Proceedings of the First IEEE International Conference on Formal Engineering Methods (ICFEM'97), Hiroshima, Japan, November 1997

Generic Engineering of Communication Protocols - Current Experience and Future Issues[1] -

Birgit Geppert
University of Kaiserslautern
Computer Science Department
P.O. Box 3049, 67653 Kaiserslautern, Germany
geppert@informatik.uni-kl.de

Frank Rößler
University of Kaiserslautern
Computer Science Department
P.O. Box 3049, 67653 Kaiserslautern, Germany
roessler@informatik.uni-kl.de

Abstract

We present an approach to the generic engineering of communication protocols, which combines pattern-based software engineering with the formal description technique (FDT) SDL. Our approach has the potential to reduce the effort for protocol design and assures high quality of the resulting communication subsystem. Actually, we suggest to keep a pool of reusable SDL patterns, which use FDTs to describe generic solutions for recurring design problems. According to specific communication requirements a protocol designer can then select patterns, which are suitably adapted and finally composed to build part of the resulting protocol specification.

1. Introduction

Due to increasing communication demands as found in many modern applications (e.g., stringent requirements on throughput, delay, delay-jitter as well as multicasting or synchronization), the communication services provided by existing general-purpose protocol stacks (such as the OSI stack or the Internet protocol suite) are not always adequate. Also, classical protocols are not designed to exploit the advantages of advanced transmission technologies (e.g., fibre optics) and high-speed networks (e.g., ATM), which combine high bandwidth with low error rates. Rather, they enforce the use of mechanisms such as error control, which may actually not be needed by a given application leading to reduced performance.

To improve this situation plenty of different customized communication subsystems would be necessary. Here, protocol configuring seems to be a promising way to reduce the additional development effort. Actually, we suggest to provide a pool of reusable protocol building blocks, from which the protocol designer can select components according to the specific communication requirements. After suitable adaptation these building blocks are ready for composition to build part of the customized communication subsystem. This characterizes protocol configuring as a generic approach with reuse of products (i.e. protocol building blocks) as a main concept of genericity (see [9] for further information on generic *requirements* engineering).

Well-known projects dealing with protocol configuring are F-CCS [16], Da CaPo [10], CogPiT [17] and ADAPTIVE [13] (see [6] for an overview). These approaches have in common that special-purpose protocol implementations are automatically configured during connection establishment. As a major drawback, the use of programming languages prevents the resulting communication subsystem from being verified. Also, the extension of the component pool appears to be difficult, because it is given as a tightly coupled library of C-modules. Here, the use of formal description techniques (FDTs) allowing an abstract, unique specification of protocol building blocks as well as their interactions seems to be advantageous.

We therefore aim to devise formal methods and techniques for the generic engineering of communication subsystems, which help to overcome additional development effort resulting from customization and assure high quality of the resulting communication subsystems. Instead of configuring a protocol implementation we aim at configuring a formal design specification, where the FDT of our choice is SDL [20]. Genericity in our approach is not achieved by low-level reuse of code but mainly by high-level *reuse of architecture and design*. The reuse of predesigned solutions for recurring design problems is of major concern in object-oriented software development. During the past few years design patterns have emerged as an

[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".

especially fruitful concept from other well-known approaches such as frameworks or toolkits (in the sense of object-oriented libraries) [2] [3]. Early experience in reuse of protocol specifications with SDL has been reported in [15], where a protocol building block was designed as a reusable library class, however, according to the authors with limited success. As their conclusions were similar to those which led from the toolkit approach to the design patterns concept, we argue that pattern-based reuse is better suited. Thus our approach is based on the concept of design patterns, which we specialize to communication protocols. Apart from SDL, we use MSC [21] in combination with OMT [12] to formally specify protocol design patterns and rules for their instantiation and composition. As the target language for pattern instantiation is SDL, we call reusable protocol building blocks *SDL patterns*.

An important advantage of our approach is that configuring leads to formal design specifications of communication protocols, which can then be used for validation and code generation. Validation techniques such as MSC-based validation, simulation or semantic analysis (see [8] for an overview of SDL based validation techniques) are supported by existing SDL development tools. Automatic code generation (rapid prototyping) is also supported by existing SDL tools. Thereby code generation is another important aspect of genericity, however, in this case genericity of the development process.

The remainder of the paper is organized as follows: in Section 2, we shortly introduce the different techniques used for pattern description, namely SDL, MSC and OMT object model. Their combined use for specifying SDL patterns is presented in Section 3, while Section 4 describes pattern-based engineering of formal SDL design specifications. So far we have developed an initial pool of SDL patterns for generic engineering of communication protocols. Its application to a non-trivial case study is sketched in Section 5. We conclude with experiences and an outlook in Section 6.

2. Description techniques suitable for pattern specification

In the following we will shortly introduce the Specification and Description Language (SDL), Message Sequence Chart (MSC), as well as the OMT object model. These three techniques will be used for the specification of SDL patterns (Section 3). Note that OMT object model and a notation similar to MSC are already used for the specification of conventional design patterns [2], so an important difference of our approach is the usage of SDL as a target language for pattern instantiation.

2.1. SDL - Specification and Description Language

SDL is a formal description technique for the specification of reactive, discrete systems. The most recent version was standardized by the International Telecommunication Union (ITU) in 1992 [20] with some updates made in 1996. It offers a textual as well as a graphical representation. SDL is an object-oriented language capable for description of architecture, behaviour, and data of distributed, communicating systems. An SDL system specification is structured into blocks (Fig. 1) where each block is

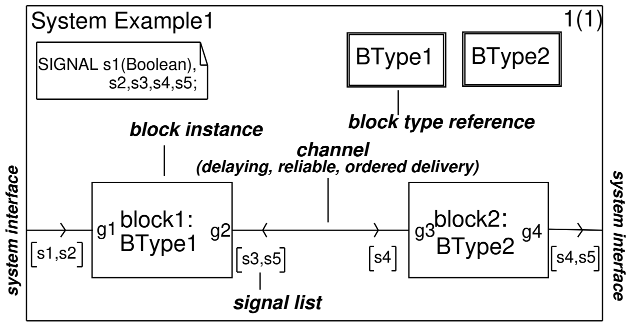


Figure 1. SDL system diagram

composed either of a set of blocks or a set of processes (Fig. 2). System behaviour is modelled as a set of communicating extended finite state machines (CEFSMs) each represented by an SDL process (Fig. 3) with an unbounded FIFO input queue. SDL processes run concurrently and communicate asynchronously by signal exchange. An SDL process may be further structured into SDL services, which themselves represent CEFSMs (Fig. 4). In this case, an SDL process appears as the product automaton of its services. The services of a process run sequentially and share the input queue of the process.

Additionally SDL allows some kind of generic specification. This is supported by parameterization and specialization of type definitions. An example of a parameterized process type definition is shown in Fig. 3. Formal context parameters (sig1, sig2, sig3, sig4) can be set to actual con-

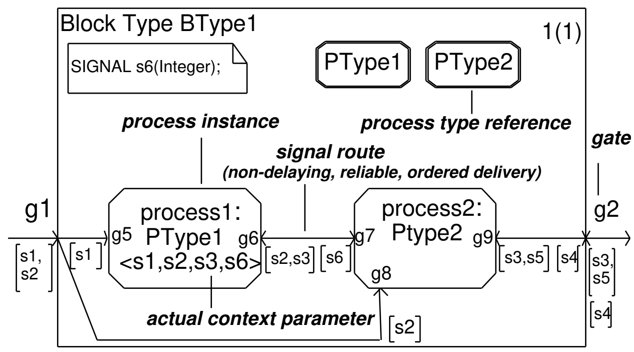


Figure 2. SDL block type diagram

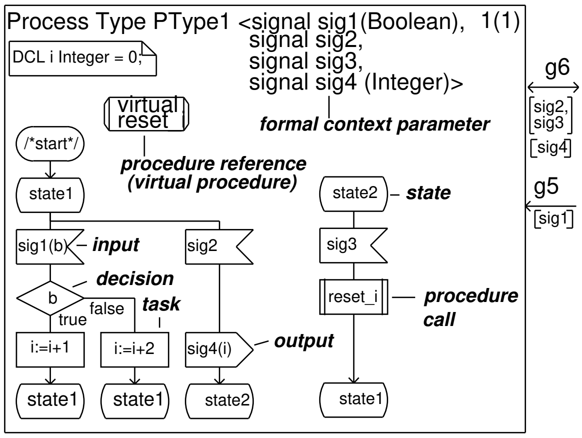


Figure 3. SDL process type diagram

text parameters (s1, s2, s3, s6) of the actual scope unit by instantiation (Fig. 2). Specialization of types results from the reuse of an existing type definition and additionally bounding some context parameters, adding properties or redefining virtual types or transitions. Specialization of types is important for pattern composition [5]. SDL also supports the specification of abstract data types with behaviour definitions of the operators given informally, axiomatically, algorithmically or by the use of another notation such as ASN.1.

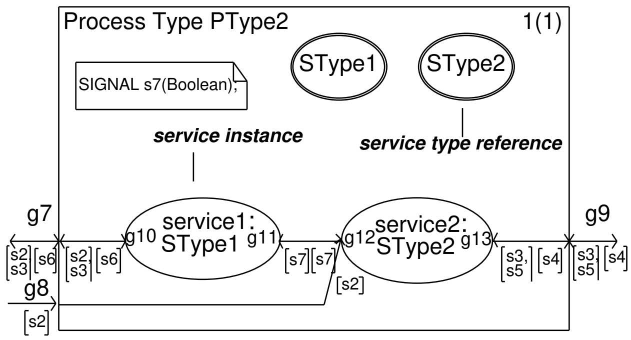


Figure 4. SDL service structure

For systems engineering SDL is usually used in combination with other languages such as OMT object model, MSC, or ASN.1. Some of these languages have also been studied by the ITU. For instance, the application of Message Sequence Charts [21] together with SDL is suggested in [20]. Though the combined use of OMT and SDL is not yet standardized it is commonly agreed on that "the use of the OMT object model notation in combination with MSC and SDL is a powerful combination that covers all aspects of system engineering" [1] . The integration of SDL with these techniques is also supported by existing development tools.

2.2. MSC - Message Sequence Chart

MSC is a formal technique for specifying the interactions between system components. It is subject of standardization by the ITU with the recent version MSC'96 defined in [21]. Like SDL it offers a textual and a graphical representation. MSC may be used for requirement specification, interface specification, simulation, validation, and documentation of distributed systems. Thereby an MSC diagram is not intended to describe the complete behaviour of a system (like SDL does), but merely defines one execution trace (e.g., a typical interaction or an exceptional case). MSC is often used in combination with SDL to give an overview specification of the interaction patterns of a distributed system, i.e. to specify selected system traces. Fig. 5 illustrates an example trace for the interaction between *process1* and *process2* of Fig. 2.

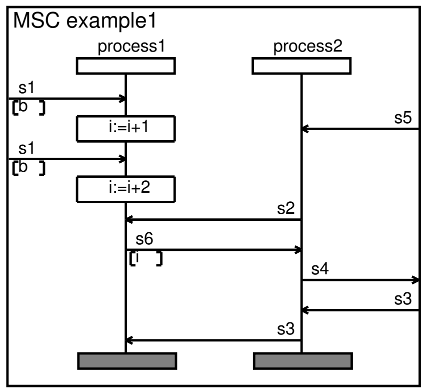


Figure 5. MSC diagram

2.3. OMT - Object Model

The Object Modeling Technique (OMT) [12] is an object-oriented methodology for system analysis and design where a system is described by three models: the *object model* specifying the static structure of the involved objects and their relations, the *dynamic model* specifying system behaviour, and the *functional model* specifying data transformations. As system behaviour and data can also be specified by SDL, especially the OMT object model is relevant for integration with SDL. The integration of SDL and OMT object model is even supported by commercial tools such as SDT or Geode. The object model for the SDL types given in Section 2.1 is illustrated in Fig. 6.

[1]This citation is borrowed from the Homepage of the SDL-Forum, which can be found under *http://www.sdl-forum.org/*

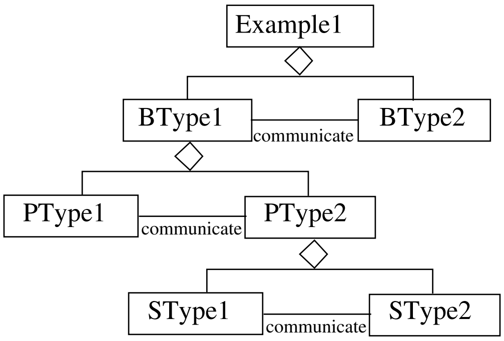


Figure 6. OMT object model

3. SDL patterns as reusable protocol building blocks

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 resemble conventional design patterns and offer the same advantages: 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. Patterns even help improving the documentation and maintenance of existing software systems, because they focus on the essentials of a design solution and thus increase the probability to be matched with a given design.

A main characteristic of conventional design patterns is their independence on a particular target language. Thus the design solution normally is described in natural language combined with an OMT object model that defines the participating components and their relations. Though sample implementations may be given, these are not strictly prescribed and only serve as implementation guidelines. The lack of a formal target language for pattern instantiation prevents a pattern's intent from being precisely specified.

We therefore combine the design patterns concept with SDL for definition of reusable protocol building blocks. SDL patterns deal, for instance, with two-way interaction between communicating peers, error control (lost or duplicated messages), lower layer interfacing, or dynamic establishment and closing of connections. Apart from syntactical rules for pattern application also semantic properties can be specified defining the pattern's intent more precisely. The use of an FDT furthermore allows to specify constraints on the redefinition of pattern instances in order to prevent a pattern's intent from being destroyed. In [5] the language support of SDL for formal pattern-based protocol design is discussed.

The conventional design patterns presented in [2] and [3] are described by the means of special description templates. Roughly speaking, a pattern is described by its name, the recurring design problem it addresses, a corresponding well-proven design solution, and a short description when the pattern should/should not be applied. In order to specify SDL patterns we propose an enhanced description template with the items listed below. A comparison to existing description templates is given in [4].

SDL pattern description template

Name. The name of the pattern, which should intuitively describe its purpose.

Intent. A short informal description of the particular design problem and its solution.

Motivation. 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 object model*. This defines the involved components 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 **versions** of the design solution are possible (realization as SDL service or procedure, interaction by message passing or shared variables, etc.), fragments for the most frequent versions are included. We plan to substitute versioning by a special kind of pattern parameterization. For each fragment, corresponding **syntactical embedding** rules are defined in terms of the SDL syntax:

- Rules for renaming of the abstract identifiers of the SDL-fragment.
- Rules for specialization of embedding SDL superclasses in order to integrate the instantiated pattern. This could, for instance, result in the *addition* of new transitions or SDL services or the *redefinition* of existing virtual types or transitions

Note that the 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 being 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 the 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.

Redefinition. An embedded pattern instance can be further redefined, e.g., by the embedding of another pattern instance in subsequent development steps. Redefinitions 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.

Fig. 12 shows an extract of an example SDL pattern, which we will apply in our case study in Section 5. For more information about specific SDL patterns see [4], [5], [7].

4. Protocol Engineering based on SDL patterns

Pattern-based configuring involves three steps: *selection*, *adaptation*, and *composition* of protocol building blocks. Starting point is a pool of predefined protocol building blocks and a context specification (written in SDL) containing an unsolved design problem. This could, for instance, be a protocol design which relies on a reliable basic service. The design problem now is to suit the protocol to an *unreliable* basic service. Based on the new requirements a proper set of SDL patterns is to be *selected*. 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 patterns 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 of protocol building blocks can start from the very beginning of protocol design. That means, the above mentioned steps are applied recursively during the design activity where appropriate. Selection, adaptation, and composition is supported by the SDL pattern description template in the following way:

- **selection:**
  before selection of a proper set of building blocks the communication requirements are analysed in order to identify the inherent design problems. Pattern selection can then be guided by comparing the resulting analysis model with the pattern descriptions. We therefore assume that an object-oriented analysis is performed before configuring and that the resulting analysis model includes an OMT object model and MSC diagrams describing typical and exceptional scenarios. This prerequisite is met by most existing SDL methodologies [1], [11], [14], [18], into which pattern-based configuring may therefore be integrated. As we demand the same techniques (OMT, MSC) for analysis and pattern description, we lower the gap between analysis and design models. By comparing OMT and MSC analysis diagrams with the *structure* and *message scenario* descriptions of the SDL patterns and by further examination of the patterns' *intent*, *semantic properties,* and *motivation,* protocol building blocks are to be selected.
- **adaptation**:
  as SDL patterns describe generic design solutions they have to be adapted before composition. Depending on the context specification into which the pattern instance shall be embedded, a suitable *version* of the pattern has to be determined. The chosen version additionally must be adapted by at least *renaming* the abstract identifiers (e.g., signals, parameters, variables) which is guided by the *syntactical embedding* rules. The result is a pattern instance ready for composition with the embedding SDL specification.
- **composition**:
  the pattern instance, i.e. the adapted SDL-fragment, finally has to be composed with the embedding SDL specification. This is done according to the *specialization* part of the syntactical embedding rules resulting in a specialization of the embedding SDL specification. Thereby, possible redefinitions of virtual constructs of the context specification are constrained by the syntactical embedding rules. An example for such a constraint would be that a redefined transition only adds a procedure call to the virtual transition and keeps the same otherwise.

The resulting SDL specification may be further refined in order to get an executable version. Therefore additional *redefinition* steps as far as allowed by the embedded SDL patterns may be necessary. Examples are the declaration of instantiated signals, sorts, or channels. The *semantic properties* of an embedded pattern may also impose additional assumptions on the context specification. They have to be checked and also taken into account in further development steps.

5. Pattern-based engineering of a resource reservation protocol

We have developed a real-time communication service running on PCs interconnected with a token-ring local area network. Thereby a system architecture evolved that comprises a *resource reservation protocol* for signalling quality of service requirements across the network and initiating admission control, which checks for the availability of network resources. The resource reservation service (also called *target service*) allows to establish and close unidirectional real-time connections between two communicating peers. For establishing a real-time connection, the calling user has to specify the required quality of service, including the expected amount of traffic load. Only the calling user is allowed to close a connection. More than one connection per node may be active at the same time. In the following we extract a development step in the pattern-based configuring of the resource reservation protocol.

5.1. Context specification and design problem

The resource reservation protocol was configured in an incremental way, where each step consisted of selecting, adapting, and composing predesigned protocol building blocks represented as SDL patterns. We started by configuring a protocol providing a subset of the target service based on a reliable underlying service. By incorporating further service requirements and/or relaxing the assumptions w.r.t. the underlying service, we finally obtained a complete solution in four development steps where we will focus on the last step in the following. For more information the reader is referred to [7].

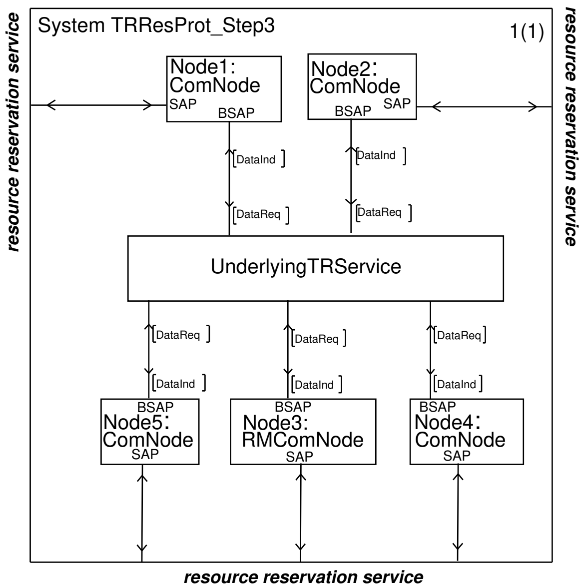


Figure 7. System architecture

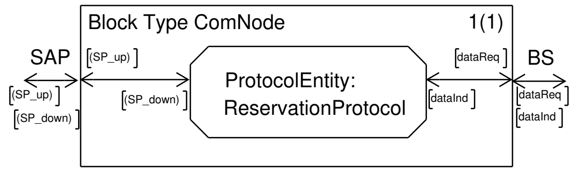


Figure 8. Communication subsystem

Starting point is the system architecture shown in Fig. 7 where we assume five nodes connected to an underlying token-ring network service. For each node, one SDL process *ProtocolEntity* is instantiated (see Fig. 8). ProtocolEntities are internally structured using the concept of SDL services (see Fig. 9). These SDL services are the result of applying SDL patterns through development steps 1-3. *LowerLayerInterfacing*, for instance, has resulted from the application of the *Codex* pattern during the second development step (Codex allows two or more entities to interact through an underlying service by means of service primitives, i.e. Codex essentially provides a translation from protocol data units to service primitives). The services *initConSetUp* and *initConTearDown* form the sender part of the protocol machine while *accConSetUp* and *accConTearDown* belong to the receiver part. Thereby initConSetUp in cooperation with accConSetUp are responsible

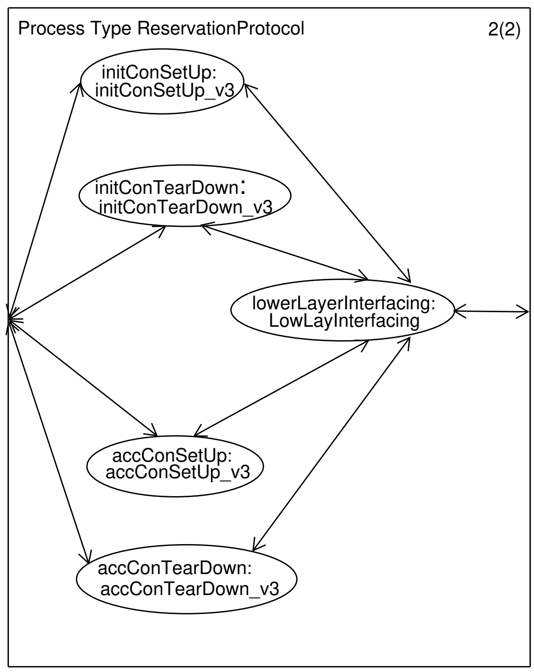


Figure 9. Protocol entity

for establishing a real-time connection. An open connection is closed by initConTearDown and accConTearDown, respectively. The behaviour for establishing a connection is partially shown in Fig. 10 and Fig. 11. When an application process requests a real-time connection a flow specification containing the quality of service requirements is provided through the service primitive *ConnectReq* (see Fig. 10). Subsequently, the calling protocol entity requests a connection from the called protocol entity (Fig. 10: procedure call *askCallee*) which itself requests necessary resources from a resource manager (Fig. 11: procedure call *askResManager*) if the receiver application is willing to accept the connection (this is checked by the procedure call *askReceiver* in Fig. 11). Note that the message exchange between calling and called protocol entity is mainly hidden inside the procedure askCallee, which is not shown here.

The given SDL specification (resulting from development steps 1-3) supports the full target service except the fact that only one instance of an outgoing and one instance of an incoming real-time connection per node is supported[1]. This substantially reduces the complexity of the resulting protocol entity. Nevertheless, it is indispensable for the offered service to support several connections per

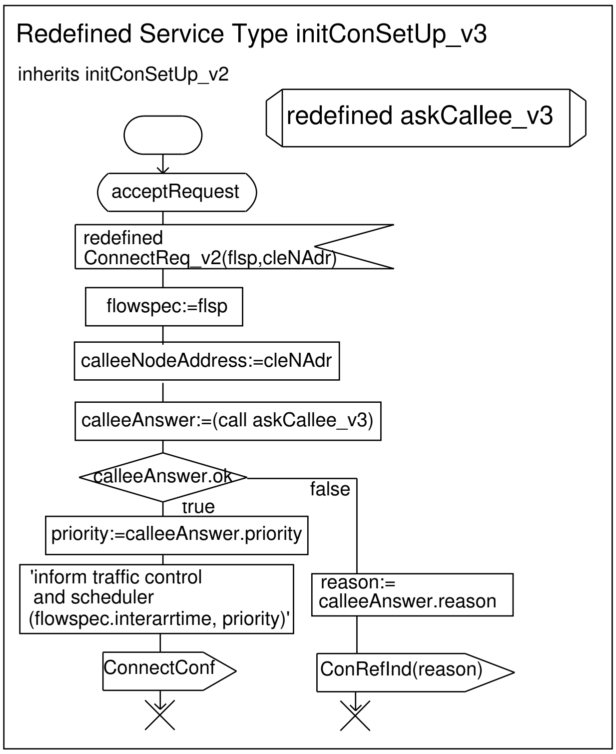


Figure 10. Partial protocol behaviour

[1]Note the several termination symbols in the service definitions of Fig. 10 and Fig. 11.

node and user even at the same time. This design problem is to be solved by the fourth development step.

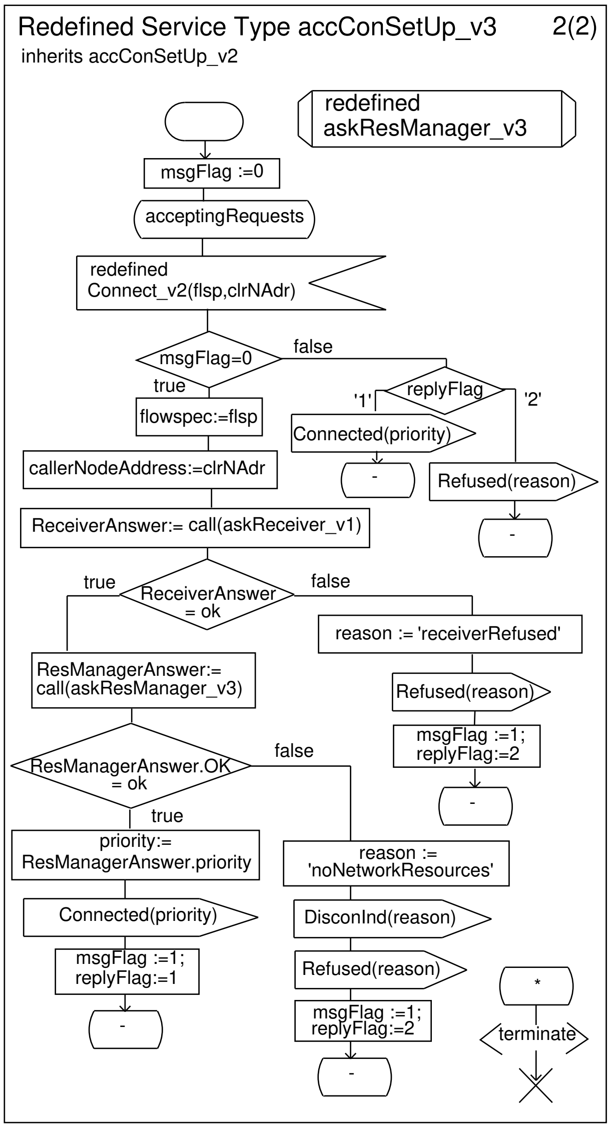
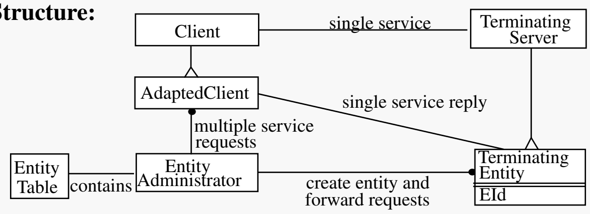


Figure 11. Partial protocol behaviour

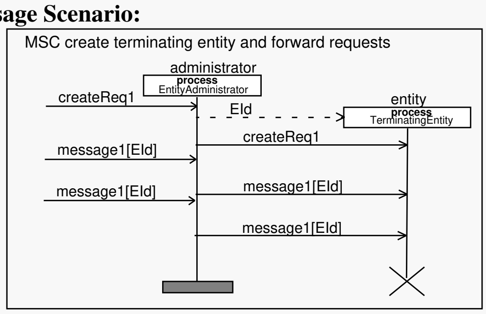
5.2. Selection of a suitable SDL pattern

An SDL pattern dealing with the design problem mentioned above is the *DynamicEntitySet* pattern (Fig. 12) as its intent suggests. The notations introduced by the pattern's object model will be applied throughout the rest of this section. From the perspective of the sending application process the calling protocol entity plays the role of a *TerminatingServer* while at the same time it is a *Client* for the called protocol entity. On the other hand the called protocol entity plays the role of a TerminatingServer for both

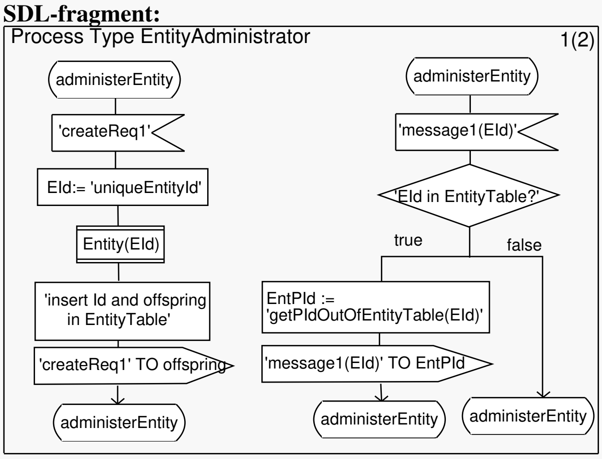
Name: DynamicEntitySet
Intent: The automaton *TerminatingServer* is capable to provide its service exactly one time and terminates afterwards. In order to offer the service several times (e.g., to more than one *Client)* the DynamicEntitySet pattern is introduced. For each service request a new server entity *TerminatingEntity* is dynamically created by *EntityAdministrator*. Thus *EntityAdministrator* acts as a proxy from the perspective of the clients, which forwards service requests to the corresponding server entity.
Motivation: If a communication subsystem supports multiple connections at the same time, each connection can be managed by a separate protocol entity, which is created and released dynamically. Each incoming message must be forwarded to the protocol entity to which it belongs.
Structure:



Message Scenario:



SDL-fragment:



Syntactical embedding
Specialization: transitions of *TerminatingServer* which send a signal back to the *client* are potential candidates for redefinition in order to inform the *client* about the local *EId*. The protocol engineer has to decide which ones are relevant or if the client is informed otherwise. In any case the *EId* will be used by the *AdaptedClient* when sending signals to the *TerminatingEntity*. Therefore all transitions which send a signal (except *createReq1*) to *TerminatingEntity* are redefined by adding the *EId* as signal parameter.

A process of type *EntityAdministrator* is added to the surrounding block diagram of *TerminatingServer.*
Renaming: *createReq1* and *message1* correspond with those messages the *client* sends to its *TerminatingServer*, where *createReq1* is the first message received. However, the concrete quantities of course have to be adapted.
Structural change: signal routes to *TerminatingServer* must be deleted and redirected to *EntityAdministrator*. The reference symbol for *TerminatingServer* must be replaced by a process set reference *Entity* with corresponding process type *TerminatingEntity* in the embedding block. *EntityAdministrator* must be connected with the process set *Entity* by a create line and additional signal routes for forwarding the messages.
Semantic properties:
If the assumption stated below holds, the same service as provided by TerminatingServer will be offered by each of the server entities of type TerminatingEntity, and each server entity will only receive messages belonging to it. The assumption is:

- The protocol designer takes care that the AdaptedClients are informed about the EIds of their corresponding server entity and always add this EId to the output signals, which are sent to the server entity (except createReq1)

Figure 12. Extract of an SDL pattern

the calling protocol entity and the receiving application process. Application of the DynamicEntitySet pattern means that each connection is managed by a separate set of protocol entities, which are created and released dynamically by the *EntityAdministrator*. Note that a calling and a called protocol entity together build one TerminatingServer so that each creation of a *TerminatingEntity* activates either the calling or the called part of it.

5.3. Adaptation and composition step

Adaptation of the DynamicEntitySet pattern and composition with the context specification is strictly prescribed by the SDL-fragment with its associated syntactical embedding rules. We will show how the pattern is applied to the context specification introduced in Section 5.1.

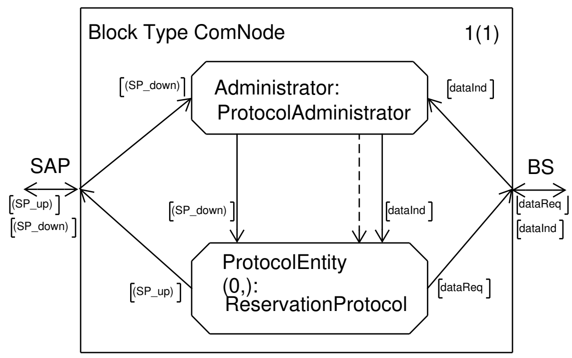


Figure 13. Completed communication subsystem

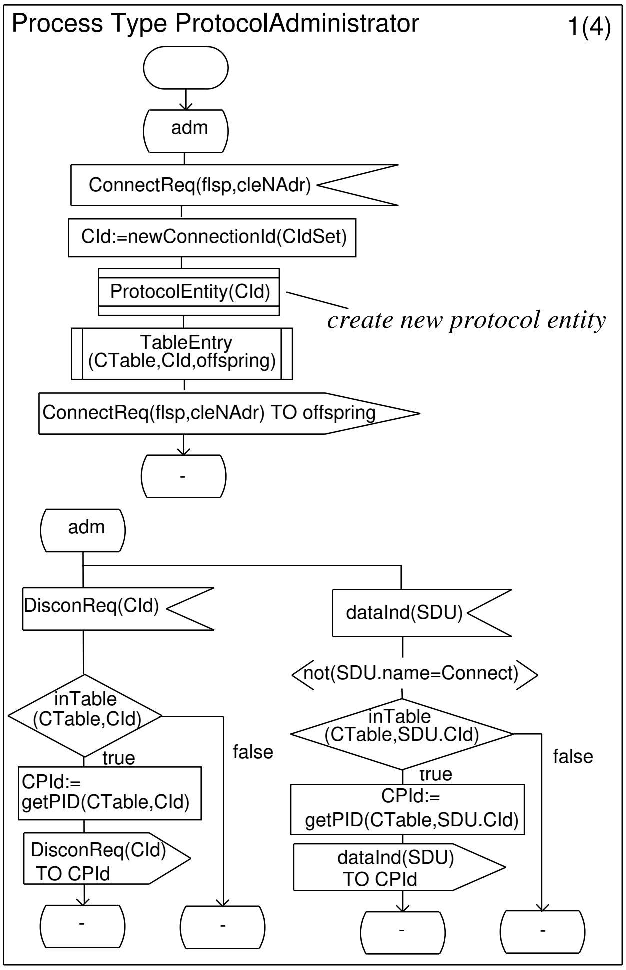


Figure 14. Behaviour of ProtocolAdministrator

The specialization part of the syntactical embedding rules requires that an administrator process is added to the context specification. Together with the structural changes additionally demanded by the syntactical embedding rules the communication subsystem of Fig. 8 is refined to the process structure of Fig. 13. Furthermore it is necessary to inform the clients of TerminatingEntities about the identifier (EId) allocated by the EntityAdministrator. The syntactical embedding rules suggest to use existing signals sent back to the clients for this purpose (see specialization part). In case of the calling protocol entity it is sufficient to let the *ConnectConf* service primitive (see Fig. 10) carry the identifier back to the application process. Therefore we specialize the signal ConnectConf by adding an additional signal parameter *CId,* which is assigned to the process instance at creation time by a global parameter (Fig. 14: create symbol). In case of the called protocol entity there are two clients to be notified of its identifier: the calling protocol entity as well as the receiving application process. Accordingly, the signals *Connected* and *ConnectInd* (hidden in the procedure *askReceiver*) of Fig. 11 are specialized in the same way as described above. Finally, the renaming part of the syntactical embedding rules guide the adaptation of the SDL-fragment, which describes the behaviour of the EntityAdministrator. The generic signal *createReq1* corresponds to *ConnectReq* in the resulting specification (Fig. 14) which is the first message received from the application process and initiates the creation of a new protocol entity. The generic signal *message1* of the pattern corresponds to those signals, which are only forwarded to their corresponding protocol entity. Fig. 14 shows two candidates, namely *DisconReq* (sent by the application process to close the connection) and *dataInd* (received from the peer via the underlying service). Signals which are directed to the receiver part of the ProtocolEntity are also handled by the ProtocolAdministrator but are not shown in Fig. 14. It is worth mentioning that the DynamicEntitySet pattern directs client requests to the right servers while the problem of relating server replies to the right clients is not addressed. Thus we decided to distinguish application processes by user provided identifiers passed at the service interface. However, this design decision is not covered by an SDL pattern and therefore not shown in the resulting specification. In case of replies from the called protocol entity, which are directed to the calling protocol entity, we could use the ProtocolAdministrator for identification. This only requires that apart from the sending application process also the called protocol entity is notified of the identifier assigned to the calling protocol entity. This design decision is also not covered by an SDL pattern and consequently not illustrated.

6. Summary and Outlook

We have presented an approach to the generic engineering of protocols where genericity is achieved by reuse of predesigned protocol building blocks. Our approach is based on the concept of design patterns, which we have specialized to communication protocols and combined with the description techniques SDL, MSC and OMT to formally specify protocol design patterns as well as rules for their instantiation and composition. To illustrate our approach, we have sketched the configuring of a resource reservation protocol using a pool of predesigned SDL patterns. When applying our approach, we have observed the following:

- As compared to an SDL specification of the same protocol that has been developed the usual way, the specification of the configured protocol is more readable, which is due to the more systematical design. Among other things, this further results in improved maintainability and less design errors, since the design decisions are well founded and documented.

- The integration of OMT and MSC with SDL for system analysis and design (as proposed by several existing SDL methodologies) has turned out to be a fruitful combination supporting pattern-based configuring.
- It has turned out that the patterns approach can be applied in an incremental way. This, too, improves maintainability due to a more systematical design. It would be interesting to see to what degree the application is commutative or reversible.
- Identification, investigation, and description of suitable protocol building blocks is a challenging and time consuming task, but provides great payoff for subsequent pattern-based protocol engineering. Note that the same experience has been made in other contexts where design patterns are used.

From these observations, we infer that our approach has the potential of reducing the effort for customizing and maintaining communication protocols, which seems to be a prerequisite for developing protocols that support applications in the best possible way. However, in order to draw a final conclusion, further experience will be needed. We are currently extending the pool of protocol building blocks and are using our approach for re-engineering of existing protocols including, for instance, ST2+ and IPv6. As good patterns mainly arise from practical experience we plan to perform more case studies, especially on field bus and ATM based protocols.

So far, implementations of configured protocols are hand-coded, because no code generator was available in our SDL tool environment. As we also plan to improve our approach concerning genericity of the development process we are currently integrating the SDT Cadvanced SDL compiler for automatic code generation.

The configuring of protocols classifies as a synthesis approach, meaning that systems are constructed from predesigned components such that by following certain rules, required system properties such as freedom of deadlocks, freedom of unspecified receptions, or conformance to the service specification can be guaranteed a priori. We see this as a fertile field for future research.

Acknowledgements. Special thanks go to our colleagues of the SFB 501 project for inspiring exchange of ideas as well as to the students implementing the case studies. The anonymous reviewers are acknowledged for their valuable comments and criticism.

References

[1] R. Bræk, Ø. Haugen. *Engineering Real Time Systems - An object-oriented methodology using SDL*. Prentice Hall, 1993

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

[4] B. Geppert, R. Gotzhein, F. Rößler. *Configuring Communication Protocols Using SDL Patterns*. SDL'97, 8th SDL Forum, Paris, France, 1997

[5] B. Geppert and F. Rößler. *Combining SDL and Pattern-based Design for the Customization of Communication Subsystems*. in [19], pp. 201-210

[6] B. Geppert and F. Rößler. *Automatic Configuration of Communication Subsystems - A Survey*. SFB 501 Report 17/96, University of Kaiserslautern, Germany

[7] B. Geppert and F. Rößler. *Pattern-based Configuring of a Customized Resource Reservation Protocol with SDL*. SFB 501 Report 19/96, University of Kaiserslautern, Germany

[8] D. Hogrefe. *Validation of SDL systems*. Computer Networks and ISDN Systems, Special Issue on SDL and MSC, 28 (1996)

[9] C. Peper, R. Gotzhein, and M. Kronenburg. *A Generic Approach to the Formal Specification of Requirements*, ICFEM'97, 1st IEEE International Conference on Formal Engineering Methods, Hiroshima, Japan, 1997

[10] T. Plagemann, B. Plattner, M. Vogt, and T. Walter. *Modules as Building Blocks for Protocol Configuration*. Proceedings of ICNP'93, International Conference on Network Protocols, San Francisco, 1993

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

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

[13] D.C. Schmidt, D.F. Box, and T. Suda. *ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment*. Concurrency Practice and Experience, Vol. 5, No. 4, 1993

[14] SDT 3.1 Methodology Guidelines: *The SOMT Method*, Telelogic, 1996

[15] A. Sinton and M. Crowther. *SDL-92 support for re-use in protocol system specifications - some early experience*. SDL'95 with MSC in CASE, Proceedings of the 7th SDL Forum, Oslo, Norway, 1995

[16] B. Stiller. *Flexible Protokollkonfiguration zur Unterstützung eines diensteintegrierenden Kommunikationssubsystems*. PhD thesis (in german), VDI-Verlag, 1994

[17] B. Stiller. *CogPiT - Configuration of Protocols in TIP*. Technical Report TR 368, University of Cambridge, Computer Laboratory, England, U.K., June 1995

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

[19] A. Wolisz, I. Schieferdecker, A. Rennoch (Ed.). *Formale Beschreibungstechniken für verteilte Systeme*, GI/ITG-Fachgespräch, 19.-20. Juni 1997, Berlin. GMD-Studien Nr. 315, ISBN 3-88457-315-2

[20] Z.100 (03/93) *CCITT Specification and Description Language (SDL)*, ITU-T, 1994

[21] Z.120 (10/96) *Message Sequence Chart (MSC)*, ITU-T, 1996