Pattern-based Configuring of a Customized Resource Reservation Protocol with SDL

Birgit Geppert, Frank Rößler

SFB 501 Report 19/96

Pattern-based Configuring of a Customized Resource Reservation Protocol with SDL

Birgit Geppert, Frank Rößler

{geppert, roessler}@informatik.uni-kl.de

Report 19/96

Sonderforschungsbereich 501

Computer Networks Group
Computer Science Department
University of Kaiserslautern
P.O. Box 3049
67653 Kaiserslautern
Germany

Pattern-based Configuring of a Customized Resource Reservation Protocol with SDL

Birgit Geppert, Frank Rößler

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

Abstract

Due to the large variety of modern applications and evolving network technologies, a small number of general-purpose protocol stacks will no longer be sufficient. Rather, customization of communication protocols will play a major role. In this paper, we present an approach that has the potential to substantially reduce the effort for designing customized protocols. Our approach is based on the concept of design patterns, which is well-established in object oriented software development. We specialize this concept to communication protocols, and - in addition - use formal description techniques (FDTs) to specify protocol design patterns as well as rules for their instantiation and composition. The FDTs of our choice are SDL-92 and MSCs, which offer suitable language support. We propose an SDL pattern description template and relate pattern-based configuring of communication protocols to existing SDL methodologies. Particular SDL patterns and the configuring of a customized resource reservation protocol are presented in detail.

1 Introduction

Today's communication systems are typically structured into several layers, where each layer realizes a defined set of protocol functionalities. These functionalities have been carefully chosen such that a wide range of applications can be supported, which has led to the development of a small number of general-purpose protocol stacks. However, due to increasing communication demands as found in many modern applications, the communication services provided by these protocol stacks are not always adequate. In particular, varying demands on throughput and delay as well as on delay jitter, synchronization and multicasting are not well supported by existing protocol stacks. 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 that may actually not be needed by a given application, for instance, the use of error control mechanisms, which leads to reduced performance.

To improve this situation, different communication architectures as well as a new generation of general-purpose protocols are currently being developed. It is expected that in order to increase flexibility and to support applications in the best possible way, also customization of special-purpose communication protocols will play a major role. Here, the configuring of protocols from reusable components (called *protocol building blocks* in this paper) seems to be a promising way to reduce the additional development effort.

Several approaches to the configuring of protocols have been reported in the literature. Early research focused on the identification and collection of suitable protocol components by reverse

engineering of existing transport and network protocols. A protocol implementation was then automatically configured from a subset of these components. Well-known projects in this area are F-CCS [30], [34], Da CaPo [15], [16] and ADAPTIVE [23], [24], [25] (see [6] for an overview). These approaches have in common that protocol *implementations* are configured. As a major drawback, the use of implementation languages prevents the resulting communication system from being verified, which is further complicated by the configuring of protocols during connection establishment. Also, the extension of the component pool appears to be difficult in these approaches because the knowledge about composition principles is not explicitly described. Here, the use of formal description techniques allowing an abstract, unique specification of protocol components and component interactions seems to be mandatory.

The reuse of predesigned solutions for recurring design problems is of major concern in object oriented software development in general. During the past few years, *design patterns* have emerged as an especially fruitful concept from other well-known approaches such as *frameworks*, or *toolkits* (in the sense of object oriented libraries) [5], [3], [17]. Early experience in reuse of protocol specifications with SDL has been reported in [28], where a protocol building block was designed as a reusable library class, however, according to the authors, with limited success.

In this report, we present a new approach for designing customized protocols. Our approach is based on the concept of *design patterns*, which we specialize to communication protocols. In addition, we use SDL-92 [35] and MSCs [36] to formally specify protocol design patterns and rules for their instantiation and composition. An important advantage of our approach is that the configuring leads to formal specifications of communication protocols, which may then be used for validation and implementation purposes. Due to the importance of currently developing SDL methodologies we discuss how pattern-based configuring relates to existing SDL methodologies, in particular, to the SDL methodology framework [18].

The remainder of this report is organized as follows: in Section 2, we propose an advanced SDL pattern description template and discuss the process of pattern employment. Additionally pattern-based configuring is incorporated into the recently proposed SDL methodology framework [18]. In Section 3, a customized resource reservation protocol, which is part of the realization of a real-time communication service based on a conventional token ring network, is configured. Thus particular SDL patterns will be presented and applied according to the process model of Section 2. We conclude with experiences and an outlook in Section 4.

2 Pattern-based protocol configuring

Protocol configuring is a promising way to cope with the enormous number of possible customized protocols. Actually we suggest to provide a pool of reusable and formally specified protocol building blocks from which the protocol designer may 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 protocol (Figure 1). During further stages of the development process the resulting design specification will finally be mapped to a conforming implementation (however, we will not consider implementation issues in this report).

The protocol building blocks are represented by SDL patterns describing a generic design solution for a communication specific problem. This is similar to the well-known design patterns concept, as introduced by the Gang-of-Four [5]. SDL patterns comprise an SDL-fragment as the syntactical part of the design solution, which will be embedded into the final protocol specification, and additional items, that ensure proper pattern application. A detailed presentation of our SDL pattern description template and a comparison to existing design pattern description templates is given in Section 2.1.

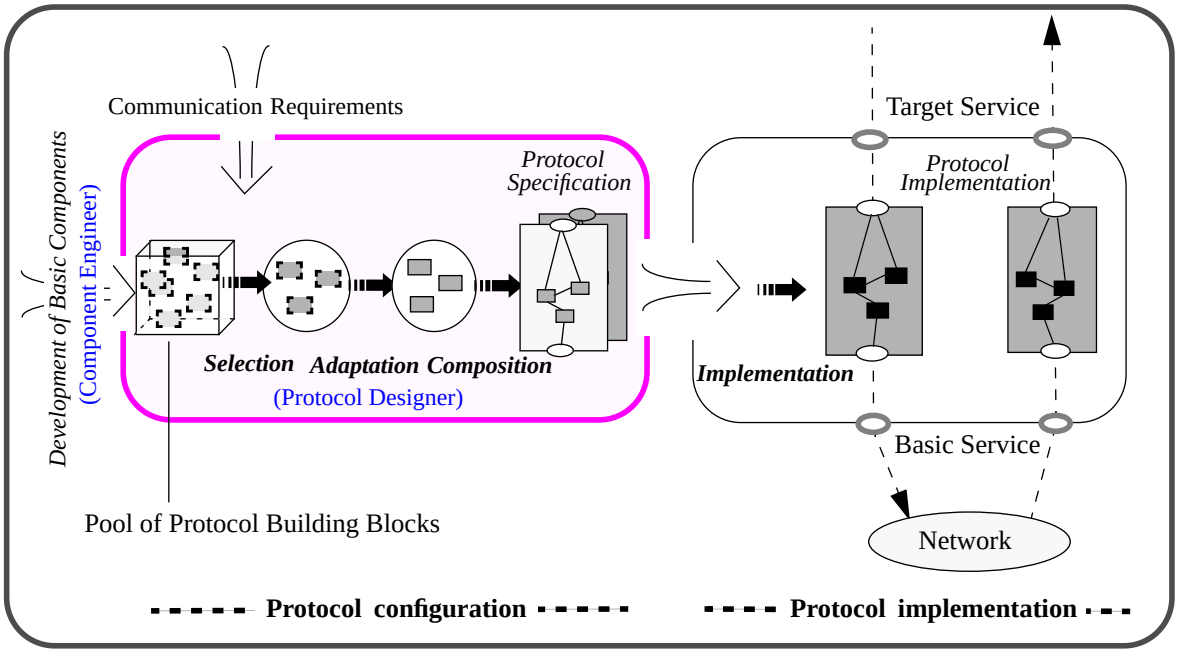


Fig. 1: configuring and implementing communication protocols

The configuration process cursory sketched above is capable to be developed into a detailed process model. Therefore we combine existing SDL design methodologies and specialize them to the domain of communication protocols. This will be explained in Section 2.2.

2.1 SDL patterns

An *SDL pattern* describes a generic solution for a context-specific design problem from the domain of communication protocols. It is assumed that the target language for pattern instantiations is SDL-92. Thus the pattern description comprises syntactical rules for pattern application as well as semantic properties defining the patterns intent more precisely. This definition of SDL pattern is similar to those of conventional design patterns used in object oriented software development:

- *„Design Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.“ [5]*
- *„A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.“ [3]*

The differences between design patterns and SDL patterns are that we choose a particular application domain (communication protocols), and that we combine the advantages of the formal description technique (FDT) SDL with the patterns‘ concept. Instead of specifying and applying the patterns rather informally, SDL offers the possibility to specify what the application of a specific pattern precisely means, under which assumptions this will be allowed, and what the consequences are. Here we are in line with the advocates of design pattern formalization inside the design patterns community, though we are even more rigorous by demanding the use of an FDT for this purpose. As a consequence, the description of SDL patterns differs in some ways from design patterns in [5], [3]. We propose an SDL pattern description template with the items listed below and relate it to the existing pattern description templates of [5], [3]. As already mentioned, instantiations of this template are called SDL patterns which, itself instantiated, form the constituent parts of an SDL protocol specification.

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 Typical scenarios describing the interactions between the involved objects 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:

- 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. Here, „specialization“ is meant in the sense of specialization of SDL types as defined in [35]. This could, for instance, result in
    - the *addition* of new transitions or SDL services
    - the *redefinition* of existing virtual types or transitions.

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. The semantic properties define the patterns intent more precisely.

Redefinition An embedded pattern instance can be further redefined, e.g. by the embedding of another SDL-fragment in subsequent development steps. Redefinitions compatible with the patterns intent and semantic properties 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.

Gamma et al.	Buschmann et al.	
Pattern Name and Classification	**Name**	Name and short description of intent
Intent		
Also Known As	**Also Known As**	Other well-known names
Motivation	**Example**	Real-world example illustrating the design problem
Applicability	**Context**	Situations in which the pattern should/ should not be applied
	Problem	General description of the design problem and the offered solution (detailed intent)
	Solution	
Structure	**Structure**	Graphical representation of participating objects (OMT) and their interactions.
Participant		
Collaborations	**Dynamics**	
Implementation	**Implementation**	Guidelines for implementation, including code fragments in C++, Smalltalk, ...
Sample Code		
	Example Resolved	Description of other important aspects of the given example not addressed so far and other possible variants or specializations of the pattern
	Variants	
Known Uses	**Known Uses**	Systems in which the pattern has been applied
Consequences	**Consequences**	Benefits and results of using the pattern.
Related Patterns	**See Also**	List of similar patterns

Table 1:

The description template for SDL patterns and existing templates for design patterns (see Table 1) have some items in common: *name*, *intent*, *motivation*, *structure*, and *message scenario*. For SDL patterns, these items are specialized to the communication systems domain. Thus participating objects typically include protocol entities, protocol functions, service users or service providers. Interactions between them can be described by Message Sequence Charts (MSC), with the additional advantage to perform MSC based validation. Furthermore, several SDL methodologies suggest to use OMT [21] and/or MSC for analysis (see e.g. [18], [29], [32]). To fit with these methodologies, we bridge the semantic gap between analysis and design models by employing OMT and MSC for pattern descriptions as well (see also Section 2.2).

Different from [5] and [3], SDL patterns are part of a dedicated pool of protocol building blocks and have a formal foundation. Thus an SDL pattern can be related to other pool components by specifying their *cooperative usage*. This is strongly supported by restriction to design problems of a certain domain. The formal foundation results from the use of the standardized FDT SDL, where, for instance, the *syntactical embedding* of the pattern, i.e. its integration into a given SDL specification, can be specified uniquely in terms of the SDL syntax. Furthermore, the formal semantics of SDL supports the formalization of a patterns intent by *semantic properties*. This includes both desired properties and necessary assumptions which have to be fulfilled to ensure the intended use of the pattern. This is important for validation of the resulting communication protocol. The possibility to simulate the design specification between consecutive development steps or before implementation is another advantage of the SDL based approach. Undetected design errors can therefore be identified in early stages of the development process.

Items not already incorporated into the SDL pattern template, for instance „Also Known As“, „Known Uses“ or „Related Patterns“, may be added in future versions. However, it seems more important to further improve the template as far as pattern interactions or system validation are concerned.

2.2 A process model for pattern-based configuring

In the following, a process model for protocol configuring is proposed, defining different steps to be followed and intermediate descriptions to be produced. We employed this model for the configuration of a resource reservation protocol (Section 3). As already mentioned, the proposed process model results from a combination and adaptation of different SDL design methodologies known from the literature.

Similar to [9], we propose a use case driven design. However, to describe the interactions between the involved objects we prefer Message Sequence Charts (MSC), which is a standardized description technique and often used in combination with SDL. Additionally, instead of producing an object-oriented implementation we only aim at an SDL design specification of the communication protocol, which may be used for automatic code generation, though.

In [29] the SOMT (SDL-oriented Object Modelling Technique) method is presented which will be supported by the SDT tool set [27]. The idea we are following is to combine the Object Modelling Technique (OMT) [21] with SDL and MSC for analysis and design. The SDL specification can then be transformed into executable software by the use of the SDT code generator. The main difference to our approach is that we focus on reuse of predesigned building blocks.

Another approach combining OMT with SDL is the INSYDE methodology [32]. INSYDE (INtegrated methods for evolving SYstem DEsign) aims at combining object-orientation and formal description techniques for developing prototypes of hybrid systems. Therefore the methodology integrates not only OMT with SDL, but also with VHDL. The development process for an SDL specification consists of analysis in OMT, system design in OMT* (a restricted, formal variant of OMT, see [10]), and detailed design in SDL. Translation rules from OMT* to SDL are given in [31]. As indicated in [8], iterative design and reuse of analysis and design models from existing systems is of major concern for the methodology‘s acceptance in industry. It is planned to integrate these missing features in the INSYDE methodology.

The methodology presented in [2] is part of the SISU project, a Norwegian technology transfer program with the intent to improve productivity and quality of companies that develop real-time systems. The engineering process is partitioned into requirements specification, design, and implementation. For requirements specification a new notation called SOON (SISU object-oriented notation) is introduced which is used in combination with natural language and MSC.

In [18] an SDL methodology framework is presented, where the engineering process consists of five activities, namely documentation, analysis, draft design, formalisation, and implementation. Each activity is characterized by its process step and its input and output documents. Apart from a

combination of MSC and SDL the framework also proposes the use of OMT. A key issue of the methodology framework is the reuse library, an archive where relevant documents are put in for later reuse. So far our process model is only a partial instantiation of the methodology framework, because reuse is only supported for the pool of protocol building blocks. Though other descriptions are also stored in the reuse library (for documentation purposes only) we actually do not provide mechanisms for their reuse. Furthermore, we slightly modify the methodology framework by introducing an additional activity called *division* into the engineering process that supports incremental

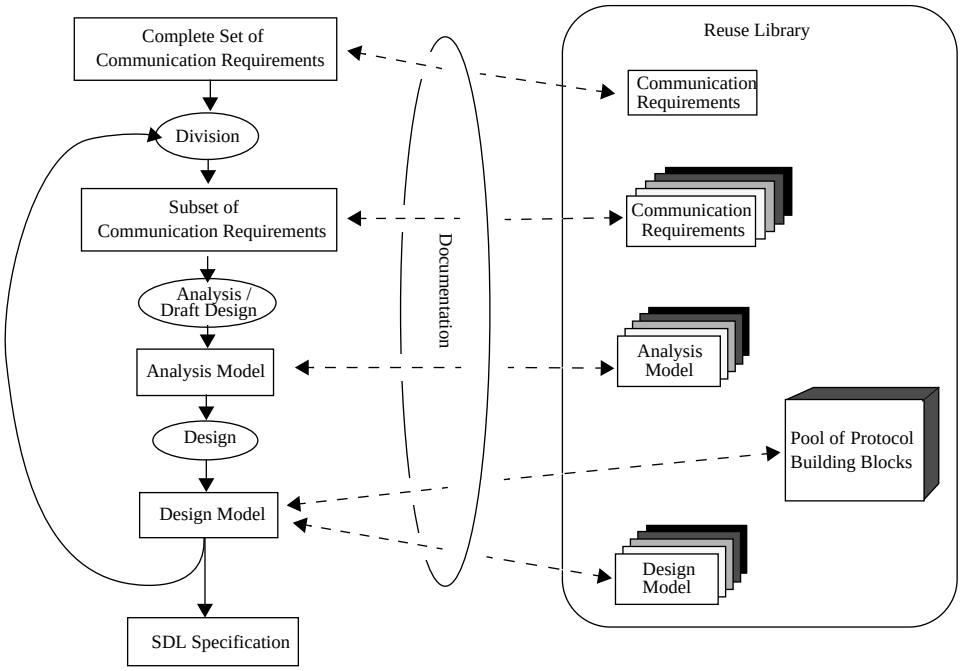


Fig. 2: activities that build a development step

design. Starting with a small initial subset of the communication requirements, system functionality is stepwise completed with each development step until all communication requirements are met.

Figure 2 illustrates this incremental process, where each development step consists of four main activities, namely documentation, division, analysis, and design. Thereby the subset of communication requirements reflects the currently implemented system functionality. Note, that only the development of an SDL specification is shown. The integration of further activities such as implementation and validation is not illustrated. Additionally the development steps in Figure 2 are ideal in the sense, that all activities are passed through exactly one time per step. If inconsistencies were found, this would result in a return to one of the previous activities and additional documentation. In the following, the activities of the process model are further elaborated and related to the activities of the methodology framework [18].

2.2.1 Documentation activity

This activity is carried out in parallel to the others. The task is to archive and administer all descriptions evolving from the current engineering process and to offer access to protocol building blocks from previous developments. This includes not only final documents like communication requirements and SDL design specification, but also intermediate products and corresponding change logs. Currently our engineering process involves documents with different levels of formality: informal natural language descriptions, MSC diagrams, OMT object models, and formal SDL specifications. Generally speaking, we correspond with the documentation activity of the methodology framework. But, as already mentioned, we only support reuse of protocol building blocks.

2.2.2 Division activity

This activity determines the requirements subset, which shall be handled by the ongoing development step. The goal is to incrementally reduce the distance between the whole set of communication requirements and the system functionalities realized so far. For this purpose the set of open requirements may be partitioned and/or simplified in order to define a manageable requirements subset. The respective decisions, however, have to be recorded. The division activity has no direct correspondence with the methodology framework.

2.2.3 Analysis activity

Compared to the methodology framework this activity includes both *analysis* and *draft design*. We decided to combine these two activities because concepts and terminology are well-known for the domain of communication systems and need not be defined separately. Thus we start with the identification of the participating objects and their relations in terms of aggregation, specialization, association, and so forth. For the case of communication systems possible objects include protocol entities, protocol functionalities, service users or service providers. The result of the analysis activity is an OMT object model. Communication relations between objects (e.g., message flow between protocol entities or data exchange between service user and provider) are represented as signal channels in an SDL overview diagram. Additionally, use cases are defined covering typical scenarios and important exceptional cases. We will describe them by the means of Message Sequence Charts.

2.2.4 Design activity

The design activity yields an executable SDL specification which is derived from the design specification of the previous development step. According to the current subset of communication requirements and the current design model the protocol engineer selects predesigned protocol building blocks represented as SDL patterns. After proper adaptation the protocol building blocks are ready to be composed with the SDL specification at hand. Based on the information provided by SDL patterns, these design steps can be explained in more detail:

- **selection:**
  we reduce the semantic gap between analysis and design models by employing OMT and MSC for pattern descriptions as well as analysis 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 protocol building blocks describe generic design solutions they have to be adapted before composition. Depending on the given SDL specification into which the pattern shall be embedded, a suitable *version* of the pattern has to be identified. The chosen version additionally must be adapted by *renaming* the abstract identifiers (e.g. signals, parameters, variables) in order to seamlessly fit the SDL specification at hand. This 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 finally has to be composed with the embedding SDL specification. This is done according to the *specialization* part of the *syntactical embedding* rules. In order to compose the SDL fragment with an embedding specification, this specification has to be specialized in the sense of the SDL standard. This results either in the addition of SDL constructs, like transitions or SDL services, or in the replacement of virtual constructs by redefinition. Thereby, possible redefinitions 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 one 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 pattern may be necessary. Examples are the declaration of new signals, sorts or channels. The *semantic properties* of an embedded pattern may also impose additional assumptions on the environment. They have to be taken into account in further development steps and must therefore be added to a list of assumptions (*checklist)*.

Compared with the methodology framework the design activity corresponds to the formalisation activity. As mentioned in [18], most work of later development steps will be done for this activity, while the work for analysis will be gradually reduced.

3 Configuring a customized resource reservation protocol

In this section, a resource reservation protocol is configured. Roughly speaking, this protocol supports connection setup in conjunction with the reservation of sufficient network resources to guarantee a specified quality of service during data transfer. Together with adequate mechanisms for traffic policing, scheduling, connection admission control, and user interfacing, it provides a real-time communication service that we have realized on the basis of a conventional token ring network [1].

3.1 Resource reservation service

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. Therefore unique local connection identifiers (CIds) are employed. The service users are informed of their CIds through the service primitives *ConnectConf* and *ConnectInd*. Additionally, several service users per node may exist, which are distinguished by local, user provided identifiers (userId) passed at the service interface (*ConnectReq, ConnectInd*). The communicating peers can therefore be globally identified by a combination of node address and userId. The calling user provides both with the connection setup request. The service primitives are listed in Table 2:. Thereby the flowspec parameter specifies the QoS requirements comprising the values: *minimum packet interarrival time, maximum packet length* and *maximum end-to-end delay.* Figure 3 shows possible interactions at the service interface.

3.2 Basic service

As an additional requirement the target service has to be build on top of a conventional token ring network. The token ring network consists of 5 Pentium-PCs running under QNX and connected with the IBM 16/4 Token Ring Network Adapter II. Thus we model our target platform as a basic service that is connectionless with the service primitives and error model described in Table 3.

3.3 The protocol configuration process

According to the process model of Section 2.2 the protocol implementing the resource reservation service was configured in an incremental way, where each design 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. In the remainder of

service primitives	parameters
ConnectReq	flowspec, callerUserId, calleeNodeAddress, calleeUserId
ConnectInd	flowspec, calleeUserId, calleeCId
ConnectResp	calleeCId
ConnectConf	callerUserId, callerCId
ConRefReq	calleeCId
ConRefInd	reason, callerUserId
Error	reason, callerUserId
DisconReq	callerCId
DisconInd	reason, calleeCId
DisconConf	callerCId

Table 2:

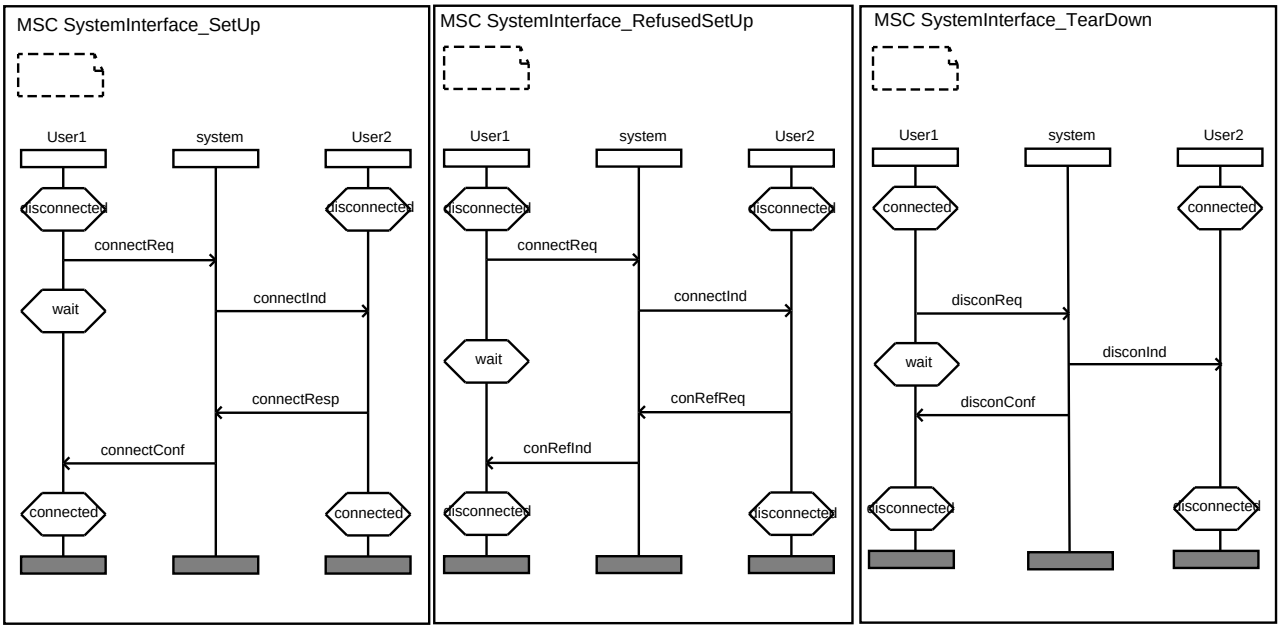


Fig. 3: MSC interface diagrams of the target service

this section, we describe the configuring of the resource reservation protocol. Apart from declarations and some block structure diagrams the design model of the first development step is given in Appendix B. 1, for step 2 and 3 selected diagrams are listed in Appendix B. 2, while Appendix B. 3 shows some diagrams that belong to development step 4.

service primitives	parameters
DataReq	nodeaddress, SDU
DataInd	SDU

error model
• frames may get lost • disruption of frames neglectable • duplication of frames not possible • ordered delivery of frames

Table 3:

3.3.1 Development step 1: dedicated sender and receiver nodes, direct communication

3.3.1.1 Division

The initial subset of the target service supports at most one unidirectional connection between a dedicated sender node (with a fixed sending user) and a dedicated receiver node (with a fixed receiving user). The service is provided exactly one time. The protocol instances are assumed to be directly connected, i.e. there is no underlying communication layer.

3.3.1.2 Analysis

In order to guarantee the requested quality of service, sufficient resources must be reserved during connection establishment and released when closing the connection. For this purpose a special entity called *resource manager* is introduced. Different from other reservation protocols such as RSVP [33] or ST2+ [4], our solution uses a centralized resource manager. Thus three main objects can be identified: caller (sending protocol instance), callee (receiving protocol instance) and resource manager. Each of them is located on its own node and can further be divided into two functional entities responsible for connection establishment and closing, or reserving and releasing resources, respectively. Figure 4 illustrates the involved objects. Their communication relations are illustrated in Fig-

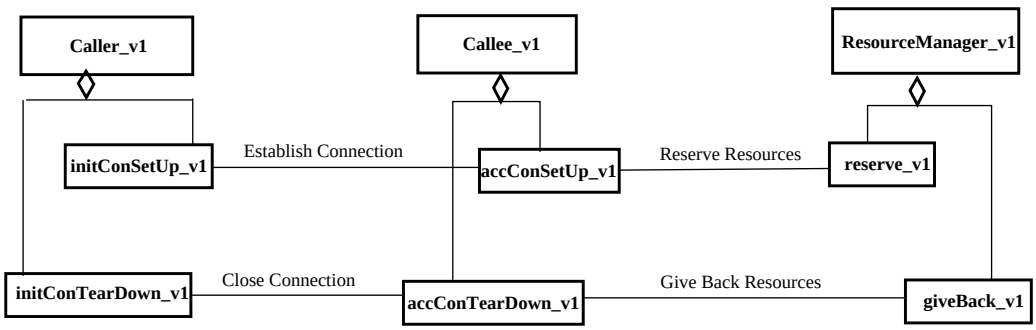


Fig. 4: object model for development step 1

ure 5. It is assumed that the callee communicates with the resource manager for reserving and giving back resources and therefore no communication path between caller and resource manager is necessary. The objects caller, callee, and resource manager are mapped onto SDL processes with the functional entities represented as separate SDL services. The communication nodes are represented as SDL blocks. Scenarios for connection establishment are shown in Figure 6, where a two-way hand-

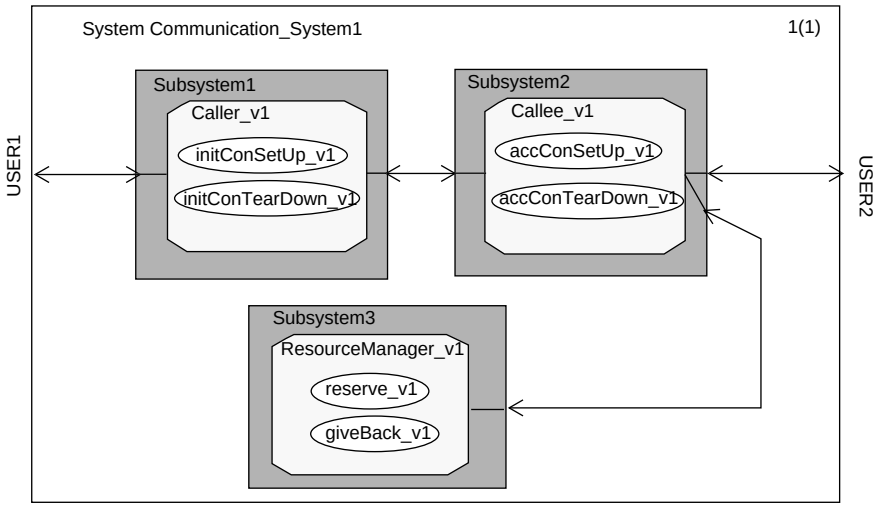


Fig. 5: SDL overview diagram for development step 1

shake is applied. The same kind of interaction is assumed between the callee and the resource manager.

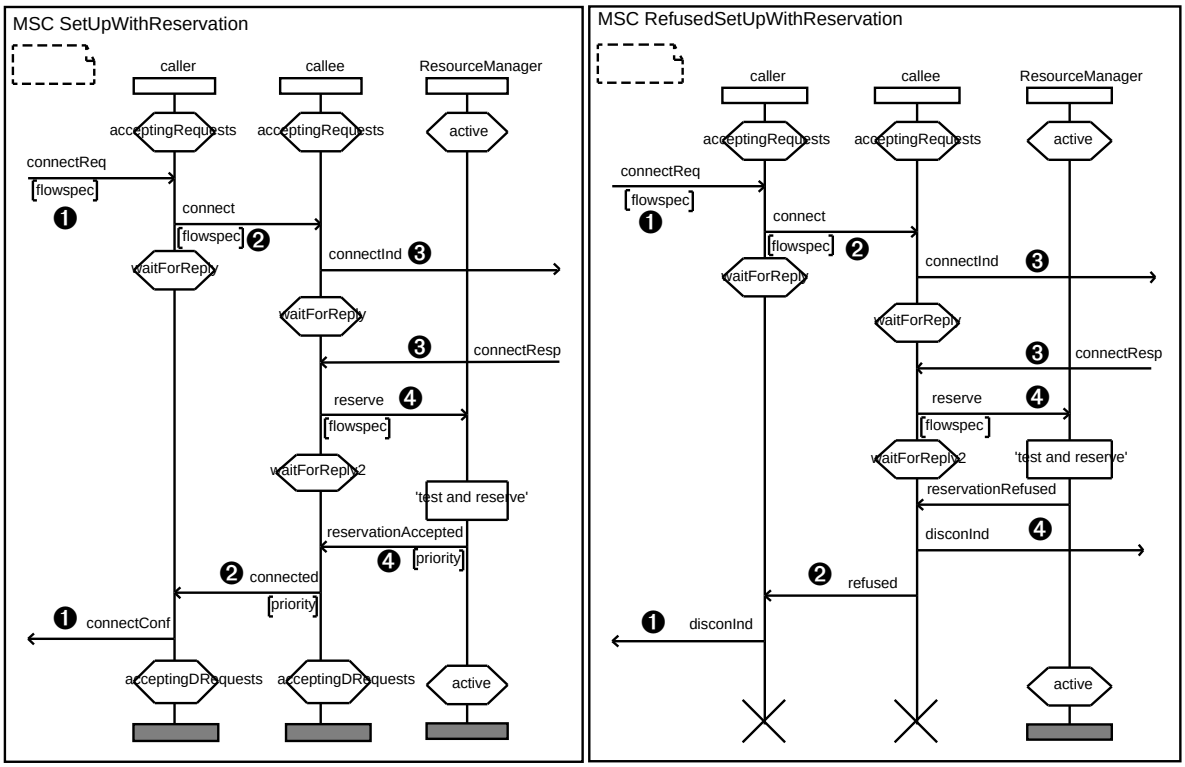


Fig. 6: MSC diagrams for development step 1: connection establishment

3.3.1.3 Design

As can be seen from the analysis model four pairs of communicating entities can be identified for connection establishment and closing, respectively: (user1, caller), (caller, callee), (callee, resource manager), and (callee, user2). Thereby user1 and user2 are part of the environment and not explicitly modelled. Each interaction corresponds to an SDL pattern called *BlockingRequestReply* (Appendix A) that introduces a two-way interaction between two given automata. The complete interaction structure can be realized by multiple application of this pattern. In detail, the following design steps have been performed to realize the interaction structure for connection establishment:

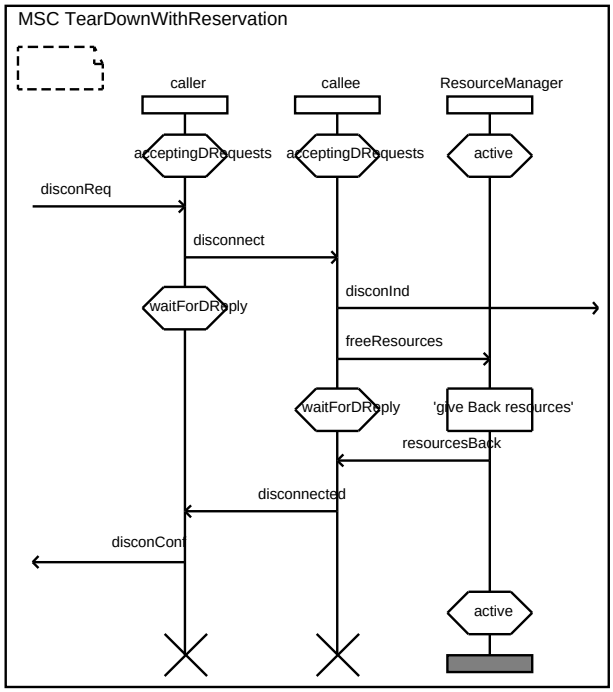


Fig. 7: MSC diagram for development step 1: connection closing

- for the interaction between user 1 and caller: ❶
  the interaction corresponds to the *BlockingRequestReply* pattern, where only *ReplyAutomaton_B* is part of the system. Version 1 is selected for this automaton. As the SDL service *Automaton_B*, into which the pattern instance has to be embedded, is empty the adaptation and composition is quite simple: signals and states can be chosen arbitrarily and the added transition of the pattern instance is the only transition of the resulting automaton. The result is the non-shaded part of *Service Type initConSetUp_v1* shown in Appendix B. 1.
- for the interaction between caller and callee: ❷
  this interaction can be realized by a *chained BlockingRequestReply* (as indicated under *cooperative usage* of the *BlockingRequestReply* pattern description, Appendix A) with ❶. Version 2 must be selected, where the new parts are shaded in *initConSetUp_v1*. The corresponding *ReplyAutomaton* is realized as an SDL service (non-shaded part of *Service Type accConSetUp_v1* listed in Appendix B. 1).
- for the interaction between callee and user2: ❸
  this interaction can be realized by a *chained BlockingRequestReply* (as indicated under *cooperative usage* of the *BlockingRequestReply* pattern description, Appendix A) with ❷. Version 2 must be selected, where the new parts are shaded in *accConSetUp_v1*. The corresponding *ReplyAutomaton* is not part of the specification.
- for the interaction between callee and resource manager: ❹
  the analysis model suggests to realize this interaction by a *knotted BlockingRequestReply* (as indicated under *cooperative usage* of the *BlockingRequestReply* pattern description, Appendix A) with ❸. Version 2 must be selected, where the new parts are shaded in *accConSetUp_v1*. The corresponding *ReplyAutomaton* is realized as an SDL service (*Service Type reserve_v1* of Appendix B. 1).

The configured chain of *BlockingRequestReply* patterns realizes the expected interaction structure between the involved communicating objects. As a consequence no signals are implicitly consumed, all (SDL) communication channels are reliable, and each *ReplyAutomaton* remains in its *startReply*

state, thus Property A.1 of the *BlockingRequestReply* pattern holds for every link of the chain. Therefore service users may rely on finite response times at this stage of the development process.

The interaction structure for closing a connection is realized analogous, except that the reply of user 2 is empty. Note, that development step 1 already yields an executable SDL specification, however, providing only a subset of the target service based on a reliable underlying service.

Checklist (assumptions that must be met in further development steps in order to ensure the properties of the SDL patterns embedded so far):

- the calling user interacts according to the *BlockingRequestReply* pattern and should behave like a *RequestAutomaton* for both connection establishment and closing. (A1)
- the called user interacts according to the *BlockingRequestReply* pattern and should behave like an ordinary *ReplyAutomaton* in case of connection establishment and like a *ReplyAutomaton* with an empty reply message in case of connection tear down. (A2)
- in order to keep finite response times, further development steps must guarantee that:
    - the request and reply signals are not implicitly consumed (A3.1)
    - the communication paths between the *RequestAutomata* and *ReplyAutomata* for transmission of request and reply signals are reliable (A3.2)
    - the *startReply* states of all *ReplyAutomata* will always eventually be reached (A3.3)

3.3.2 Development step 2: dedicated sender and receiver nodes, reliable basic service with addressing mechanism

3.3.2.1 Division

The second subset of the target service also allows at most one unidirectional connection per node, and only supports dedicated sender nodes (with a fixed sending user) and dedicated receiver nodes (with a fixed receiving user). However, this time the protocol instances operate on top of a reliable and connectionless basic service (with service primitives as described in Section 3.2). Thus basic service interfacing and receiver addressing are further issues.

3.3.2.2 Analysis

Because the protocol instances are no longer directly connected, a new object *ReliableBasicService* is introduced. This results in a ternary association between two communicating peers and the basic service. Additionally each communicating entity has to be specialized in order to integrate translation from protocol data units (PDUs) to service primitives of the basic service. The resulting object model is shown in Figure 8. The overview diagram of Figure 9 describes the communication relations between the involved objects, where the object *ReliableBasicService* is given as an SDL block, which is not further refined. Note, that the block *ReliableBasicService* is not part of the communication subsystem to be configured. Rather, it models the communication service provided by our target platform. A typical scenario is given in Figure 10.

3.3.2.3 Design

The SDL channels connecting the communication peers of the first version of the reservation protocol are replaced by an SDL block with channels connecting the protocol entities and the resource manager. This step can be seen as a structural refinement, as we still assume a reliable underlying service. The interfacing of the entities with the underlying service represented by this SDL block is configured by applying the *Codex* pattern (Appendix A) to the first version of the reservation proto-

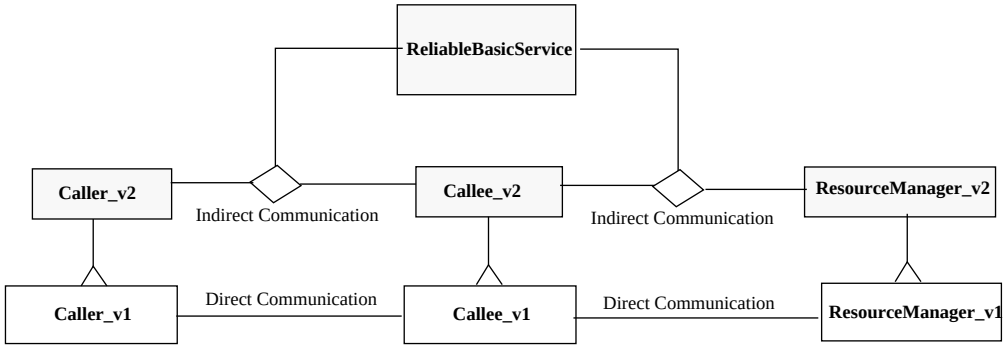


Fig. 8: object model for development step 2

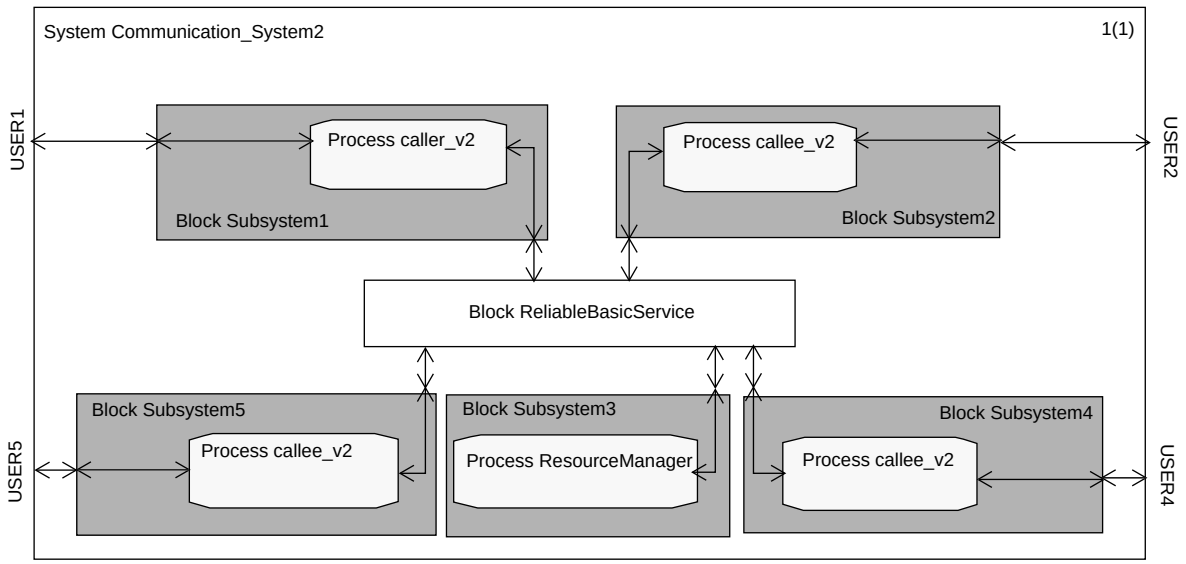


Fig. 9: SDL overview diagram for development step 2

col (development step 1). *Codex* allows two or more entities to interact through an underlying service represented by an SDL block/process by means of service primitives, i.e. *Codex* essentially provides a translation from protocol data units to service primitives.

The lower layer interface control information (ICI) needed for a correct employment of *codex* only includes the receivers node address. For the caller entity this information is provided with the upper layer service primitive *ConnectReq*, while for the callee entity it is provided with the first incoming message *Connect*. As a consequence the following *localCommunicatingEntities* (*codex* notation, see Appendix A) have to be specialized according to the syntactical embedding rules of the *codex* pattern description:

- Service Type *initConSetUp*:
  The necessary ICI to be stored consists of the callee‘s node address. Furthermore, the callee has to be informed of the caller‘s node address. This information is sent along with the *Connect* PDU.
- Service Type *accConSetUp*:
  The caller‘s node address contained in the *Connect* PDU serves as lower layer ICI to be stored for the *Codex* SDL service (*codex* notation, see Appendix A).

For the preparation of lower layer service primitives and the decoding of incoming primitives a service *lowerLayerInterfacing* is added to the surrounding process diagrams *Caller, Callee*, and *ResourceManager*. Finally, the structural changes described with the *codex* pattern have to be conducted. The changes are illustrated in Appendix B. 2 and shaded . The ICI (peer node address) is set with the first signal received, and is left unchanged. Because each protocol entity han-

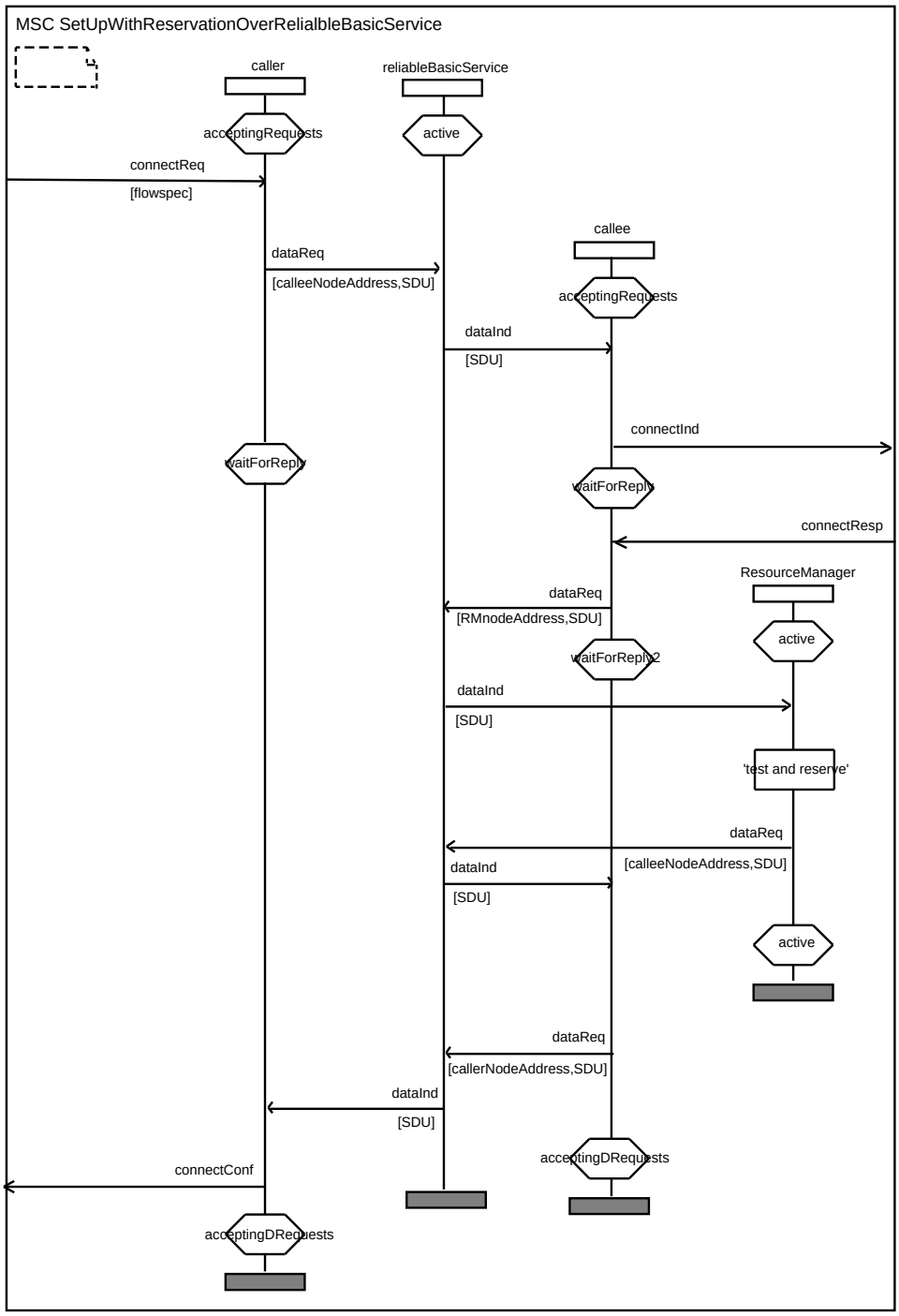


Fig. 10: MSC diagram for development step 2

dles at most one connection, the peer node address always matches with the PDU currently processed by *lowerLayerInterfacing*. As the basic service in use is reliable and connectionless, the *codex* pattern suffices to replace the previously (see development step 1) used SDL channels (Property B.1 of the *Codex* pattern). Therefore assumption A3.2 (see checklist of development step 1) is still valid. Other assumptions from the checklist of development step1 are not affected by the *Codex* pattern.

Checklist (additional assumptions that must be met in further development steps in order to ensure the properties of the SDL patterns embedded so far):

- the basic service is reliable (A4.1)
- the basic service is connectionless (A4.2)

- the interface control information retrieved by *lowerLayerInterfacing* always matches with the PDU currently processed (A4.3)

Note, that assumptions A4.1 - A4.3 replace assumption A3.2 from development step 1.

3.3.3 Development step 3: dedicated sender and receiver nodes, unreliable basic service

3.3.3.1 Division

In the third step, we relax the assumption that the underlying service be reliable by allowing frame loss. This corresponds with the error model of the basic service provided by our target platform.

3.3.3.2 Analysis

We replace *ReliableBasicService* with a new object *UnderlyingService*. Due to the changed error model the communicating objects Caller, Callee, and ResourceManager have to be specialized to cope with lost messages. The object model is shown in Figure 11. The overview diagram corresponds to the one of Figure 9, except that the *Block ReliableBasicService* has to be replaced by the *Block UnderlyingService*.

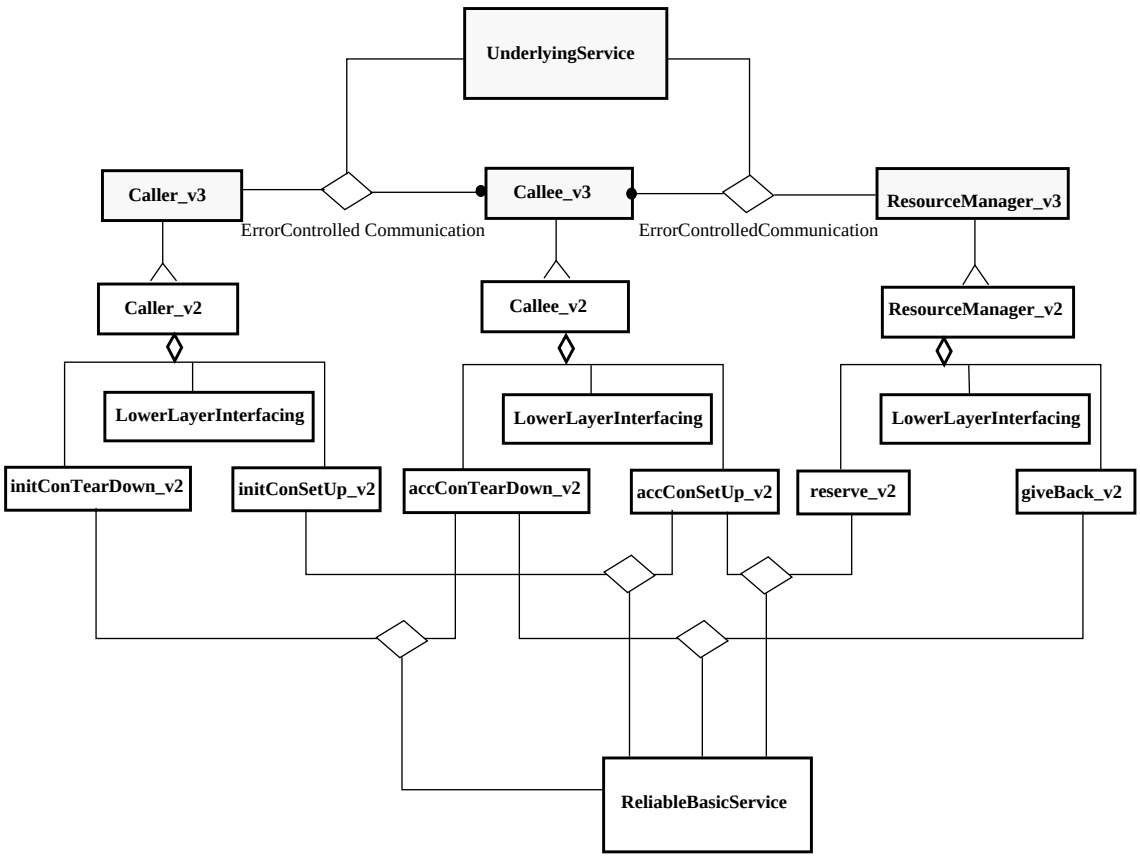


Fig. 11: object model for development step 3

3.3.3.3 Design

To cope with lost frames the *TimerControlledRepeat* pattern (Appendix A) is applied to the second version of the reservation protocol (development step 2). If an expected reply does not arrive before the expiry of a timer, the message is repeated (Positive Acknowledgement with Retransmis-

sion). Since retransmission may lead to duplication of messages (not treated by TimerControlledRepeat), the patterns *DuplicateIgnore* and *DuplicateHandle* (Appendix A) are also applied. Duplicates are detected by an unique message identifier and either discarded or, where necessary, specifically handled.

Possible message losses of the *underlyingService* affect the SDL services *initConSetUp, initConTearDown, accConSetUp,* and *accConTearDown*. As the involved *acknowledgeAutomata (TimerControlledRepeat* notation, see Appendix A) are not capable to cope with duplicate messages *DuplicateIgnore* and *DuplicateHandle*, respectively, are applied to *accConSetUp, accConTearDown, reserve,* and *giveBack* before application of *TimerControlledRepeat*. Note, that duplicates can be detected by signal names as each request must only be serviced one time during the lifetime of the SDL services. While the SDL service *accConTearDown* ignores duplicates, the SDL services *accConSetUp, reserve,* and *giveBack* handle duplicates by repeating the corresponding reply. SDL service *accConSetUp* is shown in Appendix B. 2, where the changes are shaded .

Checklist (additional assumptions that must be met in further development steps in order to ensure the properties of the SDL patterns embedded so far):

- the basic service neither disrupts nor creates messages (A5.1)
- the timer intervals for retransmission are greater than the maximal round trip time for the requests and corresponding replies (A5.2)

Note, that assumptions A5.1 - A5.2 relax assumption A4.1 from development step 2 (communication is no longer reliable, but the sender is informed about a failed transmission after a certain number of retries). As a consequence, the semantic properties of some embedded *BlockingRequestReply* patterns do no longer hold, i.e. we can not guarantee that replies will definitely arrive within finite time. However, we definitely reach an error state within finite time, if a certain number of retransmissions fail.

3.3.4 Step 4: mixed nodes, unreliable basic service

3.3.4.1 Division

In the fourth and last step, we consider the full target service supporting nodes with both sender and receiver functionality (with multiple sending and receiving users per node) as well as several connections per node and user, i.e. the service will be provided multiple times.

3.3.4.2 Analysis

Each connection is managed by a separate set of protocol entities, which are created and released dynamically. We replace *Caller* and *Caller* by a new object *ReservationProtoco*l with merged functionality (i.e. an aggregation of *initConSetUp, initConTearDown, accConSetUp, accConTearDown, and lowerLayerInterfacing).* Each instance is responsible for handling either the caller part or the callee part for one connection. The object model is shown in Figure 12.

Figure 10 illustrates the establishment and closing of one connection, where a corresponding protocol entity is created at the caller node and the callee node. In order to prevent double establishment of the same connection, incoming createReq messages must be controlled for duplicates, where duplicates are handled by forwarding them to the corresponding protocol entity.

The set of connections is managed by a new object *ProtocolAdministrator* responsible for creating new objects of type *ReservationProtocol* if additional connections are requested and forwarding messages to the right connection.

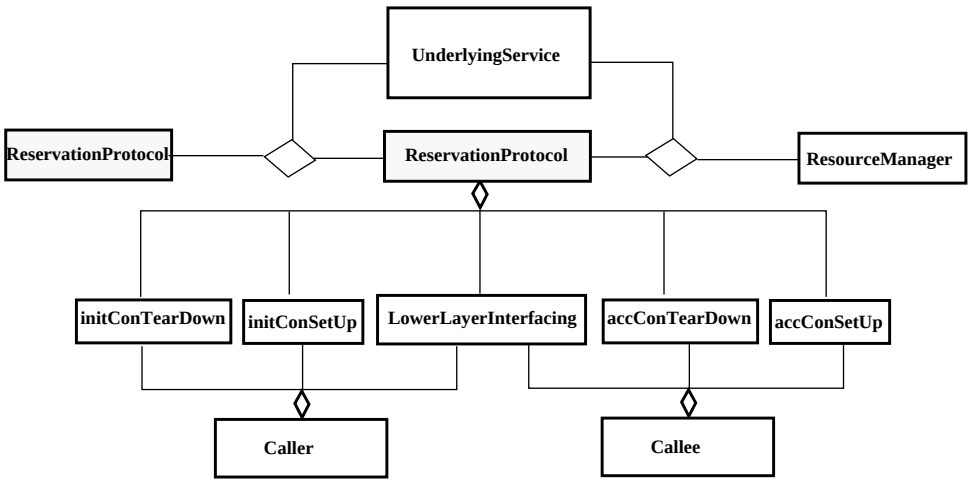


Fig. 12: object model for development step 4

3.3.4.3 Design

The pattern dealing with the dynamic creation of entities is *DynamicEntitySet* (Appendix A). For the *EntityAdministrator* (*DynamicEntitySet* notation, see Appendix A) a new process *ProtocolAdministrator* is introduced (see Appendix B. 3), where two transitions *createNewEntity* (*createReq*-signals are *ConnectReq* and *Connect*) are inserted and redefined to restrict the number of active connections per time. The communication with the user is assumed to be reliable and therefore the createReq-signal *ConnectReq* is not controlled for duplicates. For the createReq-signal *Connect* the *DuplicateHandle* pattern (Appendix A) is applied. Duplicate *Connect* messages do not create a new protocol instance, they are forwarded to the already existing ReservationProtocol entity. Thereby incoming *Connects* are identified by a combination of *callerNodeAddress* and *callerConnectionId* parameters. Each original *Connect* signal is logged by storing this pair of parameters (*callerNodeAddress, callerConnectionId*) in a so-called *PartnerList*. The instantiated *DuplicateHandle* pattern is shaded in the *ProtocolAdministrator* process (Appendix B. 3). For other incoming signals of *ProtocolAdministrator* corresponding *forwardMessage* transitions are incorporated.

The *TerminatingEntity* automaton (*DynamicEntitySet* notation, see Appendix A) is given by process type *ReservationProtocol*. The signal routes to *ReservationProtocol* are redirected to the *ProtocolAdministrator*, that is connected with the process set *ProtocolEntity* by a create line and signal routes in order to forward service requests.

The requesting peer is informed about the local connection identifier (CId) by adding a parameter *CId* to the reply message. This CId is inserted into the output signals (except ConnectReq and Connect) which are sent to the protocol entity. Therefore the service offered by *ProtocolAdministrator* is provided several times and each protocol entity will only receive messages which belong to its connection.

Checklist (additional assumptions that must be met in order to ensure the properties of the SDL patterns embedded so far):

- in order to assure that each ReservationProtocol instance is dedicated to one connection and will receive exactly those messages corresponding to its connection the following has to remain valid in future development steps:
    - the peer is informed about the CId of the protocol entity and always adds this CId to the output signals (except *ConnectReq* and *Connect*) which are sent to the protocol entity (A6.1)

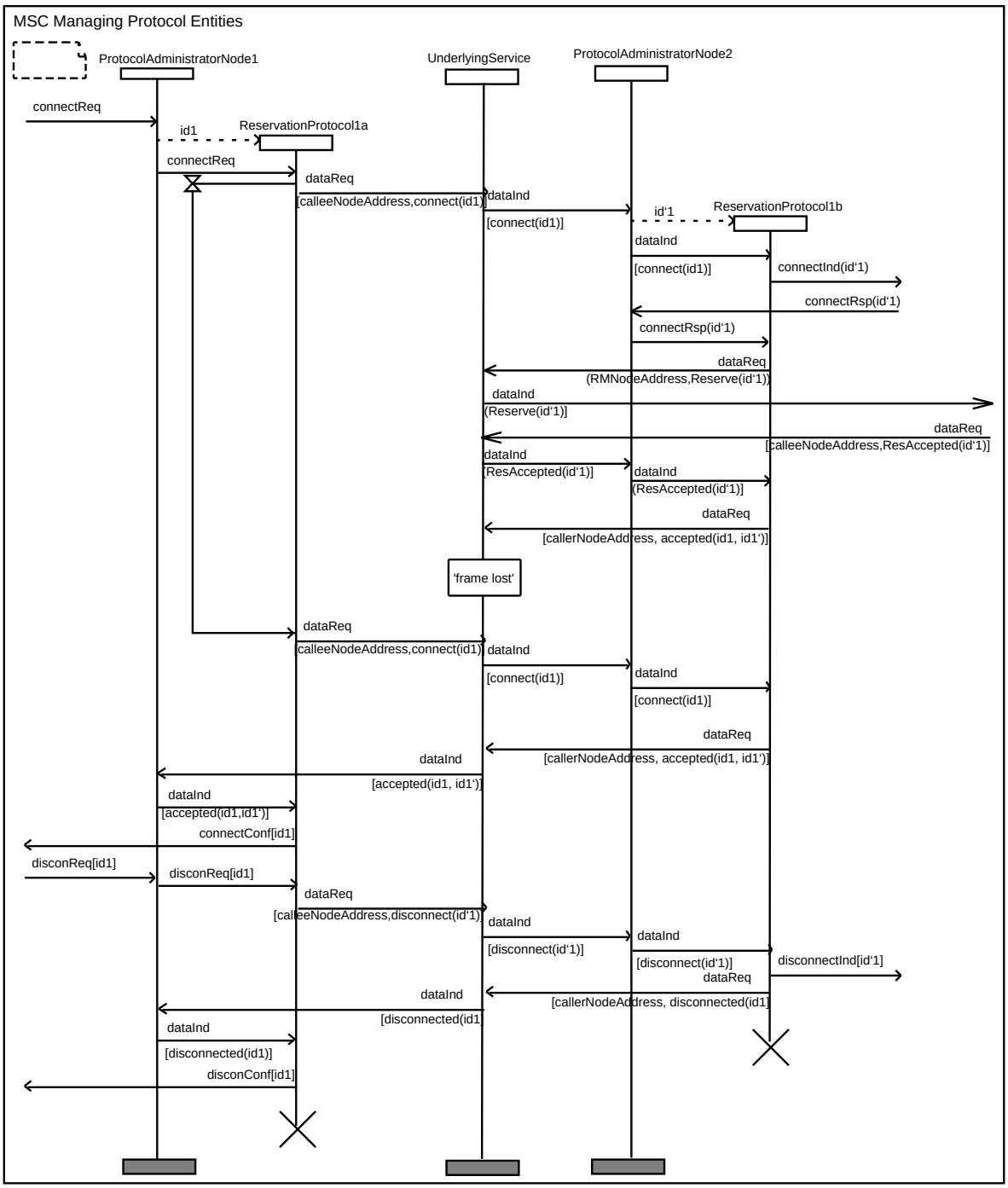


Fig. 13: MSC diagram for development step 4

It follows that all assumptions (the cumulative checklists of the separate development steps) concerning the communication subsystem are fulfilled. Furthermore, the basic service provided by our target platform meets the assumptions A4.2 and A5.1. If additionally the service users behave according to A1, A2 and the retransmission timers are set according to A5.2, the embedded patterns are applied as intended. Though this does not prove the correctness of the communication protocol in general, it assures some „design-specific“ properties that increase the confidence in the resulting specification.

4 Conclusion and future work

We have presented an approach that has the potential of substantially reducing the effort for designing customized protocols. The approach is based on the concept of design patterns, which we have specialized to communication protocols. In addition, we have used SDL-92 and MSCs to formally specify protocol design patterns as well as rules for their instantiation and composition. To illustrate our approach, we have configured a resource reservation protocol. When applying our approach, we have observed the following:

- Each of the selected SDL patterns has been applied several times when configuring the reservation protocol. This provides some evidence that the predesigned patterns have been well chosen.
- A very large portion (almost 100% of the control structure) of the final protocol specification has resulted from the application of SDL patterns. This gives some evidence for the feasibility of our approach.
- 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 results in improved maintainability and less design errors, since the design decisions are well founded and documented.
- 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 very time consuming task. Note that the same experience has been made in other contexts where design patterns are used.
- The SDL patterns applied to the configuring of the resource reservation protocol have been of rather fine granularity. Coarser patterns may have the advantage of reducing the overall development effort, since less patterns need to be applied to configure a protocol. However, this is merely a question of identifying suitable protocol building blocks and does not affect our approach itself.

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, in order to draw a final conclusion, further experience with this approach will be needed. We are currently extending the pool of protocol building blocks, and are using our approach for configuring several other protocols including, for instance, ST2+ [4].

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 a service specification can be guaranteed a priori. We see this as a fertile field for future research.

Acknowledgements. Special thanks go to Prof. Dr. R. Gotzhein for valuable comments and discussions on an early version of this report.

References

[1] C. Bobek, „Entwurf und Implementierung eines Ressourcen-Verwalters zur Echtzeitkommunikation“ (in german), diploma thesis at the University of Kaiserslautern, 1996

[2] R. Bræk and Ø. Haugen, „Engineering Real Time Systems - An object-oriented methodology using SDL“, Prentice Hall, 1993

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

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

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

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

[7] R. Gotzhein, B. Geppert, C. Peper, and F. Rößler, „Generic Layout of Communication Subsystems - *A Case Study*“, SFB 501 Report 14/96, University of Kaiserslautern, Germany

[8] „INSYDE II - Areas of Interest for Future Research - Towards a Continuation of the INSYDE Consortium“, http://info.vub.ac.be:8080/users/insyde/future_research.html

[9] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, „Object-Oriented Software Engineering - A Use Case Driven Approach“, Addison-Wesley, 1995

[10] V. Jonckers, K. Verschaeve, B. Wydaeghe, L. Cuypers, and J. Heirbaut, „OMT*, Bridging the Gap between Analysis and Design“, Proceedings of FORTE ‘95, International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols, 1995

[11] A. Kühlmeyer, „Variantenbildung in SDL-92“, Students Work at the University of Kaiserslautern, 1996

[12] M. Lang, „Spezifikationsvarianten des Alternating-Bit Protokolls in SDL/SDT“, Students Work at the University of Kaiserslautern, 1996

[13] S. van Lier, „Komponentenbasierte Dekomposition und Spezifikation des Real-Time Transportprotokolls RTP“, Students Work at the University of Kaiserslautern (in work)

[14] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J.R.W. Smith, „Systems Engineering Using SDL-92“, North-Holland, 1994

[15] 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

[16] T. Plagemann, J. Waclawczyk, and B. Plattner, „Management of Configurable Protocols for Multimedia Applications“, Proceedings of ISMM International Conference on Distributed Multimedia Systems and Applications, Honolulu, USA, 1994

[17] W. Pree, „Design Patterns for Object-Oriented Software Development“, Addison-Wesley, 1995

[18] R. Reed, „Methodology for real time systems“, Computer Networks and ISDN Systems 28 (1996), 1685-1701

[19] E. Rudolph, P. Graubmann, and J. Grabowski, „Tutorial on Message Sequence Charts“, Computer Networks and ISDN Systems 28 (1996), 1629-1641

[20] E. Rudolph, J. Grabowski, and P. Graubmann, „Message Sequence Charts (MSC 96)“, Tutorial Notes of the 9th International Conference on Formal Description Techniques, Kaiserslautern, 1996

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

[22] Ph. Schaible, „Pattern-basierte Konfigurierung des Reservierungsprotokolls ST2+ und Erstellung eines SDL-Frameworks für Ressourcenreservierungsprotokolle“, Diploma Thesis at the University of Kaiserslautern (in work)

[23] D.C. Schmidt, „An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Systems“, IEE Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems), Volume 2, No 4, Dec. 1994

[24] 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

[25] D.C. Schmidt, B. Stiller, T. Suda, and M. Zitterbart, „Configuring Function-based Communication Protocols for Multimedia Applications“, Proceedings of the 8th International Working Conference on Upper Layer Protocols, Architectures, and Applications, Barcelona, Spain, 1994

[26] M. Schwaiger, „Komponentenbasierte Dekomposition und Spezifikation des Multicast Protokolls IPv6“, Students Work at the University of Kaiserslautern (in work)

[27] SDT 3.0 Reference Manual & User‘s Guide, TeleLogic, 1995

[28] 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

[29] „The SOMT Method“, Telelogic AB, http://www.telelogic.se/products/somt.htm

[30] B. Stiller, „Flexible Protokollkonfiguration zur Unterstützung eines diensteintegrierenden Kommunikationssubsystems“, PhD thesis (in german), VDI-Verlag, Reihe 10, Nr. 306, 1994

[31] K. Verschaeve, B. Wydaeghe, V. Jonckers, and L. Cuypers, „Translating OMT* to SDL, Coupling Object-Oriented Analysis and Design with Formal Description Techniques“, Proceedings of Methods Engineering ‘96 -- IFIP WG 8.1/8.2 Working Conference on Principles of Method Construction and Tool Support, 1996

[32] 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

[33] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, „RSVP: A new Resource ReSerVation Protocol“, IEEE Network, Sep. 1993

[34] M. Zitterbart, „Funktionsbezogene Parallelität in transportorientierten Kommunikationsprotokollen“, PhD thesis (in german), VDI-Verlag, Reihe 10, Nr. 183, 1991

[35] Z.100 (1993) CCITT Specification and Description Language (SDL), ITU-T, 1994

[36] Z.120 (1993) Message Sequence Chart (MSC), ITU-T, 1994

Appendix A

A set of protocol building blocks for the case study.

BLOCKINGREQUESTREPLY

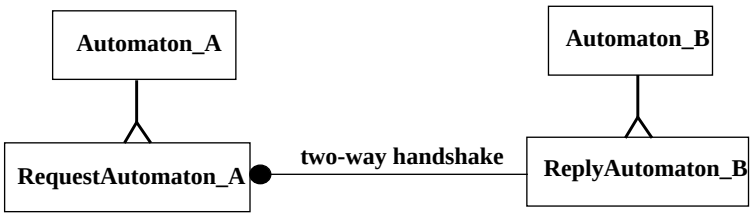
Intent:

The BlockingRequestReply pattern introduces a two-way handshake between two given automata *Automaton_A* and *Automaton_B*. Being triggered, *Automaton_A* will send a request and is blocked until receiving a reply. After reception of a request, *Automaton_B* sends a reply. To assure finite response time and proper connection, certain assumptions about the embedding environment (including the superclasses *Automaton_A* and *Automaton_B*) are in place.

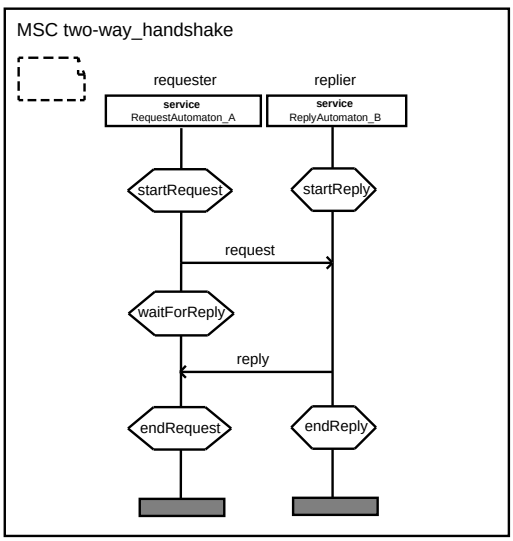
Motivation:

After initiating a connection setup, a service user waits for a reply from the service provider („accepted“, „refused by callee“, „refused due to lack of resources“,...). In case of refusal, the user may try again with lower quality of service requirements.

Structure:

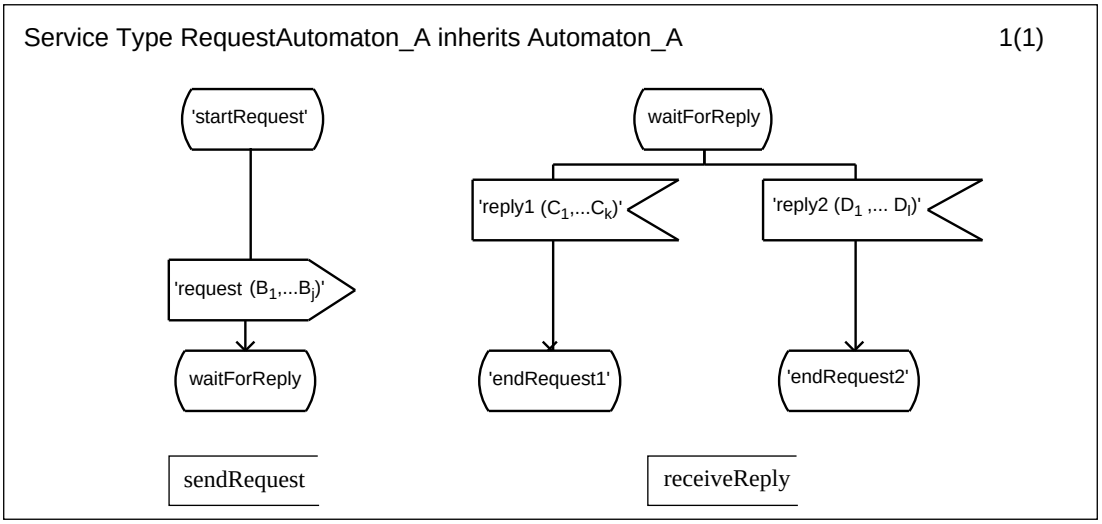


Message scenario:

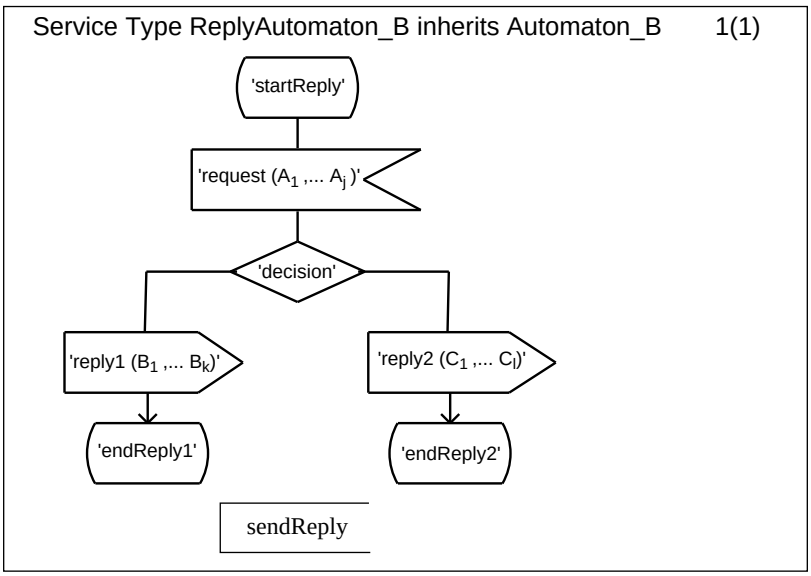


SDL-fragment (Version 1)**:**

RequestAutomaton_A



ReplyAutomaton_B



Syntactical Embedding

- Automaton_A:

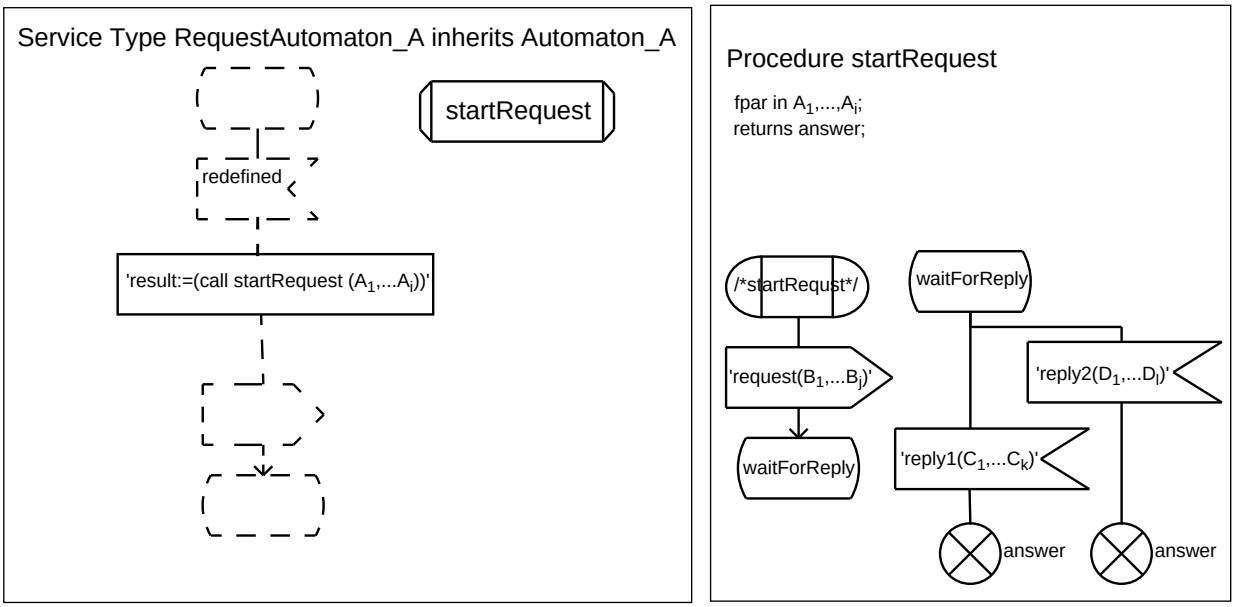
Specialization: Add transitions *sendRequest* and *receiveReply* to the given SDL service *Automaton_A.*
Renaming: The signals *request, reply1*, and *reply2* and the state *waitForReply* may be renamed but are required to be locally unique. The states *startRequest, endRequest1*, and *endRequest2* may be identified with each other or any state in the given SDL service *Automaton_A.*

- Automaton_B:

Specialization: Add transition *sendReply* to the given SDL service *Automaton_B,* which must be different to *Automaton_A.*
Renaming: The signals *request, reply1,* and *reply2* may be renamed but are required to be locally unique and of the same name as the corresponding signals in *RequestAutomaton_A.* The states *startReply, endReply1 and endReply2* may be identified with each other or any state in the given SDL service.

SDL-fragment (Version 2)**:**

RequestAutomaton_A



ReplyAutomaton_B

same as in Version 1

Syntactical Embedding

- Automaton_A:

Specialization: redefine a proper transition by mere supplementation of the procedure call *startRequest*.
Renaming: The signals *request, reply1*, and *reply2* and the state *waitForReply* may be renamed but are required to be locally unique.

- Automaton_B:

same as in Version 1

Semantic properties:

Property A.1: *If the assumptions stated below hold,* RequestAutomaton_A *will eventually receive a reply from* ReplyAutomaton_B *after sending a request. The assumptions are:*

- *The* request *and* reply *signals are not implicitly consumed by the respective superclass.*
- *Communication between* RequestAutomaton_A *and* ReplyAutomaton_B *for transmission of the* request *and* reply *signals is reliable.*
- *The state* startReply *of* ReplyAutomaton_B *will always eventually be reached.*

Redefinition:

Normally, the embedded SDL-fragment will be supplemented by additional statements e.g. to prepare signal parameters. The following property determines the allowed redefinitions of the BlockingRequestReply pattern.

Property A.2: *Property A.1 still holds, if the BlockingRequestReply pattern is redefined by the introduction of additional statements, which do not disrupt or bypass the thread of control from predefined input to predefined output statements.*

There is one mandatory redefinition, namely the replacement of the comment 'decision' in *ReplyAutomaton_B* with a real decision according to the protocol designer's needs.

Cooperative usage:

As a major feature, BlockingRequestReply may be extended to an arbitrary complex interaction structure by self-embedding. This follows from our redefinition rule, which e.g. allows the SDL-fragment embedded into *Automaton_B* to be supplemented by a procedure call initiating a new

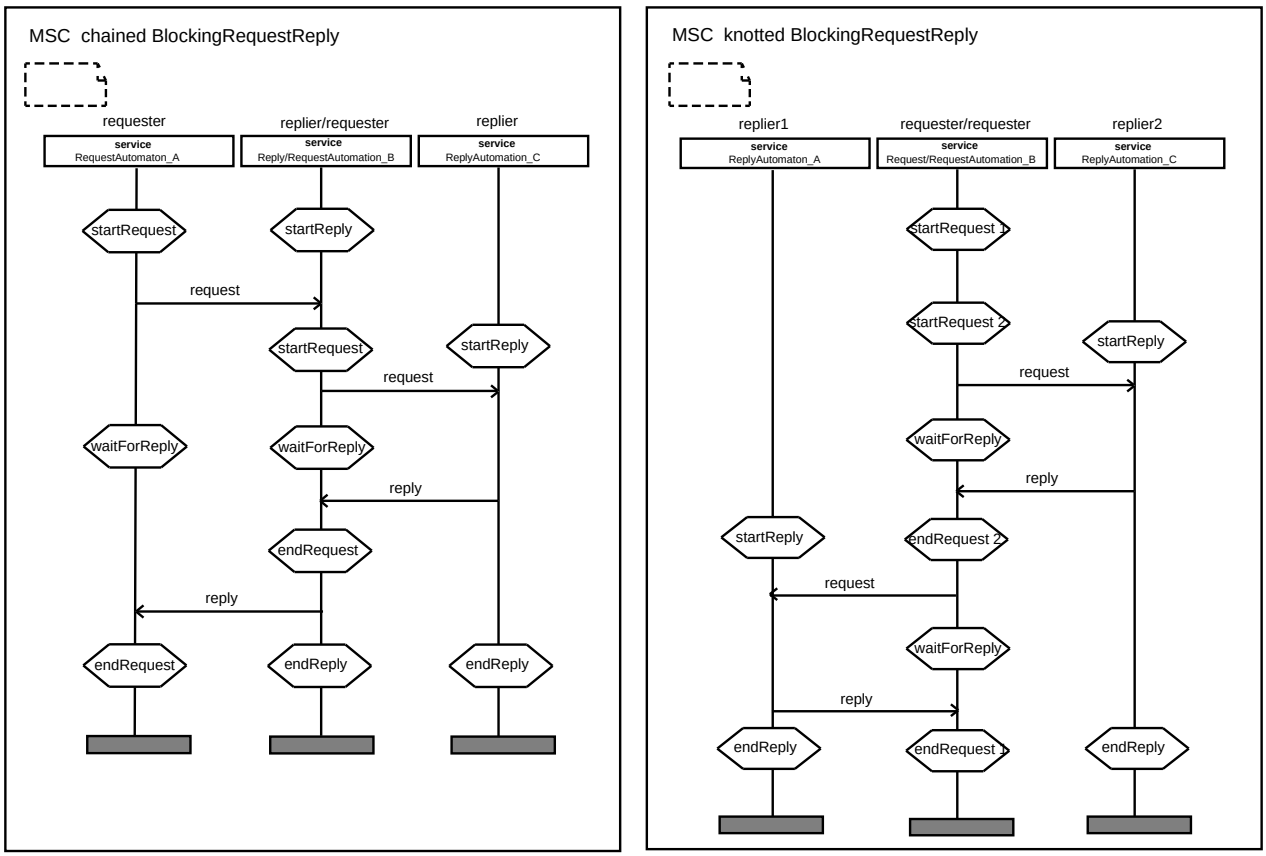


Fig. 14: multiple employment of BlockingRequestReply

request (because of property A.1, this redefinition does not disrupt or bypass the predefined thread of control). See Figure 14 for this and another simple example. It is worth mentioning that the finite response time property generalizes to chains of BlockingRequestReply patterns, if the assumptions are valid for every link of the chain. Chains of BlockingRequestReply patterns are built by successive embedding of a *startRequest* procedure call of a new pattern instance into a *sendReply* transition of a preexisting pattern instance.

Corollary A.3: *If the assumptions stated in Property A.1 hold for every link in a chain of BlockingRequestReply instances, the first RequestAutomaton will eventually receive a reply from his corresponding ReplyAutomaton after sending a request.*

In order to relax the assumption of reliable communication channels (Property A.1), BlockingRequestReply may be used in conjunction with the patterns TimerControlledRepeat and DuplicateControl.

CODEX

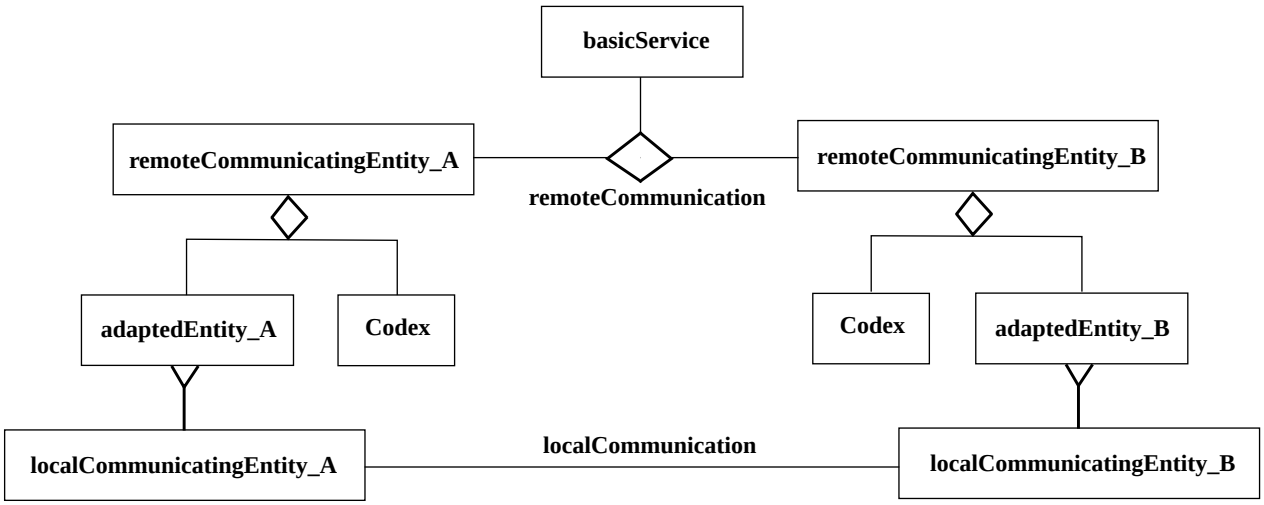
Intent:

Codex provides mechanisms to allow two (or more) entities *localCommunicatingEntity_A* and *localCommunicatingEntity_B*, which interact through SDL channels, to cooperate by the means of a given communication system *basicService*. 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 and routing decisions. The Codex pattern is only concerned about a minimal subset of these functionalities, namely interfacing with *basicService* by the means of service primitives.

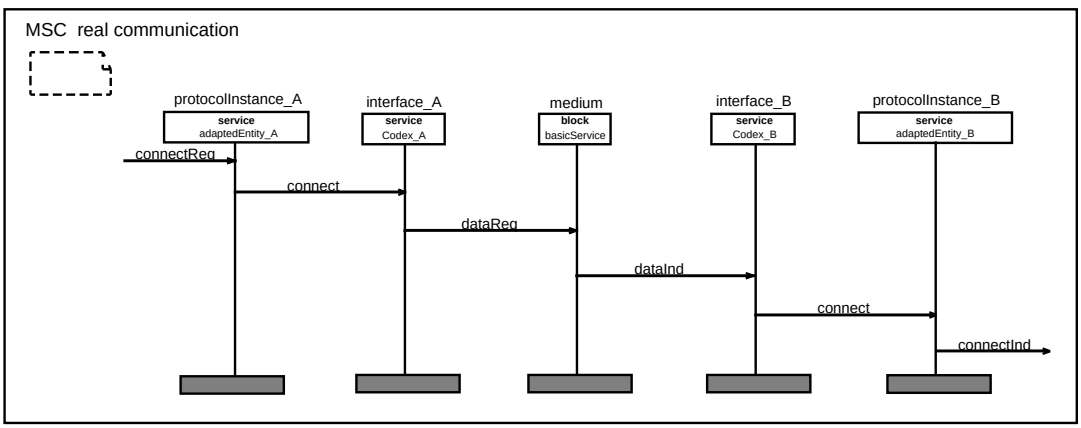
Motivation:

Conventional LANs like Ethernet or Token-Ring may play the role of a basic service. If a protocol specification happens to be put on top of such a LAN Codex may be fruitfully employed.

Structure (only two communicating entities involved):

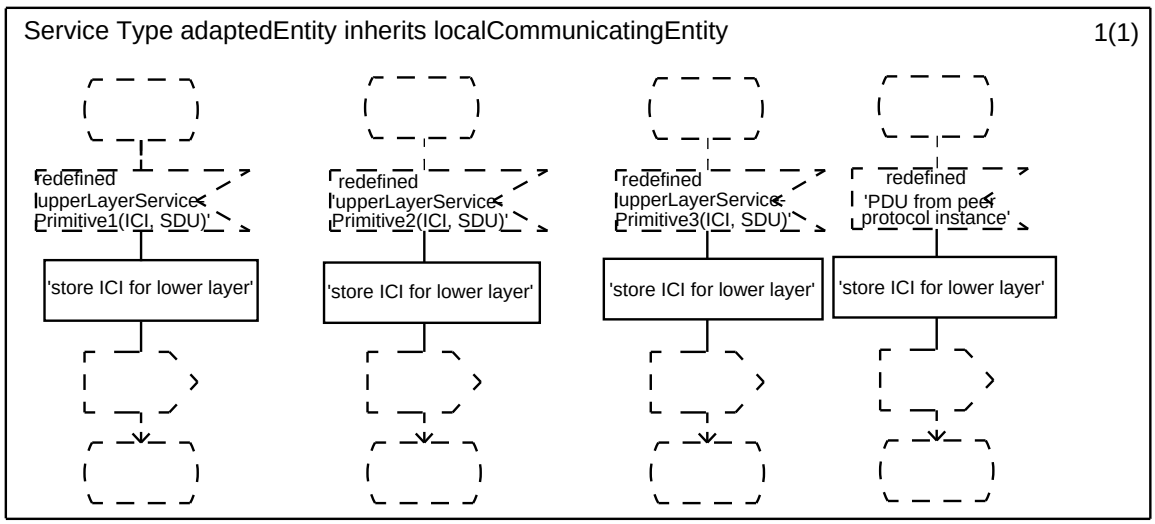


Message scenario:

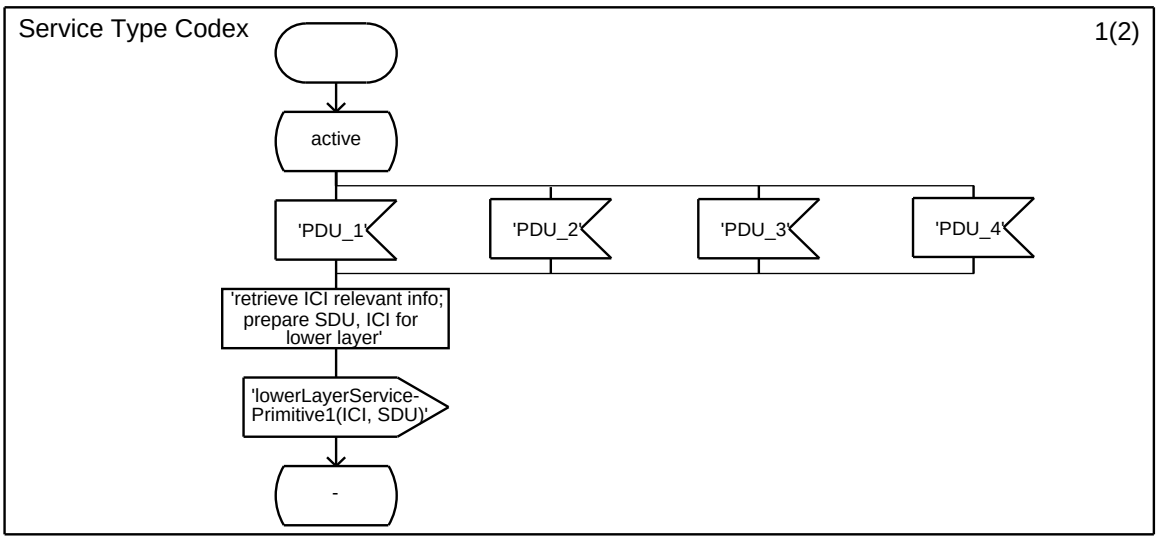


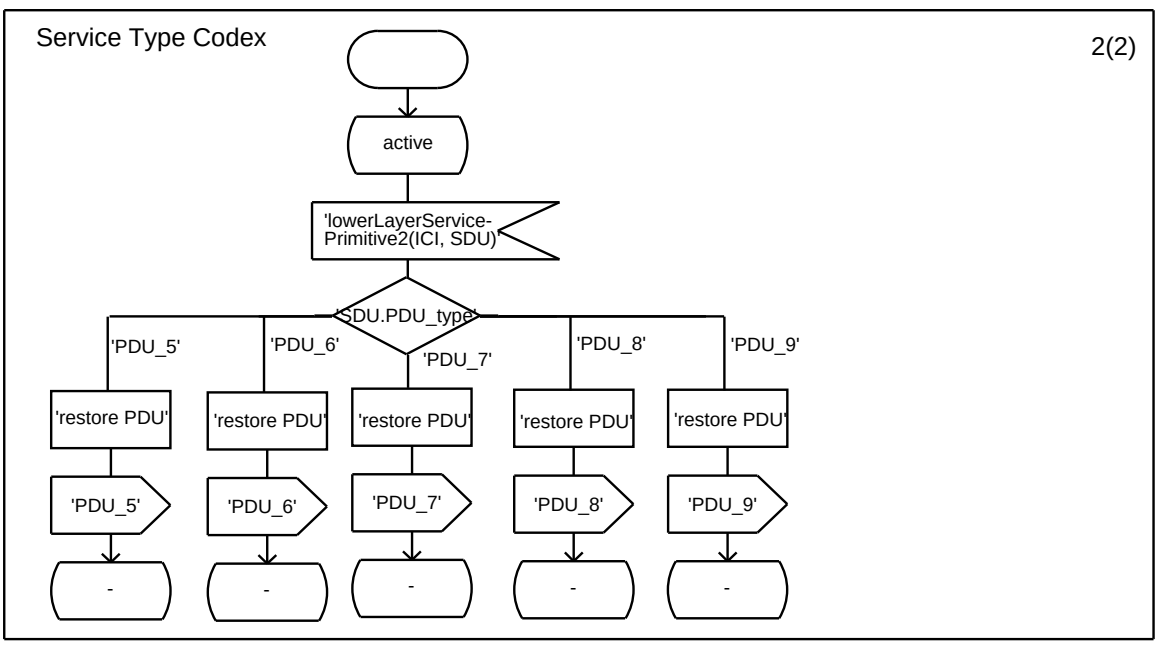
SDL-Fragment:

AdaptedEntity (not mandatory)



Codex





Syntactical embedding

Specialization: transitions of *localCommunicatingEntity* which receive service primitives *servicePrimitiveX* from the upper layer or PDUs from the peer protocol entity are potential candidates for redefinition in order to derive and store necessary lower layer interface control information (ICI) e.g. peer protocol instance addresses. The protocol engineer has to decide which ones are relevant or if the necessary information is provided elsewhere inside *localCommunicatingEntity*. In any case this information will be used when the lower layer service primitives are prepared.
For this purpose and for decoding of incoming lower layer service primitives a service of type *codex* is added to the surrounding process diagram of *localCommunicatingEntity*.
Renaming: *PDU_1* to *PDU_4* correspond with those messages *localCommunicatingEntity* sends to its peer. Accordingly *PDU_5* to *PDU_9* identify with those messages *localCommunicatingEntity* receives from its peer. However, the concrete quantities of course have to be adapted.
LowerLayerServicePrimitive1 and *lowerLayerServicePrimitive2* have to be identified with the service primitives for data transfer over the given basic service.
Structural change: the channel between *localCommunicatingEntity*_A and *localCommunicatingEntity*_B must be deleted and redirected from *adaptedEntity_A* respectively *adaptedEntity_B* to their local *codex*. Additionally the *codex* services need a channel to *basicService* to close the gap again.

Semantic properties:

Property B.1: *If the assumptions stated below hold, the codex pattern suffices to replace a SDL channel between* localCommunicatingEntity_A *and* localCommunicatingEntity_B. *The assumptions are:*

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

Redefinition:

not allowed

Cooperative usage:

It was already mentioned that the Codex pattern only solves a small subset of the problems one faces when introducing a special basic service. TimerControlledRepeat is a pattern to additionally cope with possible message losses by the basic service.

TIMERCONTROLLEDREPEAT

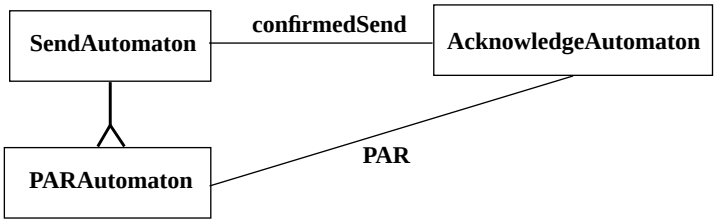
Intent:

TimerControlledRepeat extends a confirmed message exchange between two automata *SendAutomaton* and *AcknowledgeAutomaton* for the case of possible message losses during data transfer. If an expected acknowledgement does not arrive before the expiry of a timer, the message is repeated (Positive Acknowledgement with Retransmission). This pattern does not deal with the problem of message disruption or duplication.

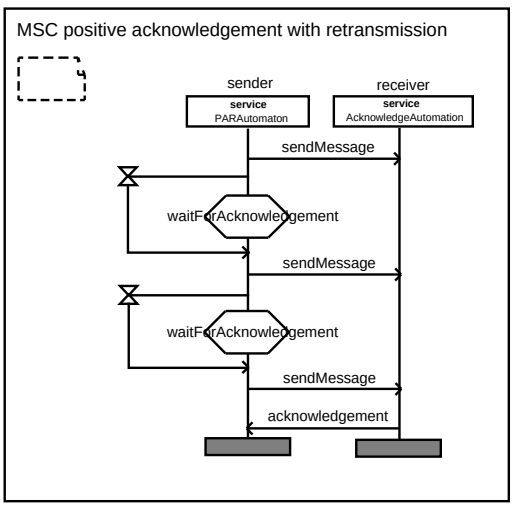
Motivation:

For a BlockingRequestReply pattern instance the requester will deadlock, if the reliable transmission of the request or reply signals is not guaranteed. Therefore replies are observed by TimerControlledRepeat in case of an unreliable basic service.

Structure:

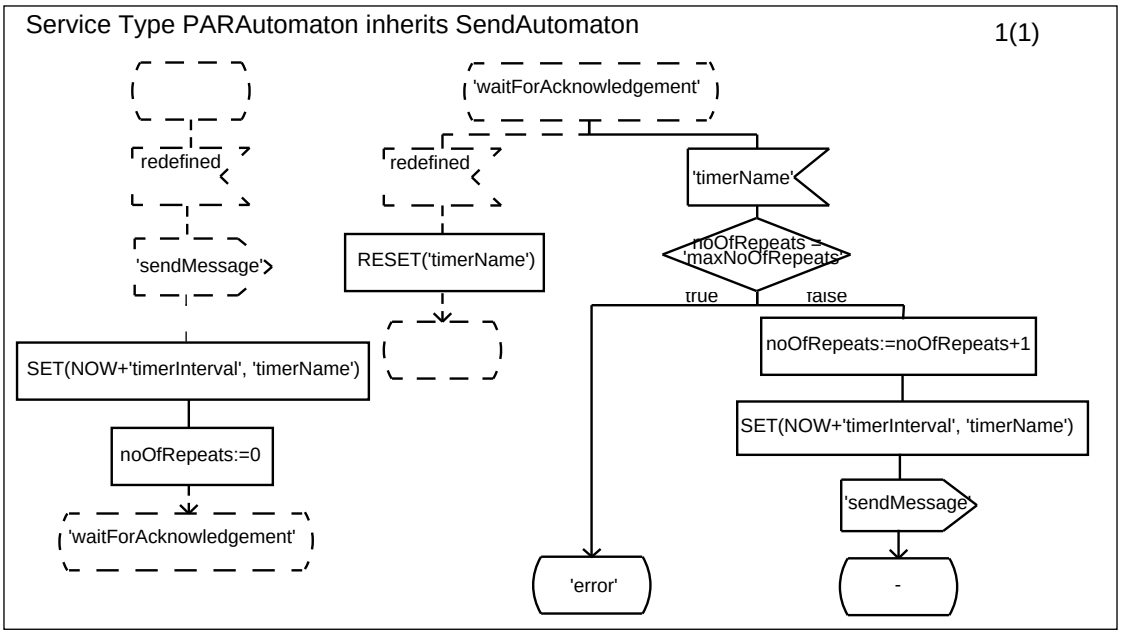


Message scenario:



SDL-Fragment:

PARAutomaton



Syntactical embedding

Specialization: redefine a given sending transition of *SendAutomaton* by supplementation of timer and counter initialization. The corresponding receiving transition(s) of *SendAutomaton* are supplemented by a timer reset. Another transition for timeout handling with retransmission is added.
Renaming: the timer *timerName* and the variable *noOfRepeats* may be renamed but are required to be locally unique. The state *error* may be identified with any state in the given service.

Semantic properties:

Property C.1: *If the assumptions stated below hold,* PARAutomaton *will eventually receive an acknowledgement from* AcknowledgeAutomaton *after sending a* sendMessage, *or* PARAutomaton *will enter the error state after* maxNoOfRepeats *unsuccessful retransmissions. The assumptions are:*

- *The communication channel between* PARAutomaton *and* AcknowledgeAutomaton *for transmission of* sendMessage *and corresponding acknowledgement signals neither disrupts nor creates messages.*
- *The communication channel may lose messages but* timerInterval *is greater than the maximal round trip time of* sendMessage *and corresponding acknowledgement.*
- AcknowledgeAutomaton *merely discards duplicate* sendMessages *or reacts on duplicates the same way (from the perspective of* PARAutomaton*) as on the original* sendMessage.

Redefinition:

The embedded SDL-fragment may be redefined e.g. for the purpose of message loss reporting or logging. The following property determines what kind of redefinition will be allowed.

Property C.2: *Property • still holds, if the TimerControlledRepeat pattern is redefined by the introduction of additional statements, which do not disrupt or bypass the thread of control from the predefined timeout input to the predefined repetitive output statement as well as the* error *state. Furthermore the timer* timerName *and the counter* noOfRepeats *must not be manipulated.*

Cooperative usage:

TimerControlledRepeat can cause duplicates of messages and may therefore be used in conjunction with DuplicateIgnore/Handle in order to ensure proper duplicate processing of *AcknowledgeAutomaton*.

DUPLICATEIGNORE

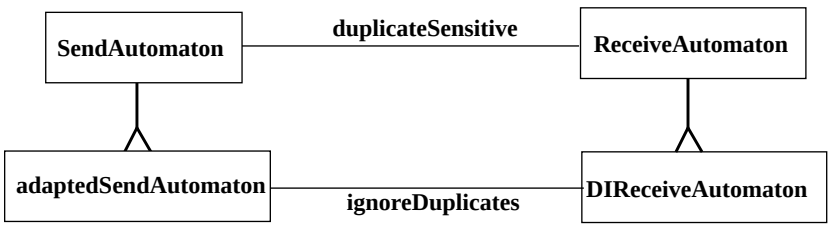
Intent:

DuplicateIgnore upgrades a message exchange between two automata *SendAutomaton* and *ReceiveAutomaton* for the case of possible message duplication. Duplicate messages are detected by a message identifier, that is unique during the lifetime of *DIReceiveAutomaton*. Furthermore, duplicate messages are simply discarded, i.e. the reaction to the original message is not repeated.

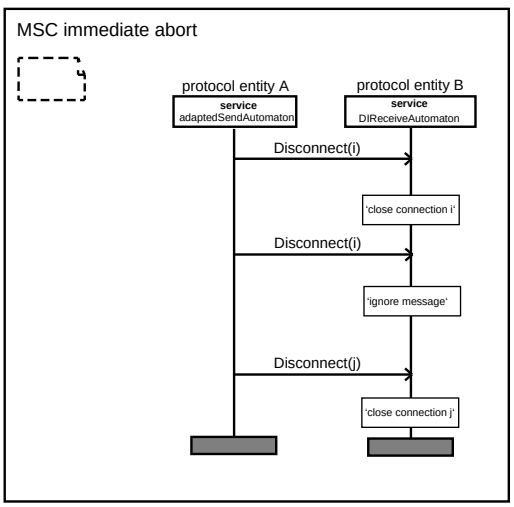
Motivation:

Retransmissions due to certain error control mechanisms may lead to duplication of messages. If no reaction to duplicate messages is expected, DuplicateIgnore can be applied to filter them out.

Structure:

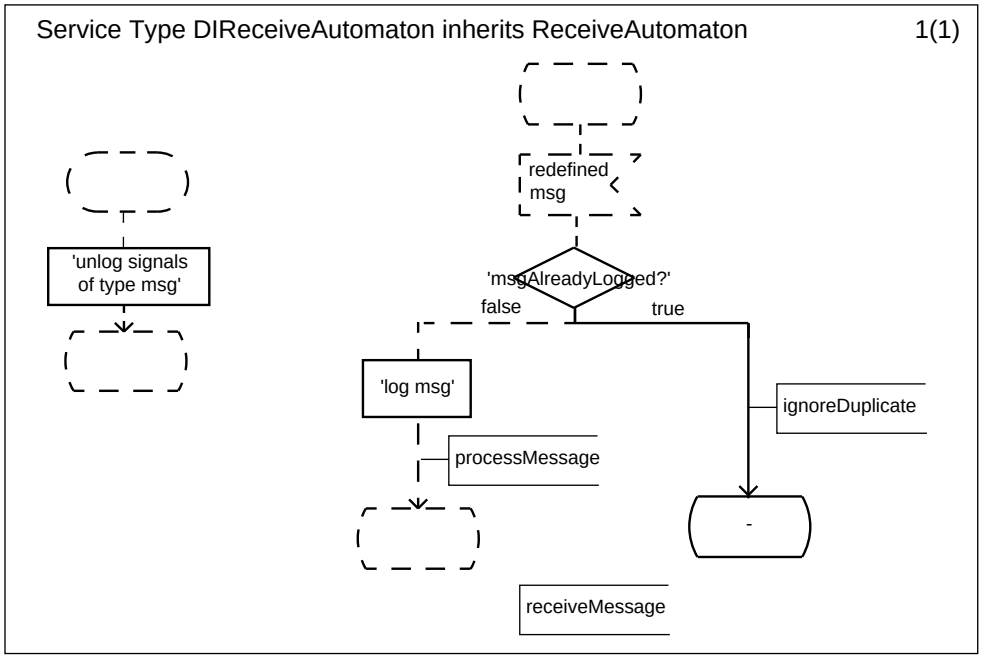


Message scenario:



SDL-Fragment:

DIReceiveAutomaton



Syntactical embedding

Specialization: redefine the start transition of *ReceiveAutomaton* by resetting all logged signals of type *msg*. Furthermore, redefine all transitions with input signal *msg* by supplementing a test if the message has already been received (*msgAlreadyLogged*?) and a transition branch *ignoreDuplicate*, that merely discards the signal. The corresponding branch that normally processes the message is supplemented by a logging mechanism for the signal *msg*.

Semantic properties:

Property D.1: *If the developer adds mechanisms for identification and logging of signals of type* msg*, duplicates of type* msg *are filtered out by DIReceiveAutomaton.*

Redefinition:

not allowed

Cooperative usage:

DuplicateIgnore may be used in conjunction with TimerControlledRepeat in order to upgrade unreliable communication channels.

DUPLICATEHANDLE

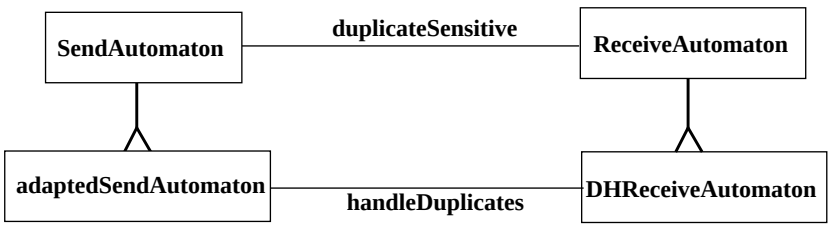
Intent:

DuplicateHandle upgrades a message exchange between two automata *SendAutomaton* and *ReceiveAutomaton* for the case of possible message duplication. Duplicate messages are detected by a message identifier, that is unique during the lifetime of *DHReceiveAutomaton*. However, duplicate messages rely on a certain reaction of *DHReceiveAutomaton*, i.e. duplicates must not be discarded.

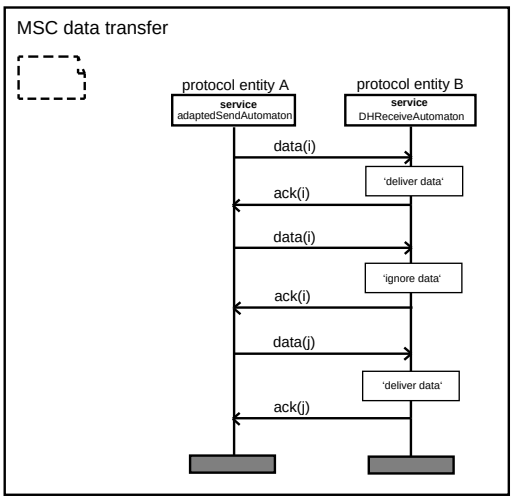
Motivation:

Retransmissions due to certain error control mechanisms may lead to duplication of messages. If a certain reaction to duplicate messages is expected (e.g., retransmission of acknowledgements), DuplicateHandle can be applied.

Structure:

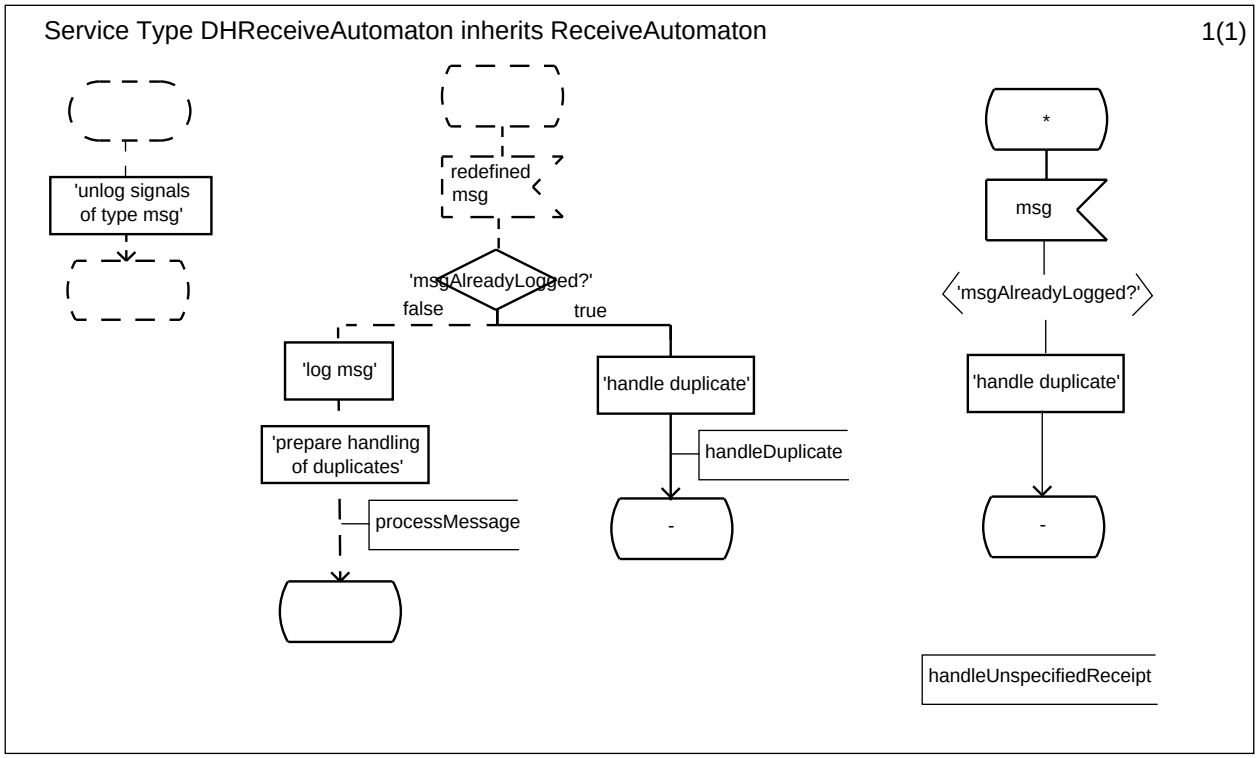


Message scenario:



SDL-Fragment:

DHReceiveAutomaton



Syntactical embedding

Specialization: redefine the start transition of *ReceiveAutomaton* by resetting all logged signals of type *msg*. Furthermore, redefine all transitions with input signal *msg* by supplementing a test if the message has already been received (*msgAlreadyLogged*?) and a transition branch *ignoreDuplicate*, that properly handles the duplicate. The corresponding branch that normally processes the message is supplemented by a logging mechanism for the signal *msg* and statements to prepare proper handling of duplicates. Additionally the transition *handleUnspecifiedReceipt* is added to the given SDL service *ReceiveAutomaton*.

Semantic properties:

Property E.1: *If the developer adds mechanisms for identification, logging and handling of signals of type* msg, *duplicates of type* msg *are always handled by DHReceiveAutomaton.*

Redefinition:

not allowed

Cooperative usage:

DuplicateHandle may be used in conjunction with TimerControlledRepeat in order to upgrade unreliable communication channels.

DYNAMICENTITYSET

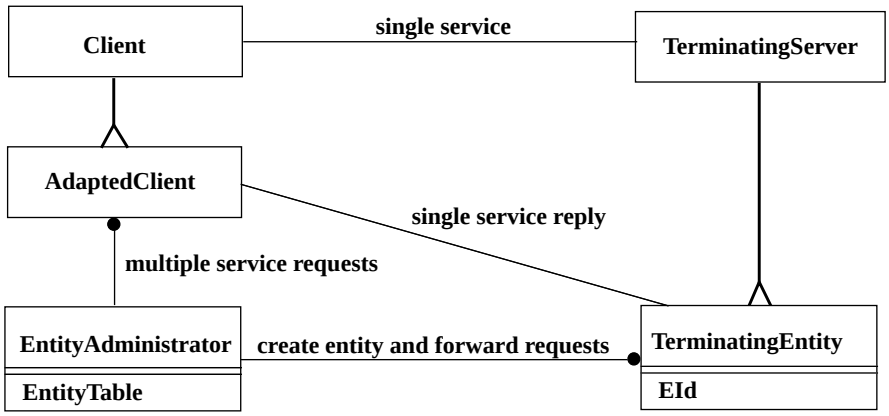
Intent:

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 client a 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.

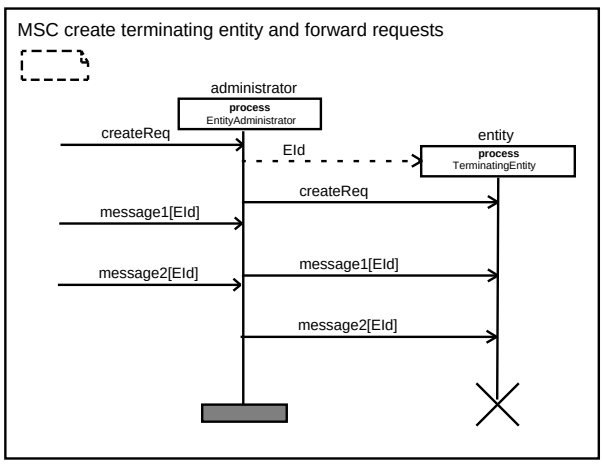
Motivation:

If a communication subsystem administers several 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:

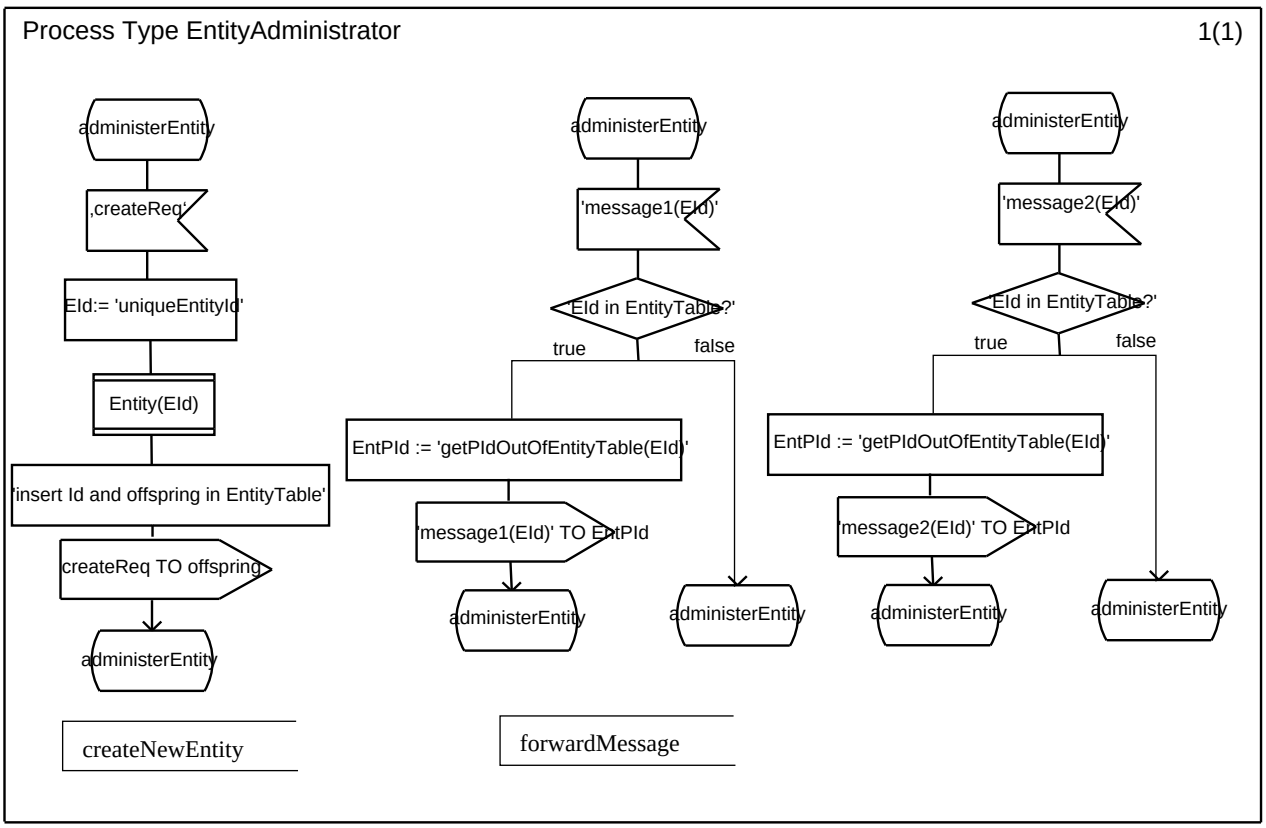


Message scenario:



SDL-Fragment:

EntityAdministrator



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 *createReq*) 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: *createReq*, *message1*, and *message2* correspond with those messages the *client* sends to its *TerminatingServer*, where *createReq* 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:

Property F.1: *If the assumptions stated below hold, the same service as provided by* TerminatingServer *will be offered several times by the server entities of type* TerminatingEntity, *and each server entity will only receive messages belonging to it. The assumptions are:*

- *The developer takes care that the AdaptedClients are informed about the EId of their corresponding server entity and always add this EId to the output signals which are sent to the server entity (except createReq)*

Redefinition:

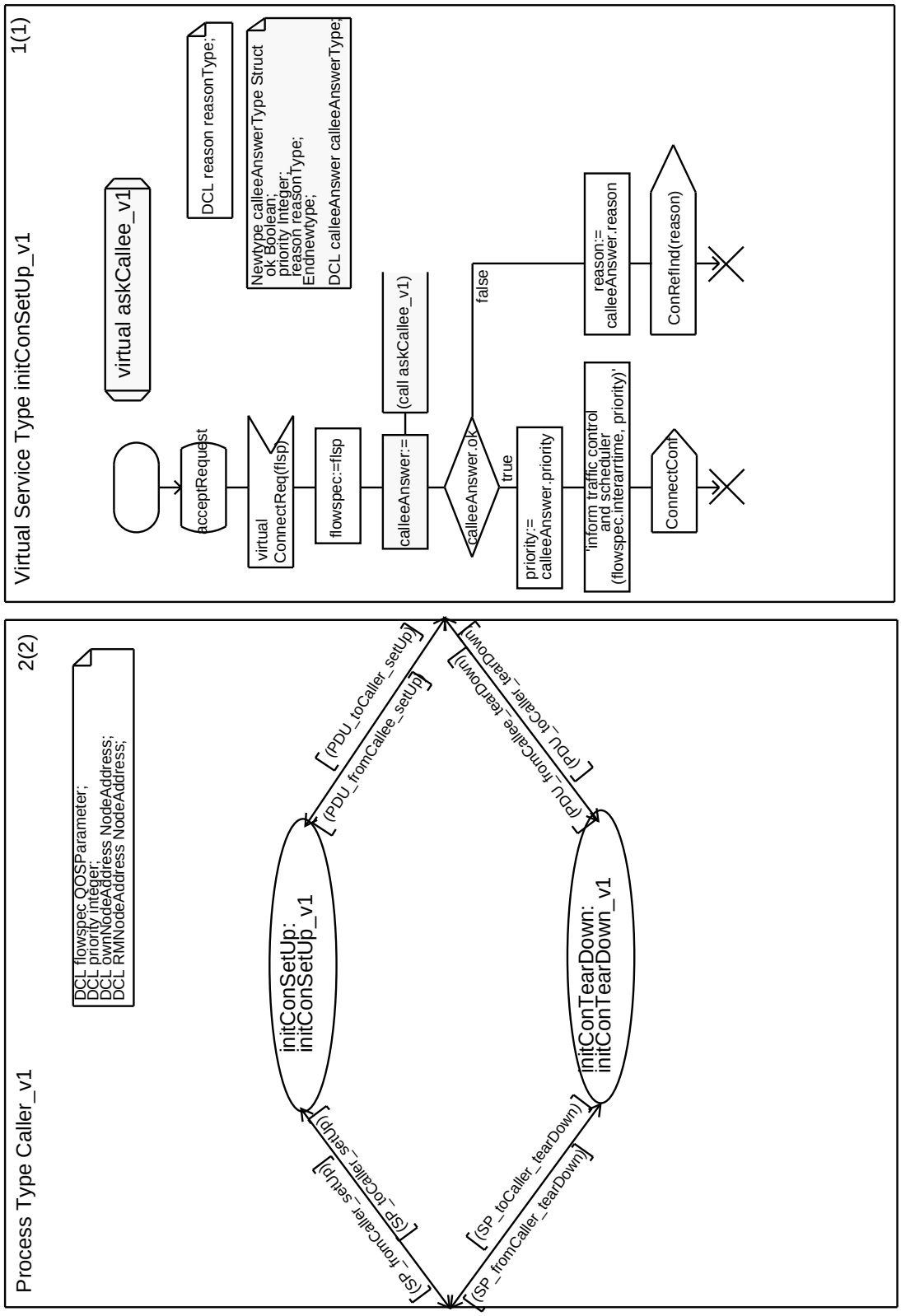
EntityAdministrator may be redefined in order to limit the number of server entities active at the same time or inform the sender of a message if no corresponding server entity could be found in the *EntityTable*. The following property determines what kind of redefinition will be allowed.

Property F.2: *Property F.1 still holds, if the DynamicEntitySet pattern is redefined by the introduction of additional statements, which do not manipulate the EId and PId entries of the* EntityTable.

Appendix B

SDL specification[1]

B. 1 Development step 1

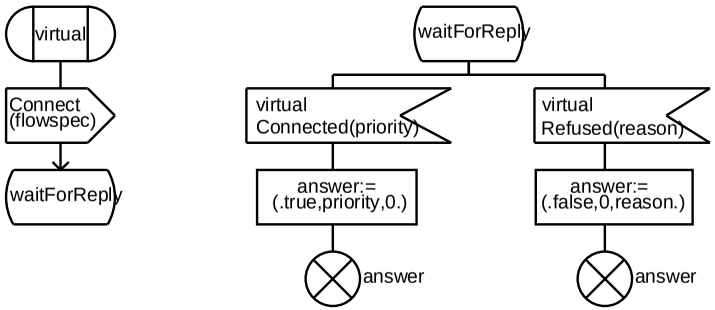


1. To get a better overview of the SDL diagrams we have omitted the names of signal routes and channels.

Procedure askCallee_v1

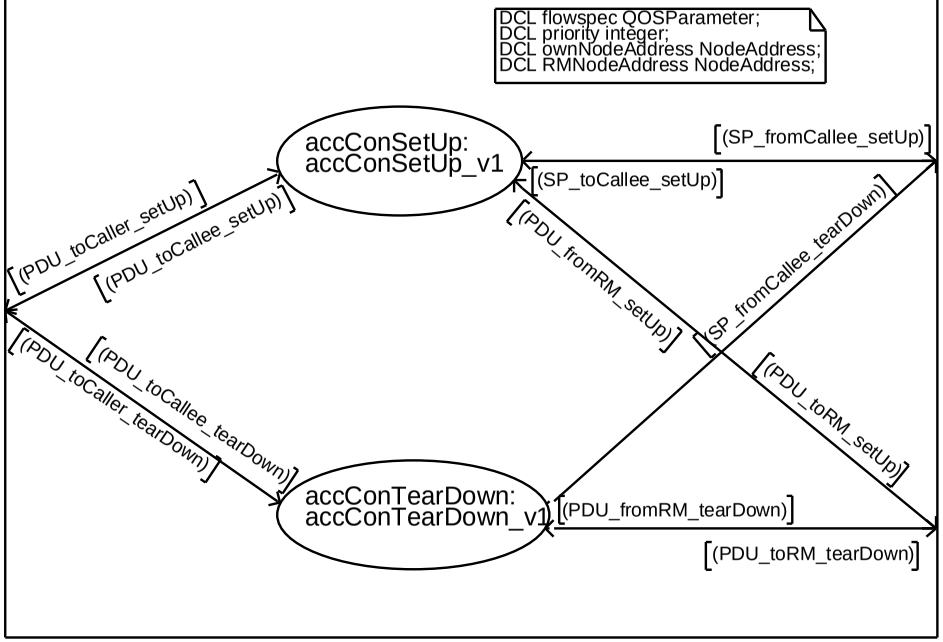
1(1)

;returns answer Answertype;



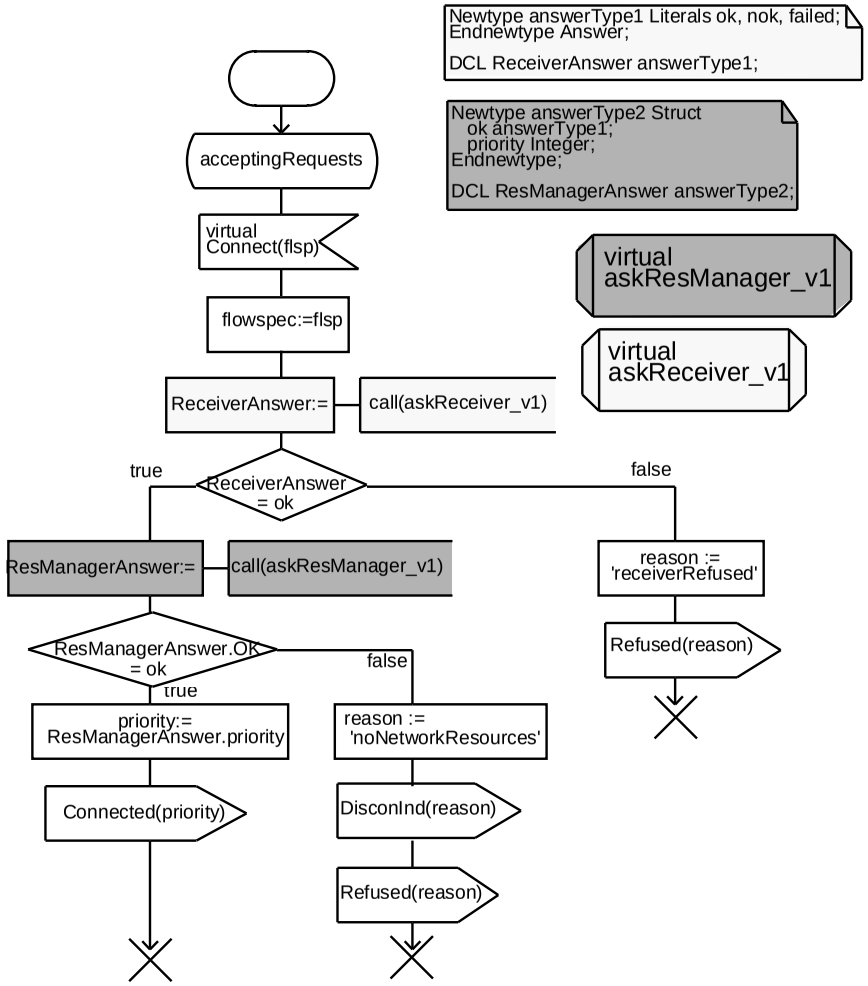
Process Type Callee_v1

2(2)



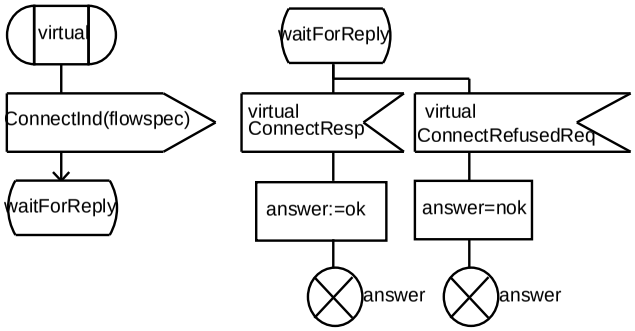
Virtual Service Type accConSetUp_v1

1(1)



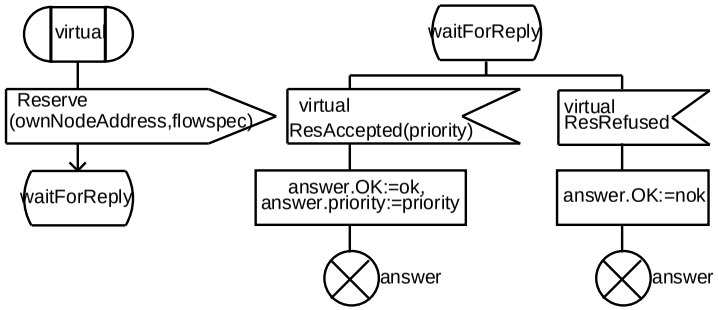
virtual Procedure askReceiver_v1 1(1)

;
returns answer ReceiverAnswer;

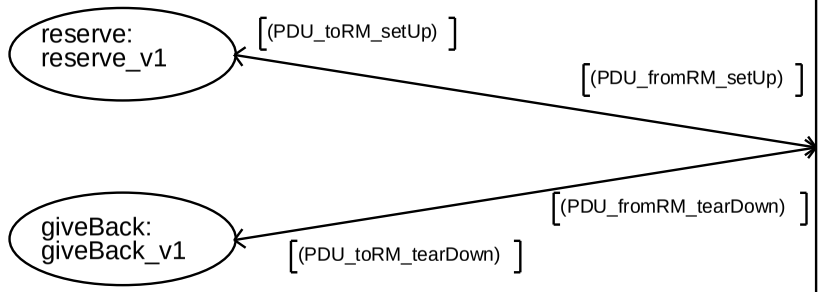


virtual Procedure askResManager_v1 1(1)

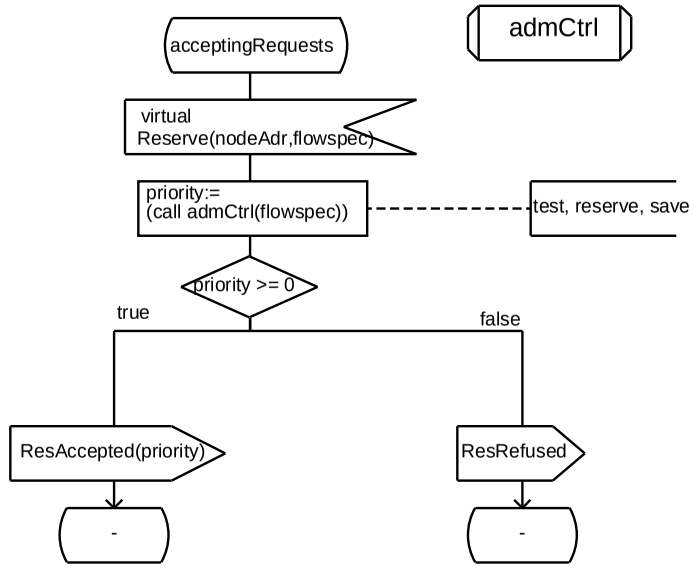
;
returns answer answerType2;



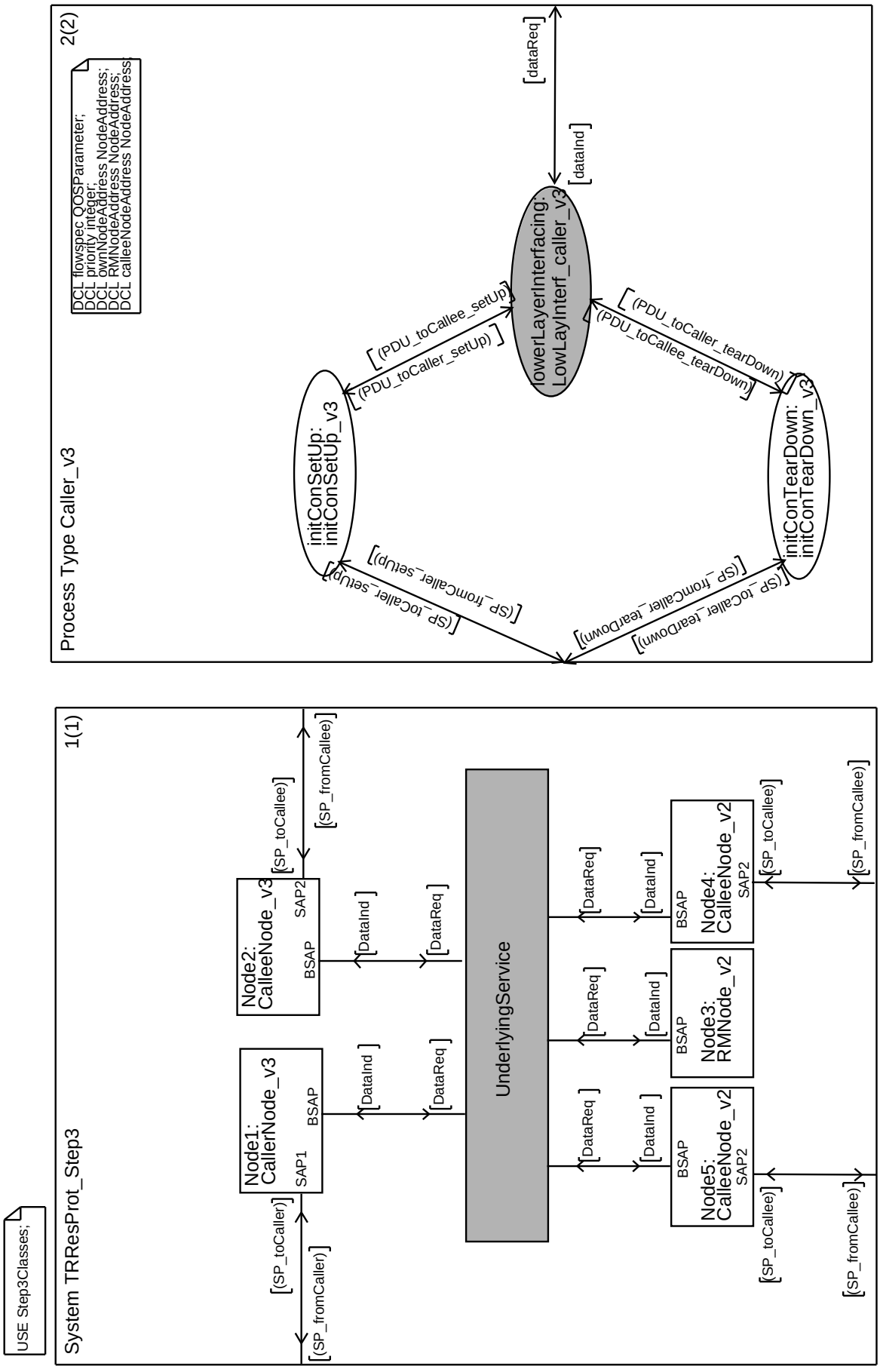
Process Type ResourceManager_v1 2(2)



Virtual Service Type reserve_v1 1(1)



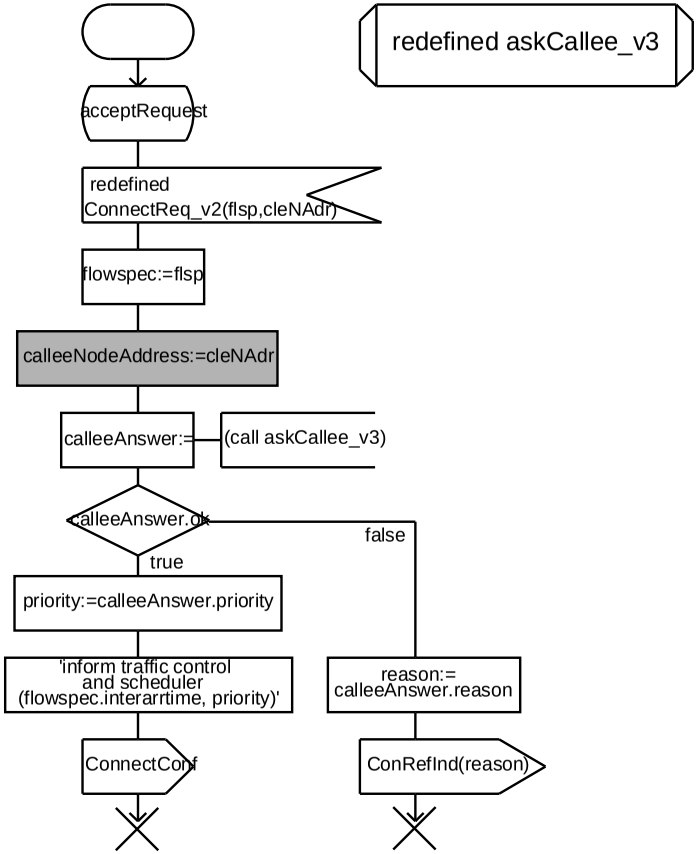
B. 2 Development step 2 and step 3



Redefined Service Type initConSetUp_v3

1(1)

inherits initConSetUp_v2

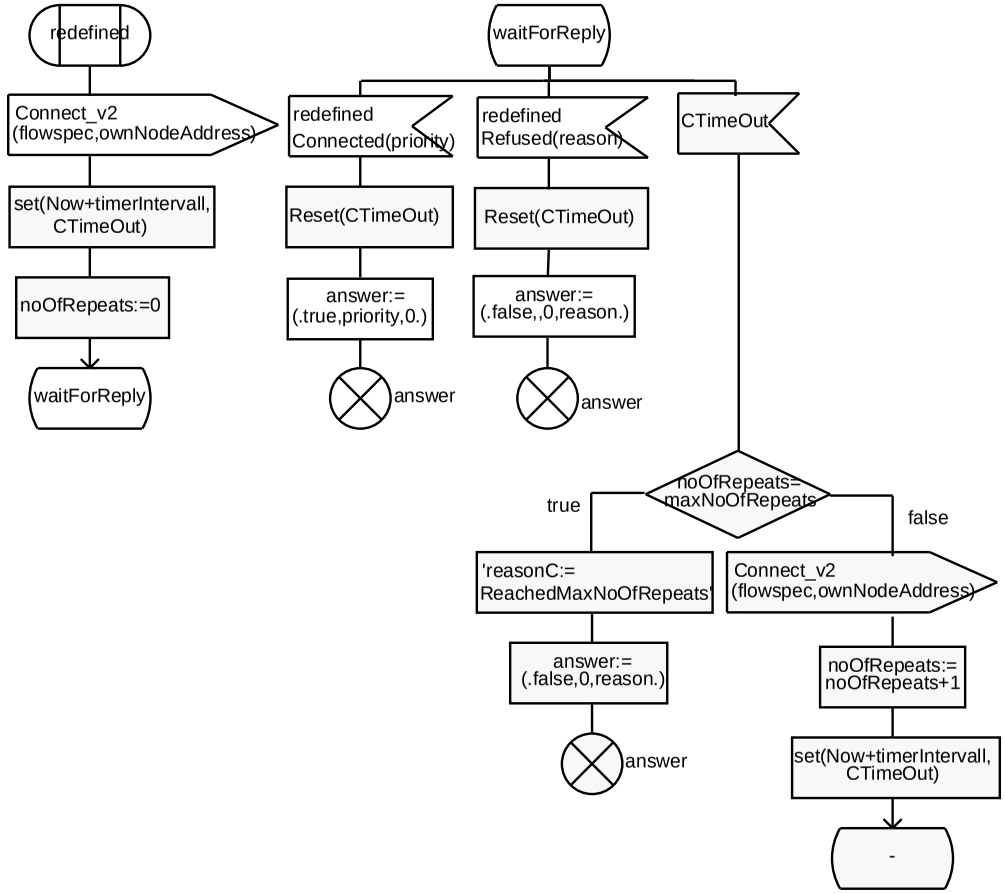


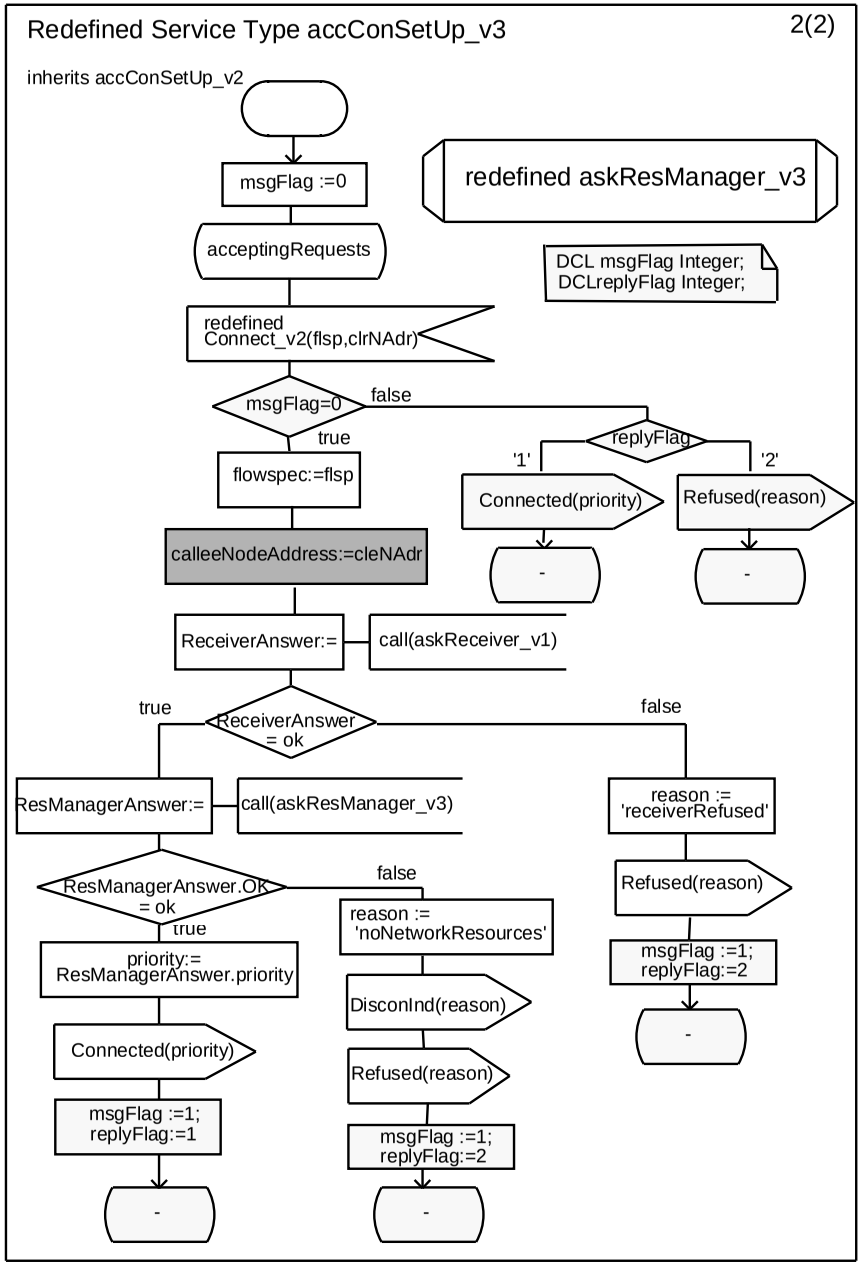
Procedure askCallee_v3

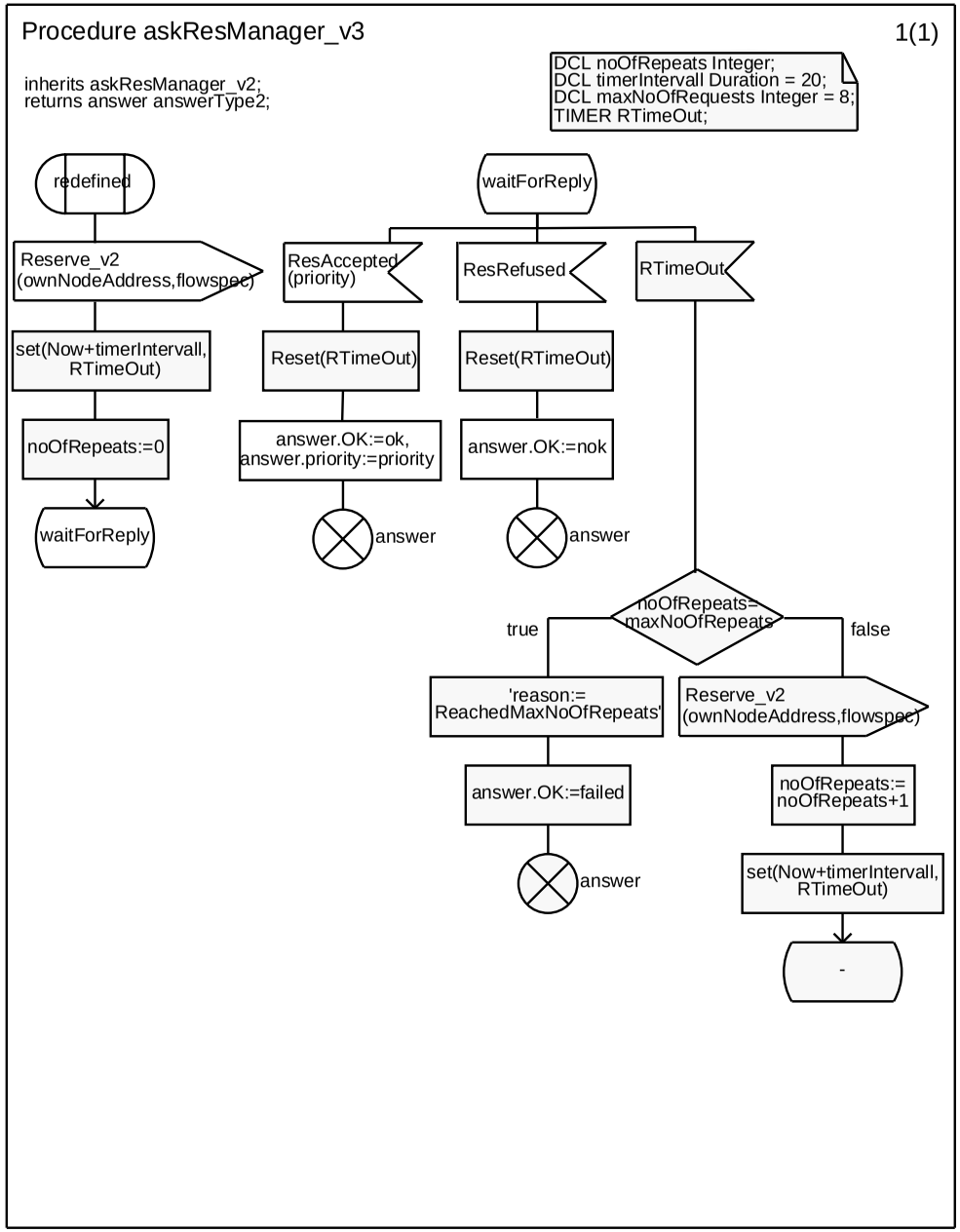
1(1)

inherits askCallee_v2
;returns answer Answertype;

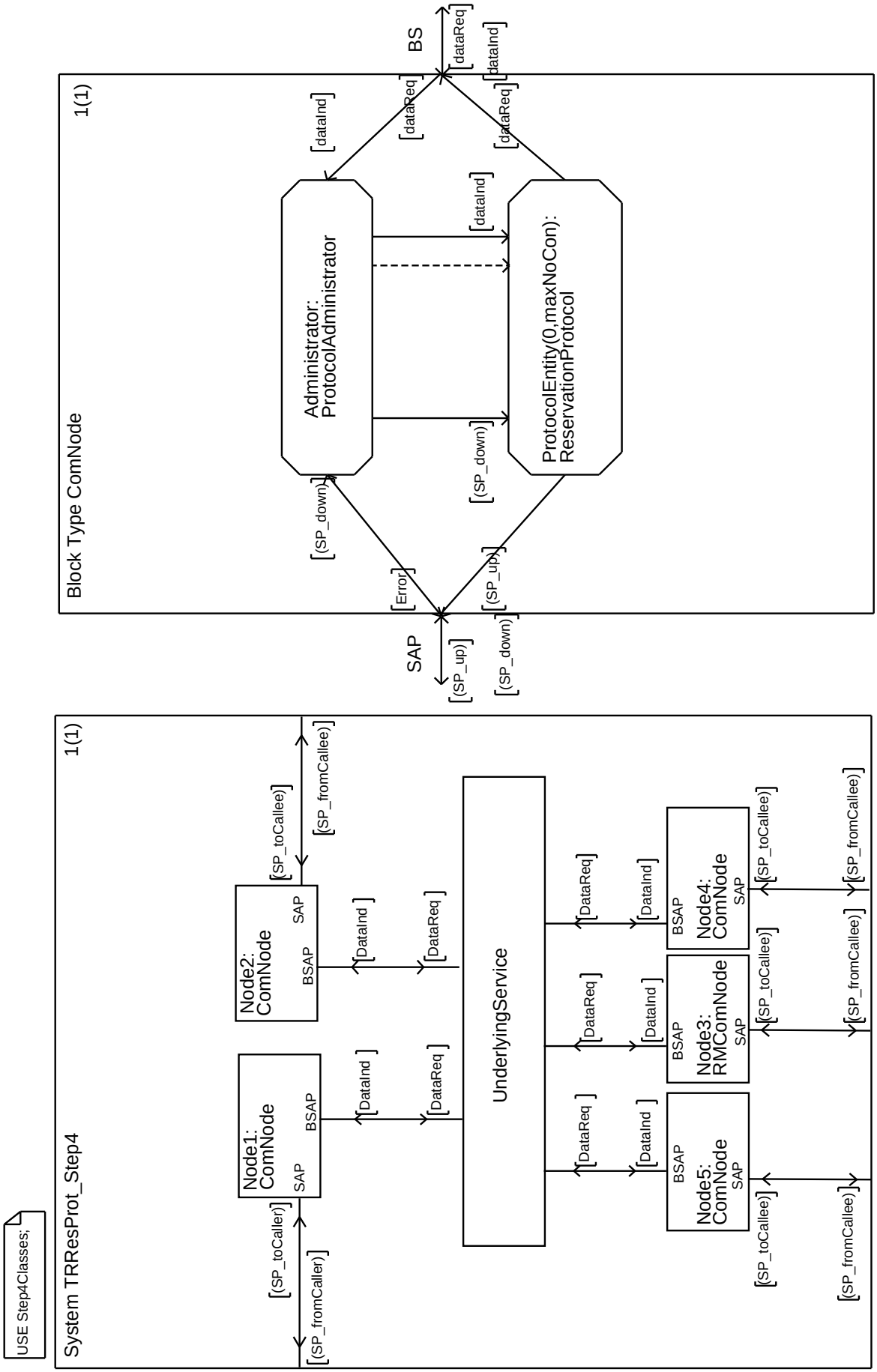
DCL noOfRepeats Integer;
DCL timerIntervall Duration = 20;
DCL maxNoOfRequests Integer = 8;
TIMER CTimeOut;



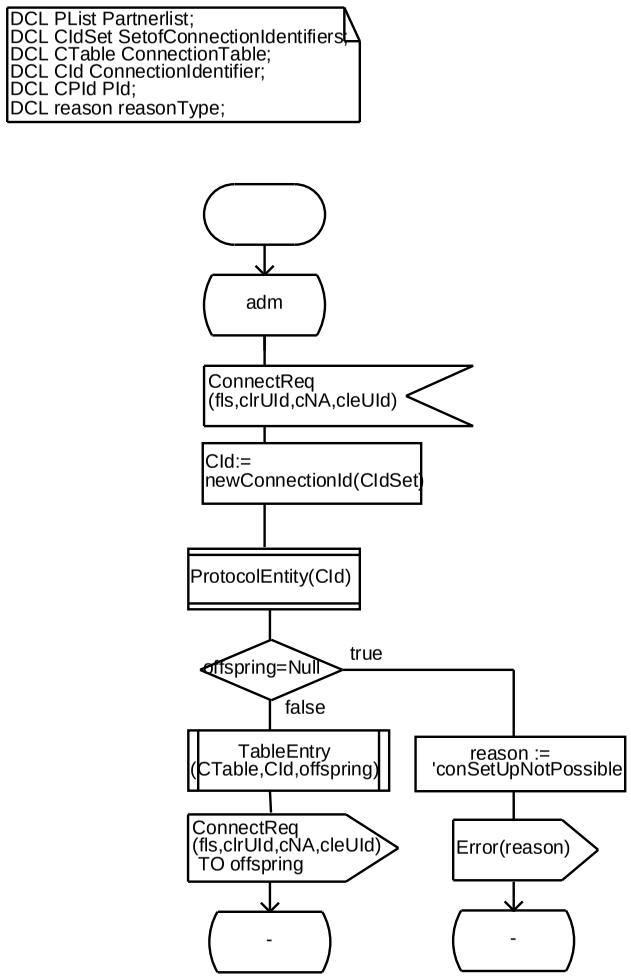

Redefined Service Type accConSetUp_v3
2(2)
inherits accConSetUp_v2
msgFlag :=0
redefined askResManager_v3
acceptingRequests
DCL msgFlag Integer;
DCLreplyFlag Integer;
redefined
Connect_v2(flsp,clrNAdr)
msgFlag=0
false
true
replyFlag
'1'
'2'
flowspec:=flsp
Connected(priority)
Refused(reason)
calleeNodeAddress:=cleNAdr
-
-
ReceiverAnswer:=
call(askReceiver_v1)
true
ReceiverAnswer
= ok
false
ResManagerAnswer:=
call(askResManager_v3)
reason :=
'receiverRefused'
ResManagerAnswer.OK
= ok
false
Refused(reason)
true
reason :=
'noNetworkResources'
priority:=
ResManagerAnswer.priority
msgFlag :=1;
replyFlag:=2
DisconInd(reason)
Connected(priority)
Refused(reason)
msgFlag :=1;
replyFlag:=1
msgFlag :=1;
replyFlag:=2
-
-
-


Procedure askResManager_v3
1(1)
inherits askResManager_v2;
returns answer answerType2;
DCL noOfRepeats Integer;
DCL timerIntervall Duration = 20;
DCL maxNoOfRequests Integer = 8;
TIMER RTimeOut;
redefined
waitForReply
Reserve_v2
(ownNodeAddress,flowspec)
ResAccepted
(priority)
ResRefused
RTimeOut
set(Now+timerIntervall,
RTimeOut)
Reset(RTimeOut)
Reset(RTimeOut)
noOfRepeats:=0
answer.OK:=ok,
answer.priority:=priority
answer.OK:=nok
waitForReply
answer
answer
noOfRepeats=
maxNoOfRepeats
true
false
'reason:=
ReachedMaxNoOfRepeats'
Reserve_v2
(ownNodeAddress,flowspec)
answer.OK:=failed
noOfRepeats:=
noOfRepeats+1
answer
set(Now+timerIntervall,
RTimeOut)
-

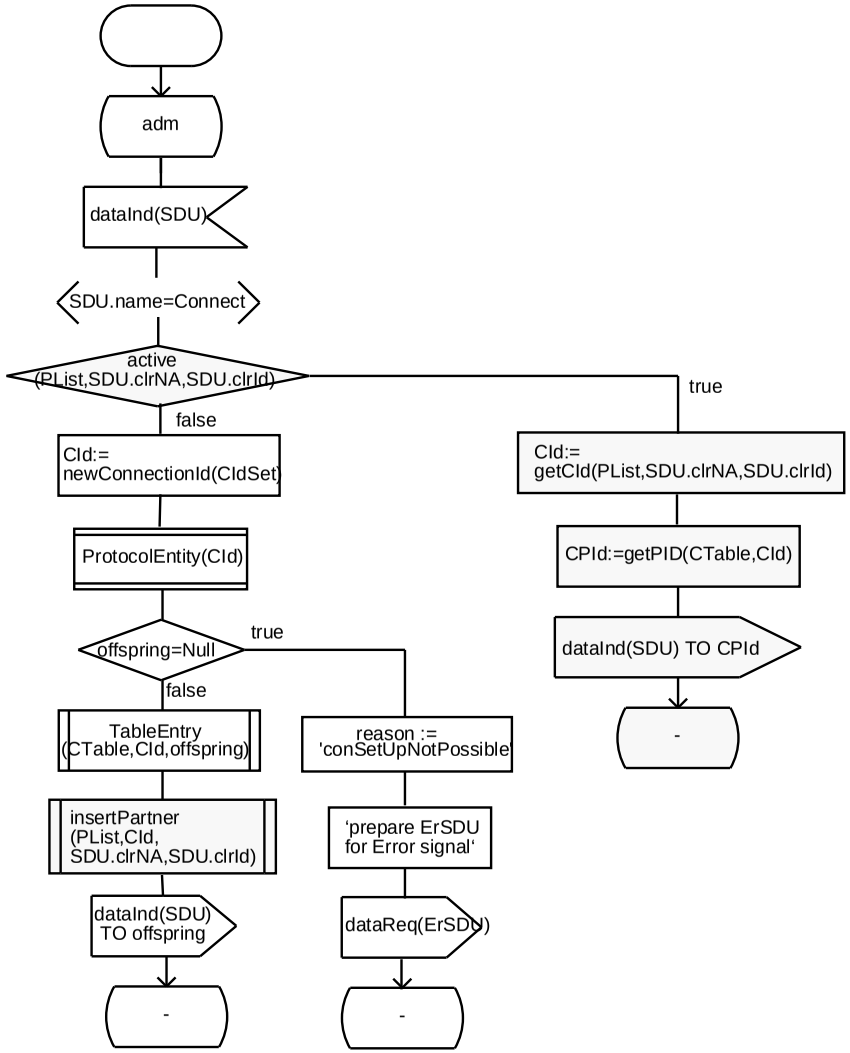
B. 3 Development step 4



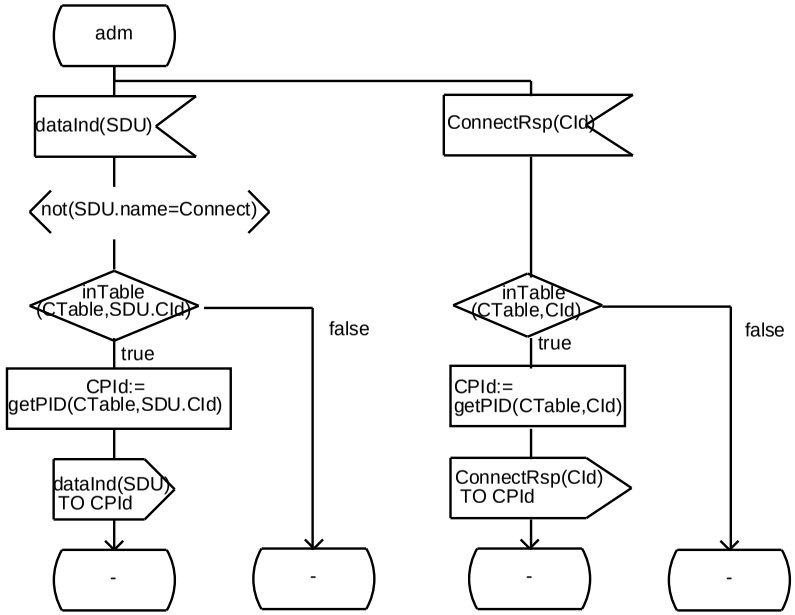
Process Type ProtocolAdministrator 1(4)



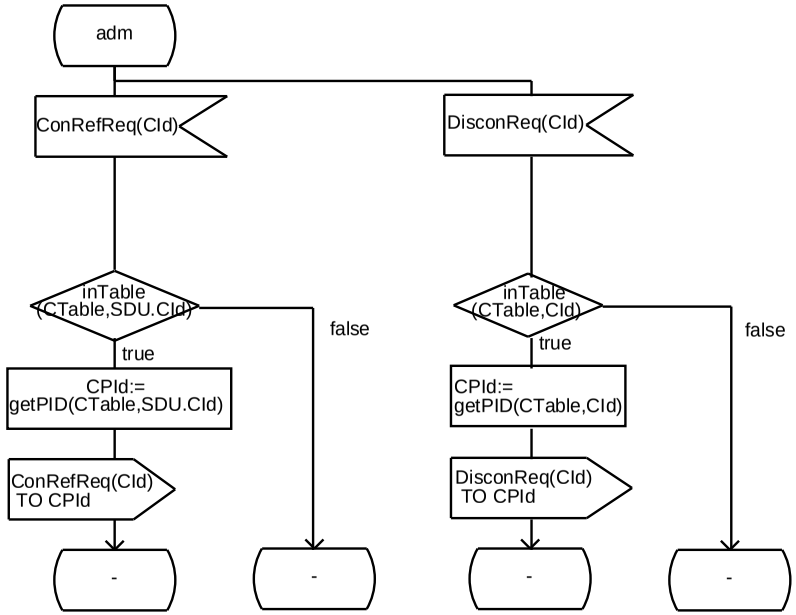
Process Type ProtocolAdministrator 2(4)

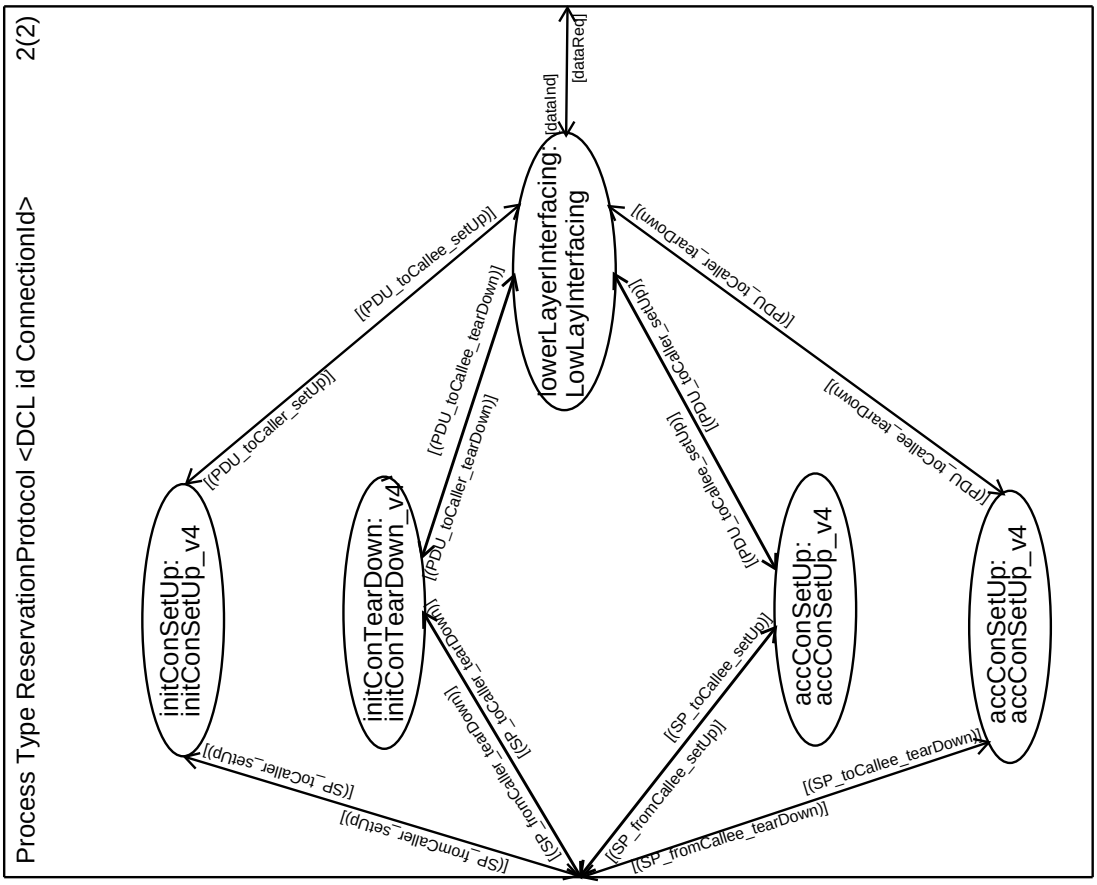


Process Type ProtocolAdministrator 3(4)



Process Type ProtocolAdministrator 4(4)




Process Type ReservationProtocol <DCL id ConnectionId>
2(2)
initConSetUp:
initConSetUp_v4
initConTearDown:
initConTearDown_v4
lowerLayerInterfacing:
LowLayInterfacing
accConSetUp:
accConSetUp_v4
accConSetUp:
accConSetUp_v4
[dataInd]
[dataReq]
[(PDU_toCallee_setUp)]
[(PDU_toCaller_setUp)]
[(PDU_toCallee_tearDown)]
[(PDU_toCaller_tearDown)]
[(PDU_toCaller_setUp)]
[(PDU_toCallee_setUp)]
[(PDU_toCaller_tearDown)]
[(PDU_toCallee_tearDown)]
[(SP_toCaller_setUp)]
[(SP_fromCaller_setUp)]
[(SP_fromCaller_tearDown)]
[(SP_toCaller_tearDown)]
[(SP_toCallee_setUp)]
[(SP_fromCallee_setUp)]
[(SP_fromCallee_tearDown)]
[(SP_toCallee_tearDown)]