Tool Support for SDL Patterns[1]

Daniel Cisowski, Birgit Geppert, Frank Rößler, Markus Schwaiger
Computer Networks Group
Computer Science Department
University of Kaiserslautern
P.O. Box 3049, D-67653 Kaiserslautern, Germany
{cisowski, geppert, roessler, schwaige}@informatik.uni-kl.de

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. Thereby we benefit from the formal basis provided by SDL. 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.

In the paper we discuss aspects of tool support for SDL-pattern application. In particular, we describe requirements, design decisions, and some implementation issues of a prototype SDL-pattern editor that is currently under development. The SDL-pattern editor mainly supports syntactical application and documentation of SDL patterns.

Keywords

SDL patterns, tool support, SDL methodology, software reuse, formal methods

1 INTRODUCTION

In [3] [5] [6] we present the SDL pattern approach[2] that integrates the well-known design pattern concept with SDL. Generally speaking, SDL patterns describe generic solutions for recurring design problems, which can be customized for a particular context. Contrary to the traditional reuse paradigm of class and function libraries, SDL patterns focus on the invariant parts of a design solution and therefore offer by far more flexibility for adaptation to the embedding context. That is, potential of reuse is substantially increased.

While conventional design patterns [1] [2] 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 still have to leave pattern application to a large degree to the personal skills of the system designer.

The specification of an SDL pattern is organized by a standard description template. The items most relevant to tool support are sketched in the following: the syntactical part of the design solution is defined by a generic *SDL-fragment*, which has to be adapted and textually embedded into the context specification when applying the pattern. An SDL-fragment may consist of several elementary units (called *elements*). After adaptation a pattern element is comparable to an SDL macro definition. Pattern elements that are to be embedded within the same scope unit of the context specification are called *associated elements*. Adaptation and composition of SDL-fragments is prescribed in terms of *syntactical embedding rules*, which, e.g., guide renaming of generic identifiers or specialization of embedding design elements. Usually, pattern semantics is not completely captured by an SDL-fragment. Thus,

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.*

2. With a major case study presented in [12].

additional *semantic properties* are included, specifying preconditions for pattern application as well as behavioral changes of the embedding context. 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.

Along with the standard template for SDL pattern description, we have defined an incremental configuration process for the application of SDL patterns. It is the design activity of the process model where SDL patterns actually come into place: starting point is a context SDL specification, obtained from the previous development step. In order to meet new requirements a number of SDL patterns are *selected* and then applied to the context specification[3]. The selection of an SDL pattern is supported by several items of the SDL pattern description template, namely *intent*, *motivation*, *structure*, *message scenario*, *semantic properties* and *cooperative usage* (some of these items are not mentioned above). As patterns represent generic design solutions, the corresponding SDL-fragment has to be *adapted* in order to seamlessly fit the embedding context. This is instructed by the *renaming* parts of the *syntactical embedding rules*. The resulting pattern instance finally is *composed* with the embedding context, which is prescribed by the *composition* part of the syntactical embedding rules and also by *refinement* rules of embedding pattern instances. The result of a design activity is an executable, intermediate SDL design specification, which subsequently is validated in the normal way and serves as the context specification for the next development step.

In the paper we discuss aspects of tool support for SDL-pattern application. In particular, we describe requirements, design decisions, and some implementation issues of a prototype SDL-pattern editor that is currently under development. The SDL-pattern editor mainly supports syntactical application and documentation of SDL patterns.

The remainder of the paper is organized as follows: Section 2 gives a general idea of tool support for SDL patterns, while concrete requirements for an SDL-pattern editor are derived in Section 3. Design decisions and some implementation issues of the SDL-pattern editor are discussed in Section 4. We summarize the results in Section 5.

2 POTENTIAL FOR TOOL SUPPORT

As already stated, formalization of SDL patterns enables validation of pattern application. Furthermore, we consider formalization to be a prerequisite for tool support in general. Here were are in line with [9, 11] where tool support for conventional design patterns is discussed. These papers clearly reveal the problems when formalization of patterns is missing. In the following we briefly discuss the potential for tool support that SDL patterns provide on the whole. In the next section we then derive the requirements for a prototype SDL-pattern editor that is currently under development.

According to an imaginary life-cycle of an SDL pattern, we identify several activities where tool support is possible.

- **Pattern development:** for the pattern engineer (the person, who develops a new pattern) it is convenient to have some kind of repository available where SDL patterns are placed. It is necessary that this repository can generate different views of a pattern, according to the different roles (pattern engineer, system designer, pattern tools, pattern novices, ...) that agents have when working with SDL patterns. Of course information must not be stored several times. An adequate meta language for specification of SDL patterns is necessary to fulfil these requirements. As mentioned above, SDL pattern specifications follow a certain description template, in which each item applies a standard description language such as MSC, UML, temporal logic, SDL, or natural language.

- **Pattern application:** application of an SDL pattern is characterized by the more syntactical subtasks of adaptation and composition as well as semantic issues that determine pattern selection and validation of pattern application.

  - Pattern **selection** can be supported by a tool, which allows the user to browse through an existing SDL pattern pool and search for patterns that match certain criteria. Here, the pattern's intent and motivation are of special interest to get a quick overview of its field of application.
  - Assistance in the renaming of generic identifiers (e.g., signals, states, variables) can, e.g., be provided by preparing lists of identifiers from the embedding context. This will help to avoid incorrect **adaptation** of a pattern. During **composition** of the pattern instance with the embedding context, new SDL constructs are added, while parts of the given context are replaced by the inserted pattern instance. In order to ensure the intended behaviour, embedding rules that are precisely specified in terms of the SDL syntax can be observed automatically.
  - **Validation** of pattern application has to ensure that the pattern's intent (as defined by the pattern engineer) is met in the given context. Partially, this is achieved by an accurate syntactical application as mentioned

[3] Note that for some design problems the pool of predefined building blocks may not contain an adequate solution. This gives rise to the development of a new SDL pattern or an ad hoc solution.

above. In addition an SDL pattern is also characterized by semantic properties that state certain assumptions on the embedding context. These assumptions can be expressed in a temporal logic and therefore be checked by suitable verification tools (e.g., a model checker).

  - When rework is necessary after a review or during maintenance or re-engineering of a specification, tool support for a convenient **deletion** of embedded SDL patterns is also conceivable.

- **Pattern documentation:** an important advantage, commonly attributed to pattern-based design, is the implicit documentation of the resulting product. That is, with the necessary pattern knowledge available, it is possible to understand whole parts of the specification at once. Tool support can additionally improve readability of a specification, if convenient ways are provided to navigate and browse through the design and change the appearance of the SDL code currently inspected (e.g., to change the colour of a certain pattern in order to emphasize its occurrence). Also guided design transformations such as collapsing of pattern instances are possible.

- **Code generation:** it has often been stated that automatic code generation from formal specifications is not efficient enough for real-life applications. Here, additional semantic information that is given by the embedded SDL patterns can help compilers to generate more optimized code.

- **Quality improvement of SDL patterns and configuration process:** usually reusable artifacts and reuse procedures improve as experience grows. In [7] it is shown how evaluation and continuous improvement of the SDL pattern approach can be achieved, while the approach is applied in real projects.

3 FEATURES OF AN SDL-PATTERN EDITOR

Currently, we are developing a prototype **SDL-p**att**e**rn **edi**tor (called SPEEDI) that covers parts of the functionalities mentioned in Section 2. In particular, pattern documentation, adaptation and composition are supported. In the following, we focus on the requirements, while further details on design decisions and implementation issues are given in Section 4.

3.1 Pattern documentation

A pattern-based SDL specification gains an inherent structure that can significantly increase intelligibility. However, when reviewing, maintaining, or re-engineering the specification an embedded SDL pattern is not always easy to recognize. Consider, e.g., that SDL-fragments often contain several elements, which are embedded in scattered places of the context specification. As a consequence, one would have to start an inconvenient search for corresponding pattern elements in order to understand the pattern application (especially its embedding context). Due to the additional overhead, detailed comments from the system designer, that may help to navigate through the specification, are also not an adequate solution. Rather, we expect SPEEDI to reveal pattern locations automatically.

We suggest a multi-stage approach to discover embedded SDL patterns and navigate through the design. First of all SPEEDI should always display the names of the patterns the current cursor position belongs to. The list of patterns must be updated automatically, when the cursor moves. Note that elements from different pattern instances may overlap, so that pattern instances cannot be uniquely selected by cursor positioning alone. Nevertheless, this feature provides an overview of the locally involved pattern instances. For further investigation of an individual pattern instance an item must be selected from the list and subsequently, the complete pattern instance (including the scattered parts) will automatically be highlighted in the specification. That is, SPEEDI scans the specification for corresponding pattern elements and colors the identified pattern instance for easy recognition. The applied color should be user-specified. As already mentioned, pattern elements may be scattered all over the context specification and thus it may be impossible to show the complete pattern instance on the editing window. For a first orientation SPEEDI therefore displays a tree-like view of the specification's structure (containing blocks, processes, services, and procedures) where those automatons that contain some pattern elements are emphasized. Based on this information the user can decide where to jump in the context specification to investigate other parts of the embedded pattern. SPEEDI should provide jump functionalities, so that corresponding pattern parts are found by a simple mouse click.

By the features described so far it is possible to navigate conveniently through an SDL-pattern-based specification. For example, the user may follow an interaction scenario by traversing a sequence of *BlockingRequestReply* pattern instances. Thereby it comes out that a request signal is sent over an underlying basic service and that the transformation from protocol data units to service primitives is performed by a *Codex* pattern instance. Further investigation reveals that error control is achieved by a *TimerControlledRepeat* and a *DuplicateHandle* instance, while the sequence of *BlockingRequestReply*s is continued thereafter.

If an embedded pattern is well understood within its context and the internal structure is of no interest currently, the pattern instance might be collapsed with its name and some semantic information displayed instead. As we assume a priori knowledge about SDL patterns, only a few information can be enough to keep the compressed specification completely understandable. If necessary the pattern instance can be expanded again.

In some cases it is possible to build higher-level structural units from basic SDL patterns. Thus SPEEDI should also facilitate the creation of pattern clusters from cooperating pattern instances and treat them in the same way as normal pattern instances. That is, the features discussed above are to be extended by grouping mechanisms.

3.2 Pattern adaptation and composition

The process of adapting and composing an SDL pattern is illustrated by the flowchart of Figure 1. Actions taken by the system designer are represented as rectangles with solid lines, while the dashed parts describe intervention from SPEEDI.

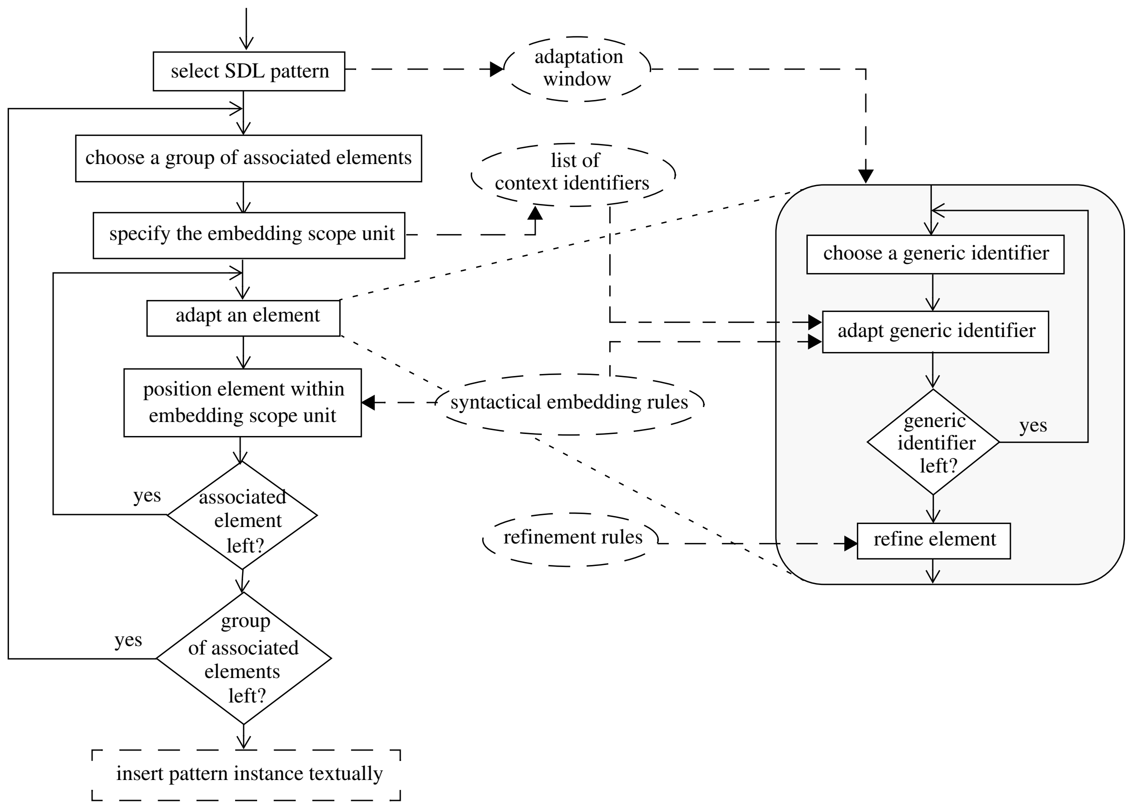


Figure 1: Embedding an SDL pattern

After pattern selection SPEEDI opens an adaptation window with a corresponding template containing the SDL-fragment and several placeholders for the items that must be adapted (e.g., signal identifiers or state names). When adapting an SDL pattern, generic identifiers are often replaced by identifiers from the embedding scope unit. In this case the tool needs to know where the pattern instance will be located within the context specification, so that an adequate list of context identifiers can be prepared. As the designer is only allowed to choose identifier substitutes from the offered list, we reduce possible adaptation errors. If all generic identifiers are adapted, a pattern element can be further refined. Note that an SDL pattern only represents a generic solution for a design problem and therefore usually needs some refinement. As shown in the flowchart each adaptation step must observe certain syntactical embedding and refinement rules. The rules are contained in the pattern description and SPEEDI should ensure that the designer keeps to them. After adaptation a pattern element is ready to be composed with the embedding context. Therefore SPEEDI needs exact positioning information that is collected for each element of a pattern. However, in order to keep syntactical correctness of the context specification SPEEDI should only insert complete pattern instances into the specification.

During later development steps it is possible that embedded patterns are further refined. Whenever an existing pattern instance is touched, we expect SPEEDI to treat this refinement in the same way as the first one. That is, the adaptation window is opened again, containing the complete pattern instance just as before the instance was composed with the embedding context. Adaptation can now be continued by further refinement of each pattern element, while SPEEDI observes the corresponding refinement rules.

4 DESIGN DECISIONS AND IMPLEMENTATION ISSUES

A first analysis of the requirements from the previous section turns out that SPEEDI partially goes beyond the functionality of a normal editing tool. For instance, the embedding of SDL patterns (Section 3.2) is comparable to the template-oriented style of a syntax-directed editor. Furthermore, a tree-like view of the specification's structure (Section 3.1) needs a syntactical analysis of the SDL specification, while the preparation of context identifiers for pattern adaptation (Section 3.2) requires semantic analysis. For the development of a prototype, however, our main goal is to demonstrate the practicability of SDL-pattern-based design. We therefore made some general design decisions that should help to save resources:

- SPEEDI only supports SDL/PR, though it is expected that the advantages of SDL-pattern-based design increase with a graphical representation.
- SPEEDI must integrate into an existing SDL tool environment with a suitable parser and semantic analyser.
- SPEEDI is not fully syntax-directed. For pattern application the editor must support some template-oriented editing style, but this does not hold for other SDL syntactical units.
- Advanced editing features such as pretty printing or search functionalities are not required for SPEEDI.

In the following, we explain some more specific design considerations. It turns out that those considerations concerning pattern documentation apply equally to different SDL patterns, while pattern adaptation and composition sometimes require pattern-specific solutions.

4.1 Pattern documentation

In order to realize pattern documentation, embedded pattern elements have to be located within the context specification. Therefore the SDL specification must be enriched with start and end delimiters for each pattern element. This can be done by pattern-based reverse engineering of an existing specification or directly during pattern-based system design. Because SDL patterns do not belong to the SDL standard, they cannot be introduced by a special keyword. Rather, pattern information is coded by annotations that allow to locate and highlight pattern instances. However, in order to keep readability of the resulting specification, the annotations should be made invisible. Furthermore, in order to prevent unintentional destruction of pattern information, editing the pattern annotations should not be allowed. We decided to code this information as two special SDL comments that mark begin and end of an element. The first comment line additionally contains identification information. This information determines the membership of a particular group of associated elements, a particular pattern, and also a pattern cluster, if existent. The pattern comments should be placed in the SDL-fragment and are inserted automatically by SPEEDI, when composing a new pattern instance. During syntax analysis the comments are evaluated and the parse tree is enriched with the pattern information. For each pattern element its position and identification information (element ID, associated element ID, pattern ID, and possibly the cluster ID) are stored as attributes of the tree nodes belonging to the pattern instance.

With this information we are able to identify the pattern instances the cursor currently points to: starting from the current cursor position, the specification is searched backwards and the traversed pattern marks finally identify the involved pattern instances. In order to highlight a selected pattern instance, its elements must first be located throughout the specification. Therefore we need a mapping from pattern IDs to text positions of corresponding pattern elements. This information is managed by a position map, i.e., an internal database where the current positions of pattern elements are stored. The position map must be updated, while editing the specification. For lack of an incremental SDL parser, this is not always possible for the parse tree. First of all, it would be quite inefficient to start a complete pass of the parser after each keystroke and secondly we would have to deal with syntactically incorrect specifications. A new parse tree is only generated, if information from the semantic analyzer is needed or a new pattern was embedded (i.e., before and after pattern application). Intermediate changes of the specification are only reflected in the position map.

The jump feature and collapsing functionality are also based on the position information. In order to jump to another element of the selected pattern instance, the user is offered a tree-like view of the specification, which is displayed in a separate window. The tree reflects the SDL structure and is derived from the parse tree. The nodes belonging to the selected pattern instance are highlighted and additional information such as the name of the included element is listed. By pointing at a pattern element in the tree, its position in the specification is derived from the position map and the cursor is moved to the element's location.

In order to collapse a selected pattern instance, the SDL code of its elements are cut, kept in memory, and replaced by some pattern-specific identification information such as pattern name or pattern ID. Which information to display should be prescribed by a special item of the pattern description template, so that the replacement can be performed automatically by SPEEDI. If a pattern instance contains another instance, the identification information of each of them is placed in separate lines. When expanding a collapsed pattern instance, the stored SDL code simply replaces the identification information again.

For user defined pattern clusters special cluster IDs are introduced. The above mentioned functions such as coloring, jumping, and collapsing are generalized for those cluster IDs.

4.2 Pattern adaptation and composition

In order to support adaptation and composition of SDL patterns, SPEEDI needs information from various sources (Figure 2). In Section 3.2 we already discussed how the user (system designer) interacts with the tool. Among other things, she specifies the embedding scope unit for each group of associated pattern elements, so that SPEEDI can prepare a list of possible substitutes for the generic identifiers. Therefore a semantic analyser is invoked. Figure 3

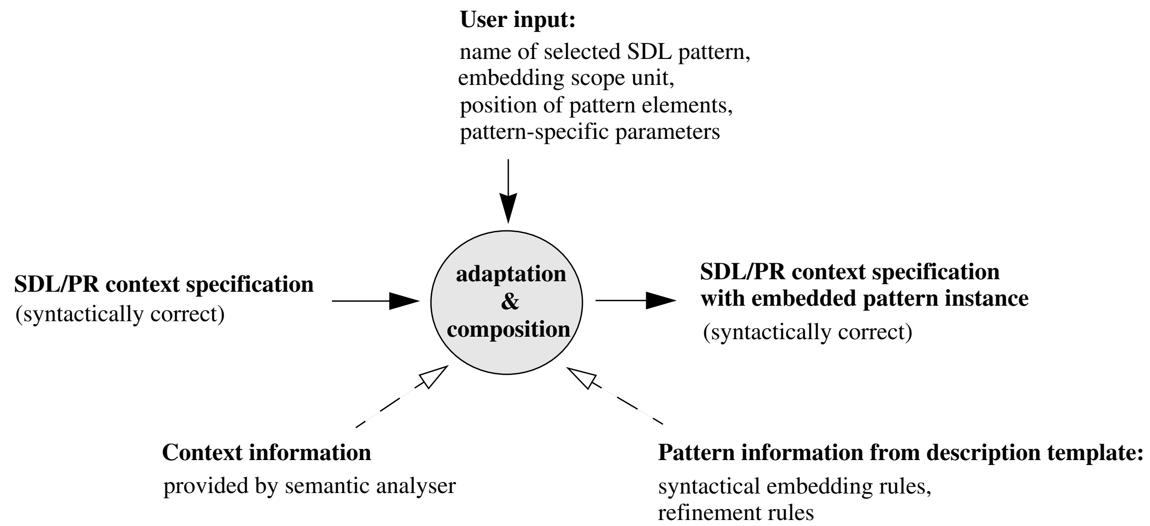


Figure 2: Flow of information during adaptation and composition of an SDL pattern

shows the template for an element of the *BlockingRequestReply* pattern[4], which is presented to the system designer in the adaptation window. The generic identifiers are marked with inverted commas (e.g., '*reply_1*'). Preparing a list of adequate context identifiers for '*reply_1*' would cause SPEEDI to request a list of all signal identifiers from the semantic analyser, that lie within the embedding scope unit. Similarly, SPEEDI requests a list of all state names, if asked for the generic state name *'endRequest1'*. The need for semantic analysis during the adaptation of an SDL pattern requires the context specification to be syntactically correct. As a consequence, SPEEDI initiates a syntax check, when the adaptation window is opened. To keep the parse tree up-to-date for visualization of pattern locations (Section 4.1), the parser is also invoked, when pattern application is completed.

The example template shown in Figure 3 allows two possible reply signals (*'reply_1'* and *'reply_2'*) for the two-way handshake. However, the multiplicity of reply signals is not restricted by the *BlockingRequestReply* pattern (this would otherwise reduce potential of reuse). In order to generate an appropriate template the user is asked for the number of different reply signals, when the pattern has been selected. This and similar kinds of variations of SDL-fragments require pattern-specific algorithms, when adapting a pattern. Since the pattern pool can always grow, SPEEDI must be extendable on that score.

4. The *BlockingRequestReply* pattern introduces a two-way handshake between two automata. Being triggered, the first automaton (Requester) sends a request and is blocked until receiving a reply. After receiving a request, the second automaton (Replier) replies immediately. Refer to [4] for a complete specification of the pattern.

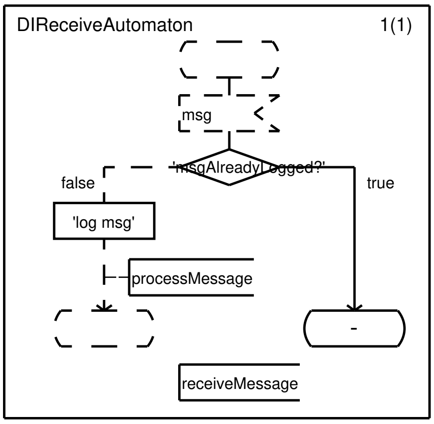
```
state waitForReply;
      input 'reply_1';
      nextstate 'endRequest1';
endstate;
state waitForReply;
      input 'reply_2';
      nextstate 'endRequest2';
endstate;
```

Figure 3: The *ReceiveReply* element of the *BlockingRequestReply* pattern

The same applies to the observation of syntactical embedding and refinement rules. For instance, the substitution of a generic identifier typically depends on syntactical embedding rules such as: "identifier A may be renamed but is required to be locally unique". Since it is expected that the quantity of different rules be not too large, a small set of standard routines may be sufficient in most cases. Note that the observation of a pattern's syntactical embedding rules also requires information from the semantic analyzer (e.g., the set of identifiers from the embedding scope unit must be known to check local uniqueness).

Generally, composition of a pattern instance both adds and replaces parts within the embedding context. Having this in mind, we can imagine the following special case: syntactical units of the context are completely replaced or expanded by the pattern instance. In this case an automatic insertion is not too difficult to handle. Unfortunately, most of the time we have to deal with the situation, that syntactical units of the embedding context are only partially replaced, i.e., inner parts remain unchanged and are installed into the embedded pattern. This often results in extensive structural changes of the context specification. An example is given below.

The SDL-fragment in Figure 4 is a graphical representation of a template for the *ReceiveMessage* element as specified by the *DuplicateIgnore* pattern (refer to [4] for the complete description). The dashed symbols represent design elements from the embedding context. Since the intent of our example pattern is to discard duplicate messages of type *msg*, it should be inserted only after an input of that signal type. The '*msgAlreadyLogged?*' decision with its two branches will be added to the context specification as well as the '*log msg*' task. The *processMessage* comment indicates that after logging the message, it is processed normally. As a consequence, the application of this pattern changes the structure of the embedding transition, because the normal processing of the message will be moved into a branch of the newly added '*msgAlreadyLogged?*' decision.




```
transition 'receiveMsg'
├state 'foo'
├input 'msg'
└decision 'msgAlreadyLogged?'
     ├ false:
     │   ├task 'log msg'
     │   ├comment 'process msg'
     │   └nextstate 'foo'
     └ true:
          nextstate -
```

Figure 4: The *ReceiveMessage* element of the *DuplicateIgnore* pattern

In order to avoid numerous elements containing single tasks ('*log msg*') or decision symbols we decided to provide a pattern template, which additionally contains information about the required context. The idea is to include placeholders for the parts of the context that will be used unchanged within the pattern. Instead of searching an appropriate insertion point for a multitude of small elements, we only copy the corresponding parts of the context into the pattern template. Replacing placeholders with context information is similar to the adaptation of generic identifiers and leads to a more gradual composition.

4.3 SITE - SDL Integrated Tool Environment

As mentioned above, SPEEDI relies on existing tools to support functionalities that do not directly involve pattern-based design. For this purpose we seize the opportunity of using some of the back-end tools of the *SITE* project (SITE - SDL Integrated Tool Environment) [8] [14]. It is planned that the editor cooperates with the following tools:

- sdl96-parser (for the syntax check and building of a parse tree)
- sdl96-crossreferencer (to access the semantic information of a specification)
- sdl96-pretty printer (for a well-formatted output in the editing window)

The parser checks whether a given SDL/PR specification is syntactically correct. If it is, a parse tree is built and stored using the *common representation* (*CR*) - an exchange format for the SITE tools [13]. In this way a CR instance (the parse tree) is built, which contains the pattern information stored in the SDL comments and serves as the basis for semantic analysis. The cross-referencer (sdl96-cross) acts as a semantic analyzer with extended functionalities. It reads a CR instance, performs the semantic analysis and then supplies semantic information about the SDL specification or some parts of it. For example, it can return all identifier definitions within the namespace of an SDL structural unit as well as their corresponding positions in the textual representation. Refer to [10] for the whole functionality of the cross-referencer. The pretty printer generates a well-formatted textual output from a given CR instance, that is displayed in SPEEDI's main window.

Some of these tools have an IDL-specified software interface. To communicate with them, we use the same interface in our editor. Since the possibility to store pattern information as attributes of the corresponding CR nodes is not supported yet, some of SPEEDI's features (especially those concerning pattern documentation) currently make use of our internal pattern database instead of the tree structure.

5 CONCLUSION

SDL patterns describe generic solutions for recurring design problems, which can be customized for a particular context. It is required that the target language for SDL pattern instantiation is SDL. The formal basis provided by SDL enables validation of pattern application and serves as a prerequisite for tool support in general.

In this paper we have first sketched the potential for tool support that SDL patterns provide on the whole. Furthermore, we have discussed the requirements and design decisions for a prototype SDL-pattern editor, called SPEEDI, that is currently under development. Roughly speaking, SPEEDI supports documentation as well as adaptation and composition of SDL patterns. Implicit documentation of an SDL specification by highlighting embedded pattern instances allows to understand whole parts of the specification at once. Readability is further improved by providing a way to navigate through the design and to collapse inspected pattern instances. During adaptation and composition of a pattern, SPEEDI assists in the renaming of generic identifiers and the observation of the pattern's embedding rules.

The aim of SPEEDI as a prototype tool is to illustrate the advantages of SDL-pattern-based design. Functionalities that do not directly involve pattern-based design are outside the primary scope of SPEEDI. Therefore the editor relies on existing tools such as the SDL parser, cross-referencer, or pretty printer of the SITE project. It is planned to interface SPEEDI with this tool environment.

Acknowledgements

Our gratitude is extended to the research group of Prof. Dr. Joachim Fischer from the Humboldt University, Berlin for their cooperation and the support regarding the SITE tool environment. Special thanks go to Ralf Schröder for his cooperativeness in several email discussions.

6 REFERENCES

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley & Sons, 1996

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

[3] 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

[4] 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

[5] 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

[6] 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

[7] B. Geppert, F. Rößler, R. L. Feldmann, and S. Vorwieger, *Combining SDL Patterns with Continuous Quality Improvement: An Experience Factory Tailored to SDL Patterns*, Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC, SAM98, Berlin, 1998

[8] M. v. Löwis and R. Schröder, *Simulation of Telecommunication Systems in SDL-92 using SITE*. In Y.M. Teo, W.C.Wong, T.I.Oren, and R.Rimane, editors, World Congress on Systems Simulation, Singapur, 1997

[9] M.Meijers, G.Florijn, *Support for Object-Oriented Design Patterns*, Department of Computer Science, Utrecht University, The Netherlands, April 1996

[10] Ulf von Mersewsky, *Crossreferencer, Ein Beispiel für eine CORBA-basierte Integration in die SDL-Entwicklungsumgebung SITE*, Diplomarbeit, HU-Berlin, 1998

[11] B. Pagel, M. Winter, *Towards Pattern-Based Tools*, University of Hagen, 1996

[12] F. Rößler, B. Geppert, and P. 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

[13] A.Schade, *Eine Common Representation für SDL'92 Spezifikationen*, Jahresarbeit, HU-Berlin, 1993

[14] http://www.informatik.hu-berlin.de/Institut/struktur/systemanalyse/SITE/e_index.html