Joint International Conference on
Formal Description Techniques & Protocol Specification, Testing, and Verification,
FORTE XI / PSTV XIII '98, Paris, France, 1998

SDL-Pattern based Development of a Communication Subsystem for CAN *

B. Geppert, A. Kühlmeyer, F. Rößler, and M. Schneider
University of Kaiserslautern, Computer Science Department
P.O. Box 3049, 67653 Kaiserslautern, Germany
{geppert, akuehlm, roessler, schneider}@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 specification of communication protocols based on 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. We also applied the SDT Cadvanced code generator to implement the resulting design specification on a PC cluster. 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 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. There are several advantages commonly attributed to 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.

* This work is supported by the German Science Foundation (DFG) as part of the Sonderforschungsbereich SFB 501 *Development of Large Systems with Generic Methods.*

In [6] [7] we present the SDL-pattern approach that integrates the design patterns concept with SDL [11]. Generally speaking, SDL patterns describe generic solutions for recurring design problems, which can be customized for a particular context. While conventional design patterns are specified independently from a possible design language, it is assumed that the target language for SDL pattern instantiation is SDL. Thereby we benefit from the formal basis provided by SDL, so that SDL patterns actually characterize as formalized design patterns. 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 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. However, we do not deal with formalizing design patterns in general. Instead of formalizing reuse concepts we aim to support reusability within the formal methods area.

In this paper, we present a case study on the specification of communication protocols based on 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) [3] installation. We demonstrate how the protocols supporting user communication and certain management tasks were configured. We also applied the SDT Cadvanced code generator to implement the resulting design specification on a PC cluster running under the real-time operating system QNX.

The remainder of the paper is organized as follows: Section 2 introduces the SDL-pattern approach. In Section 3 we demonstrate how a real working communication subsystem for CAN was configured using SDL patterns. The development steps for (semi-)automatic code generation of the resulting SDL design are outlined in Section 4. Finally, we summarize the results in Section 5.

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 SDL patterns and configure them to a customized, formal protocol specification.

2.1. SDL patterns

An SDL pattern describes a generic solution for a recurring, context-specific design problem. It is assumed that the target language for pattern instantiation is SDL. Though the concept is not restricted to a specific application domain, we are mainly concerned with communication protocols.

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 when applying the pattern. SDL fragments represent context invariant parts of the 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 behavioral changes of the embedding context. Though semantic properties are currently stated in natural language, it is possible to express them precisely in a temporal logic. Also, restrictions on the *refinement* of pattern instances are specified in order to prevent a pattern's intent from being destroyed by subsequent development steps. A 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 behavior 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.

- **BlockingRequestReply:** The *BlockingRequestReply* pattern introduces a confirmed interaction (two-way handshake) between two given automata. After a trigger from the embedding context, the first automaton sends a request and is blocked until receiving a reply. The request is eventually received by the second automaton, which replies and finally releases the first automaton from its waiting state.
- **DynamicEntitySet:** Consider a given server entity that provides its service exactly one time and terminates thereafter. In order to offer this service several times (e.g., to more than one client), the *DynamicEntitySet* pattern is applied. For each client a new server entity is dynamically created by a special entity administrator. Subsequently, the administrator acts as a proxy from the perspective of the clients, which forwards service requests to the corresponding server entity.
- **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. 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 corresponding simplification. Thereby each development step divides into analysis, design, and validation and yields an executable SDL design specification. 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 UML [1] object model and an MSC [12] use case model, which together identify participating objects and typical interaction scenarios.

The analysis model is implemented 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. 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 design steps are performed that apply separate SDL patterns 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 ad hoc 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. This is instructed by the *renaming* parts of the *syntactical embedding rules* and the *refinement rules.* The resulting pattern instance finally has to be composed with the embedding context, which is prescribed by the *composition* part of the syntactical embedding rules and also by the refinement rules of embedding pattern instances.

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.

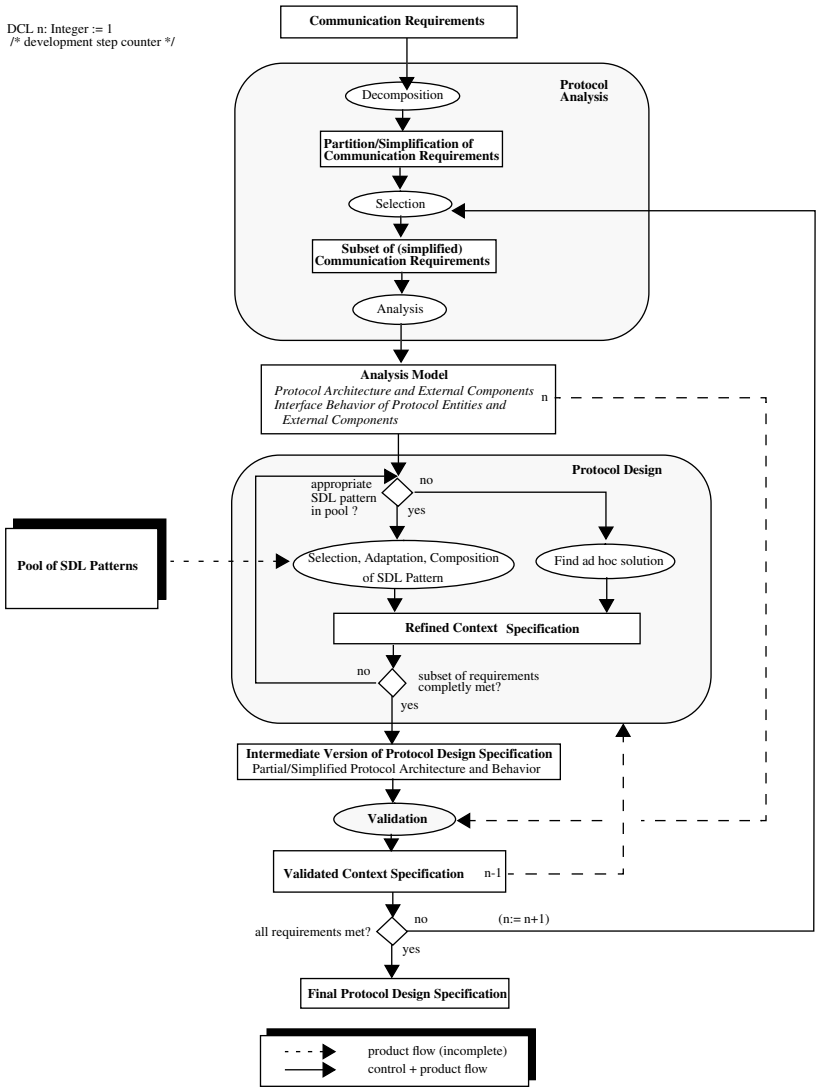


Figure 1. Configuration process (excerpt)

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. Note that the presentation below follows the configuration process introduced in Section 2.2.

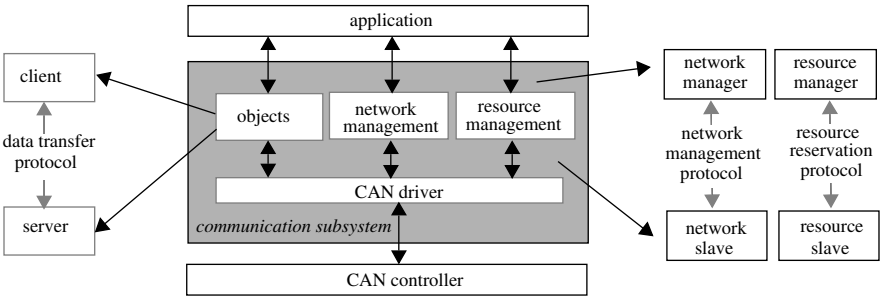


Figure 2. CAN subsystem architecture

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 normally precedes 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.

3.2. Decomposition

The real-time CAN subsystem (Figure 2) is based on a generic QoS architectural framework [8]. This framework was also previously instantiated for a real-time communication subsystem on top of a Token-Ring network [9]. 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). 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

3.3.1. Development step I: Initializing CAN modules, directly connected peers. Before communication can start, some CAN modules must be initialized (e.g., programs or parameters are downloaded). This process would consume many CAN identifiers that are permanently assigned to individual modules. In order to save resources, it is suggested to share CAN identifiers for initialization purposes. As a consequence, a special control protocol for medium access is needed, which was specified in the first development step. The protocol is implemented by a network slave on each CAN module and a central network manager.

The interaction between the network manager and a network slave is structured into different phases. When the slave requests access to the CAN bus, the corresponding request message contains no data in order to avoid collisions. Thus, the network manager must identify the slave before it can be addressed. This is done by a binary search that broadcasts identification messages, which contain an interval of valid module identifiers. Only those slaves with a valid module identifier are allowed to reply.

A slave enters the next phase, when an initialization message with its own module identifier is received. As already mentioned, the initialization protocol employs CAN identifiers that are shared among all slaves. In order to avoid collisions on the CAN bus their use is coordinated by sequencing the initialization phases of individual slaves.

The interaction scenario is implemented by cooperative application of several SDL patterns such as *SendReceive*, *BlockingRequestReply* or *SingleRequestMultipleReplies*. For instance, the exchange of the request and initialization message follows the *BlockingRequestReply* pattern, while the binary search for identification of network slaves applies *SingleRequestMultipleReplies*. Because of space limitations we skip further details here.

3.3.2. Development step II: Resource reservation, directly connected peers. Before communication can take place a certain amount of network resources must be reserved in order to guarantee quality of service (QoS). Figure 4 illustrates the interaction between a resource slave and the central resource manager. The application asks its local resource slave to establish a connection with certain QoS (*SP_id.req*) and is

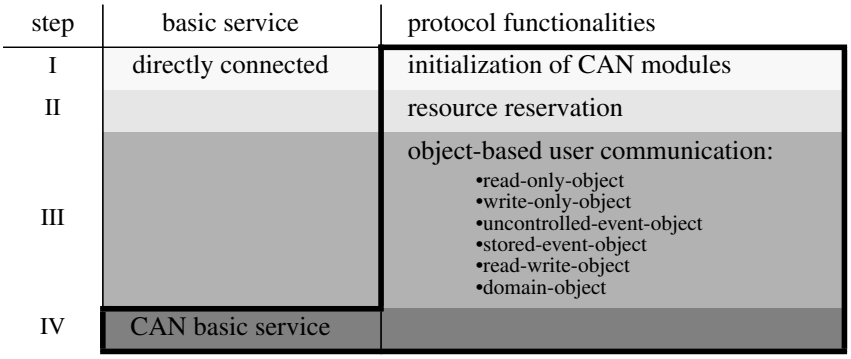


Figure 3. Decomposition and chronological order of selection

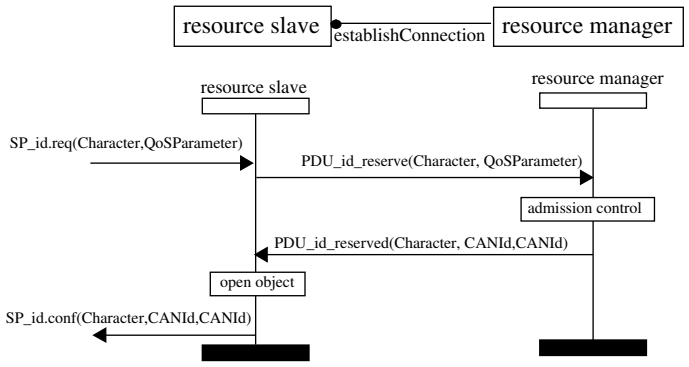


Figure 4. Analysis model of development step II (excerpt)

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 unique CAN identifiers to the communication object that will be used for data transfer. The identifiers are returned within the *PDU_id_reserved* message. The interaction is realized by a cascaded application of the *BlockingRequest-Reply* pattern. The shaded parts of Figure 5 determine the renamed SDL fragments of the embedded pattern instances.

3.3.3. Development step III: Object-based communication, directly connected. The CAN subsystem supports object-based user communication. That is, a connection is established by common access to the same communication object, which, e.g., represents a physical quantity or event of a technical process. Access to remote objects is realized according to the Client/Server paradigm. Thereby the server side is responsible for updating the logical communication object with its physical counterpart, while the client side offers remote access. We defined six types of communication objects with different access characteristics, such as read-only or write-only objects (Figure 6). An object consists of a certain number of clients and servers, which are usually located on different nodes. In case of a read-write object (RWobject) the object consists of one server and a maximum of one clients (Figure 6). The MSC diagrams describe interactions between server and client for opening, reading, writing, and closing such an object. In the following, we will only consider RWobjects. It should be noted that communication based on other objects can be specified independently from each other.

Design of object-based communication divides into two major steps. At first, it is assumed that no more than one server and one client of each type of communication object exists per node. Figure 7 shows a possible architecture. Two processes are added to the context specification that resulted from development step II (Figure 5), one for the client and one for the server of a RWobject. The client is assumed to be located on node *Com1* and the corresponding server on node *Com3*. For definition of the behavior SDL patterns are applied. In the following, we will concentrate on the

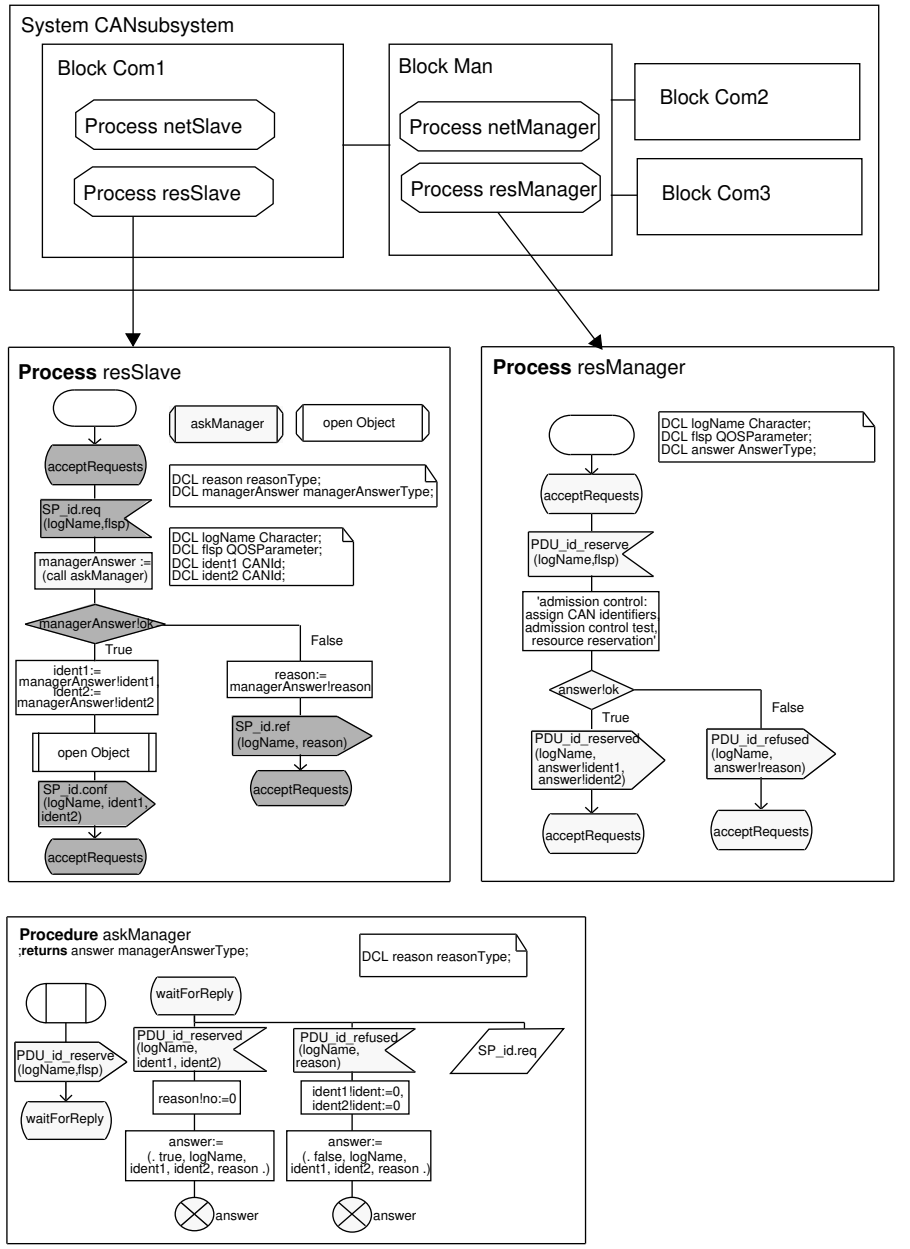


Figure 5. Design specification of development step II (excerpt)

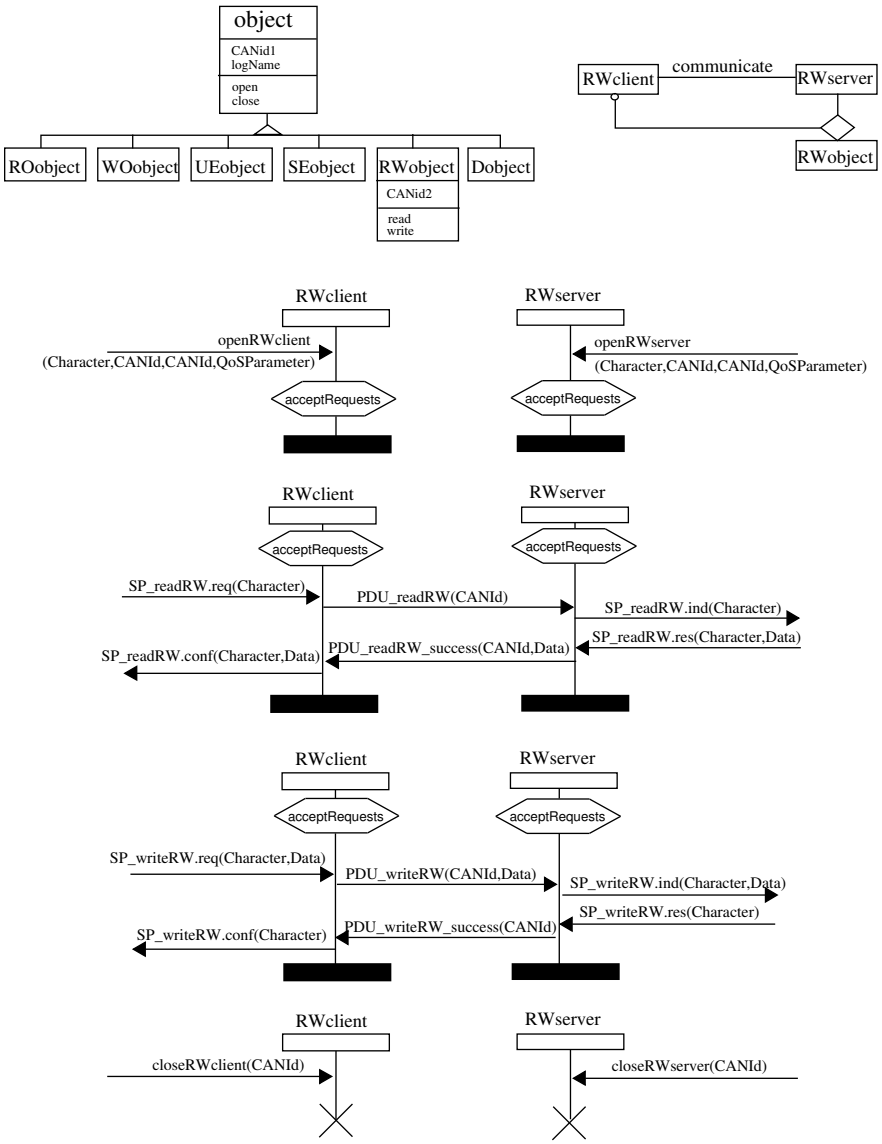


Figure 6. Analysis model of development step III (excerpt)

interactions for reading the object. Write access is specified accordingly. It follows from the corresponding MSC of Figure 6 that the interactions actually describe three cascaded two-way handshakes between the application and *RWclient*, *RWclient* and *RWserver*, and finally between *RWserver* and the application. An SDL pattern generating a two-way handshake is *BlockingRequestReply*. We applied the pattern three times, resulting in the chained *BlockingRequestReply* instances of Figure 7. For the first pattern instance, which is shaded only the replier is specified, because the request-

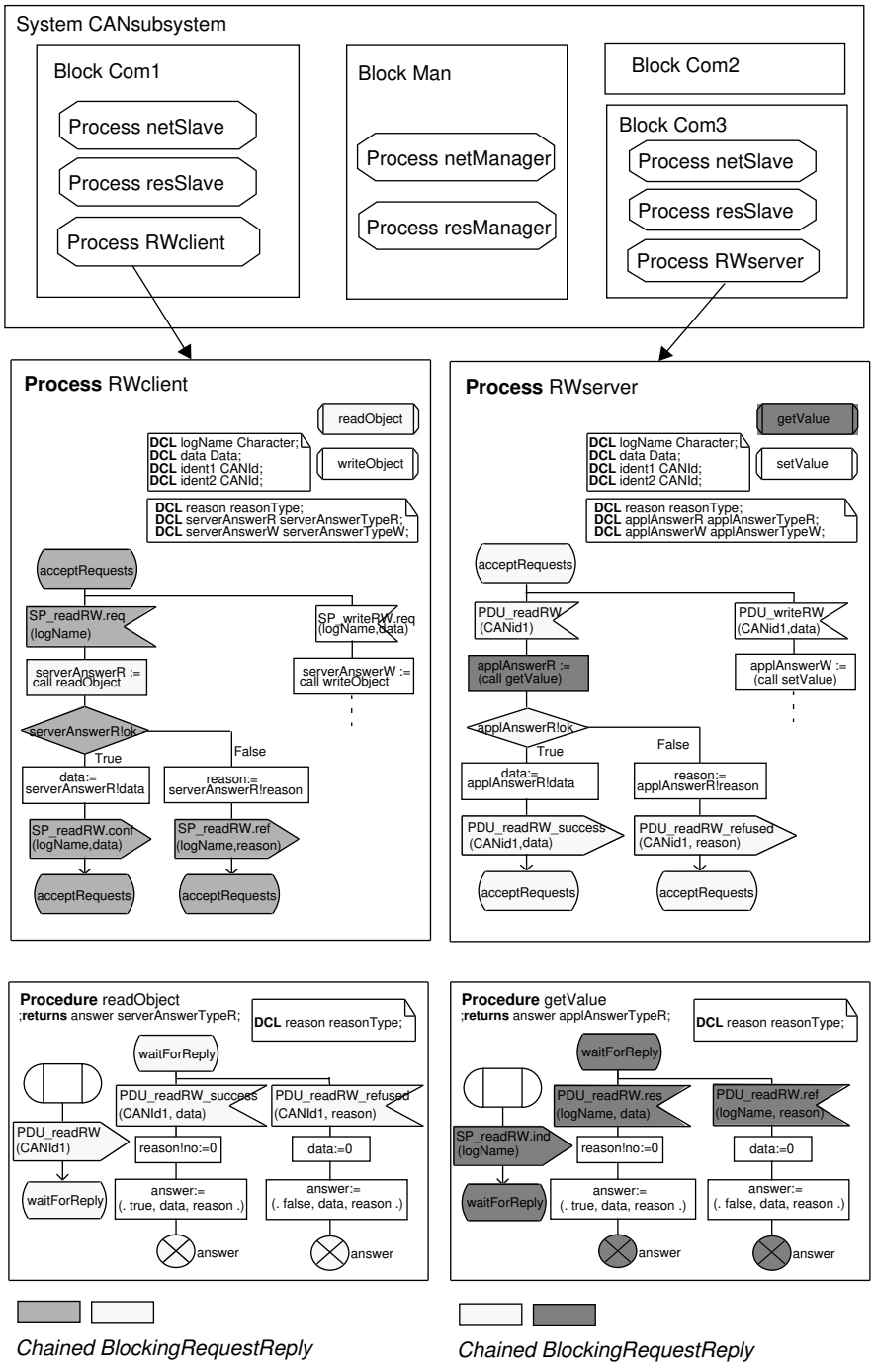
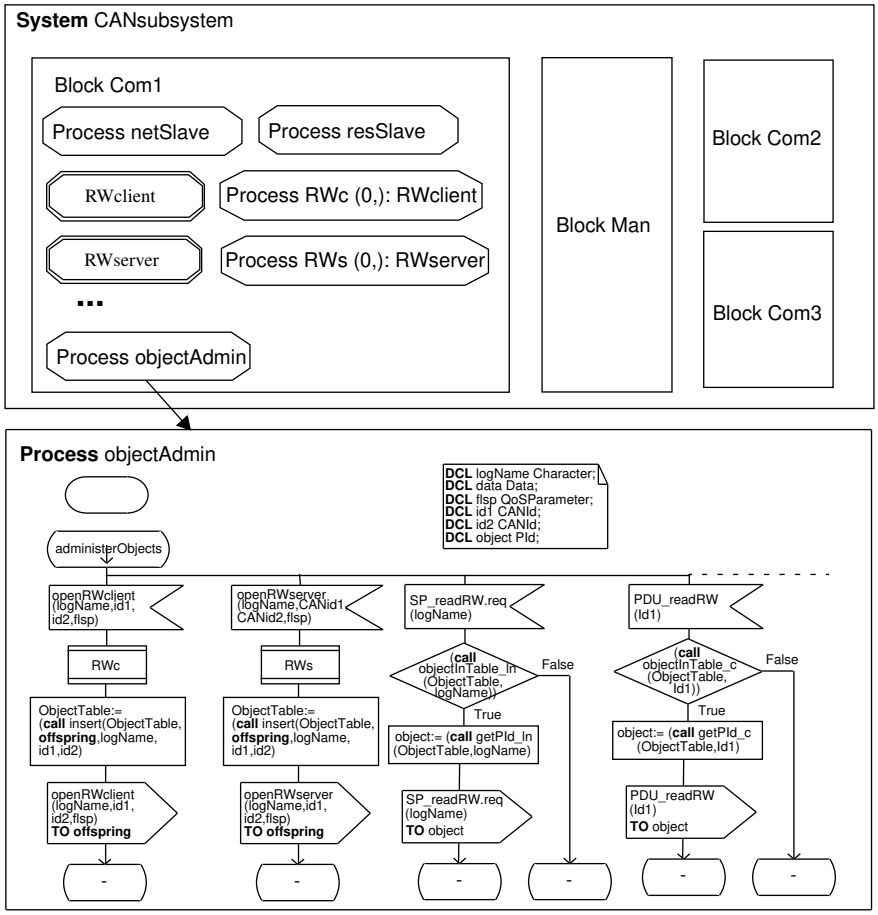


Figure 7. Design specification of development step III: Part 1 (excerpt)



DynamicEntitySet

Figure 8. Design specification of development step III: Part 2 (excerpt)

ing application is part of the environment. The requester of the second pattern instance (shaded ▭) is embedded in the first one and defined by the procedure *readObject*. The corresponding replier is located in *RWserver* and also shaded ▭. The requester of the third *BlockingRequestReply* instance is given by the procedure *getValue* (shaded ▭). The corresponding replier is part of the application on the server side and therefore not explicitly specified.

In the second major design step we relax the assumption that no more than one server or client of each object type be allowed per node. In order to realize this we applied the *DynamicEntitySet* pattern. According to the syntactical embedding rules the server and client processes are replaced by process sets with corresponding process types. For instance, the process *RWserver* is replaced by a process type of the same name and a corresponding process set *RWs* (Figure 8). Additionally, an administrator

process *objectAdmin* is introduced, which is responsible for dynamically creating the processes, whenever a new object is opened. Subsequently, *objectAdmin* forwards incoming signals to the right instance of the process set. As a consequence, certain signal routes must be re-routed, i.e., disconnected from the client and server processes and connected to the administrator, which itself must be connected to the process sets. For reasons of readability, channels and signal routes are not shown in Figure 8.

3.3.4. Development step IV: Introducing the CAN basic service. Finally, we replace the direct connections between the communicating peers by the CAN basic service. 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 process for each communication node. We skip further details here.

4. Generating the implementation

The design specification must finally be transformed into executable code. Note that this is actually not part of the configuration process discussed in Section 2.2, but is added to cover the whole development cycle. Implementation can be done manually or using a code generator such as the SDT Cadvanced Compiler. For our case study, we followed the implementation activity of the SOMT method [10] belonging to the SDT development tool set. That is, we first partitioned the SDL design to define the different run-time modules according to the light integration for the real-time operating system QNX. Environment functions are to be implemented, which are responsible for interfacing the generated code with its physical environment. Next the implementation must be generated using the SDT Cadvanced compiler with its run-time library for QNX. As the development tool set does not run under QNX but is implemented on our SUN cluster running Solaris, the generated C files must be downloaded and compiled on the QNX-PCs.

4.1. Partitioning and light integration

Cadvanced allows two types of integration into other operating systems, which mainly differ in the way SDL processes are mapped to operating system (OS) processes. The tight integration creates an OS process for every SDL process, while the light integration creates a single OS process that handles all SDL processes of the SDL partition. We decided to use light integration, because at the time of implementation only the light integration supported SDL services, which our specification heavily relied on.

We prepared two different partitions. One that covers the protocol functionalities of a slave node (where only network and resource slaves reside, Figure 2) and one for the manager node (where additionally the network and resource managers reside, Figure 2). The hardware-specific low-level CAN driver was hand-implemented. For performance reasons, we also decided to integrate the communication objects (Figure 2) into the low-level driver.

4.2. Interfacing with the environment

The generated modules communicate with the environment by the environment functions xInEnv and xOutEnv. xInEnv provides the functionality to somehow receive messages from the environment and transmit the messages as signals to the generated module. xOutEnv is the counterpart of xInEnv. It forwards SDL signals from the specification to the environment.

4.2.1. Interface to the low-level driver. The low-level driver provides the functionality to transmit messages of up to 8 bytes (the maximum length of a CAN frame). A second part of the interface relates to the object-based user communication. We focus on the basic routines for sending and receiving simple messages, because the generated modules only communicate this way. The send routine has three parameters: the CAN identifier of the message type, a pointer to a buffer, where the message is stored, and the size of the message. The send procedure blocks the calling process until the message is delivered. The receive function has the same parameters, while it is non-blocking.

4.2.2. Interface to the applications. The SDL design defines signals to indicate the initialization phase and to start communication. To realize this we provide two functions that block the application until the corresponding signal is received. Additionally, we define a function that calls the local reservation slave via QNX specific IPC mechanisms in order to establish a communication object.

4.3. Environment functions

For every signal that enters or leaves the specification, we define a C data type named by the signal. For example, for the signal PDU_iden_rem, specified as

```
SIGNAL PDU_iden_rem( integer, integer )
```

we define the data type

```
typedef struct {
      char sigtype;
      short int1, int2;
} PDU_iden_rem_T;
```

`sigtype` is a unique number to identify the signals. Instances of these types are then transferred via the CAN bus.

The environment function *xOutEnv*. There are two types of signals: the signals that have to be transferred via the CAN bus and the signals that realize the communication to the application.

The signals to the low-level driver are implemented by simply allocating an instance of the corresponding data type, filling it with the supplied arguments of the SDL signal, and sending it by calling the low-level send function. For example, the code for the PDU_iden_rem signal looks as follows:

```
...
} else if( (*S)->NameNode == PDU_iden_rem ) {
      PDU_iden_rem_T data;
      data.sigtype = SIG_PDU_iden_rem;
      data.int1 = ((yPDef_PDU_iden_rem *)(*S))->Param2;
      data.int2 = ((yPDef_PDU_iden_rem *)(*S))->Param3;
      send( ((yPDef_PDU_iden_rem *)(*S))->Param1.ident,
            /* ID */ &data, sizeof( data ) );
} else if ...
```

`SIG_PDU_iden_rem` is a unique number for this signal. Expressions such as `((yPDef_PDU_iden_rem *)(*S))->Param1` are the Cadvanced standard method to provide signal parameters to the environment.

Communication with the application is realized differently: an application requests a service via IPC. When the request is handled, the protocol entity first stores the application's process identifier. If the answer is available, the stored process identifier will then be used to reply the answer to the right requester.

The environment function *xInEnv*. Again there are two types of signals: requests from applications and messages from the low-level driver.

To receive signals from the low-level driver, one has to poll the driver by calling its receive function. For every message type there normally exists a different CAN identifier. So, the xInEnv function calls the receive function sequentially for every possible CAN identifier of incoming messages. A typical piece of code looks as follows:

```
if( receive( ID_PDU_iden_rem, data, sizeof(PDU_iden_rem_T)) > 0 ) {
     S = xGetSignal( PDU_iden_rem, xNotDefPId, xEnv );
     ((yPDef_PDU_iden_rem *)(S))->Param1.ident = ID_PDU_iden_rem;
     ((yPDef_PDU_iden_rem *)(S))->Param2 =
         ((PDU_iden_rem_T *)(data))->int1;
     ((yPDef_PDU_iden_rem *)(S))->Param3 =
         ((PDU_iden_rem_T *)(data))->int2;
     SDL_Output( S, xSigPrioPar(xdefaultPrioSignal) (xIdNode *)0 );
}
```

The code inside the **`if`** block is the Cadvanced standard method for creating a signal instance, defining the parameters, and sending it to the generated module. The code for receiving a request from the application is basically the same, only that the message is received via the QNX specific "Creceive" call and the signals are differentiated by the first byte (the signal identifier sigtype).

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 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 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. It is worth mentioning, that a very large portion of the final specification resulted from SDL patterns, where each pattern has been 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. Though these statements result from experience of several test projects, we intend to validate them in a stronger sense. In [4] we present an approach for the experimental evaluation of empirical properties of SDL patterns.

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, *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

[8] 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

[9] 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

[10] Telelogic, *TAU 3.3 Methodology Guidelines – Part1: The SOMT Method*, Telelogic, Sweden, 1998

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

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