Using SDL Patterns for the Design of a CAN-based Communication Subsystem

Birgit Geppert*, Frank Rößler**, and Michael Schneider***
Computer Science Department, University of Kaiserslautern,
P.O. Box 3049, 67653 Kaiserslautern (Germany)
* Computer Networks Group, Email: geppert@informatik.uni-kl.de
** Computer Networks Group, Email: roessler@informatik.uni-kl.de
*** Integrated Communication Systems (ICSY), Email: mschneid@informatik.uni-kl.de

Abstract

SDL patterns are reusable software artifacts. They represent generic solutions for recurring design problems with SDL as applied design language. We have developed a construction set of protocol building blocks consisting of a pool of SDL patterns and an accompanying methodology for the incremental design of communication protocols. In this paper we present a case study on the design of communication protocols using SDL patterns. The case study is part of a more comprehensive project where a real-time communication subsystem was developed on top of a Controller Area Network (CAN) installation. We demonstrate how the protocols supporting user communication and certain management tasks were configured. Generally, it turned out that SDL-pattern based configuring of communication protocols yields more systematic designs, i.e. readability and maintainability is improved and less design errors occur, since the design decisions are well founded and documented.

1 Introduction

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 especially fruitful approach to software reuse [2] [5]. Contrary to the traditional paradigm of class and function libraries, which are 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.

In [6] [8] [9] we present an approach, which on principle adopts the patterns concept for the design of communication protocols. However, in order to assure high quality of the resulting communication subsystem we produce formal design specifications. Furthermore, we extend the approach, so that tool support is possible and pattern application can be validated. For this purpose pattern application rules are to be provided that can be checked and pattern semantics must be defined that captures behavioural changes of the embedding context. We attack this problem by applying SDL [15] as the target language for pattern instantiation. Thus, e.g., instantiation of a pattern can be defined by precise embedding rules in terms of the SDL syntax. Also we can precisely state assumptions for adequate pattern application as well as resulting properties on the embedding context in terms of the SDL semantics. This is a major improvement compared to conventional design patterns, which mainly rely on natural language based pattern description and

In Proc. of:
Formale Beschreibungstechniken für verteilte Systeme, 8. GI/ITG-Fachgespräch, Cottbus, Germany, 1998

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 a formal description technique with its formal syntax and semantics.

We have applied the construction set of protocol building blocks for a number of case studies (e.g., [7] [11]) including the development of a real-time communication subsystem on top of a Controller Area Network (CAN) installation [10].

The remainder of the paper is organized as follows: Section 2 shortly introduces the concepts underlying the proposed construction set of protocol building blocks including the SDL patterns concept and the corresponding process model. In Section 3 we present part of a case study on the design of a CAN-based communication subsystem that follows this configuration process and covers almost 100% of the resulting SDL control structure by SDL patterns. We summarize the results in Section 4.

2 The construction set of protocol building blocks

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. For further details the reader is referred to [6] [8] [9].

2.1 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 conventional design patterns: patterns capture solutions, which have evolved over time. 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, different 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 the application of a specific pattern is performed, under which assumptions this will be allowed, and what properties result for the embedding context.

For the specification of SDL patterns we have defined a standard description template with the main items 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. 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 generic identifiers or specialization of embedding design elements. Usually pattern semantics is not completely captured by an SDL-fragment. Due to language constraints this would otherwise result in an overspecification of the design solution and reduce 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 more detailed discussion of the SDL-pattern description template and a comparison to existing description templates of conventional design patterns 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. To further illustrate the functional scope of SDL patterns we shortly introduce some examples. 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].

- **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 sends a request and is blocked until receiving a reply. After receiving a request, the second automaton replies immediately.
- **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.
- **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. That is, Codex essentially provides a mapping from protocol data units to basic service primitives and vice versa.

2.2 Incremental 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 1). The configuration process suggests an incremental protocol design, 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 for 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 corre-

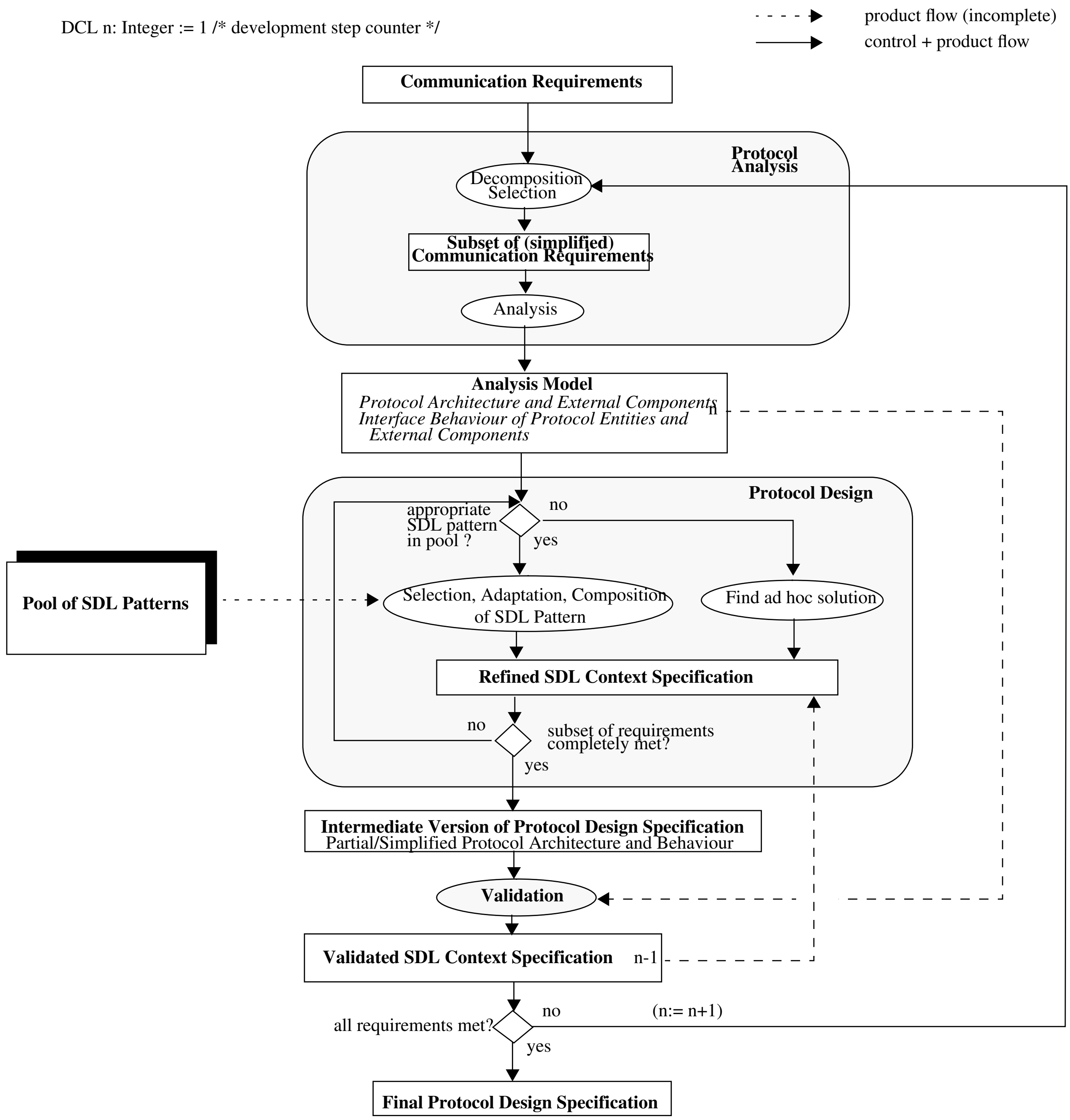

Figure 1 Configuration process (extract)

sponding simplification. Thereby each development step is divided into analysis, design, and validation and yields an executable SDL design specification (Figure 1). In the following the different 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 [13] or UML [1] object model and an MSC [16] use case model, which together identify participating objects and typical interaction scenarios.

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. In order to meet the new requirements a number of design steps are performed where individual SDL patterns are applied to the context specification. Note that for some design problems the pool of predefined protocol building blocks may not contain an adequate solution, so that an adhoc solution must be found. 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 c*ooperative usage*. As patterns represent generic design solutions the corresponding SDL-fragment has to be adapted in order to seamlessly fit the embedding context. The resulting pattern instance finally has to be composed with the embedding context. Adaptation and composition is instructed by *syntactical embedding rules* of the pattern.

The result of this design activity is an intermediate SDL design specification, which is subsequently validated against the analysis use case model. 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 or design steps is needed (not shown in Figure 1). 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.

3 Configuring a communication subsystem for CAN

Controller Area Network (CAN) is a field-bus standard originally developed for mobile systems. However, due to exceptional features such as high data rates and fault tolerance CAN became also established in other areas of the field-bus domain [3]. We have configured a real-time communication subsystem for CAN using SDL patterns.

3.1 Communication requirements

Communication between intelligent sensors, actuators and controller components imposes stringent requirements on timeliness and predictability. Thus CAN employs bitwise bus arbitration based on message identifiers to support deterministic media access. Message identifiers are to be interpreted as priorities and therefore a global, priority-driven scheduling of message transfers is realized. As timeliness and predictability strongly depend on worst case traffic load, field-bus installation is normally preceded by a configuration step where communicating peers as well as traffic loads are identified and priorities are assigned accordingly. This step is carried out prior to system operation and works fine for static settings. Nevertheless the operator or owner of a field-bus installation may change communicating devices or applications dynamically (e.g., in building automation). Though dynamic insertion and removal of communicating devices is generally supported by CAN, timeliness guarantees have to be concerned from scratch.

Therefore a communication subsystem shall be developed on top of CAN that performs admission control, priority assignment, and traffic policing *automatically during system operation*. This can be seen as a kind of “plug and play” functionality for timeliness guarantees, i.e., when, for instance, a new communicating device is inserted, the system automatically handles priority assignment with respect to timeliness requirements and under consideration of the already existing traffic load. If there are not enough network resources left, the system shall deny additional communicating peers.

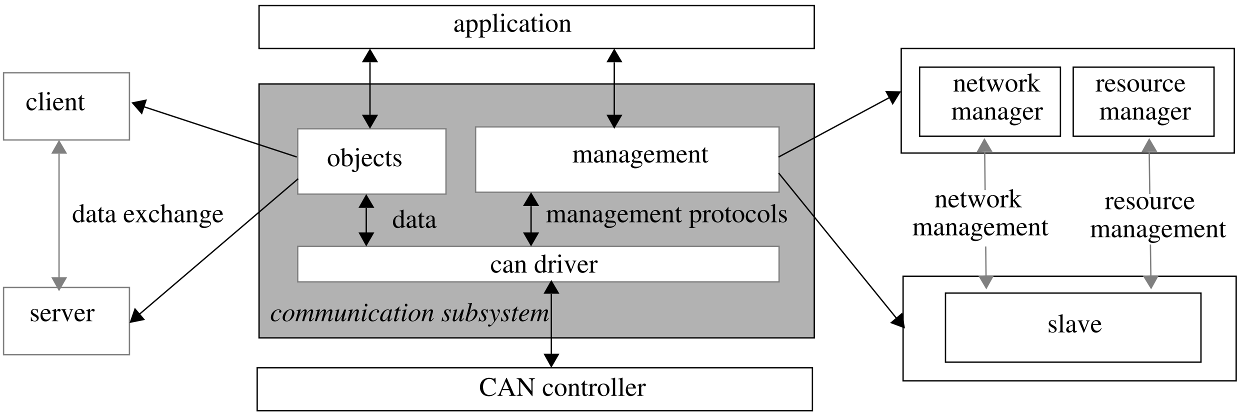

Figure 2 CAN subsystem architecture

3.2 Decomposition

The real-time CAN subsystem (Figure 2) is based on a generic QoS architectural framework [10]. This framework was also previously instantiated for a real-time communication subsystem on top of a Token-Ring network [7] [12]. Essentially, the subsystem for CAN contains protocol functionalities for initialization of CAN modules (network management), resource management, and object-based user communication (e.g., access to remote read-only-variables) [10]. With the additional simplification that communicating peers exchange protocol data units (PDU) directly, i.e., without using an underlying basic service, we get the decomposition of Figure 3. The chronological order of the performed development steps is also illustrated.

3.3 Development steps

Development step I: Initializing CAN modules, directly connected peers
Before communication can start, CAN modules have to be initialized. This is illustrated by the MSC in Figure 4. The interaction scenario between the network manager and the participating slaves is structured into different phases. First some slave starts the identification phase by sending a PDU_connect message to the network manager and waits for a corresponding reply (PDU_prep message). In order to avoid collisions on the CAN bus the PDU_connect messages contain no data. This necessitates an identification of the slaves, which is done by a binary search. The network manager broadcasts a PDU_iden message containing an interval of valid module identifiers (MId). Only those slaves with a valid module identifier are allowed to reply. After a

step	basic service	protocol functionalities
I	directly connected	initialization of CAN modules
II		resource reservation
III		object-based user communication
IV	CAN basic service	

Figure 3 Decomposition and chronological order of selection

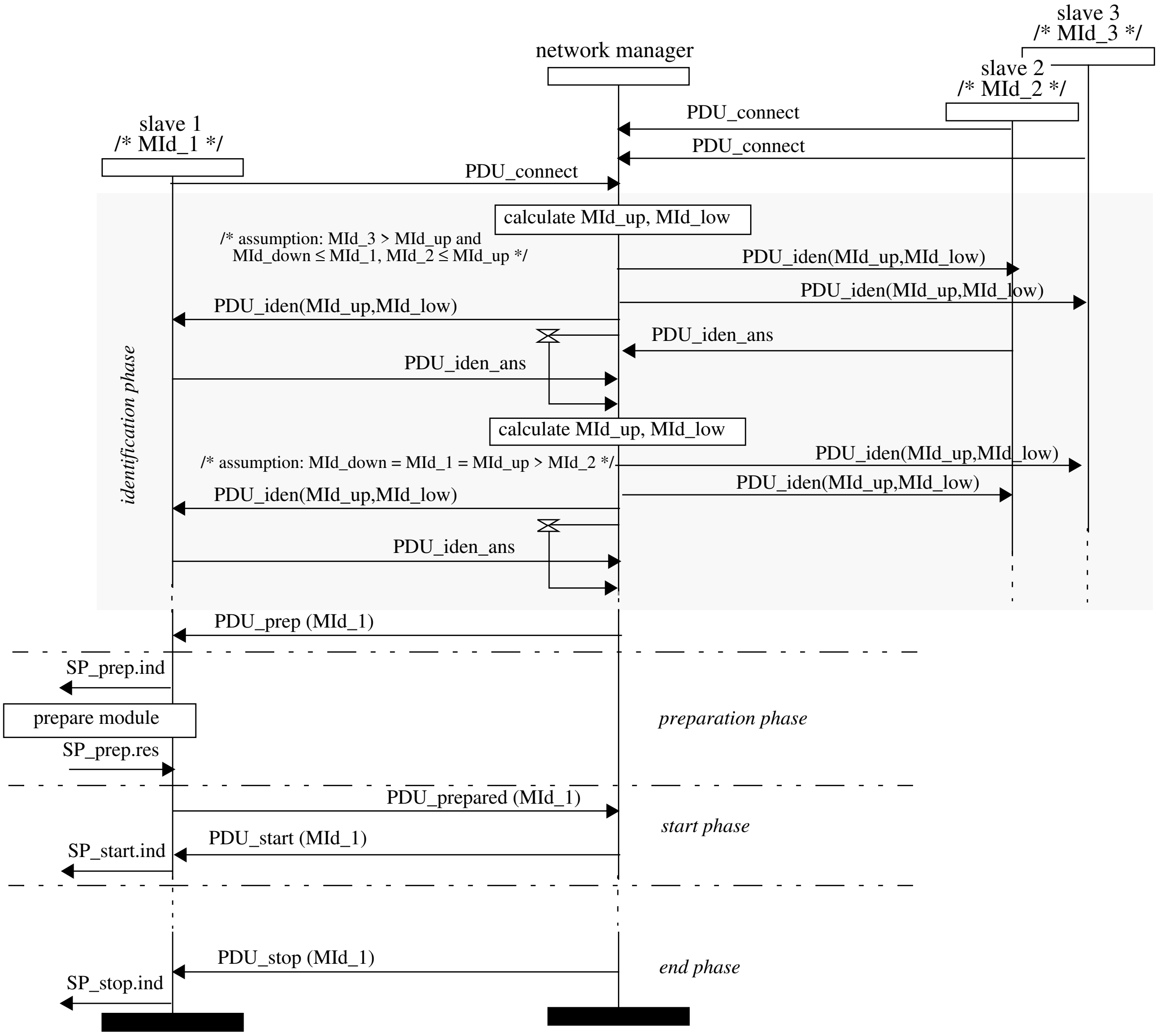

Figure 4 Interaction scenario of development step I (extract)

constant time interval all replies are expected to be received. By iterated bisection a slave is uniquely identified. This is illustrated in the grey shaded part of Figure 4 (slave_1 is identified). The search is repeated until all announced slaves are identified (not shown in Figure 4).

A slave may enter the preparation phase when a PDU_prep message with the own MId is received. The application is informed and communication is prepared (e.g., programs or parameters are downloaded, not illustrated in Figure 4). For this activity CAN identifiers are employed that are shared among all slaves. In order to avoid collisions on the CAN bus their use is coordinated by sequencing the preparation phases of the individual slaves. Thus the network manager sends the *PDU_prep* message for the next slave only after finishing the current preparation (not illustrated in Figure 4).

If preparation is finished for all slaves, communication can start. Therefore each slave is informed by a *PDU_start* message. The slaves must finish communication when receiving a *PDU_stop* message.

The whole scenario can be implemented in SDL by cooperative usage of certain interaction patterns such as SendReceive, BlockingRequestReply or SingleRequestMultipleReplies (Section

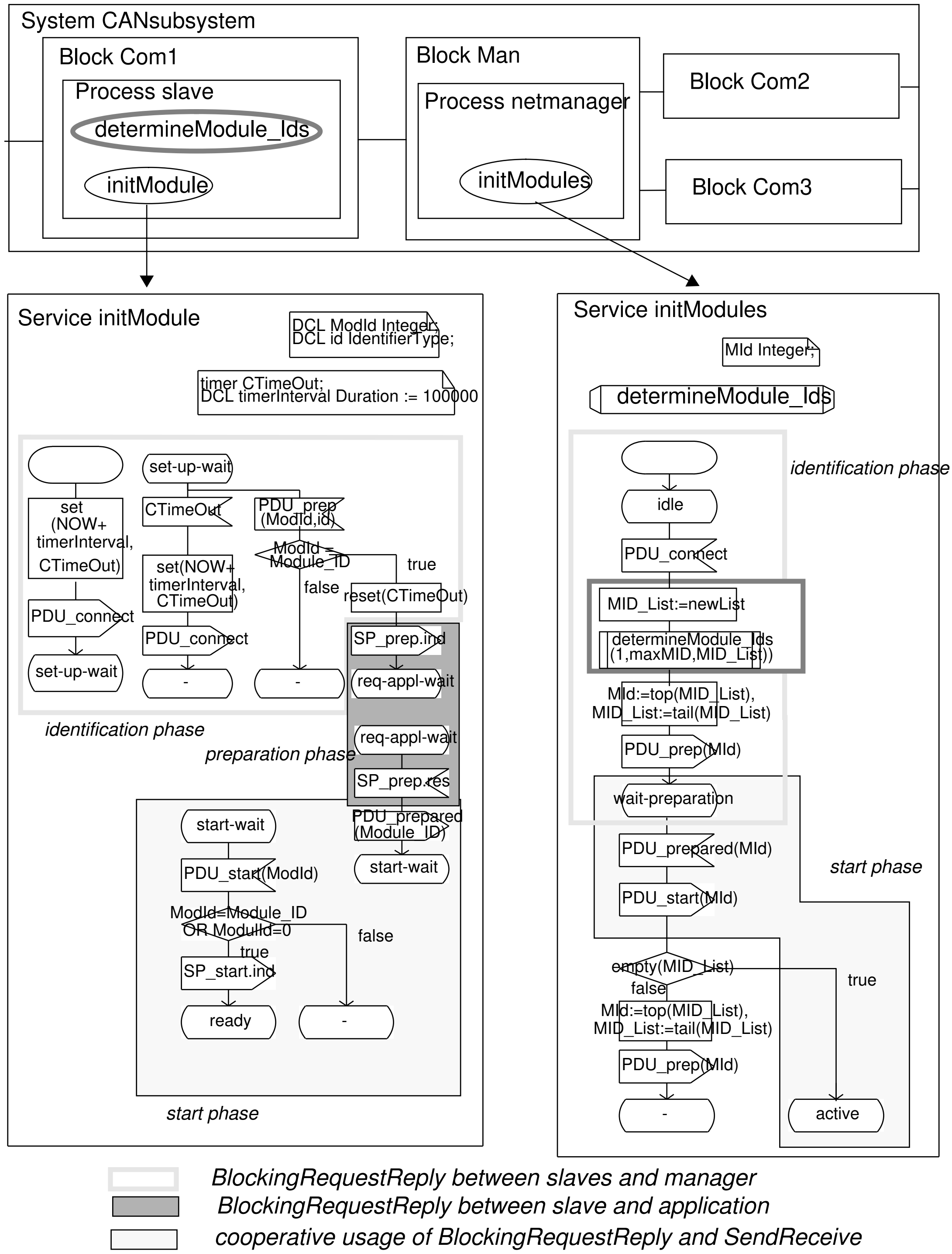

Figure 5 Design specification of development step I (extract)

2.1). Figure 5 shows an extract of the resulting design specification. The different phases can easily be identified and are illustrated using different shadings. For the identification phase (excluding the determination of Module-Ids), e.g., the BlockingRequestReply pattern was applied. A slave sends a request (*PDU_connect*) and blocks until receiving a reply (*PDU_prep*). The manager replies when identification is finished. As illustrated in Figure 5 most of the control structure is covered by SDL patterns.

Development step II: Resource reservation, directly connected peers
Before communication can take place a certain amount of network resources has to be reserved in order to guarantee quality of service (QoS). Figure 6 illustrates the interaction between a reservation slave and the central resource manager. The application asks its local reservation slave to establish a connection with certain QoS and is informed about the result by a corresponding reply (*SP_id.conf*). Embedded into this interaction is a two-way handshake between the slave and the resource manager. The slave sends a *PDU_id_reserve* message with a flow specification containing all relevant QoS parameters. The resource manager then decides if the connection will be established and assigns a unique identifier to the communication object that will be used for data transfer. This identifier is returned within the *PDU_id_reserved* message. The whole interaction is realized by a chained BlockingRequestReply [7] (Figure 7).

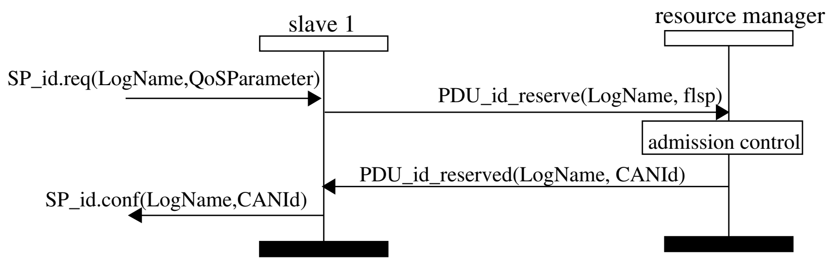

Figure 6 Interaction scenario of development step II (extract)

Development step III: Object-based communication, directly connected peers
The CAN subsystem supports object-based user communication. That is, a connection is defined by common access to the same data object. Access to remote objects is realized according to the Client/Server paradigm, where the interaction patterns from Section 2.1 are applied again. For further information the reader is referred to [14].

Development step IV: Introducing the CAN basic service
Finally, we replace the direct connections between the communicating peers by the basic service provided by CAN. We therefore have to map PDUs to service primitives offered by CAN. A pattern dealing with that problem is Codex (Section 2.1). Its application results in a new SDL service for each communication node. See [14] for further details.

4 Conclusion

We have presented a case study on the SDL-pattern based configuring of communication protocols. SDL patterns classify 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.

As compared to SDL specifications that have been developed the usual way, SDL-pattern based configuring leads to a more systematic 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 [11]. It is worth mentioning, that a very large portion (almost 100 % of the control structure) of the final specification resulted from SDL patterns, where each pattern has been

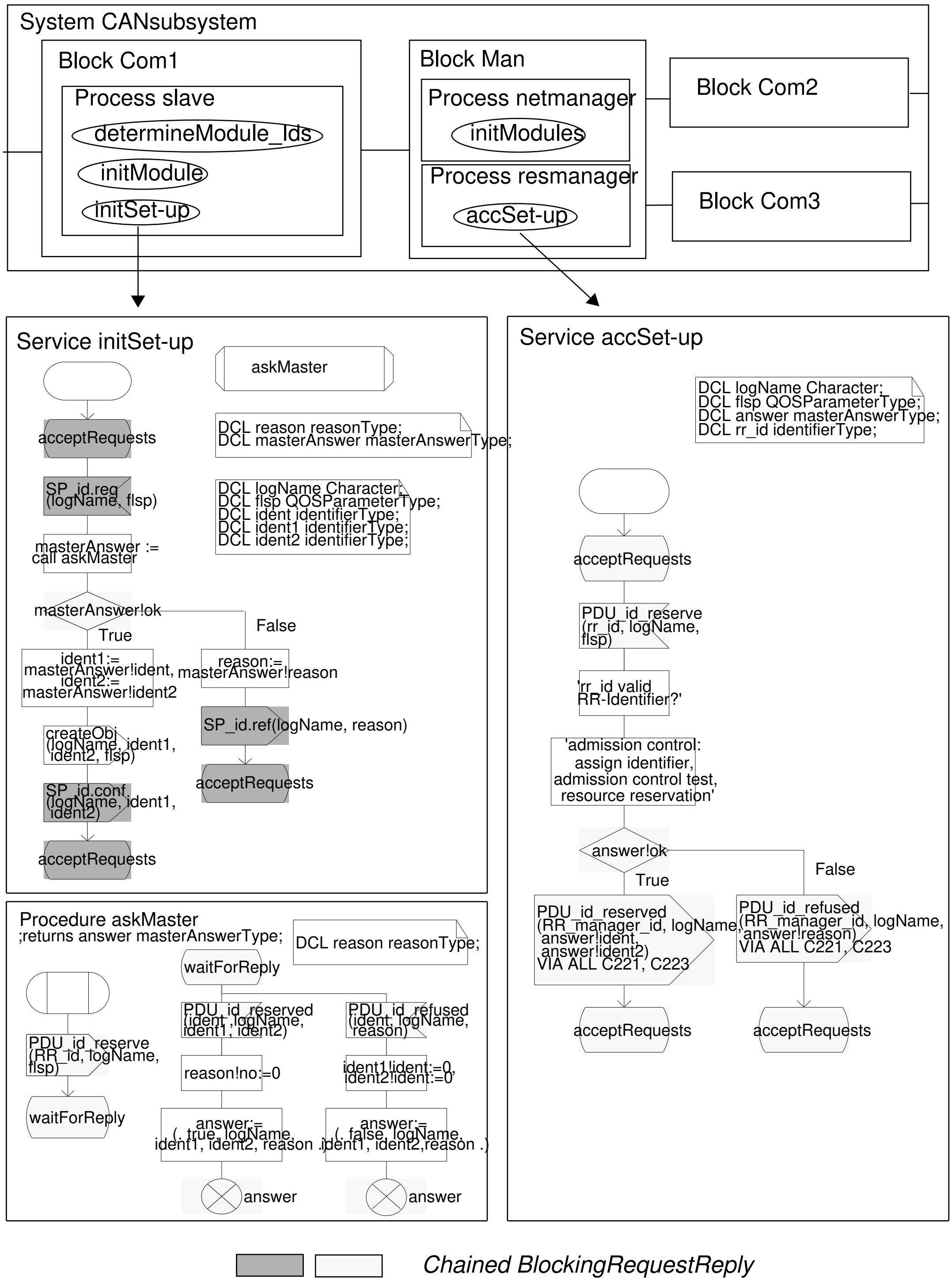

Figure 7 Design specification of development step II (extract)

applied several times. This provides some evidence that the predesigned patterns have been well chosen. From these observations we infer that our approach has the potential of substantially 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, this statement needs further validation. In [4] we present an approach for the experimental evaluation of empirical properties of SDL patterns.

5 References

[1] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language*, Version 1.0, Rational Software Corporation, 1997

[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] K. Etschberger, *CAN Controller-Area-Network - Basics, Protocols, Building Blocks, Applications* (in german), Hanser Verlag, 1994

[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*, SDL'97 Time for Testing - SDL, MSC and Trends, Proceedings of the 8th SDL Forum, Paris/Evry, France, 1997

[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] B. Geppert and F. Rößler, *Combining SDL and Pattern-based Design for the Customization of Communication Subsystems*, in: A. Wolisz, I. Schieferdecker, A. Rennoch (Ed.): Formale Beschreibungstechniken für verteilte Systeme, GI/ITG-Fachgespräch, 19.-20. Juni 1997, Berlin, (GMD-Studien Nr. 315)

[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] F. Rößler and B. Geppert, *Applying Quality of Service Architectures to the Field-Bus Domain*, Proceedings of the 2nd IEEE International Workshop on Factory Communication Systems, WFCS'97, Barcelona, Spain, 1997

[11] F. Rößler, B. Geppert, and Ph. Schaible, *Re-Engineering of the Internet Stream Protocol ST2+ with Formalized Design Patterns*, accepted for the 5th IEEE International Conference on Software Reuse, ICSR'98, Victoria, Canada, 1998

[12] F. Rößler, A. Kühlmeyer, *Implementing Real-Time Communication on a Token-Ring Network*, Proceedings of the 6th Open Workshop on High Speed Networks, Stuttgart, October, 1997

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

[14] Michael Schneider, *Design and Implementation of a real-time communication subsystem for CAN,* diploma thesis (in german), University of Kaiserslautern, 1998

[15] Z.100 *CCITT Specification and Description Language (SDL)*, ITU-T, 1996

[16] Z.120 *Message Sequence Chart (MSC)*, ITU-T, 1996