Fifth IEEE International Conference on Software Reuse, ICSR5, Victoria, British Columbia, Canada, 1998

Re-Engineering of the Internet Stream Protocol ST2+ with Formalized Design Patterns [1]

Frank Rößler, Birgit Geppert, and Philipp Schaible
University of Kaiserslautern, Computer Science Department
P.O. Box 3049, D-67653 Kaiserslautern, Germany
{roessler, geppert, schaible}@informatik.uni-kl.de

Abstract

The paper presents a case study on the design of communication protocols using formalized design patterns. On principle, the approach adopts the well-known design patterns concept for reusing design solutions. However, a formal description technique (FDT) is applied as design language and therefore knowledge about pattern instantiation or its impact on the embedding context can be precisely specified. We consider formalization to be a prerequisite for increased correctness of resulting products and tool-supported pattern application. The FDT of our choice is the specification and description language SDL. Thus pattern application follows strict rules in terms of the SDL syntax. Additionally, assumptions and effects stated in terms of the SDL semantics help to ensure the intended working of an SDL-pattern within the embedding context. An SDL-pattern based re-engineering of the Internet Stream Protocol ST2+ is presented. In particular, it is demonstrated how SDL-pattern based design can boost the developer's confidence in the resulting product.

1. Introduction

Due to the increasing variety of modern applications and evolving network technologies, the communication services provided by today's general-purpose protocol stacks are not always adequate. It is expected that in order to increase flexibility and to support applications in the best possible way, also customization of special-purpose protocols will play a major role. Here, the configuring of communication protocols from reusable protocol building blocks seems to be a promising way to overcome the additional development effort.

The reuse of predesigned solutions for recurring design problems is of major concern in object oriented software development in general. During the past few years, design patterns have emerged as an especially fruitful approach to software reuse [1] [2] [5] [11]. Contrary to the traditional paradigm of class and function libraries, which is solely concerned with *code* reuse, design patterns aim to focus on the invariant parts of a design solution and offer by far more flexibility for adaptation to the embedding context. That is, the potential of reuse is substantially increased. Additionally, design patterns proved helpful in guiding the instantiation and documentation of frameworks, which provide larger-scale reuse of the overall architecture and design from a certain application domain.

In [6] [8] [9] we present the SDL-pattern approach, which on principle adopts the patterns concept for the design of communication protocols. However, in order to increase productivity and to assure high quality of the resulting communication subsystem, we preferably apply a formal description technique (FDT) as design language. In this context we suggest to formalize design patterns, too. That is, we precisely specify knowledge about pattern application or its impact on the embedding context. We consider formalization to be a prerequisite for tool support and validation of pattern application.

The FDT of our choice is SDL [16]. Thus, e.g., instantiation of an SDL pattern can be defined by precise *embedding rules* in terms of the SDL syntax. Similarly, the semantic model of SDL allows to precisely state assumptions for adequate pattern application and it is possible to express resulting properties of 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. Note, however, that the enhancements work within the scope of an FDT with its formal syntax and semantics.

As an additional advantage, protocol design based on SDL patterns leads to formal specifications of communication protocols, which can be further used for validation and

[1] This work has been supported by the German Science Foundation (DFG) as part of the Sonderforschungsbereich 501 "Development of Large Systems with Generic Methods".

code generation. Simulating or analysing a distributed system (especially its flow of control) before it is implemented helps detect design errors in early stages of development.

The remainder of the paper is organized as follows: Section 2 shortly introduces the FDT SDL. In Section 3 we summarize the SDL-pattern approach to the configuring of communication protocols. This section also includes a short description of the SDL-patterns applied in the case study. SDL-pattern based re-engineering of the Internet Stream Protocol ST2+ [3] is presented in Section 4, where we especially focus on validation issues. We summarize the results in Section 5.

2. The formal description Technique SDL

SDL is a description technique with a formal syntax and semantics and is designed for the specification of reactive, distributed systems, in particular communicating systems. The most recent version was standardized by the International Telecommunication Union (ITU) in 1992 [16] with some updates made in 1996. It offers a textual as well as a graphical representation. SDL is an object-oriented language capable for the description of architecture, behaviour, and data. It is in widespread use in industry and is well-supported by commercial and public domain development environments.

An SDL system specification is hierarchically structured into blocks. Each block is composed either of a set of blocks or a set of processes. System behaviour is modelled as a set of communicating extended finite state machines (CEFSMs) each represented by an SDL process. SDL processes run concurrently and communicate asynchronously by signal exchange. An SDL process may be further structured in SDL services, which themselves represent CEFSMs. In this case, an SDL process appears as the product automaton of its services. For defining data, SDL offers a number of predefined data types such as integer or boolean. New data types are either derived by using built-in constructs (e.g., structs or arrays) or by using the abstract data type concept. For the latter the behaviour of the operators can be defined axiomatically, algorithmically, or by interfacing to another language such as C.

As an object-oriented language, SDL allows parameterized type definition of blocks, processes, services and signals as well as specialization by bounding parameters, adding properties (e.g., new input signals), or redefining virtual types or transitions. Object oriented concepts not supported by SDL are multiple inheritance and dynamic binding.

As a main advantage, SDL specifications are executable and can be used for simulation, validation against test cases, and validation of general properties (such as freedom from deadlock or unspecified reception). Also code can be automatically derived from an SDL specification. Validation, simulation as well as automatic code generation are well supported by existing SDL development tools.

3. SDL-Pattern based Protocol Engineering

This section summarizes concepts for a construction set of protocol building blocks from which a protocol designer can select components and compose them to a customized formal protocol specification. In order to get highly flexible building blocks and to increase quality of resulting products, we combine the design patterns approach with SDL. For further details the reader is referred to [6] [7] [9].

3.1. SDL patterns

An SDL pattern describes a generic solution for a recurring context-specific design problem from the domain of communication protocols. It is assumed that the target language for pattern instantiation is SDL.

SDL patterns are expected to offer the same advantages as commonly attributed to conventional design patterns: patterns capture solutions, which have evolved over time and serve as an elegant way to make designs more flexible, modular, reusable, and understandable. They reflect experiences gained in prior developments and therefore help designers to 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 as hypotheses that have to be validated. In [4] we present an experimental approach for the evaluation of empirical properties of SDL patterns.

Contrary to conventional design patterns, SDL patterns add the advantages of a mathematical foundation. Instead of specifying and applying the patterns rather informally, a formal target language such as SDL offers the possibility to precisely specify how to apply a specific pattern, under which assumptions this will be allowed, and what properties result for the embedding context. This information is pattern specific and organized by a standard description template. The main items of our SDL pattern description template are sketched in the following.

The mere syntactical part of the design solution is defined by a generic *SDL-fragment*, which has to be instantiated and textually embedded into the context specification when applying the pattern. SDL-fragments represent context invariant parts of a design solution. Instantiation and embedding of SDL-fragments is prescribed in terms of *syntactical embedding rules*, which, e.g., guide renaming of abstract identifiers or specialization of embedding design elements. Usually pattern semantics is not com-

pletely captured by an SDL-fragment. Due to language constraints this would otherwise result in an overspecification of the design solution and reduce potential of reuse. Thus additional *semantic properties* are included specifying preconditions for pattern application as well as behavioural 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 *redefinition* 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 [2] [5] is given in [6].

The current pool of protocol building blocks contains SDL patterns that deal with interaction behaviour of distributed objects, error control (lost or duplicated messages), lower layer interfacing, or dynamic creation of protocol entities.

3.2. A pool of SDL patterns

The specification of the Internet Stream Protocol ST2+ uses a set of predesigned SDL patterns. In the following each pattern is introduced. Note that the SDL patterns below are not completely specified. We basically summarize a pattern's intent and skip the description items explained above. For a complete description of the SDL patterns the reader is referred to [7]. The presented pool of SDL patterns is sufficient to specify nearly 100 % of the control structure of ST2+.

- **SendReceive**
  The SendReceive pattern introduces an unconfirmed interaction between two automata. Being triggered, the first automaton sends a signal and continues work. The signal is eventually received by the second automaton.
- **BlockingRequestReply**
  The BlockingRequestReply pattern introduces a confirmed interaction between two automata. Being triggered, the first automaton (Requester, Figure 1) sends a request and is blocked until receiving a reply. After receiving a request, the second automaton (Replier, Figure 1) replies immediately. The generic SDL-Fragment is shown in Figure 1 [1].
- **MultipleRequestsMultipleReplies** and **SingleRequestMultipleReplies**
  The MultipleRequestsMultipleReplies (SingleRequestMultipleReplies) pattern introduces a confirmed 1:n (1:1) interaction between one sending automaton and multiple receiving automata (one receiving automaton). Being triggered, the sender initiates several (one) requests and waits until all corresponding replies are received. After reception of a request, a receiver can send several replies.

[1] We assume that the reader is familiar with the graphical symbols to express states and state transitions in SDL [16].

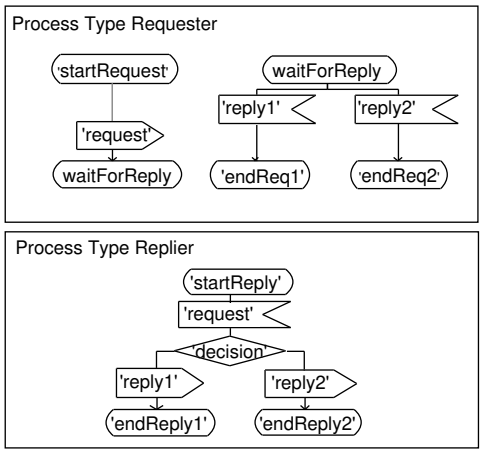

Figure 1. SDL-Fragment - BlockingRequestReply pattern

- **Codex**
  The Codex pattern provides mechanisms to allow two (or more) entities, which interact directly through SDL channels, to cooperate by the means of a given communication service. In general, the introduction of a basic service involves many specialities. Among others these are segmentation, reassembly, upgrade of basic service quality (e.g., in case of loss, disruption or duplication of messages), lower layer connection setup, or routing decisions. The Codex pattern is only concerned about a minimal subset of these functionalities, namely interfacing with the basic service by the means of service primitives (Figure 2). That is, Codex essentially provides a mapping from protocol data units to basic service primitives and vice versa.

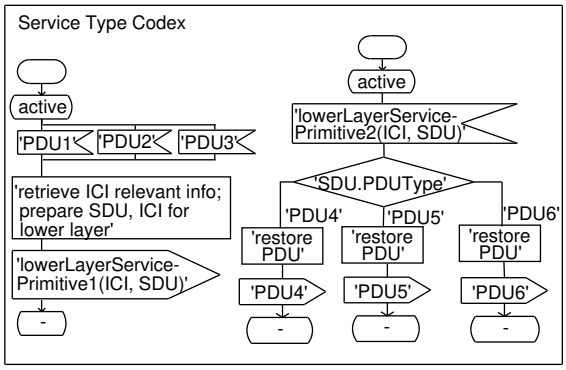

Figure 2. SDL-Fragment - Codex pattern

- **TimerControlledRepeat**
  The TimerControlledRepeat pattern upgrades a confirmed message exchange between two automata in case of message loss. If an expected acknowledgement

does not arrive before the expiry of a timer, the message is repeated (positive acknowledgement with retransmission, Figure 3). This pattern does not deal with the problem of message disruption or duplication.

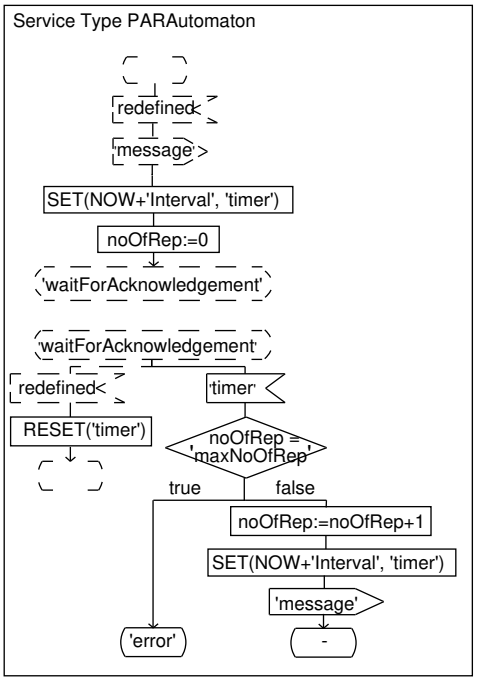

Figure 3. SDL-Fragment - TimerControlledRepeat pattern

- **DuplicateIgnore** and **DuplicateHandle**
  DuplicateIgnore and DuplicateHandle upgrade a message exchange between two automata in case of message duplication. Duplicate messages are detected by a message identifier, which is unique during the lifetime of the receiving automaton. DuplicateIgnore simply discards duplicates, i.e., the reaction to the original message is not repeated. Duplicate messages rely on a certain reaction of the receiving automaton when DuplicateHandle is applied.
- **DynamicEntitySet**
  Consider a given server entity that provides its service exactly one time and then terminates. In order to offer this service several times (e.g., to more than one client), the DynamicEntitySet pattern is introduced. Thereby an administrator dynamically creates a new server entity for each service request and acts as a proxy subsequently (Figure 4).

3.3. Configuration process

For the design of SDL protocol specifications we have defined a configuration process supporting the reuse of protocol building blocks represented as SDL patterns (Figure 5). The configuration process suggests an incremental proto-

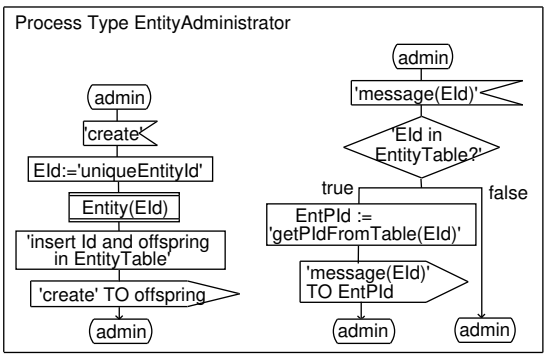

Figure 4. SDL-Fragment - DynamicEntitySet

col development, where the whole set of communication 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 instead of an unreliable basic service. Experience has shown that protocol functionalities can often be specified one after the other and - in addition - be stepwise completed (e.g., adapted to the non-ideal properties of an underlying basic service). This suggests that we perform an individual development step in order to incorporate an additional protocol functionality or relax a corresponding simplification. Thereby each development step further divides into analysis and design of the current protocol functionality and yields an executable SDL design specification. In the following the analysis and design activities within a development step are sketched.

First an object-oriented analysis of the current protocol functionality is performed. This results in an updated analysis model from the previous development step. It is suggested to provide an OMT object model [13] and an MSC [17] use case model, which together identify participating objects and typical interaction scenarios. This integrates our approach into existing SDL methodologies [12] [14] [15], which also combine OMT and/or MSC with SDL for analysis and design, respectively.

The analysis model is realized in the following design activity. Here SDL patterns come into place. Starting point is the context SDL specification, i.e., the SDL design specification obtained from the previous development step[1]. This may, e.g., be a protocol specification, which relies on a reliable basic service. Hence the design problem (stated in the analysis model) could then be to suit the protocol to an unreliable basic service. In order to meet the new requirements a number of SDL patterns are *selected* and

[1]For the first development step the initial context specification is either empty or given by an instantiated SDL framework.

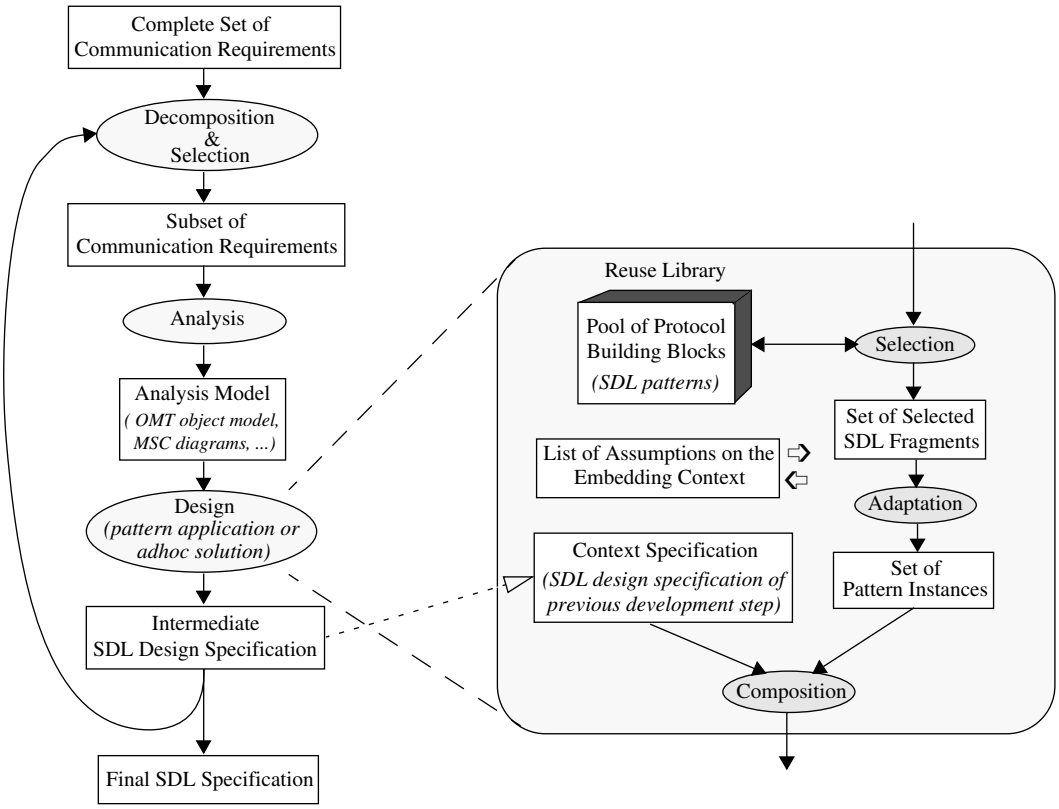

Figure 5. Incremental configuration process

applied to the context specification[1]. 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 in Section 3.1). 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 has to be *composed* with the embedding context, which is prescribed by the *specialization* part of the syntactical embedding rules and also by *redefinition* rules of embedding pattern instances, if existent.

The result of this design activity is an intermediate SDL design specification, which is subsequently validated against the analysis use case model by performing MSC based validation and simulation (not shown in Figure 5). Also the correctness of the SDL specification concerning general properties such as freedom from deadlocks is checked. If any faults are discovered, a return to one of the previous development steps is needed (not shown in Figure 5). 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.

As a special feature, SDL-patterns help to increase the developer's confidence in the correctness of the resulting product. This is because he or she can immediately check a pattern's constraints (stated by the semantic properties) on the embedding context and thus ensure its proper working. As an example a semantic property of the *BlockingRequestReply* pattern (Section 3.2) is listed below:

Property of BlockingRequestReply: *If the assumptions stated below hold, the Requester will eventually receive a reply from the Replier after sending a request. The assumptions are:*

- *The request and reply signals are not implicitly consumed by the embedding context.*
- *Communication between Requester and Replier for transmission of request and reply signals is reliable.*
- *The startReply state of the Replier will always eventually be reached.*

Care has to be taken not to destroy a pattern's intent in sub-

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

sequent development steps. Therefore the assumptions are collected in a so-called *list of assumptions* and have to be checked in each development step. We will especially focus on this point when presenting the case study in Section 4.

4. Re-Engineering of ST2+ using SDL patterns

ST2+ is the second version of the Internet Stream Protocol and is informally specified in natural language [3]. We have performed an SDL-pattern based re-engineering of most parts of ST2+, resulting in a systematical and formal SDL design specification. The presentation below follows the configuration process introduced in Section 3.3, where we focus on the decomposition of communication requirements and the resulting design activities. Furthermore, it is demonstrated how the SDL-pattern approach can boost the developer's confidence in the correctness of the resulting product. For a more detailed description the reader is referred to [10].

4.1. Basic operation of ST2+

The Internet Stream Protocol ST2+ is a connection-oriented protocol operating at the same layer as the well-known IP protocol. ST2+ supports efficient multicast delivery of data with guaranteed quality of service (QoS). It contains a resource reservation protocol for the establishment of real-time streams, that have sufficient network resources reserved to guarantee the required QoS. Each stream consists of one origin agent and several intermediate and target agents. The protocol agents form a tree-shaped structure, where a possible topology is illustrated in Figure 6. The resource reservation protocol defines several operations for establishing, maintaining, and closing real-time streams.

First, the origin application initiates the establishment of a stream by an OPEN request. Subsequently, protocol agents request communication paths to each target node and at the same time initiate resource reservations. The origin application is informed about target applications, that accept or refuse the connection or could not be reached due to lack of resources.

Once a stream is established, its QoS can be later changed by a CHG operation. Thereby resources are either released or supplemented along the communication paths of the stream.

Also the multicast group can be changed. This is done by the origin adding (ADD operation) or removing (DROP operation) individual target applications. Adding a new target is similar to establishing a stream: the path to the new target is fixed but resources are only reserved for those sections, that were not already part of the stream. Target applications can also change the multicast group themselves. In order to join an already established stream a target agent sends a JOIN message to the corresponding origin agent. If this message traverses a node that is part of the stream, this node is responsible to establish a new path to the target. Each target can remove itself from the multicast group by a LEAVE operation.

The origin closes the complete stream by a CLOSE operation. Removing of a target and closing of a stream are non-confirmed operations. Note that stream operations can perform in parallel and over an unreliable basic service. This substantially increases the complexity of ST2+ .

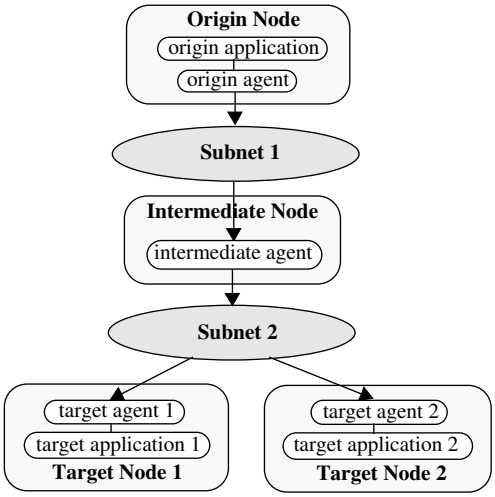

Figure 6. Example of an ST2+ stream

4.2. Incremental configuring of ST2+

In order to obtain the final design specification we have performed eight development steps. Figure 7 summarizes the chronological order and the supported subset of communication requirements. Note that requirements, which result from a simplification of original requirements, are relaxed in later development steps. The set of requirements, that is realized by the final protocol specification, is marked with a thick border (Figure 7). The initial subset of requirements, for instance, only supports setup and tear down of a single unicast stream. Furthermore, it is assumed that origin, intermediate, and target agents are directly connected by SDL channels. We stepwise relax the assumptions w.r.t. the underlying communication service, complete the protocol operations, introduce multicast communication, and finally allow several streams to be active at the same time.

There is an important requirement, which every resource reservation protocol has to meet: during connection establishment the route of traversed nodes is fixed and

step	multiplicity	mode	basic service	operations
I	single stream	unicast	directly connected	OPEN, CLOSE
II				CHG
III			reliable basic service	
IV		multicast		
V				ADD, JOIN, DROP, LEAVE
VI			disruption, loss, or duplication of messages	
VII	several streams			
VIII			breakdown of nodes or links	

Figure 7. Decomposition of requirements and design steps

sufficient resources are reserved along the resulting communication paths, so that required QoS is guaranteed during data transfer. Furthermore, after requesting a connection the calling application must eventually receive a reply stating, whether the connection establishment has succeeded or not. That is, *the calling application must not deadlock*.

In the following each development step is discussed. Additionally, it is shown how applying SDL patterns can boost the developer's confidence in the correctness of the resulting protocol w.r.t. the above mentioned requirements.

Design step I and II. The initial subset of communication requirements supports a single unicast connection over a series of nodes, that are directly connected through SDL channels (Figure 8). Thus we only consider virtual communication within the ST2+ layer of the protocol stack. Thereby the first step only realizes the OPEN and CLOSE operations. Next we have extended the protocol by allowing changes (CHG) in QoS of the established stream.

For the OPEN and CHG operation the interactions between adjacent protocol agents as well as the interactions between an origin or target agent and its application process follow the BlockingRequestReply pattern: after requesting a connection the sending process is blocked until receiving one of the expected replies. Figure 9 shows the resulting behaviour of an origin agent in case of an OPEN operation. The parts shaded and realize the interactions with the application process and intermediate agent, respectively. Note that both cases are instances of the generic SDL-fragment of Figure 1. However, only the corresponding requester or replier parts of an instance are shown, because the communicating peers belong to the application process and the intermediate agent, respectively.

The point is, that the semantic properties of the BlockingRequestReply pattern guarantee certain properties ((P1.1) and (P1.2), below) of the resulting protocol design, if some assumptions ((A1.1) - (A1.4), below) on the embedding

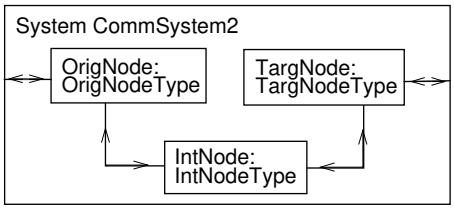

Figure 8. Protocol architecture of step I and II

context hold[1]. Note that these assumptions and derived properties are part of the SDL pattern description (semantic properties, Section 3.1).

(A1.1): Applications behave like a requester/replier as defined by the BlockingRequestReply pattern.
(A1.2): Request/reply signals (e.g., OPEN.req) are not implicitly consumed.
(A1.3): If a request signal arrives, the StartReply state of the corresponding replier will eventually be reached.
(A1.4): Communication between the different requester/replier pairs is reliable.

⇩

(P1.1): After invoking OPEN.req the origin application will eventually receive one of the expected replies from the origin agent (e.g., OPEN.accept_ind).
(P1.2): After sending a CONNECT signal each origin and intermediate agent will eventually receive one of the expected replies (e.g., ACCEPT signal).

(A1.1) is an assumption on the system environment whereas the other assumptions have to be fulfilled by the SDL specification itself. For instance, assumption (A1.4) holds, because the requester and replier are directly connected by SDL channels. (A1.2) and (A1.3) can also be validated by inspection of the SDL design [10]. The resulting properties state that no agent participating in an OPEN operation, in particular the calling application, will deadlock.

From these properties of the individual pattern instances we can additionally prove by induction that all protocol agents on the chosen communication path have been reached during connection setup. If we further assume that the resource managers on each traversed node work properly (in the sense that a sufficient amount of local resources is calculated and also reserved [2]), we can be confident that sufficient resources are reserved for the whole stream.

The CLOSE operation is non-confirmed and therefore realized by applying the SendReceive pattern, where the sending agent immediately continues work after sending a close message.

Design step III. In the third step we replace the direct

[1](A1.2), e.g., is a short cut for the second assumption of development step 1 and (P1.1), e.g., is a short cut for the first derived property of development step 1.

[2]This requirement lies outside the scope of ST2+.

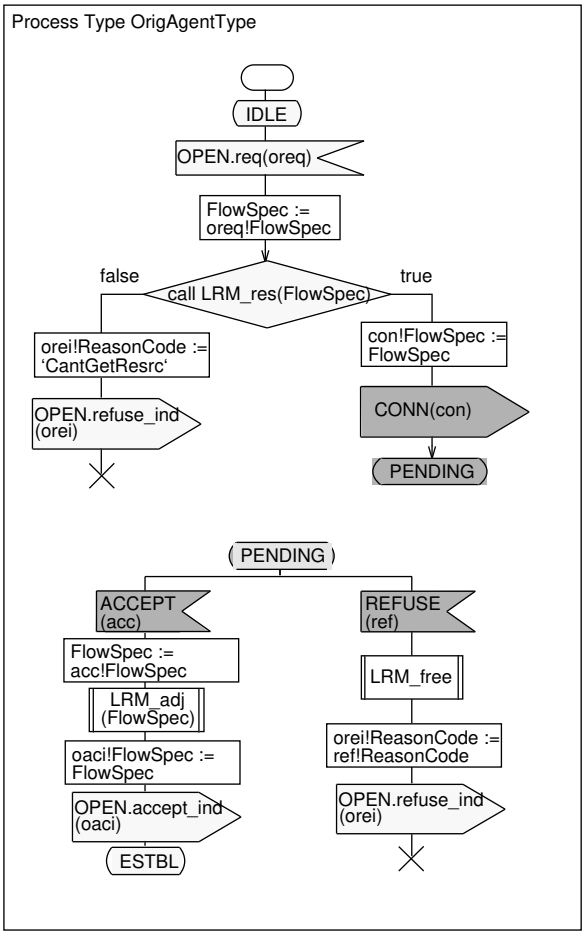

Figure 9. Replier with embedded Requester (BlockingRequestReply pattern)

connection of protocol agents by a reliable basic service (Figure 10). Therefore protocol data units (PDU) have to be mapped on basic service primitives and vice versa. To solve this problem, we have applied the Codex pattern to each process type resulting from development step 2 (e.g., OrigAgentType, Figure 9). Thereby the SDL-Fragment of Figure 2 was instantiated accordingly, i.e., with one instance of 'PDUx' for each incoming and outgoing PDU of the corresponding process type. According to the semantic properties of Codex, the pattern properly substitutes an SDL channel between communicating SDL processes, if the following assumptions hold:

(A3.1): The basic service in use must be connectionless.
(A3.2): The basic service in use must be reliable.
(A3.3): The developer adds mechanisms to handle the preparation of lower layer interface control information and takes care that the interface control information retrieved by the codex service always matches with the PDU currently processed.

Note that assumptions (A3.1)-(A3.3) replace (A1.4), where (A3.1) and (A3.2) are assumptions on the underlying basic service and assumption (A3.3) can be validated by inspection of the SDL design specification [10]. Together with the remaining assumptions of the previous development steps, (P1.1) and (P1.2) still hold.

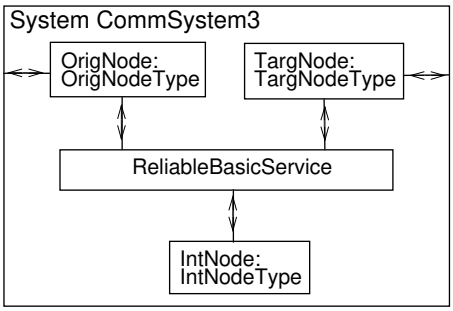

Figure 10. Protocol architecture of step III

Design step IV and V. So far we have only supported unicast communication between one origin and one target application. In the next steps we allow multicast communication with several target applications and we incorporate further stream operations, that allow the origin application to add new targets to the multicast group (ADD Operation) or remove already participating targets (DROP Operation). Furthermore, a target application itself is allowed to join (JOIN Operation) or leave (LEAVE Operation) the stream.

It turns out that multicast communication cannot be integrated by pure extensions of the SDL specification, such as in earlier development steps. Rather, the BlockingRequestReply instances that implement unicast communication have to be replaced. However, due to the pattern based approach it is straightforward to identify and eliminate those parts of the specification. To realize multicasting, we subsequently apply the SingleRequestMultipleReplies and MultipleRequestsMultipleReplies patterns. As a result, the specification partially shown in Figure 11 is obtained where the correspondence to step I and II (Figure 9) is illustrated by shadings. Essentially, the MultipleReplies patterns add a loop for sending and receiving several signals (see the parts not shaded in Figure 11). The adaptation and refinement of the patterns, however, are adopted from the unicast case (not completely shown in Figure 11). The patterns SingleRequestMultipleReplies and MultipleRequestsMultipleReplies are also applied in order to realize the ADD stream operation. Note that interactions between stream operations further increase complexity and cannot be discussed within the scope of this paper.

The updated list of assumptions (for their validation see [10]) resulting from the semantic properties of SingleRequestMultipleReplies and MultipleRequestsMultipleReplies is given below. Furthermore, the derived properties

(P1.1) and (P1.2) have to be replaced by (P4.1) and (P4.2). The properties of the resulting protocol specification w.r.t. deadlocks during connection setup and reservation of sufficient resources follow from an analogous argumentation to step I.

(A3.1)-(A3.3)
(A4.1): Applications behave like a requester/replier as defined by the SingleRequestMultipleReplies and MultipleRequestsMultipleReplies patterns.
(A4.2): Request/reply signals are not implicitly consumed.
(A4.3): If a request signal arrives, the StartReceive state of the corresponding replier will eventually be reached.
(A4.4): The reception loop of a requester must terminate iff the last reply was processed.

⇩

(P4.1): After invoking OPEN.req the origin application will eventually receive one of the expected replies (e.g., OPEN.accept_ind) for each addressed target application.
(P4.2): After sending a CONNECT packet each origin and intermediate agent will eventually receive one of the expected replies (e.g., ACCEPT packet) for each addressed target.

Design step VI. The assumption that the underlying basic service be reliable does not always hold with existing networks. We therefore extend the protocol specification to additionally cope with message loss by applying the TimerControlledRepeat pattern several times. Thereby instances of the SDL-fragment (Figure 3) are inserted where a PDU is sent. As a consequence, we must also deal with duplicated messages, which is realized by applying the DuplicateIgnore and DuplicateHandle patterns. According to the semantic properties of the applied patterns, assumption (A3.2) has to be replaced by the new assumptions listed below.

(A3.1)+(A3.3) + (A4.1)-(A4.4)
(A6.1): Messages are not disrupted.
(A6.2): Incoming messages can be uniquely identified and are logged.

⇩

(P6.1): After invoking OPEN.req the origin application will eventually receive one of the expected replies (e.g., OPEN.accept_ind) for each addressed target application.
(P6.2): After sending a CONNECT packet each origin and intermediate agent will eventually receive one of the expected replies (e.g., ACCEPT packet) for each addressed target or an error handling is started.

The TimerControlledRepeat and DuplicateControl patterns cannot guarantee that an acknowledgement will be received for each PDU, but an error handling will be launched otherwise (P6.2). It follows that after invoking OPEN.req the origin application will eventually be notified, which targets have been reached and participate in the stream. Additionally, sufficient resources must have been reserved, if the connection setup is successful (argue analogous to step I).

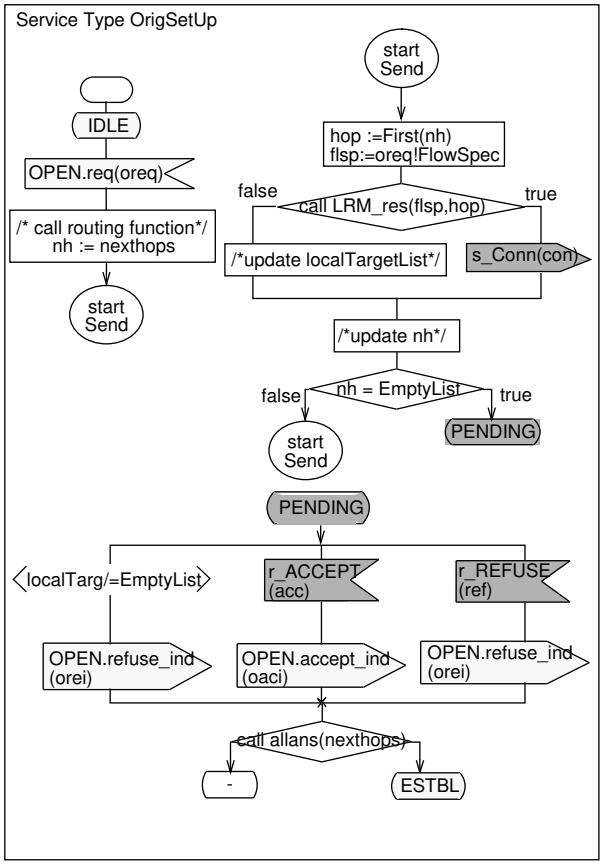

Figure 11. Replier (SingleRequestMultipleReplies) with embedded Requester (MultipleRequestsMultipleReplies)

Design step VII and VIII. Finally, we complete the protocol specification by allowing several streams to be active at the same time and additionally deal with failures of network nodes and links. The DynamicEntitySet pattern is capable to manage several streams at the same time. Its generic behaviour results from the SDL-fragment of Figure 4. In each traversed network node a stream is handled by its own SDL process and from the pattern's semantic properties ((A7.1) and (P7.1)) it follows that previous considerations are still valid for each stream.

(A7.1): Each ST agent (including ST applications) is informed about the unique identifier of its peer agent and always adds this identifier to the output signals (except for OPEN.req).

⇩

(P7.1): The stream behaviour offered by the replicated SDL processes is equivalent to the behaviour obtained in previous development steps.

The patterns dealing with failure detection and recovery do not affect the properties derived for a single stream and therefore are not further elaborated.

As a result we have the following list of assumptions and derived properties, which has to be taken into account for further changes of the SDL specification.

(A3.1), (A3.3)
(A4.1) - (A4.4)
(A6.1) - (A6.2)
(A7.1)

⇩

(P6.1): After invoking OPEN.req the origin application will eventually receive one of the expected replies (e.g., OPEN.accept_ind) for each addressed target application.

(P6.2): After sending a CONNECT packet each origin and intermediate agent will eventually receive one of the expected replies (e.g., ACCEPT packet) for each addressed target or error handling is started.

(P7.1): The stream behaviour offered by the replicated SDL processes is equivalent to the behaviour obtained in previous development steps.

5. Conclusion

We have presented a case study on the SDL-pattern based configuring of communication protocols. SDL patterns characterize as formalized design patterns. However, we do not deal with formalizing design patterns in general. That is, instead of formalizing reuse concepts we aim to increase reusability within the formal methods area. Thereby we naturally benefit from the formal basis provided by SDL. SDL-pattern based protocol configuring leads to formal specifications, which can be further used for validation and code generation. This is well supported by existing SDL development tools.

As compared to SDL specifications that have been developed the usual way, SDL-pattern based configuring leads to a more systematical design. This is due to the fact that design decisions are well founded and documented. As a consequence, readability of the specification and communication among team members is improved. Also confidence in the correctness of the resulting product increases.

It is worth mentioning, that a very large portion (almost 100 % of the control structure) of the final ST2+ specification resulted from SDL patterns (10 different patterns were used), where the original specification in natural language took about 70 pages.

Acknowledgements

Thanks go to the anonymous reviewers. Furthermore, we especially want to thank Yair Lahav for valuable comments and fruitful discussions on a draft of this paper.

References

[1] F. Buschmann, *Software Architecture and Reuse - An Inherent Conflict?*, Proceedings of the 3rd International Conference on Software Reuse, Rio de Janeiro, Brazil, Nov. 1994

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

[3] L. Delgrossi and L. Berger (Ed.), *Internet Stream Protocol Version 2 (ST2), Protocol Specification - Version ST2+*, RFC 1819, 1995

[4] 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

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

[6] 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

[7] 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

[8] 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

[9] 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

[10] B. Geppert, F. Rößler, and P. Schaible, *Pattern-based Configuring of the Internet Stream Protocol ST2+ with SDL*, SFB 501 Report 12/97, Computer Science Department, University of Kaiserslautern, Germany, 1997

[11] W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995

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

[13] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991

[14] SDT 3.2 Methodology Guidelines: *The SOMT Method*, Telelogic, 1997

[15] D. Witaszek, E. Holz, M. Wasowski, S. Lau, and J. Fischer, *A Development Method for SDL-92 Specifications Based on OMT*, in: R. Bræk, A. Sarma (Eds.): SDL'95 with MSC in CASE, Proceedings of the 7th SDL Forum, Norway, 1995

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

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