process trace for each project. Thus, we can exactly tell for each project what worked out well and should be packaged for reuse in other projects.

Relations between Experience Elements

From the description of the experience elements it is apparent that they are not independent of each other. Rather, there are many relations between them. Especially, when they are to be (re-)used together in a project. For example, measurement plans for validating or rejecting a certain hypothesis only can be (re-)used if the object that is to be measured (e.g., a SDL framework or a certain SDL pattern) is also being (re-)used. The project documentations in the project area give a first glance of which experience elements have already been used together successfully. But since all experience elements that have been especially packed for reuse in other projects are stored in areas other than the project area, additional relations are needed. These relations have to express how these experience elements are interrelated with each other in general, and how one can find project documentations in the project area in which they have been used together.

The following bidirectional relations (illustrated in Figure 4.1) between experience elements of different areas have been proven useful in our instantiation of the reuse repository:

- `measures/measured_by`: This relation is defined between experience elements of the measurement area and experience elements stored in the component repositories, process modeling, or qualitative experience areas. It is used to identify the experience elements that are measured, for example, by a measurement plan or standardized questionnaire, that are stored in the measurement area. In the other direction it helps find a measurement experience element that measures a certain experience element.
- `is_about/has_part`: This relation is defined between lessons learned stored in the qualitative experience area and experience elements of the component repositories, process modeling, or measurement area. It is used to identify the experience element that the lesson learned is concerned with. In the other direction it helps to find lessons learned dealing with a certain experience element.
- `used_in/uses`: This relation is defined between experience elements of the process modeling, measurement, or quantitative experience area, and project documentations in the project area. It allows to identify the projects in which an experience element has been used, or which experience elements have been used in a certain project. It can be assumed that an experience element that has been often successfully reused is quite valid. Thus, the number of used_in relations for a certain experience element gives a first glance about its validity.
- `gained_in/gains`: This relation is defined between a lesson learned in the qualitative experience area and a single project documentation in the project area. It identifies in which project a lesson learned was gained, or which lessons learned have been gained in a certain project.

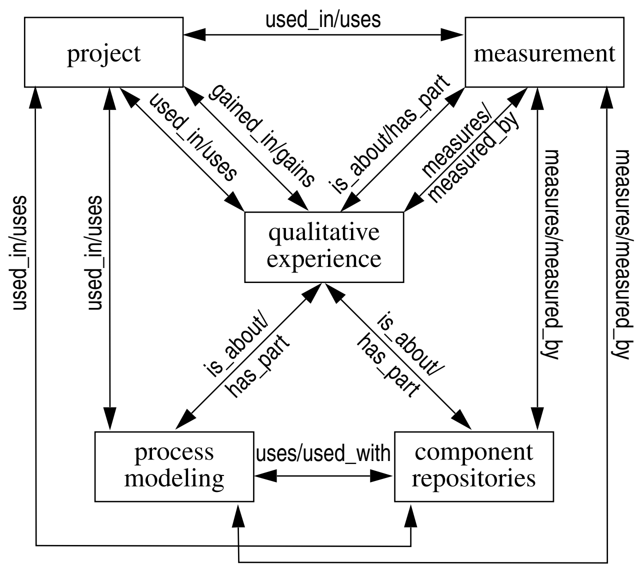

Fig. 4.1: The organizational structure of the repository

- `uses/used_with`: This relation is defined between experience elements of the process modeling area and the component repositories area. It documents that a certain experience element is used in the description of a process model. The opposite direction gives hints in which process descriptions a certain experience element is used.

Figure 4.1 summarizes the complete organizational structure of our repository with all areas and the defined relations between the experience elements.

4.3 Representation of Experience Elements and Relations

In our intranet-based implementation of the repository, we represent experience elements with the help of so-called "*characterization vectors*". A characterization vector consists of three parts. The first part holds references (HTML-Links) to the different representation forms of the experience element. In the second part, a list of existing relations to other experience elements is given (again, realized by HTML-Links). The last part consists of a number of attribute-value-pairs (e.g., <application domain, real-time systems>, <design language, SDL>). With the help of these attribute-value-pairs it is possible to describe and characterize an experience element, independently from the format in which it is stored in the repository. All characterization vectors together form a complex network that is the basis for accessing and browsing the repository by simply following the offered links. It has been shown throughout a number of projects at the University of Kaiserslautern that already a simple implementation of the repository, like ours, effectively supports the improvement cycle and the reuse cycle of reuse-driven SDL system development.

5 RELATED WORK

In our description of related work we focus on the two major parts of our approach: the improvement process and the reuse repository. First, we discuss the existing approaches for developing reusable SDL artifacts. We also take a short look at how to discover conventional design patterns. Then, we compare our repository with other descriptions of reuse repositories.

5.1 Developing Reusable (SDL) Artifacts

In [18] 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) [23] is a methodology that supports the definition and reuse of SDL frameworks. Given a system specification with an infrastructure that seems to be common for a family of systems, TIMe advocates the redesign of that system into an SDL framework. It is defined how 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 do they discuss the administration of a reuse repository.

The ***BACKDOOR*** approach [21] 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 a 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. Compared to our approach, pattern improvement is not considered. Furthermore, we came to a different decision regarding the separation of pattern development 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 [13] 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.

The ***Experience Factory (EF)*** approach by Basili et. al. [2] describes an organizational approach for building up software competencies in a company and transferring them systematically into projects. As a main part of the approach, a repository that is called ***Experience Base*** stores all relevant knowledge and offers it to new projects of the organization on demand. The idea of comprehensive reuse of all kinds of software engineering experience that is manifested by the experience base is also realized in our approach. In contrast to Basili, our reuse repository includes the single project databases in the form of project documentations in the project area. Nevertheless, our repository structure can be seen as a specialized instantiation of an experience base tailored to SDL system development. However, the main concern of an experience factory is the establishment of a global organizational improvement process, while our approach places the focus more on the concrete improvement and completion of single experience elements like SDL patterns and SDL frameworks.

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 approach for the continuous improvement of reuse-driven SDL system development. The approach directly exploits practical reuse experience and applies a measurement program for systematic identification of possible improvements. To transfer reuse experience between development projects, a specific reuse repository is defined that stores and interrelates experience elements that go far beyond the reusable SDL artifacts defined in Section 2. The approach offers several advantages compared to ad hoc learning cycles:

- Incremental investment in reuse that is triggered by software developers on request.
- Quality increase, as good artifacts depend on practical and long-term experience and several engineers contribute to quality improvement over time.
- Adaptation of the reuse repository to an application domain, i.e., domain-specific artifacts are developed from universal artifacts contained in a basic pool.
- An evaluation of reuse benefits can easily be integrated into the measurement program.

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. We plan to provide part of the SDL pattern pool on the Internet in order to get broader feedback for our improvement cycle from the SDL community.

ACKNOWLEDGMENTS

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, H. D. Rombach. Experience Factory. in J.J. Marciniak (ed), *Encyclopedia of Software Engineering, Volume 1*, John Wiley & Sons, 1994, 469-476.

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

[4] A. Birk, C. Tautz. Knowledge management of software engineering lessons learned. in *Proc. of the 10th Conference on Software Engineering and Knowledge Engineering (SEKE'98)*. (San Francisco, CA, June 1998), Knowledge Systems Institute, Skokie, Illinois, USA, 116–119.

[5] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language, Version 1.1., Rational Software Corporation, 1997.

[6] R. Bræk and O. Haugen. Engineering Real-time Systems: An Object-oriented language Methology using SDL. Prentice Hall, London, 1993.

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

[8] R. L. Feldmann, B. Geppert, 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/>.

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

[10] B. Geppert, R. Gotzhein, F. Rößler. Configuring Communication Protocols Using SDL Patterns, in: A. Cavalli, A. Sarma (Eds.): *SDL'97 - Time for Testing · SDL, MSC and Trends, Proc. of the 8th SDL Forum*, France, 1997.

[11] B. Geppert, F. Rößler. Pattern-based Configuring of a Customized Resource Reservation Protocol with SDL. *SFB 501 Report 19/96*, Computer Science Department, University of Kaiserslautern, Germany, 1996, On-line at <http://www.sfb501.uni-kl.de/sfb/reports/>.

[12] B. Geppert, 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.

[13] S. Henninger. Supporting the Construction and Evolution of Component Repositories. in *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.

[14] T. Ihme. An SDL Framework for X-ray Spectrometer Software. in *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (SAM'98)*, Berlin, Germany, 1998, 125-134.

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

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

[17] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, 1995.

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

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

[20] SDL Forum Society - Tools Web Site, On-line at <http://www.sdl-forum.org/Tools/index.htm>.

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

[22] Telelogic. SDT 3.3 Methodology Guidelines – Part1: The SOMT Method. Telelogic, Sweden, 1998.

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