Combining SDL Patterns with Continuous Quality Improvement: An Experience Factory Tailored to SDL Patterns [1]

Birgit Geppert, Frank Rößler
Computer Networks Group
Computer Science Department,
University of Kaiserslautern
{geppert, roessler}@informatik.uni-kl.de

Raimund L. Feldmann, Stefan Vorwieger
Software Engineering Group
Computer Science Department,
University of Kaiserslautern
{r.feldmann, s.vorwieger}@computer.org

Abstract

An SDL pattern is a reusable software artifact representing a generic solution for a recurring design problem. It is required that SDL be the applied design language. However, just offering an SDL pattern pool to the system designer will not result in the expected benefits of software reuse, such as shorter development cycles, improved quality, or easier maintenance of the resulting products. Advanced SDL methodologies are also needed, which are tailored to support the reuse of SDL artifacts.

To guide the application of SDL patterns during system design, an incremental configuration process is defined. However, software reuse can be characterized as a particularly dynamic concept with steady improvements of artifacts and processes as experience grows. We therefore further develop the SDL pattern approach into an SDL Experience Factory that supports evaluation and continuous improvement, while the approach is applied in projects. As its main component, the SDL Experience Factory contains a central repository for different kinds of expert knowledge.

Keywords

SDL patterns, continuous quality improvement, Experience Factory, SDL methodology, software reuse

1 INTRODUCTION

The reuse of pre-designed solutions for recurring design problems is of major concern in object-oriented software development. During the past few years, design patterns have emerged as a particularly fruitful approach to software reuse [7, 15]. Contrary to the traditional paradigm of code reuse in the sense of class and function libraries, design patterns favor high-level reuse of architecture and design and focus on the invariant parts of a design solution. Thus they offer by far more flexibility for adaptation to the embedding context and substantially increase the potential of reuse.

In [16, 18, 19] we present the SDL pattern approach that integrates the design pattern concept with SDL [21]. Generally speaking, SDL patterns describe generic solutions for recurring design problems, which can be customized for a particular context. While conventional design patterns are specified independently from a possible design language, it is assumed that the target language for SDL pattern instantiation is SDL. Thereby we benefit from the formal basis provided by SDL, so that SDL patterns are actually characterized as formalized design patterns. Instead of specifying and applying the patterns rather informally, a formal target language such as SDL offers the possibility of precisely specifying how to apply a specific pattern, under which assumptions this will be allowed, and what properties result for the embedding context. This is a major improvement compared to conventional design patterns, which mainly rely on natural language-based pattern description and, to a large degree, must still leave pattern application to the personal skills of the system designer. We also consider formalization to be a prerequisite for increased correctness of resulting products [26] and tool support [9]. However, we do not deal with formalizing design patterns in general. Instead of formalizing reuse concepts, we aim to support reusability within the formal methods area.

SDL patterns are expected to offer the same advantages as those commonly attributed to conventional design pattern, namely, pattern capture solutions, which have evolved over time and serve as an elegant way to make de-

[1] This work is supported by the German Science Foundation (DFG) as part of the Sonderforschungsbereich SFB 501 *Development of large systems with generic methods*

signs more flexible, modular, reusable, and understandable. They reflect experiences gained in prior developments and therefore help designers reuse successful designs and architectures. As a consequence, the design process becomes faster and the number of design errors decreases. However, such statements often seem to be subjective in nature. That is, they characterize hypotheses that must be validated. In [12] we present an approach for the experimental evaluation of empirical properties of SDL patterns. This approach is based on the Quality Improvement Paradigm (QIP) [1] and the Experience Factory (EF) approach [2], and serves two purposes: first, the expected properties of SDL patterns that involve human interaction can be validated and second, one gets detailed information for guiding stepwise improvement of the SDL pattern approach. Every project (i.e., case studies and controlled experiments) is carefully planned, executed, and analyzed. Goal-oriented measurement plans according to the Goal Question Metric paradigm (GQM) [3] are defined in the planning phase of each project. The data that is collected according to our measurement plans during the execution phase of a project is finally analyzed and used to prove or reject our hypotheses concerning SDL patterns. Furthermore, the results are used for continuous improvement. To support our approach, a central reuse repository is used where project data, process models, SDL patterns, measurement data, lessons learned, or other experiences concerning SDL patterns are collected: the so-called SDL Experience Base [2][2]. With the help of our Experience Base, tailored to SDL patterns, we transfer the gained knowledge into new projects and experiments to allow continuous improvement of the SDL pattern approach. The SDL Experience Base, together with the Experience Factory approach, enriches existing SDL methodologies with two promising paradigms of modern software engineering: design patterns and continuous quality improvement. Note, however, that we do not aim to substitute existing SDL methodologies. Rather, we integrate smoothly by adding concepts for reuse and continuous improvement of reuse procedures.

The remainder of the paper is organized as follows: Section 2 discusses software reuse with SDL in general and introduces the SDL pattern approach. In Section 3 we develop the approach into an SDL Experience Factory that supports evaluation and continuous improvement of SDL patterns and accompanying processes. We summarize the results in Section 4.

2 SDL PATTERNS AND METHODOLOGY

In the following we summarize reuse concepts already supported by SDL and motivate the idea of SDL patterns. Subsequently, we introduce SDL patterns and an accompanying configuration process that guides the application of SDL patterns during system design.

2.1 Software reuse with SDL

Benefits of software reuse are, for example, reduction of development effort and therefore shorter development cycles, as well as better quality and easier maintenance of the resulting products. Reuse can already be performed within a single project by using procedures, macros, or class inheritance. More interesting, however, is the reuse of software artifacts between several projects by providing some kind of software artifact repository. There are several kinds of reusable artifacts which differ in their abstraction level and flexibility.

Component libraries are concerned with code reuse and offer only little flexibility for tailoring them to a specific problem. As a consequence, the developer does not need to go to much expense before being able to reuse the component. Library components have well-defined interfaces and offer black-box reuse, i.e., reuse without the need to understand how the components are realized. However, they are rather unflexible: "applications seem infinitely variable, and no matter how good a component library is, it will eventually need new components“ [23].

Different from component libraries, frameworks are concerned with reuse of architecture and design. A framework defines a domain-specific architecture represented as a "reusable, semi-complete application that can be specialized to produce custom applications“ [10]. Therefore, an essential property for frameworks is their extensibility to ensure timely customization. Frameworks are often combined with design patterns which enable documentation of frameworks 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. Design reuse is more flexible, but the learning curve required before a framework or design pattern can be reused could be very high. „Most design reuse is informal and happens through using experienced developers“ [23].

It is generally accepted that just offering a software repository to the engineers will not result in the above-mentioned benefits of software reuse. Additionally, advanced methodologies are needed which are tailored for supporting the reuse of these artifacts. That is, specialized process models, guidelines for applying the artifacts, lessons learned as well as experiences from prior projects are to be provided to the development team.

Some of the above-mentioned reuse concepts are also supported by SDL:

- Macros:
  If a section of an SDL specification appears in several places, it can be defined once as an SDL macro and called

[2] In [2] the concept of an Experience Base is much more general. We restrict the ideas to the SDL patterns context and therefore call it SDL Experience Base.

wherever it is needed. Before interpretation of the system the macro calls are replaced by a copy of the corresponding macro definition. A macro can be parameterized and is visible to the whole system, no matter at which level it is defined. A macro definition may call other macros, but a recursive call (directly or indirectly) is not allowed.

- Object-orientation:
  SDL allows the parameterized type definition of blocks, processes, services, procedures, signals, and data. The types can be specialized by redefining virtual types or transitions, adding attributes (e.g., gates, signals), or by bounding formal context parameters (e.g., signal context parameter). Not supported by SDL are multiple inheritance and dynamic binding. Defining a type which is instantiated several times and which could also be specialized to define a subtype, enables reuse within one system. Parameterization makes a type 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 exactly match the signals of the process type.
- Packages:
  In order to support reuse between several projects, SDL offers the possibility of collecting type definitions in libraries. These libraries are called packages in SDL. The type definitions of a package are made visible to an SDL specification by a special use-clause. Until SDL-92, this was only allowed at system level, so that the type definitions of a used package were visible to the whole system. Since SDL-96, a package can also be used at a lower level, such as block or process diagrams. Packages can themselves use other packages. The concept of parameterized types (see above) is most important for packages in order to make types independent from the context in which they are to be instantiated.

It turns out that SDL only supports code reuse. More flexible reuse concepts would be gained by combining SDL with the idea of frameworks or design patterns. There are already some projects dealing with SDL frameworks (see, e.g., [6, 28]). A framework comprises an SDL system type with several virtual block, process, and service types as well as virtual procedures. Furthermore, documentation is prepared for the framework which describes three main aspects: the purpose of the framework, how to instantiate it, and its detailed design. The framework is instantiated by specialization and adaptation. In [16, 18, 19] we describe how to combine SDL with the design pattern concept. The approach is discussed in more detail in Section 2.2.

There are several SDL methodologies, but only some of them are tailored to support the reuse of SDL artifacts. For instance, a key issue of the SDL methodology framework presented in [25] is the reuse library, an archive where relevant documents are placed for later reuse. However, what kind of documents shall be stored or how they are reused is not prescribed. This has to be defined when instantiating the methodology framework for a certain development project.

2.2 SDL patterns

What are 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. 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 the applied design language*. SDL patterns - like other software patterns - serve three main purposes:

- First, patterns are descriptive in nature. That is, they describe design expertise and experiences gained in prior projects and allow to pass this knowledge on to other developers. A collection of patterns can therefore be seen as some kind of textbook.
- SDL patterns generate a design solution for a given problem. Therefore, the solution is described by an SDL-fragment that consists of several syntactical elements (e.g., transition fragments) which may be scattered over the context specification[3]. Additionally, application guidelines describe when the pattern could be applied and how to adapt the SDL fragment and compose it with the given context specification.
- Furthermore, patterns help to document a given design. The information about embedded pattern instances can be inserted during pattern-based design, i.e., when applying a pattern (top-down documentation). However, pattern information can also be inserted into a given SDL specification by pattern-based reverse engineering (bottom-up documentation). Therefore the SDL specification is scanned for pattern instances and enriched with suitable pattern comments.

[3]. Note that SDL-fragments differ considerably from SDL macros. For instance, an SDL-fragment includes several syntactical elements that are normally scattered over the context specification. Furthermore, embedded SDL pattern instances may overlap with the context specification (i.e., parts of the context are replaced when composing it with the pattern instance).

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. An example (from the area of communication systems) 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 *OMT or UML object model*. This defines the involved components (design elements) and their relations.

Message scenario. Example scenarios illustrating typical interactions between the involved objects (e.g., protocol entities, service users, service providers, protocol functions) are specified by using *MSC diagrams.*

SDL-fragment. The mere syntactical part of the design solution is defined by a generic *SDL-fragment*, which is adapted and syntactically embedded when applying the pattern. If more than one SDL **version** of the design solution is possible (realization as SDL service or procedure, interaction by message passing or shared variables, etc.), fragments for the most frequent versions are included. For each fragment, corresponding **syntactical embedding** rules are defined in terms of the SDL syntax:

- Rules for renaming of the generic identifiers of the SDL-fragment.
- Rules for composing the pattern instance with the embedding context. This could, for instance, result in the addition of new transitions or SDL services or in a refinement of existing types or transitions.

Note that an SDL-fragment is generally not a syntactically complete SDL specification but a fragment, which has to be adapted and composed with an embedding context. From a semantical point of view, SDL-fragments therefore only become formally meaningful when they are part of a complete specification.

Semantic properties. Properties of the resulting specification that are introduced by the embedded pattern. This also includes a description of assumptions under which these properties hold. Care has to be taken not to destroy a pattern's assumptions during further development steps, e.g., by manipulating the context specification. The semantic properties define the pattern's intent more precisely.

Refinement. An embedded pattern instance can be further refined, e.g., by the embedding of another pattern instance in subsequent development steps. Refinements compatible with the pattern's intent are specified.

Cooperative usage. Possible usage with other patterns of the pool is described. This is feasible and especially useful for a specific application domain as in our case. This distinguishes a pool of SDL patterns from a mere pattern catalogue where the patterns are unrelated or only loosely related.

Figure 1: SDL pattern description template

What do SDL patterns look like?

The specification of an SDL pattern is organized by a standard description template (Figure 1). Its main items are sketched in the following: the syntactical part of the design solution is defined by the generic *SDL-fragment*, which has to be instantiated and textually embedded into the context specification when applying the pattern. Instantiation and composition of SDL-fragments is prescribed in terms of *syntactical embedding rules*, which, e.g., guide the renaming of generic identifiers or specialization of embedding design elements. Usually pattern semantics is not completely captured by an SDL-fragment. Due to language constraints, this would otherwise result in an overspecification of the design solution and reduce the potential of reuse. Thus, additional *semantic properties* specify preconditions for pattern application as well as behavioral changes of the embedding context. Though semantic properties are currently stated in natural language, it is possible to express them precisely in a temporal logic. Also, restrictions on the *refinement* of pattern instances are specified in order to prevent a pattern's intent from being destroyed by subsequent development steps. A comparison to existing description templates for conventional design patterns is given in [16]. Figure 2 shows an instance of the description template.

What is the application domain of SDL patterns?

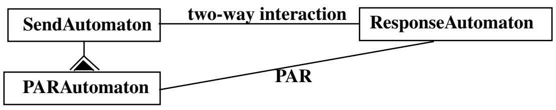
In general, the SDL pattern concept is not restricted to any application domain. This means that SDL patterns can be defined for any domain where SDL is applicable and can deal with system architecture or behavior description. In the SFB 501 project we are concerned with the generic engineering of communication software. Therefore, the current pool of SDL patterns contains building blocks from the domain of communication systems that deal, for instance, with the interaction behavior of distributed objects, error control (lost or duplicated messages, see Figure 2), lower layer interfacing, or dynamic creation of protocol entities [17]. Not included so far are SDL patterns dealing with system architecture. This is a matter of future work.

2.3 Configuration process

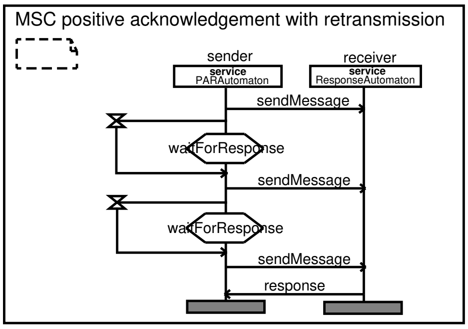
Along with the standard template for SDL pattern description, we have also defined a process model for the application of SDL patterns (Figure 3). The configuration process suggests an incremental design, where the whole set of requirements is first decomposed, i.e., partitioned and (where appropriate) simplified. Decomposition classifies as an analysis task that identifies separate protocol functionalities. Thereby it is possible to consider a protocol functionality under different assumptions. For instance, interaction sequences during connection establishment are less complex on top of a reliable basic service rather than an unreliable basic service. Experience has shown that protocol functionalities can often be specified one after the other and - in addition - be completed stepwise (e.g.,

Name: TimerControlledRepeat
Intent:
TimerControlledRepeat extends a two-way interaction between two automata *SendAutomaton* and *ResponseAutomaton* for the case of possible message losses during data transfer. If an expected response does not arrive before the expiry of a timer, the message is repeated (Positive Acknowledgement with Retransmission). This pattern does not deal with the problem of message disruption or duplication.
Motivation:
For a BlockingRequestReply pattern [17] instance the requester will deadlock, if the reliable transmission of the request or reply signal is not guaranteed. Therefore replies are observed by TimerControlledRepeat in case of unreliable basic service.

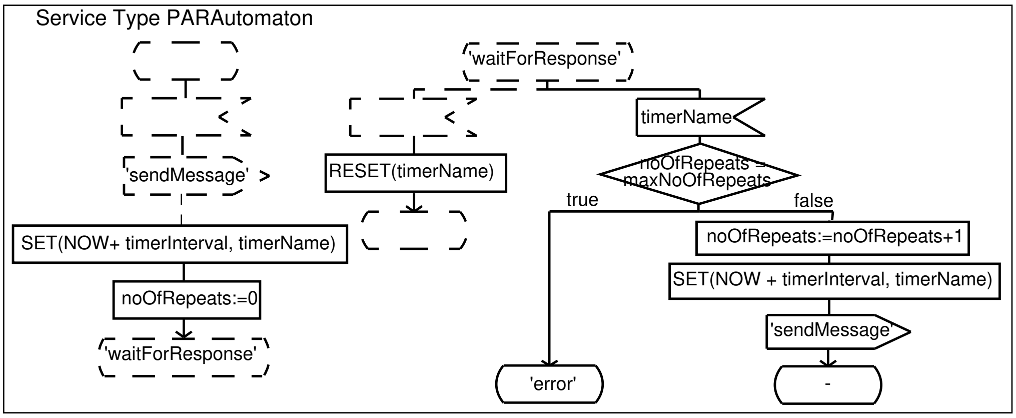
Structure:



Message Scenario:



SDL-fragment:



Semantic properties:
Property C.1.: *If the assumptions stated below hold, PARAutomaton will eventually receive a response from ResponseAutomaton after sending a sendMessage, or PARAutomaton will enter the error state after maxNoOfRepeats unsuccessful retransmissions. The assumptions are:*

- *The communication channel between* PARAutomaton *and* ResponseAutomaton *for transmission of* sendMessage *and corresponding response signals neither disrupts nor creates messages.*
- *The communication channel may lose messages but* timerInterval *is greater than the maximum round trip time of* sendMessage *and corresponding response.*
- ResponseAutomaton *reacts on duplicates the same way (from the perspective of* PARAutomaton*) as on the original* sendMessage.

Figure 2: Excerpt of an SDL pattern

adapted to the non-ideal properties of an underlying basic service) [20, 26]. This suggests that we perform an individual development step in order to incorporate an additional protocol functionality or relax a corresponding simplification.

A development step is further divided into an object-oriented analysis and the actual design of the current functionality. It is the design activity where SDL patterns actually come into place: starting point is the context SDL specification, i.e., the SDL design specification obtained from the previous development step[4]. Based on the new requirements, a proper set of SDL patterns is to be *selected*[5]. As SDL patterns represent generic design solutions, a selected pattern, respectively its corresponding SDL-fragment, has to be *adapted* to seamlessly fit the embedding (context) specification (e.g., generic signal names have to be set according to the pattern's syntactical embedding rules). Finally, the pattern instance is ready to be *composed* with the embedding specification. Application of patterns in the sense of selection, adaptation, and composition is specifically supported by certain items of the SDL pattern description template.

The result of this design activity is an intermediate SDL design specification, which is to be validated in the normal way. If any faults are discovered, a return to one of the previous development steps is needed (not shown in Figure 3). Otherwise the validated specification serves as the context specification for the next development step. If all simplifications are eliminated and all requirement subsets are implemented, the final design specification is given by the validated design specification of the last development step.

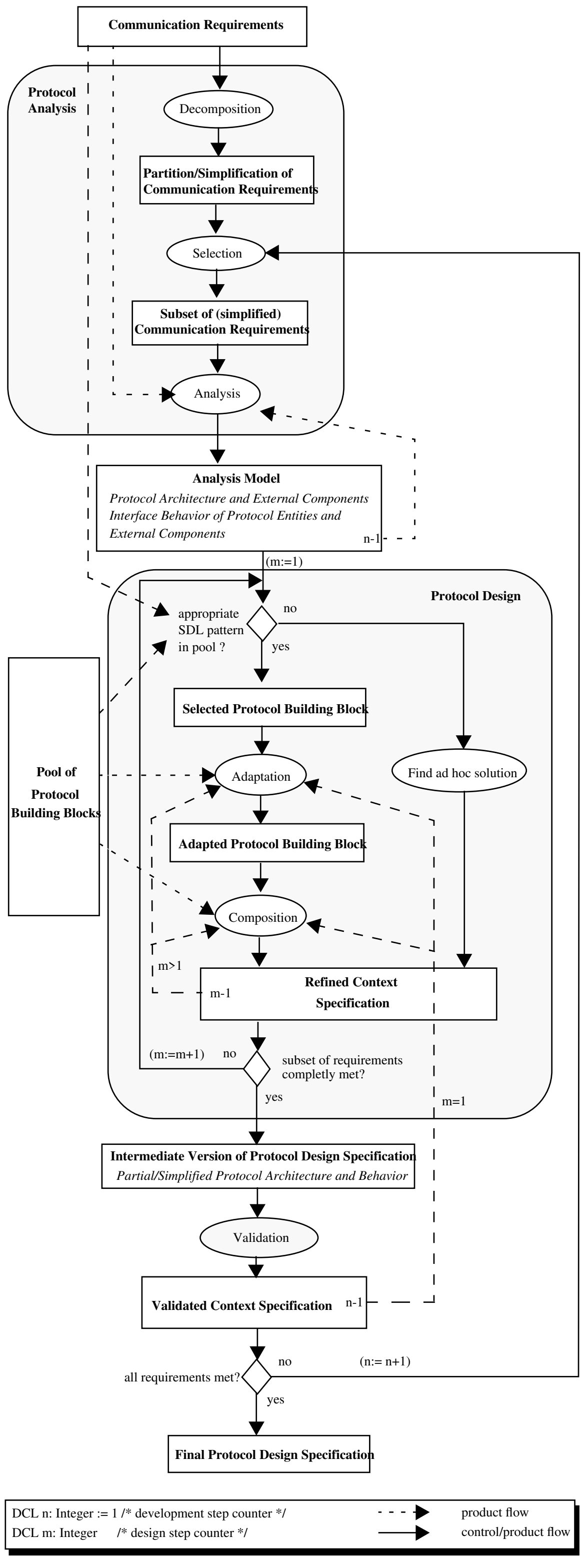


Figure 3: Process model (excerpt)

3 CONTINUOUS IMPROVEMENT OF THE SDL PATTERN APPROACH

To show the feasibility of the SDL-pattern approach, several test projects have been conducted. However, in order to validate empirical properties as mentioned in Section 1, a more systematic method is needed. Additionally, it is essential to have an infrastructure available that helps to continuously improve the concepts, as good patterns mainly arise from practical and well-founded experiences. Thus we combine SDL patterns with the Experience Factory approach.

3.1 Test projects

In order to show the feasibility of the SDL-pattern approach, we conducted several test projects. For instance, we have performed an SDL pattern-based re-engineering of most parts of the Internet Stream Protocol ST2+ [8], resulting in a systematic and formal SDL design specification [26]. Thereby a very large portion (almost

4. For the first development step, the initial context specification is either empty or given by an instantiated SDL framework.

5. Note that for some design problems the pool of pre-defined building blocks may not contain an adequate solution. This gives rise to the development of a new SDL pattern or an ad hoc solution.

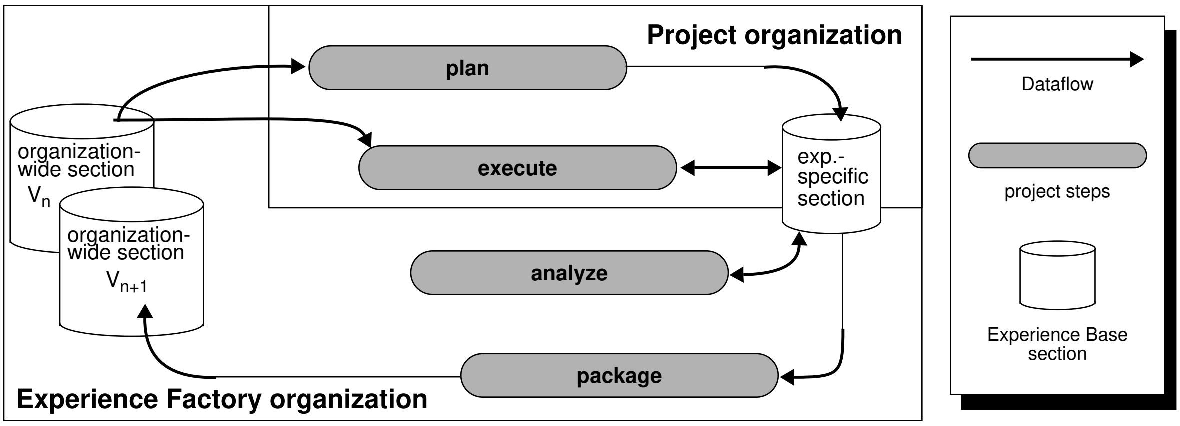


Figure 4: The Experience Factory

100% of the control structure) of the final ST2+ specification resulted from SDL patterns (10 different patterns were used and the original specification in natural language took about 70 pages). In particular, we demonstrated how SDL pattern-based design can boost the developer's confidence in the correctness of the resulting product [26]. Another test project was part of a more comprehensive project where a real-time communication subsystem was developed on top of a Controller Area Network (CAN) installation. In [20] we demonstrate how the protocols supporting user communication and certain management tasks were configured using SDL patterns. We also applied the SDT Cadvanced code generator to implement the resulting design specification on a PC cluster. Again, it turned out that SDL pattern-based configuring of communication protocols yields more systematic designs, i.e., readability and maintainability is improved and less design errors occur, since the design decisions are well founded and documented.

Test projects are an adequate means of demonstrating the feasibility of the approach. They also helped to develop an initial pool of SDL patterns. However, with the current status of the SDL pattern approach we wish to have a more systematic method to investigate certain details of the approach. Additionally, it would be helpful to record other kinds of experiences concerning SDL patterns (we currently only support informal feedback on the configuration process and the patterns themselves). To alleviate these problems, we are developing an SDL Experience Factory with a central repository for all kinds of SDL pattern-specific experience, the SDL Experience Base.

3.2 Logical structure of the SDL Experience Factory

The Experience Factory (EF) concept, as suggested by Basili et. al. [2], is a logical and/or physical organization that supports project development by a) analyzing and synthesizing all kind of experience, b) acting as a repository for such experience, and c) supplying that experience to future projects on demand. Every single project is consequently seen as an experiment from which the organization can gain new experience to improve its competence. Therefore, the Experience Factory organization completes the standard project organization, so that each project is conducted in four main steps: plan, execute, analyze, and package. While the planning and execution steps are mainly performed by the project organization, the Experience Factory organization is responsible for the analysis and packaging steps, i.e., to systematically analyze each conducted project in order to gain new experience or validate existing experience, and (re-)structure and store the results from the analysis step, so that they can serve as valuable input for future projects.

One main component of an EF is the so-called Experience Base (EB). An EB acts as a central repository for all kinds of experiences regarding software development, since not only the classical code or pattern reuse is supported. In our SDL Experience Factory we distinguish between an *organization-wide section* and an *experiment-specific section* of the EB. The organization-wide section stores experience common to several projects, e.g., the SDL pattern pool, while all information concerning single projects is kept in the experiment-specific section according to predefined templates. These templates are completed while the project is planned, executed, and analyzed, and serve as a basis for the last step, which packages the gained experience into the organization-wide section of the EB. Therefore, the experiment-specific section is similar to commonly known project databases. Figure 4 summarizes how our EB sections are involved in the conduction of the single project steps.

In our instantiation of an EF tailored to SDL patterns, we refine the organization-wide section of the EB into different *areas*. They help us structure the stored experience and can be seen as disjunct modules. An example for an area is the glossaries area which provides definitions for terms commonly used in all projects and experience elements stored in the EB. Just as modules can be refined into functions, areas can be further refined. For example, the glossaries area is split up into a GQM glossary defining terms concerning GQM-based [3] measurement activities, an SDL glossary with SDL-related definitions, and a general Software Engineering glossary providing definitions for terms like process, product, etc.. Besides the glossaries area, the following areas have been instantiated:

- **The component repositories area:**
  This area offers components that can be reused in different projects. For example, SDL patterns such as the TimerControlledRepeat pattern (Figure 2), or C++ code for checksum algorithms are stored here.
- **The process modeling area:**
  Process, product, and resource models, describing how to conduct a project or apply a technique, are offered in this area. For example, the configuration process for the application of SDL patterns (Section 2.3), the SOMT process model [29], or the Bræk and Haugen model for developing real-time systems with SDL [5] are provided.
- **The technologies area:**
  For different techniques, methods, and tools, so-called *technology packages* [24] are stored in this area. These packages contain basic information about the technologies and help to select the appropriate techniques when setting up a new project. The SDT (SDL Design Tool) package, for instance, helps newcomers get into the SDL development environment.
- **The measurement area:**
  Pre-defined GQM plans [3] that can be easily adapted to new projects, e.g., to empirically validate and improve the SDL pattern pool, or the configuration process are stored in this area.
- **The qualitative experiences area:**
  All lessons learned, i.e., experiences that were not planned to be made[6], but turned out to be useful, from the planning, execution, and analysis steps of projects, are represented in this area. They are categorized according to the topics they deal with. Currently we deposit experiences about adequate decomposition of communication requirements in this area (Section 2.3).
- **The literature area:**
  Background knowledge in the form of (external) references, on-line documents, and contact addresses is provided within this area. For instance, a reference to the SDL forum society web page[7], or relevant papers dealing with communication protocols can be found here.

The described areas help find experience elements of a concrete type within the organization-wide section of the EB. Additionally, different *relations* have been defined between the areas and the experiment-specific section to support the search for experience elements in a given project context. The areas, sections, and predefined relations together form a framework that represents the logical structure of our EB instantiation. It is described in detail in [12]. A discussion of the technical realization of the EB using HTML-pages that are accessible via the SFB 501 intra-net can be found in [14].

3.3 Process model for experimental evaluation and improvement

After we have discussed the basic structure of our Experience Factory, we will now describe how projects using SDL patterns are conducted in this infrastructure and how the experimental evaluation and improvement is supported. Therefore, for each of the four project steps - plan, execute, analyze, and package - we list which processes have to be conducted, and which products are produced, extracted from, and inserted into the EB sections. A more general discussion on how an EB supports systematic reuse in projects can be found in [13].

Step 1: Planning the project
As in common project environments, the new project first has to be characterized by means of "*What are the deliverables of the project?*", "*In which environment is the project to be conducted?*", and "*Are there time restrictions for the project?*". This characterization that serves as a basis for all planning activities is recorded and stored in a new project entry that is created in the experiment-specific section of the EB. In accordance with some pre-defined template (see [11]), all documents produced for the project will be stored in this entry. The project characterization is used to search the organization-wide section of the EB for similar project plans and/or process descriptions suitable for the new project (remember, in an Experience Factory reuse is extended to all kind of project experience). Based on the retrieved information, the project plan for the new project is defined and stored in the project entry of the experiment-specific section of the EB. For instance, the process model for the application of SDL patterns (see Section 2.3) that is stored in the organization-wide section of the EB usually serves as a basis for the new project plan. Information about the resources, i.e., people and tools, that will be needed for the project, and milestones of the project will be added. Technology packages [24], from the technology area of the organization-wide section of the EB, can help to select the appropriate technologies for the project.

Since every project is seen as an experiment that helps to systematically improve and validate the used technologies, project plans, SDL patterns, etc., any goals regarding the validation of hypotheses that should be examined

6. Experience which is expected to be collected is captured with the help of GQM-based measurement programs that are defined in the planning step of a project.

7. http://www.sdl-forum.org/

in the project must be formulated in a quantitative manner. Therefore, metrics are defined with the help of GQM plans [2], and data collection forms, e.g., in the form of questionnaires, are created. This task is supported by some GQM plans, e.g., regarding the quality of SDL patterns, that are stored in the measurement area of the organization-wide section of the EB. All documents for the projects measurement program are saved in the project entry.

Finally, it must be checked if all required resources are ready to be used in the project. This includes the setup and installation of the needed tools and maybe the preparation of training of the people with regard to the technologies used and the development process of the project. Technical help for the installation and usage of tools is again provided by the technology packages of the technology area, or by literature references from the literature area of the organization-wide section of the EB. Reading those lessons learned from past projects that deal with technologies can help to avoid problems in the new project.

Step 2: Executing the project

In this step, the project is executed according to the project plan that is stored in the project entry. For the development process, the process model for the application of SDL patterns (Figure 3), as part of the project plan, is used. Developers extract and reuse SDL patterns and/or code fragments from the component repository area of the organization-wide section of the EB, if they are suitable for the problem that is to be solved. With the help of the questionnaires from the project entry, measurement data is collected and stored in the project entry. Unexpected problems and/or new solutions that occur in the project are recorded by the developers and are added to the project entry, to be analyzed in later project steps and maybe packaged for future reuse in other projects. Finally, the developed documents, i.e., SDL specifications, code, requirements descriptions, etc., are stored in the project entry.

Step3: Analyzing the project

The measurement data that has been collected and stored in the project entry during the execution of the project is now processed and analyzed. With its help the questions concerning the measurement program goals are answered, i.e., the hypotheses that were formulated at the beginning of the project are tested to see if they have been validated or must be rejected. The results are recorded and stored in the project entry. This is usually done by members of the Experience Factory organization. Additional reported experiences, e.g., about problems and new solutions concerning the usage of tools or SDL patterns, are completed and carefully judged to see if they were project-specific, or if they can be of relevance for future projects. Therefore, the people from the development team and project management team are interviewed. Again, the results are stored in the project entry.

Step 4: Package the project experience

From the experiences in the project entry of the experiment-specific section, lessons learned are formulated and stored in the organization-wide section of the EB if they are of interest for future projects. Both the analyzed measurement data and the captured lessons learned can then be used to systematically improve and/or adapt the technologies (e.g., tools), process models (e.g., the SDL-pattern process), and components (e.g., the SDL patterns) that were used in the actual project, for similar future projects. Even if no problems occurred and everything worked out fine, the results are useful for future projects: this is because the used processes and products can be more trusted since they have been successfully tested in practice and therefore can guarantee a minimum of quality assurance in a project if they are selected for reuse. So whenever a future project uses the organization-wide section of the EB to support its planning and execution step, the experience gained in the actual project is systematically transferred into the new projects. And when each future project is also seen as an experiment which is used to gain experience and therefore, is conducted according to the four steps: plan, execute, analyze, and package, the cycle starts again and continuous improvement is established.

4 CONCLUSION

We have introduced the SDL-pattern approach which integrates the well-known design pattern concept with the formal design language SDL. As a major advantage, SDL patterns allow to precisely specify knowledge about pattern application and its impact on the embedding context. SDL patterns focus on the reuse of architecture and design as opposed to the reuse concepts that are already supported by SDL. To show the feasibility of the approach, several test projects were conducted. However, with the current status of the approach, we wish to have a more systematic method to investigate certain details concerning SDL patterns. Additionally, it is essential to have an infrastructure available that helps to continuously improve the concepts, as good patterns mainly arise from practical and well-founded experiences. For this purpose we combined the SDL pattern approach with the Experience Factory approach. The SDL Experience Factory contains a central reuse repository for all kinds of SDL pattern-specific experiences and allows us to effectively set up new projects with a corresponding measurement program. Knowledge is systematically packaged and transferred into new projects, so that the SDL pattern approach can be continuously improved. A model for the SDL Experience Factory was introduced where the main activities are planning, executing, and analyzing the project as well as packaging the project experiences.

Initial experience with the SDL Experience Factory [12] has shown that it is a valuable means for evaluation and continuous improvement, while the approach is applied in real projects. Thus we are currently planning a re-engineering of two former test projects within this context.

Acknowledgements
Our gratitude is extended to our project leaders Prof. Dr. Reinhard Gotzhein and Prof. Dr. H. Dieter Rombach. Additionally, the assistance provided by Sonnhild Namingha from the Fraunhofer Institute for Experimental Software Engineering (IESE) in reviewing early versions of this paper is much appreciated.

5 REFERENCES

[1] V. R. Basili and H. D. Rombach, *The TAME Project: Towards improvement–oriented software environments*, IEEE Transactions on Software Engineering, SE-14(6):758–773, June 1988.
[2] V. R. Basili, G. Caldiera, and H. D. Rombach, *Experience Factory*, In John J. Marciniak, editor, Encyclopedia of Software Engineering, volume 1, pages 469–476, John Wiley & Sons, 1994.
[3] V. R. Basili, G. Caldiera, and H. D. Rombach, *Goal Question Metric Paradigm*, In John J. Marciniak, editor, Encyclopedia of Software Engineering, volume 1, pages 528–532, John Wiley & Sons, 1994.
[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language, Version 1.0.* Rational Software Corporation, 1997
[5] R. Bræk and Ø. Haugen. *Engineering Real-time Systems: An Object-oriented language Methology using SDL.* Prentice Hall, London, 1993.
[6] R. Bræk, Ø. Haugen, G. Melby, B. Møller-Pedersen, R. Sanders, and T. Stålhane, *TIMe - The Integrated Method.* SINTEF, TIMe Report, 1997
[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley & Sons, 1996
[8] L. Delgrossi and L. Berger (Ed.), *Internet Stream Protocol Version 2 (ST2), Protocol Specification - Version ST2+*, RFC 1819, 1995
[9] D. Cisowski, B. Geppert, F. Rößler, and M. Schwaiger, *Tool Support for SDL Patterns* (this volume)
[10] M. E. Fayad and D. C. Schmidt, *Object-Oriented Application Frameworks*, Communication of the ACM, Volume 40, Number 10, Oct. 1997
[11] M. Fechtig. *Fixing the case studies' structure for the access and storage system of the experiment-specific section in the SFB 501 Experience Base* (in German). Projektarbeit, Dept. of Computer Science, University of Kaiserslautern, Germany, Jan. 1998.
[12] 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
[13] R. L. Feldmann, J. Münch, and S. Vorwieger, *Towards Goal-Oriented Organizational Learning: Representing and Maintaining Knowledge in an Experience Base*, In Proceedings of the Tenth International Conference on Software Engineering and Knowledge Engineering (SEKE'98), San Francisco, USA, June 1998
[14] R. L. Feldmann and S. Vorwieger, *Providing an Experience Base in a research Context via the Internet*, In Proceedings of the ICSE 98 Workshop on "Software Engineering over the Internet", Kyoto, Japan, April 1998
[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
[16] B. Geppert, R. Gotzhein, and F. Rößler, *Configuring Communication Protocols Using SDL Patterns*, in: A. Cavalli, A. Sarma (Eds.): SDL'97 - Time for Testing · SDL, MSC and Trends, Proceedings of the 8th SDL Forum, France, 1997
[17] B. Geppert and 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
[18] B. Geppert and F. Rößler, *Combining SDL and Pattern-based Design for the Customization of Communication Subsystems*, in: A. Wolisz, I. Schieferdecker, A. Rennoch (Eds.): Formale Beschreibungstechniken für verteilte Systeme, GMD-Studien No. 315, GI/ITG-Fachgespräch, ISBN 3-88457-315-2, 1997
[19] B. Geppert and F. Rößler, *Generic Engineering of Communication Protocols - Current Experience and Future Issues*, Proceedings of the 1st IEEE International Conference on Formal Engineering Methods, ICFEM'97, Hiroshima, Japan, 1997
[20] B. Geppert, F.Rößler, and M. Schneider, *Using SDL Patterns for the Design of a Communication Subsystem for CAN*, accepted for GI/ITG-Fachgespräch: Formale Beschreibungstechniken für verteilte Systeme, Cottbus, Germany, 1998
[21] ITU–T. Recommendation Z.100 (03/93) – *CCITT Specification and Description Language (SDL)*, 1994
[22] ITU–T. Recommendation Z.120 (10/96) – *Message Sequence Chart (MSC)*, 1996
[23] R. E. Johnson, *Frameworks = (Components + Patterns)*, Communication of the ACM, Volume 40, No. 10, Oct. 1997
[24] F. Kollnischko, S. Vorwieger, M. Ciolkowski, S. Haubrichs, D. Muthig, *Online-Support For Techniques And Tools In An Software Engineering Lab* (in German), SFB 501 Report 06/97, Computer Science Department, University of Kaiserslautern, Germany, 1997
[25] R. Reed, *Methodology for Real Time Systems*, Computer Networks and ISDN Systems 28 (1996)
[26] F. Rößler, B. Geppert, and Ph. Schaible, *Re-Engineering of the Internet Stream Protocol ST2+ with Formalized Design Patterns*, accepted for the 5th IEEE International Conference on Software Reuse, ICSR'98, Victoria, Canada, 1998
[27] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991
[28] *STEPS - The SDL Template for Protocol Stacks*, S&P Media, 1998
[29] Telelogic. *SDT 3.3 Methodology Guidelines – Part1: The SOMT Method.* Telelogic, Sweden, 1998