Pattern-based Configuring of the Internet Stream Protocol ST2+ with SDL

Birgit Geppert, Frank Rößler, Philipp Schaible

SFB 501 Report 12/97

Pattern-based Configuring of the Internet Stream Protocol ST2+ with SDL

Birgit Geppert, Frank Rößler, Philipp Schaible

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

Report 12/97

Sonderforschungsbereich 501

Computer Networks Group

Computer Science Department

University of Kaiserslautern

P.O. Box 3049

67653 Kaiserslautern

Germany

Pattern-based Configuring of the Internet Stream Protocol ST2+ with SDL

Birgit Geppert, Frank Rößler, Philipp Schaible

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

Abstract

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

1 Introduction

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

The reuse of predesigned solutions for recurring design problems is of major concern in object oriented software development in general. During the past few years, design patterns have emerged as an especially fruitful approach to software reuse [1] [2] [4] [9]. Contrary to the traditional paradigm of class and function libraries, which is solely concerned with *code* reuse, design patterns aim to focus on the invariant parts of a design solution and offer by far more flexibility for adaptation to the embedding context. That is, the potential of reuse is substantially increased. Additionally, design patterns proved helpful in guiding the instantiation and documentation of frameworks, which provide larger-scale reuse of the overall architecture and design from a certain application domain.

In [5] [7] [8] we present the SDL-pattern approach, which on principle adopts the patterns concept for the design of communication protocols. However, in order to increase productivity and to assure high quality of the resulting communication subsystem, we preferably apply a formal description technique (FDT) as design language. In this context we suggest to formalize design patterns, too. That is, we precisely specify knowledge about pattern application or its impact on the embedding context. We consider formalization to be a prerequisite for tool support and validation of pattern application.

The FDT of our choice is SDL [15]. Thus, e.g., instantiation of an SDL pattern can be defined by precise *embedding rules* in terms of the SDL syntax. Similarly, the semantic model of SDL allows to precisely state assumptions for adequate pattern application and it is possible to express resulting properties of the embedding context. This is a major improvement compared to conventional design patterns, which mainly rely on natural language based pattern description and still have to leave pattern application to a large degree to the personal skills of the system designer. Note, however, that the enhancements work within the scope of an FDT with its formal syntax and semantics.

As an additional advantage, protocol design based on SDL patterns leads to formal specifications of communication protocols, which can be further used for validation and code generation. Simulating or analysing a distributed system (especially its flow of control) before it is implemented helps detect design errors in early stages of development.

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

2 The Formal Description Technique SDL

SDL is a description technique with a formal syntax and semantics and is designed for the specification of reactive, distributed systems, in particular communicating systems. The most recent version was standardized by the International Telecommunication Union (ITU) in 1992 [15] with some updates made in 1996. It offers a textual as well as a graphical representation. SDL is an object-oriented language capable for the description of architecture, behaviour, and data. It is in widespread use in industry and is well-supported by commercial and public domain development environments.

Figure 1 shows an example SDL specification. An SDL system specification is structured into blocks (*Example1*) where each block is composed either of a set of blocks or a set of processes (as in the case of *block1*). System behaviour is modelled as a set of communicating extended finite state machines (CEFSMs) each represented by an SDL process (see *PType1*) with an unbounded FIFO input queue. SDL processes run concurrently and communicate asynchronously by signal exchange. If the signal at the head of the process input queue cannot be processed in the current state it is implicitly consumed, i.e., discarded by default. An SDL process may be further structured in SDL services (e.g., *process2*), which themselves represent CEFSMs. In this case, an SDL process appears as the product automaton of its services. The services of a process run sequentially and share the input queue of the process. SDL structural concepts such as blocks, processes, or services are well-suited for mapping a pattern's design elements, i.e., those parts of the context specification that are to be refined by pattern application [7].

As an object-oriented language, SDL allows parameterized type definitions of blocks, processes, services and signals as well as specialization by bounding parameters, adding properties (e.g., new input signals), or redefining virtual types or transitions. An example for a parameterized process type definition is PType1 in Figure 1. Formal context parameters (*sig1, sig2, sig3, sig4*) can be set to actual context parameters (*s1, s2, s3, s6*) of the actual scope unit by instantiation (see *process1*). Object oriented concepts not supported by SDL are multiple inheritance and dynamic binding.

3 Protocol Engineering Based on SDL Patterns

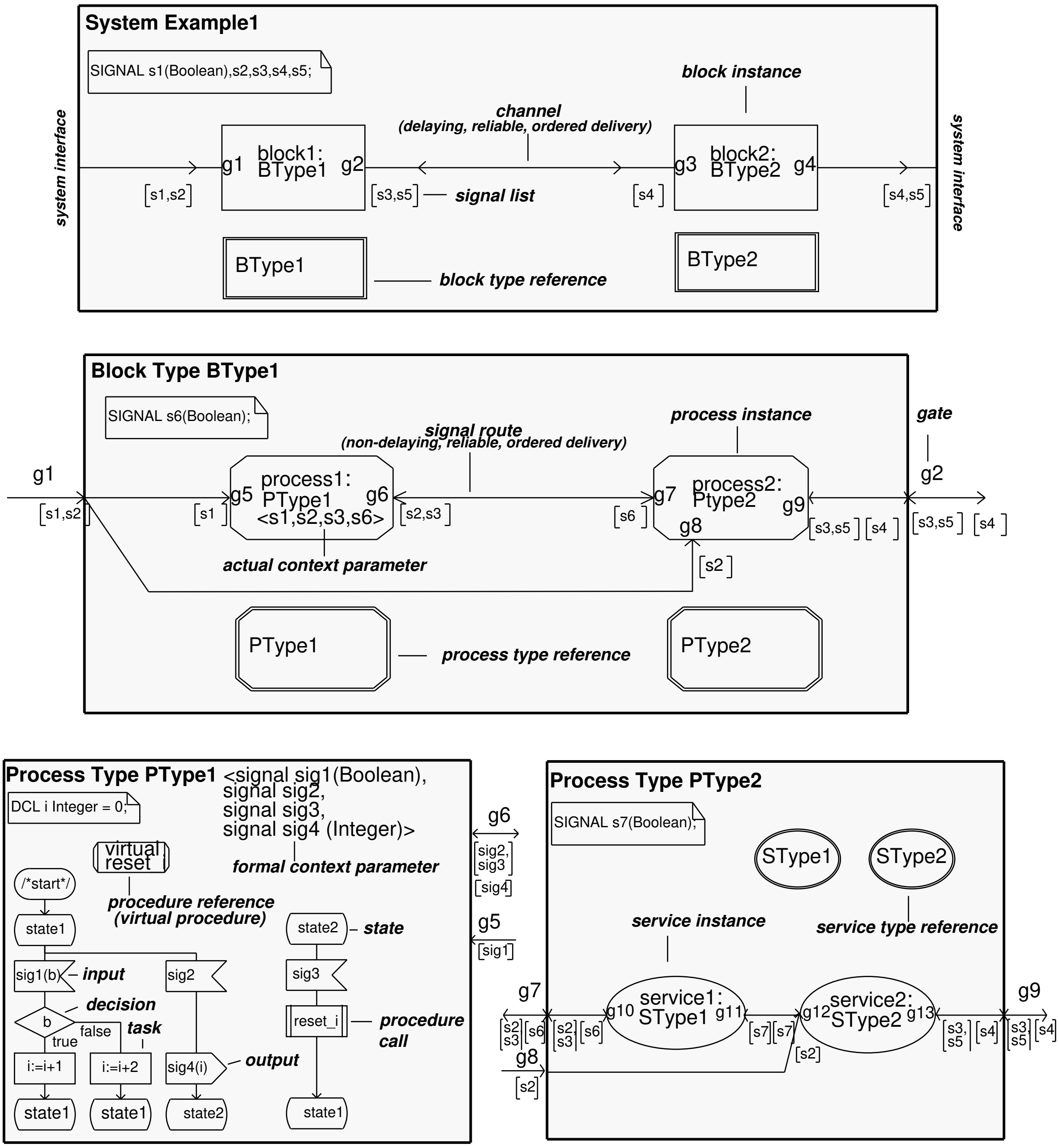


Fig.1: Example of an SDL specification

This section summarizes concepts for a construction set of protocol building blocks from which a protocol designer can select reusable artifacts and configure them to a customized, formal protocol specification. In order to get highly flexible artifacts and to increase the quality of resulting products, we combine the design patterns approach with SDL. For further details the reader is referred to [5] [6] [8].

3.1 SDL patterns as reusable protocol building blocks

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

SDL patterns are expected to offer the same advantages as commonly attributed to conventional design patterns: patterns capture solutions, which have evolved over time and serve as an elegant way to make designs more flexible, modular, reusable, and understandable. They reflect experiences gained in prior developments and therefore help designers to reuse successful designs and architectures. As a consequence, the design process becomes faster and the number of design errors decreases.

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

The mere syntactical part of the design solution is defined by a generic *SDL-fragment*, which has to be instantiated and textually embedded into the context specification when applying the pattern. SDL-fragments represent context invariant parts of a design solution. Instantiation and embedding of SDL-fragments is prescribed in terms of *syntactical embedding rules*, which, e.g., guide renaming of abstract identifiers or specialization of embedding design elements. Usually pattern semantics is not completely captured by an SDL-fragment. Due to language constraints this would otherwise result in an overspecification of the design solution and reduce potential of reuse. Thus additional *semantic properties* are included specifying preconditions for pattern application as well as behavioural changes of the embedding context. Though semantic properties are currently stated in natural language, it is possible to express them precisely in a temporal logic. Also restrictions on the *redefinition* of pattern instances are specified in order to prevent a pattern's intent from being destroyed by subsequent development steps. A comparison to existing description templates for conventional design patterns [2] [4] is given in [5].

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

3.2 A pool of SDL patterns

The specification of the Internet Stream Protocol ST2+ uses a set of predesigned SDL patterns. In the following each pattern is introduced. Note that the SDL patterns below are not completely specified. We basically summarize a pattern's intent and skip the description items explained above. For a complete description of the SDL patterns the reader is referred to [6] [12]. The presented pool of SDL patterns is sufficient to specify nearly 100% of the control structure of ST2+.

- **SendReceive**
  The SendReceive pattern introduces an unconfirmed interaction between two automata. Being triggered, the first automaton sends a signal and continues work. The signal is eventually received by the second automaton.
- **BlockingRequestReply**
  The BlockingRequestReply pattern introduces a confirmed interaction between two automata. Being triggered, the first automaton (Requester, Figure 2) sends a request and is blocked until receiving a reply. After receiving a request, the second automaton (Replier, Figure 2) replies immediately. The generic SDL-Fragment is shown in Figure 2[1].
- **MultipleRequestsMultipleReplies** and **SingleRequestMultipleReplies**
  The MultipleRequestsMultipleReplies (SingleRequestMultipleReplies) pattern introduces a confirmed 1:n (1:1) interaction between one sending automaton and multiple receiving automata (one

1. We assume that the reader is familiar with the graphical symbols to express states and state transitions in SDL [15]

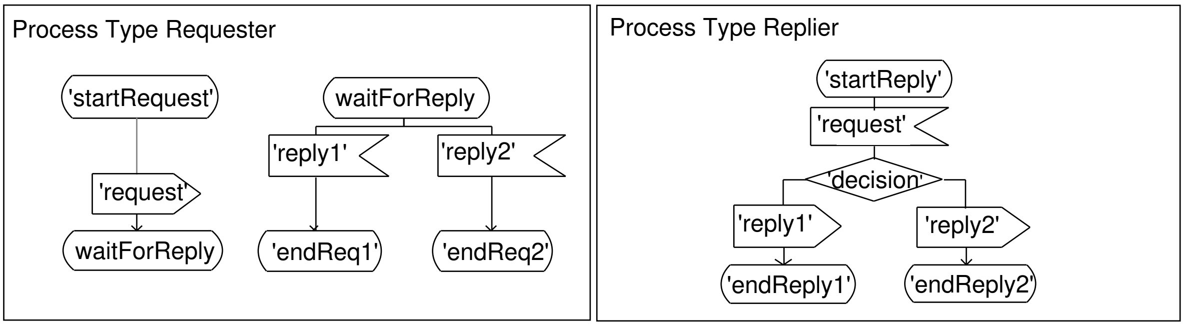


Fig.2: SDL-Fragment: BlockingRequestReply pattern

receiving automaton). Being triggered, the sender initiates several (one) requests and waits until all corresponding replies are received. After reception of a request, a receiver can send several replies.

- **Codex**
The Codex pattern provides mechanisms to allow two or more entities, which interact directly through SDL channels, to cooperate by the means of a given communication service. In general, the introduction of a basic service involves many specialities. Among others these are segmentation, reassembly, upgrade of basic service quality (e.g., in case of loss, disruption or duplication of messages), lower layer connection setup, or routing decisions. The Codex pattern is only concerned about a minimal subset of these functionalities, namely interfacing with the basic service by the means of service primitives (Figure 3). That is, Codex essentially provides a mapping from protocol data units to basic service primitives and vice versa.

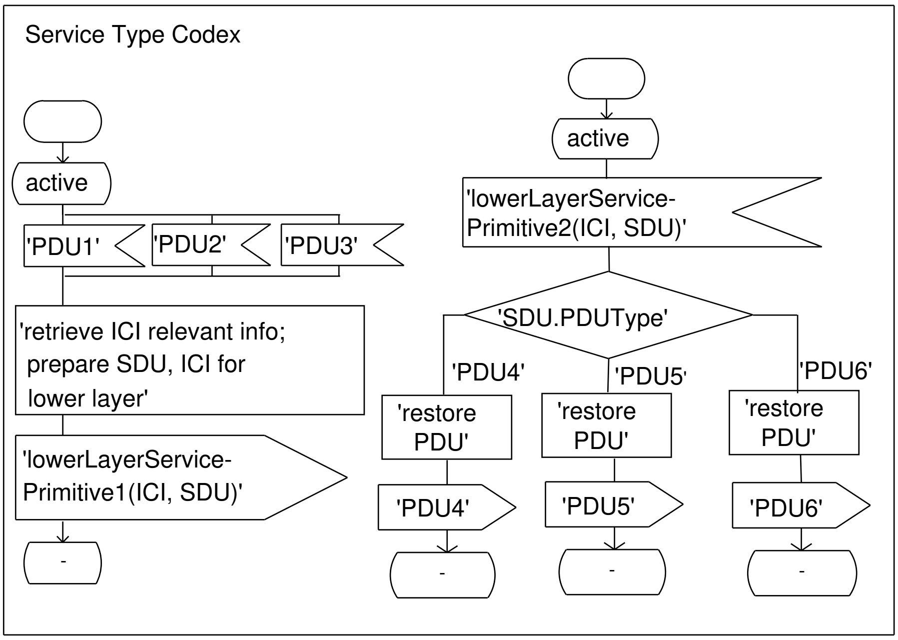


Fig.3: SDL-Fragment: Codex pattern

- **TimerControlledRepeat**
The TimerControlledRepeat pattern upgrades a confirmed message exchange between two automata in case of message loss. If an expected acknowledgement does not arrive before the expiry of

a timer, the message is repeated (positive acknowledgement with retransmission, Figure 4). This pattern does not deal with the problem of message disruption or duplication.

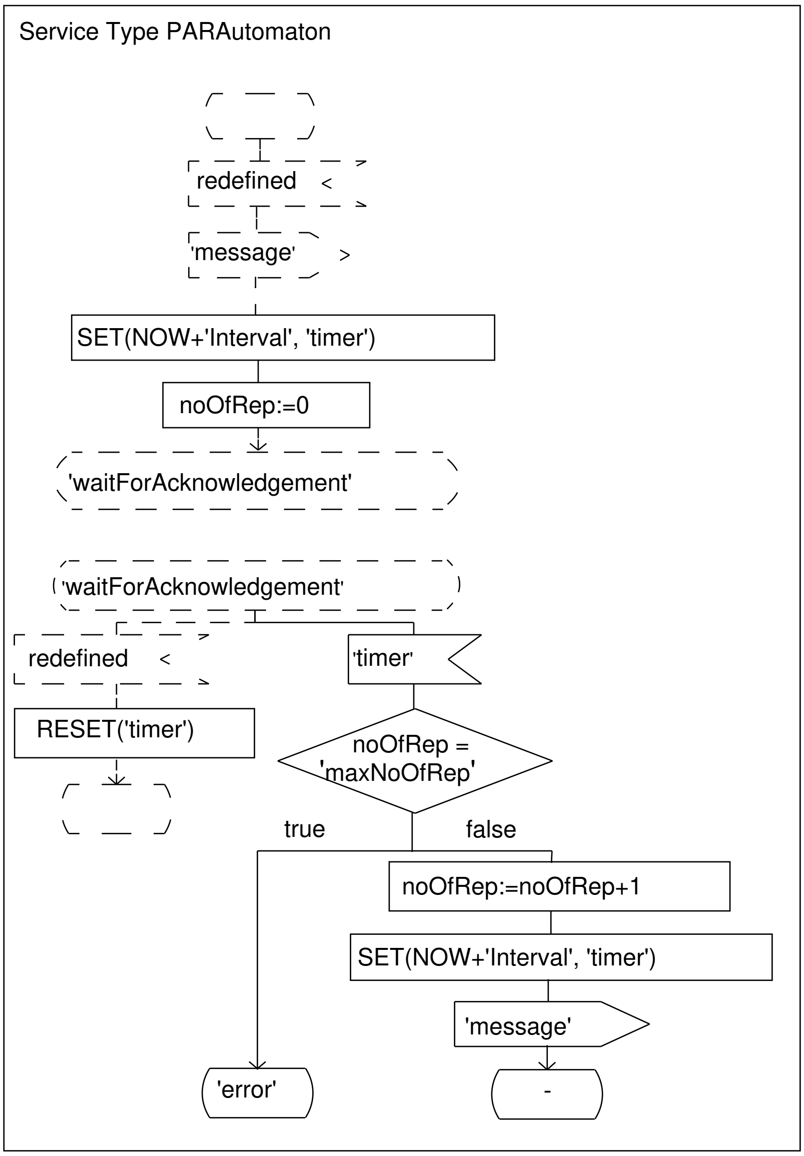


Fig.4: SDL-Fragment: TimerControlledRepeat pattern

- **DuplicateIgnore** and **DuplicateHandle**
  DuplicateIgnore and DuplicateHandle upgrade a message exchange between two automata in case of message duplication. Duplicate messages are detected by a message identifier, which is unique during the lifetime of the receiving automaton. DuplicateIgnore simply discards duplicates, i.e., the reaction to the original message is not repeated. Duplicate messages rely on a certain reaction of the receiving automaton when DuplicateHandle is applied.
- **DynamicEntitySet**
  Consider a given server entity that provides its service exactly one time and then terminates. In order to offer this service several times (e.g., to more than one client), the DynamicEntitySet pattern is introduced. Thereby an administrator dynamically creates a new server entity for each service request and acts as a proxy subsequently (Figure 5).

3.3 Incremental configuration process

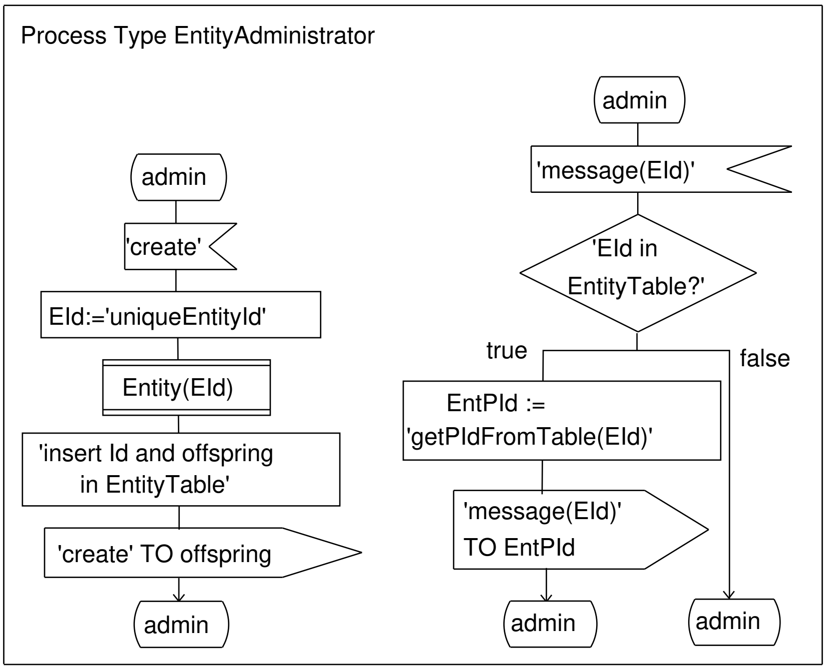


Fig.5: SDL-Fragment: DynamicEntitySet

For the design of SDL protocol specifications we have defined a configuration process supporting the reuse of protocol building blocks represented as SDL patterns (Figure 6). The configuration process suggests an incremental protocol development, where the whole set of communication requirements is first decomposed, i.e., partitioned and (where appropriate) simplified. Decomposition classifies as an analysis task that identifies separate protocol functionalities. Thereby it is possible to consider a protocol functionality under different assumptions. For instance, interaction sequences during connection establishment are less complex on top of a reliable basic service instead of an unreliable basic service. Experience has shown that protocol functionalities can often be specified one after the other and - in addition - be stepwise completed (e.g., adapted to the non-ideal properties of an underlying basic service). This suggests that we perform an individual development step in order to incorporate an additional protocol functionality or relax a corresponding simplification. Thereby each development step further divides into analysis and design of the current protocol functionality and yields an executable SDL design specification. In the following the analysis and design activities within a development step are sketched.

First an object-oriented analysis of the current protocol functionality is performed. This results in an updated analysis model from the previous development step. It is suggested to provide an OMT object model [11] and an MSC [16] use case model, which together identify participating objects and typical interaction scenarios. This integrates our approach into existing SDL methodologies [10] [13] [14], which also combine OMT and/or MSC with SDL for analysis and design, respectively.

The analysis model is realized in the following design activity. Here, SDL patterns come into place. Starting point is the context SDL specification, i.e., the SDL design specification obtained from the previous development step[2]. This may, e.g., be a protocol specification, which relies on a reliable basic service. Hence the design problem (stated in the analysis model) could then be to suit the protocol to an unreliable basic service. In order to meet the new requirements a number of SDL patterns are *selected* and applied to the context specification[3]. The selection of an SDL pattern is supported by several items of the SDL pattern description template, namely *intent*, *motivation*, *struc-*

2. For the first development step the initial context specification is either empty or given by an instantiated SDL framework.
3. Note that for some design problems the pool of predefined protocol building blocks may not contain an adequate solution. This gives rise to the development of a new SDL pattern or an ad hoc solution.

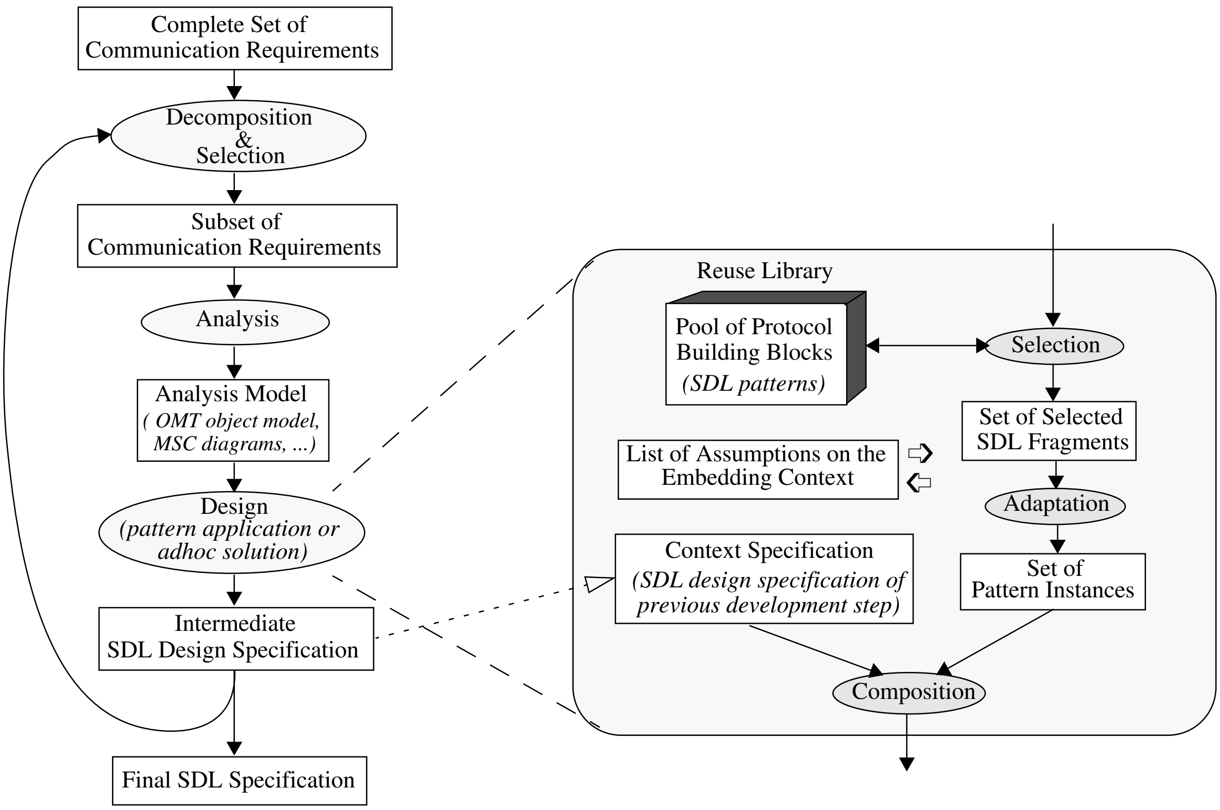


Fig.6: Incremental configuration process

ture, *message scenario*, *semantic properties* and *cooperative usage* (some of these items are not mentioned in Section 3.1). As patterns represent generic design solutions the corresponding SDL-fragment has to be *adapted* in order to seamlessly fit the embedding context. This is instructed by the *renaming* parts of the *syntactical embedding rules*. The resulting pattern instance finally has to be *composed* with the embedding context, which is prescribed by the *specialization* part of the syntactical embedding rules and also by *redefinition* rules of embedding pattern instances.

The result of this design activity is an intermediate SDL design specification, which is subsequently validated against the analysis use case model by performing MSC based validation and simulation (not shown in Figure 6). Also the correctness of the SDL specification concerning general properties such as freedom from deadlocks is checked. If any faults are discovered, a return to one of the previous development steps is needed (not shown in Figure 6). Otherwise the validated specification serves as the context specification for the next development step. If all simplifications are eliminated and all requirement subsets are implemented, the final design specification is given by the validated design specification of the last development step.

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

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

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

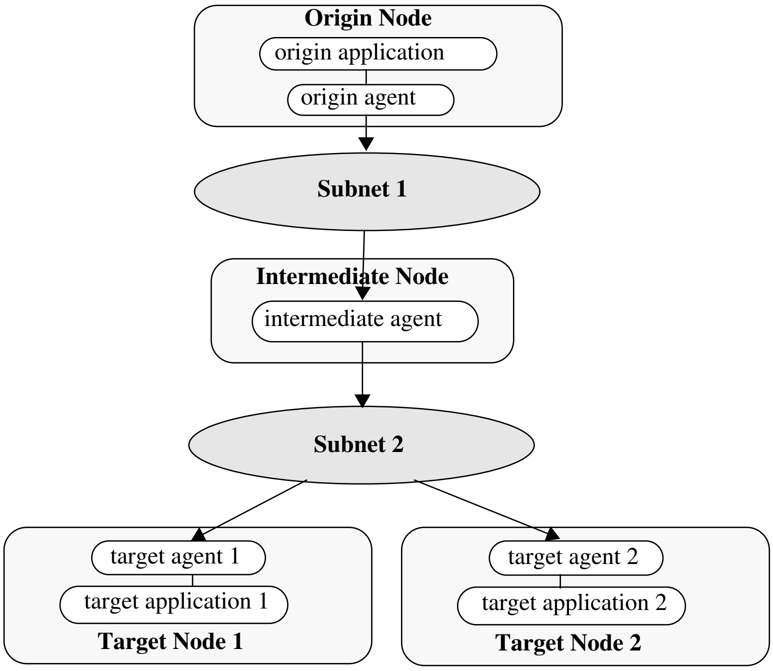


Fig.7: Example of an ST2+ stream

Care has to be taken not to destroy a pattern's intent in subsequent development steps. Therefore the assumptions are collected in a so-called *list of assumptions* and have to be checked in each development step. We will especially focus on this point, when presenting the case study in Section 4.

4 Re-Engineering of ST2+ Using SDL Patterns

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

4.1 Basic operation of ST2+

The Internet Stream Protocol ST2+ is a connection-oriented protocol operating at the same layer as the well-known IP protocol. ST2+ supports efficient multicast delivery of data with guaranteed quality of service (QoS). Real-time streams are established by a special resource reservation protocol, which is part of ST2+. Each stream consists of one origin agent and several intermediate and target agents. The protocol agents form a tree-shaped structure such as the topology illustrated in Figure 7. The stream direction from the origin to the target nodes is called downstream, while the reverse direction towards the origin node is called upstream. We call an ST agent previous-hop or next-hop, if it is one hop away in upstream or downstream direction, respectively.

The resource reservation protocol defines several operations for establishing, maintaining, and closing real-time streams. First, the origin application initiates the establishment of a stream by an OPEN request. Subsequently, protocol agents request communication paths to each target node and

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

Fig.8: Decomposition of requirements and chronological order of development steps

at the same time initiate resource reservations. The origin application is informed about target applications that accept or refuse the connection or could not be reached due to lack of resources.

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

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

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

4.2 Incremental configuring of ST2+

In order to obtain the final design specification we have performed eight development steps. Figure 8 summarizes the chronological order and the supported subset of communication requirements. Note that requirements, which result from a simplification of original requirements, are relaxed in later development steps. The set of requirements that is realized by the final protocol specification is marked with a thick border (Figure 8). The initial subset of requirements, for instance, only supports setup and tear down of a single unicast stream. Furthermore, it is assumed that origin, intermediate, and target agents are directly connected by SDL channels. We stepwise relax the assumptions w.r.t. the underlying communication service, complete the protocol operations, introduce multicast communication, and finally allow several streams to be active at the same time.

There is an important requirement, which every resource reservation protocol has to meet: during connection establishment the route of traversed nodes is fixed and *sufficient resources are reserved along the resulting communication paths*, so that required QoS is guaranteed during data transfer. Furthermore, after requesting a connection the calling application must eventually receive a reply stating, whether the connection establishment has succeeded or not. That is, *the calling application must not deadlock.*

In the following all development steps except the last one are discussed. Additionally, it is shown how SDL patterns can boost the developer's confidence in the correctness of the resulting protocol (w.r.t. the above mentioned requirements).

4.2.1 Design step I

The initial subset of communication requirements supports a single unicast connection over a series of nodes, that are directly connected through SDL channels. Thus we only consider virtual communication within the ST2+ layer of the protocol stack. Thereby the first step only realizes the OPEN and CLOSE operation.

The specification is configured as follows. First of all, we identify three types of ST2+ protocol entities, depending on their role in a stream: *OrigAgent*, *IntAgent* and *TargAgent*. Each of them resides on its own node. The involved objects and their associations are captured by the OMT object model of Figure 9. We assume that multiplicity is only determined, if a numerical value is given

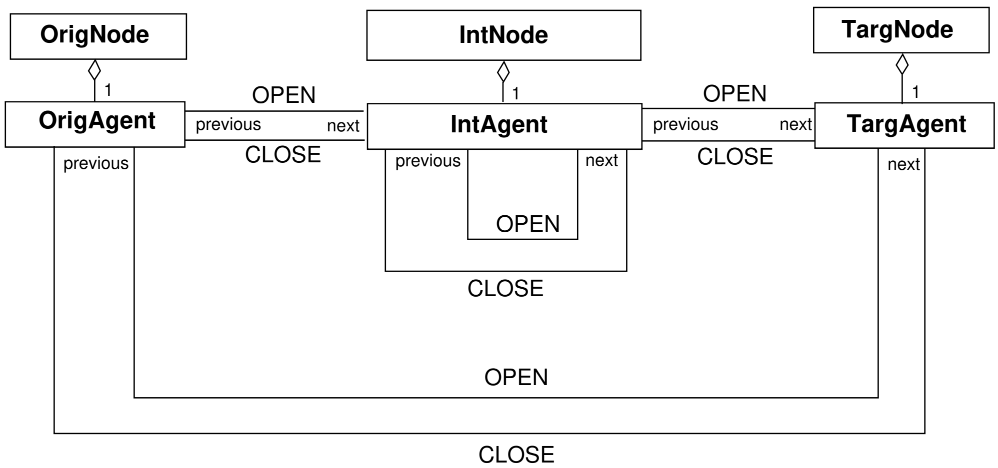


Fig.9: OMT object model: design step I/II

explicitly. Thus we informally state that for each instance of type *OrigAgent* and *IntAgent* the following holds: the instance relates to one and only one instance of type *IntAgent* or *TargAgent* through the association OPEN and the role *previous*. Similarly, each instance of type *TargAgent* and

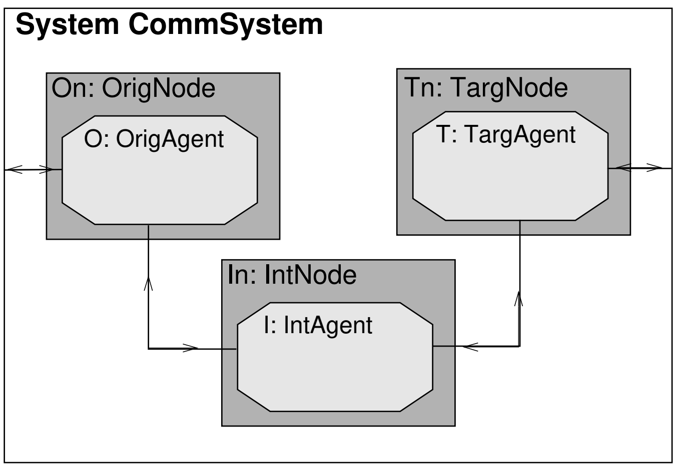


Fig.10: Protocol architecture: design step I

IntAgent relates to one and only one instance of type *IntAgent* or *OrigAgent* through the association

OPEN and the role *next*. Figure 10 shows an SDL overview diagram as a possible instantiation of the OMT object model.

The analysis model is completed by modelling the interaction behaviour between the participating objects in form of MSC diagrams. As we perform a re-engineering of ST2+ the interactions between protocol entities are prescribed. Figure 11 shows typical scenarios that result from an OPEN stream operation[4]. When an *OrigAgent* is triggered by OPEN.req it invokes a local resource manager

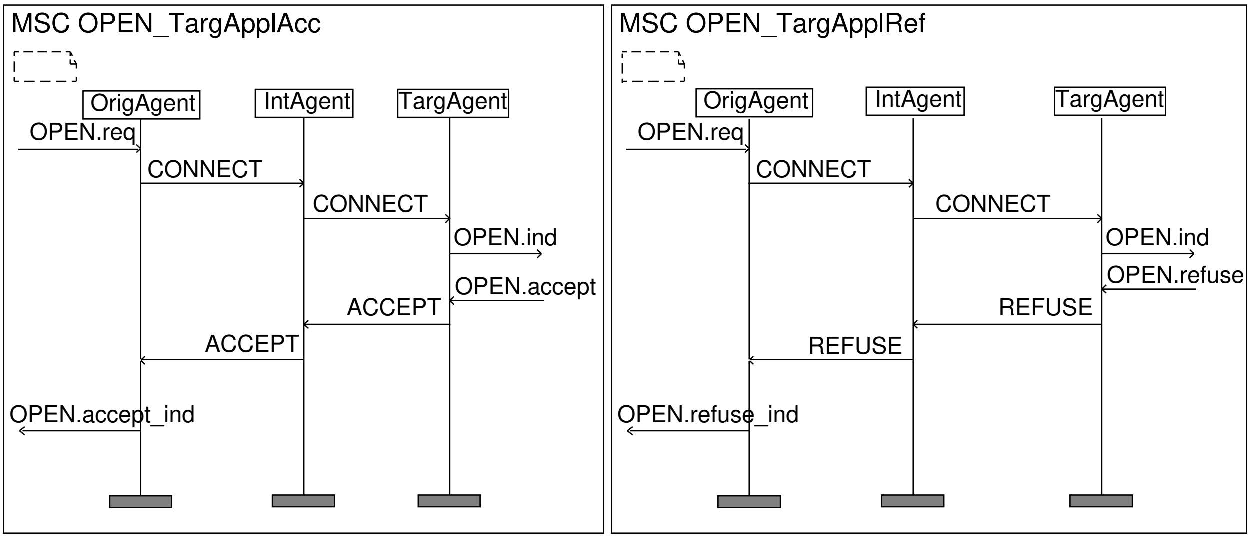


Fig.11: MSC diagram: design step I, OPEN operation

(*LRM*), which tries to reserve sufficient resources for transmitting the real-time data later on. If the reservation succeeds, a CONNECT PDU is sent to the next hop. An *IntAgent* proceeds the same way and forwards the CONNECT message after successful resource reservation towards the target node until a *TargAgent* is reached. An OPEN.ind primitive asks the target application for acceptance of the connection. The application answers with an OPEN.accept or OPEN.refuse. In the first case an ACCEPT PDU is generated and sent towards the origin, while resource reservations may be relaxed by the intermediate agents. Finally, the *OrigAgent* receives the ACCEPT message and informs the origin application about the successful establishment of the stream. If not enough resources could be reserved or the target application does not want to take part in the stream, a REFUSE message is passed back with all resources released.

As can be seen from the analysis model, the interactions between adjacent protocol agents and between an origin/target agent and its upper layer application follow the *BlockingRequestReply* pattern: after requesting a connection the sending automaton is blocked until receiving one of the expected replies (compare Figure 11).

We illustrate the application of the *BlockingRequestReply* pattern for the type *OrigAgent*. An *OrigAgent* plays the role of a replier when communicating with the origin application. As the SDL process into which the pattern instance has to be embedded is empty, the adaptation and composition is quite simple: signals and states can be arbitrarily named and the added transitions of the pattern instance are the only transitions of the resulting automaton. The pattern-specific part of the *OrigAgent* process is shaded in Appendix A, STEP I. The generic decisions of the *BlockingRequestReply* pattern cannot all be made locally. Thus communication with a next-hop *IntAgent* is necessary. Regarding this interaction an *OrigAgent* operates as a requester of the *BlockingRequestReply* pattern. Different replies of the pattern instance determine different outcomes for the decision of the first *BlockingRequestReply* instance. The second pattern instance within the *OrigAgent* process is shaded in Appendix A, STEP I.

4. Note that the processing of the OPEN stream operation is typical for ST2+. Other confirmed stream operations will be realized in the same manner.

The whole interaction scenario of the OPEN stream operation can be configured by the same kind of self-embedding of *BlockingRequestReply* pattern instances.

The point is, that the semantic properties of the *BlockingRequestReply* pattern guarantee certain properties ((P1.1) and (P1.2), below) of the resulting protocol design, if some assumptions ((A1.1) - (A1.4), below) on the embedding context hold[5]. Note that these assumptions and derived properties are part of the SDL pattern description (semantic properties, Section 3.1).

(A1.1): The origin application invokes OPEN.req only one time.
(A1.2): Request/reply signals (e.g., OPEN.req) are not implicitly consumed.
(A1.3): If a request signal arrives, the StartReply state of the corresponding replier will eventually be reached.
(A1.4): Communication between the different requester/replier pairs is reliable.

⇩

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

(A1.1) is an assumption on the system environment whereas the other assumptions have to be fulfilled by the SDL specification itself. For instance, assumption (A1.4) holds, because each requester is directly connected to the corresponding replier by an SDL channel. (A1.2) and (A1.3) can also be validated by inspection of the SDL design [12]. The resulting properties state that no agent participating in an OPEN operation, in particular the calling application, will deadlock.

From these properties of the individual pattern instances we can additionally prove by induction that all protocol agents on a chosen communication path have been reached during connection setup. If we further assume that the resource managers on each traversed node work properly (in the sense that a sufficient amount of local resources is calculated and also reserved[6]), we can be confident that sufficient resources are reserved for the whole stream. From now on we have to preserve these essential properties of the configured SDL system. This is done by ensuring the correctness of the corresponding assumptions throughout the following development steps.

The CLOSE operation is non-confirmed as can be seen in Figure 12. It is therefore realized by

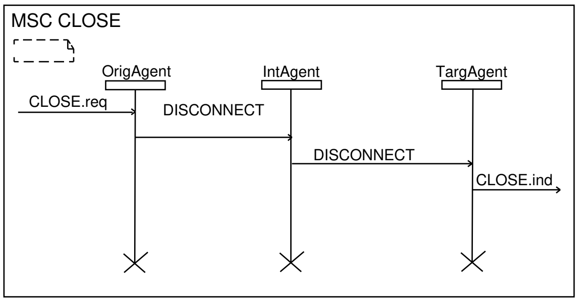


Fig.12: MSC diagram: design step I, CLOSE operation

applying the *SendReceive* pattern, where the sending agent immediately continues work after sending a DISCONNECT message. The whole interaction scenario is specified by linking the individual *SendReceive* pattern instances (Appendix A, Step I).

When configuring the protocol incrementally, we have to take into account possible interactions with newly introduced operations. Though ST2+ assumes that at the user interface a CLOSE.req is

5. (A1.2), e.g., is a short cut for the second assumption of development step 1 and (P1.1), e.g., is a short cut for the first derived property of development step 1.
6. This requirement lies outside the scope of ST2+.

only invoked after an OPEN.req, the CLOSE and OPEN operations may internally be processed in parallel. Thereby, closing a stream aborts a running OPEN operation, because the traversed ST agents are all terminated. Strictly speaking, *BlockingRequestReply* does not allow this kind of refinement, which actually gives rise to an extension of the pattern. For this case study, however, we handled the abortion of the OPEN scenario on an ad hoc basis. As a consequence, some of the assumptions above and also the two properties must be adapted for the exceptional case of operation interaction. This is of course not guided by the pattern description so far. For reasons of simplicity, we will only consider the assumption property lists for the normal case in this paper.

4.2.2 Design step II

In the second design step we have extended the protocol by allowing changes in the quality of service of the established stream. The CHG stream operation is configured the same way as the OPEN operation. However, an appropriate linkage between the two scenarios must be provided. As described in [3], a CHG operation can only start when the OPEN operation has completely finished. This defines separate protocol phases that are to be composed sequentially. Thus the involved pattern instances for the CHG operation are attached to the exit states (ESTBL) of the OPEN pattern instances (Appendix A, Step II). The assumptions and properties of design step I apply correspondingly to design step II with the exception that (A1.1) is changed to:

(A2.1): Concerning the CHG operation the target applications behave like a request/reply automaton as defined by the *BlockingRequestReply* pattern.Thereby a CHG.req is only invoked when the OPEN operation is finished.

4.2.3 Design step III

In the third step we replace the direct connection of protocol agents by a reliable basic service (Figure 13). Therefore protocol data units have to be mapped on basic service primitives and vice

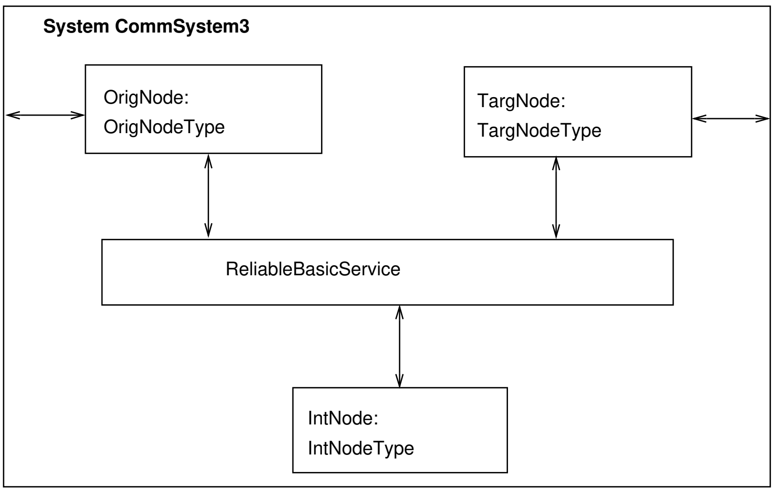


Fig.13: Protocol architecture: design step III

versa (Figure 14). To solve this problem, we have applied the *Codex* pattern to each process type resulting from development step II (e.g., OrigAgentType). As a consequence, a process type was split into two services, where one service handles the mapping of PDUs and service primitives and the other service contains the complete specification of the former process type (Appendix A, Step III).

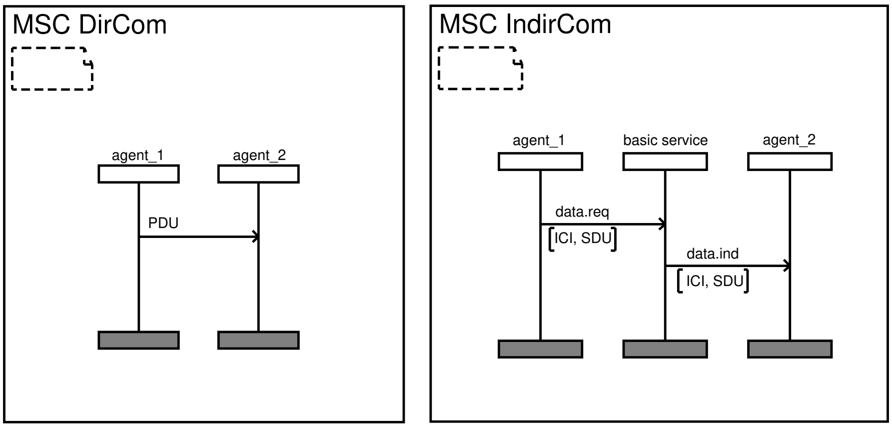


Fig.14: MSC diagram: design step III

According to the semantic properties of *Codex*, the pattern properly substitutes an SDL channel between communicating SDL processes, if the following assumptions hold:

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

Note that assumptions (A3.1)-(A3.3) replace (A1.4)/(A2.4), where (A3.1) and (A3.2) are assumptions on the underlying basic service and assumption (A3.3) can be validated by inspection of the SDL design specification: each PDU is supplemented by necessary interface control information before sending it to the codex component. Together with the remaining assumptions of the previous design steps, (P1) - (P4) still hold.

4.2.4 Design step IV

So far, we have only supported unicast communication between one origin and one target application. In the next steps we allow multicast communication with several target applications. We illustrate the necessary steps for configuring the OPEN operation. Most other stream operations are handled the same way.

For multicasting the service primitives and PDUs get an additional parameter *TargList*, which contains a list of target applications to be included in the stream. After receiving an OPEN.req(*TargList*) the *OrigAgent* invokes a routing function to determine the next-hops on the way to the targets. Then the LRM reserves resources and sends a CONNECT(*TargList*) message to each of them. According to the targets that are reachable through the next-hops, the initial *TargList* is divided into several *TargLists*. Every next-hop proceeds in the same way and forwards the CONNECT messages downstream. The targets do not distinguish between multicast and unicast and therefore the reactions are identical to the former design steps. In particular, the ACCEPT(Target) or REFUSE(Target) messages are transmitted the same way but in reverse direction as the CONNECT PDUs, so that reservations of the intermediate agents can be adjusted. If for some of the hops the resource reservations do not succeed, the agents generate an OPEN.refuse_ind(*TargList*) message for the affected targets and send it upstream. So finally, the sender application gets an OPEN.accept_ind or an OPEN.refuse_ind for each target (Figure 16).

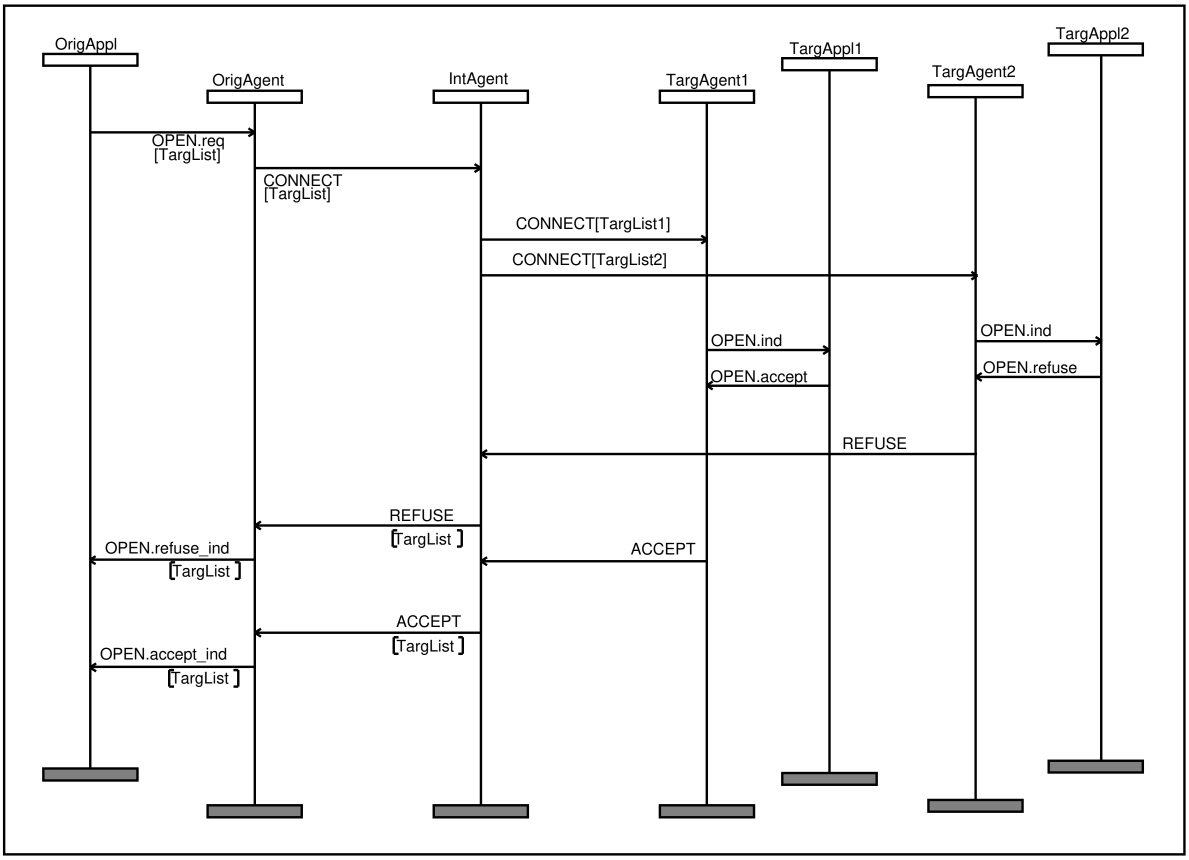


Fig.15: MSC diagram: design step IV

It turned out that multicast communication cannot be integrated by pure extensions of the SDL specification, such as in earlier design steps. Rather, the *BlockingRequestReply* instances that implement unicast communication have to be replaced. However, due to the pattern based approach it is straightforward to identify and eliminate those parts of the specification. To realize multicasting, we applied the *SingleRequestMultipleReplies* and the *MultipleRequestsMultipleReplies* pattern. As a result, the specification partially shown in Appendix A, step IV is obtained where the correspondence to step I and II is illustrated by shadings. Essentially, the *MultipleReplies* patterns add a loop for sending and receiving several signals (see the parts not shaded in Appendix A, step IV). The adaptation and refinement of the patterns, however, are adopted from the unicast case.

The CLOSE operation is also involved when reorganizing the protocol specification for multicasting. Similar to the OPEN stream operation the corresponding *SendReceive* instances must also be replaced by their multicast versions.

The updated list of assumptions (for their validation see [12]) resulting from the semantic properties of *SingleRequestMultipleReplies* and *MultipleRequestsMultipleReplies* is given below. Furthermore, the derived properties (P1.1) and (P1.2) have to be replaced by (P4.1) and (P4.2). The properties of the resulting protocol specification w.r.t. deadlocks during connection setup and reservation of sufficient resources follow from an analogous argumentation to step I.

(A2.1), (A3.1)-(A3.3)
(A4.1): The origin application invokes OPEN.req only one time.
(A4.2): Request/reply signals are not implictly consumed.
(A4.3): If a request signal arrives, the StartReceive state of the corresponding replier will eventually be reached
(A4.4): The reception loop of a requester must terminate iff the last reply was processed.

⇩

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

4.2.5 Design step V

Design step V newly introduces the ADD, JOIN, DROP, and LEAVE operation. The DROP operation allows the origin application to remove individual targets from the stream and operates very similar to the CLOSE stream operation. By means of the LEAVE operation each target can leave the stream itself. It is also an unconfirmed operation, that transmits a *REFUSE(Target)* message upstream in order to inform all intermediate and the origin agent that the target is no longer part of the stream and resources can be released. This operation is specified by concatenation of the *SendReceive* pattern just like the CLOSE operation (design step I). Through the ADD operation the origin application can add new targets to an existing stream. As the configuration hardly differs from the OPEN operation, we omit further details here.

The JOIN stream operation (Figure 16) is another way for a target application to influence the multicast group. First, the target application collects information about the stream it wants to join and communicates its request to the local ST agent by a JOIN.req service primitive. Then the agent sends a JOIN message towards the stream's origin. Eventually, this message reaches a node, which belongs to the referenced stream[7]. The ST agent of this node is now responsible for integrating the target. Its detailed reaction depends on the stream's *authorisation level*, which is assigned during stream establishment. With authorisation level 0, no target is allowed to join the stream. JOIN_REJECT is sent in this case. In case of authorisation levels 1 or 2 the target can be integrated into the stream. The corresponding procedure is almost identical to the OPEN and ADD operations and needs no further explanation. Authorisation levels 1 and 2 prescribe different strategies whether the stream's origin is notified of new targets after successful integration. With authorisation level 1 the origin is always informed about all the targets, while the intermediate agents do not necessarily inform the origin about new targets when authorisation level 2 is used.

It turns out that the target agent either receives a JOIN_REJECT or a CONNECT message in response to its JOIN message. Thus the JOIN operation can be realized by a generalization of the *BlockingRequestReply* pattern that allows the embedding of a connection setup procedure. The resulting specification is shown in Appendix A, step V. It is worth mentioning that the mere transportation of JOIN and JOIN_REJECT messages is not realized so far, as we only consider ST agents belonging to the stream. So, for this design step, we assume that JOIN and JOIN_REJECT messages are forwarded properly to their destinations.

Up to now, we mostly considered stream operations separately. That is, we have implicitly assumed that they do not overlap, which is not always true. Only some of the possible interactions

7. In the worst case the origin node itself is reached first.

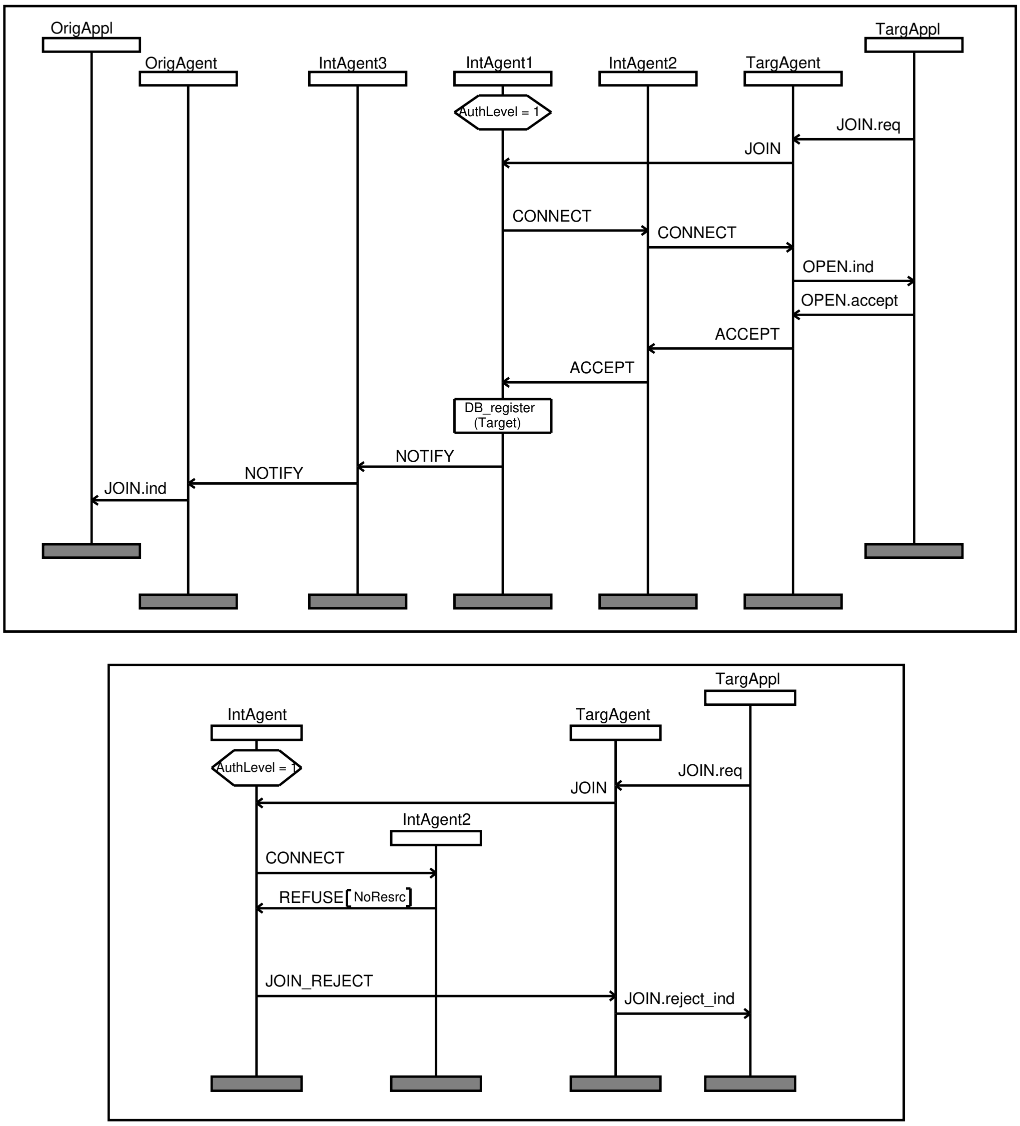


Fig.16: MSC diagram: design step V, JOIN operation

are resolved this way (e.g., the ADD operation can only start when the OPEN operation has completely finished). Others should be treated explicitly (e.g., a target that is deleted by a DROP operation during a CHANGE). The fact that some operations can be initiated on different nodes makes this task even more complicated. Each pair of interacting operations must be considered separately (such as the OPEN/CLOSE-interaction, design step I) and increases the complexity of the protocol. Note that most of them are not discussed in the textual protocol specification [3]. Nevertheless, we have tried to complete the specification by a proper interaction handling for all possible interactions.

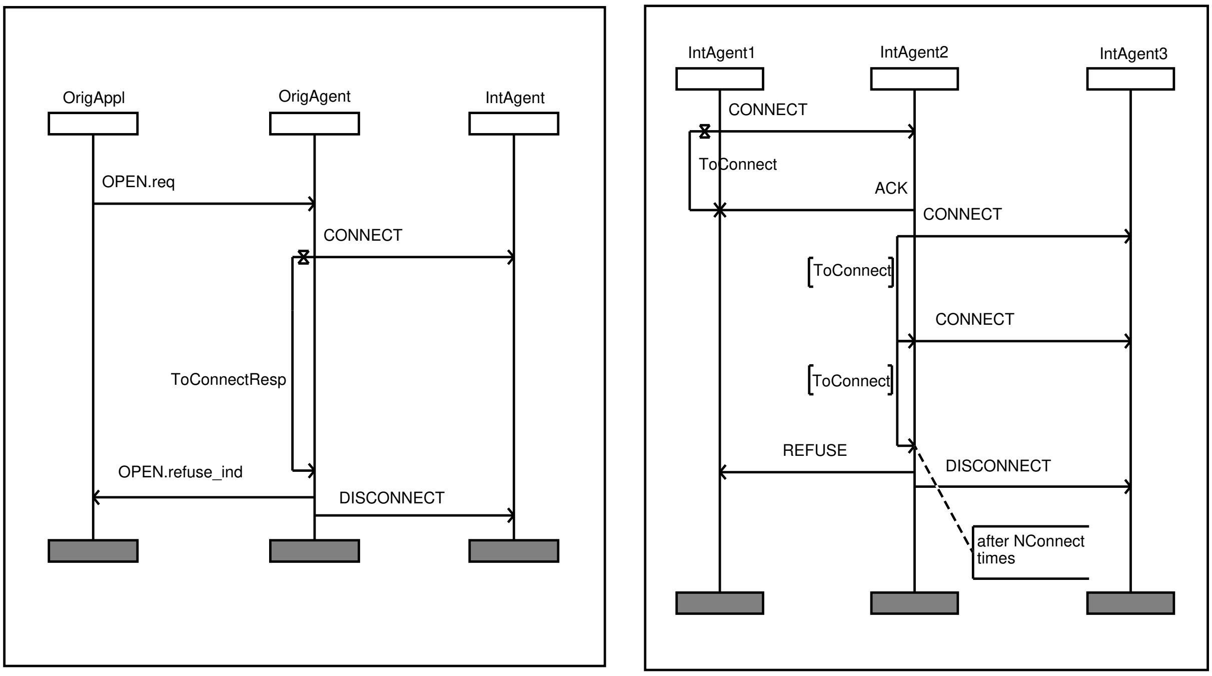
Roughly speaking, the stream operations can be divided into two groups: the confirmed operations OPEN, ADD, JOIN and CHANGE and the unconfirmed operations CLOSE, DROP and LEAVE. Interactions between operations of the same group can mostly be resolved by holding on one operation while the other operation is active on the considered node which results in a sequential composition such as the OPEN/CHANGE-composition of design step II. The interactions between confirmed and unconfirmed operations are more complicated and influence each other. When an OPEN, ADD, JOIN, or CHANGE is interrupted by a CLOSE, DROP or LEAVE, we take the targets referenced in CLOSE, DROP or LEAVE out of the confirmed operation. This can imply that a confirmed operation does not receive all expected replies. However, an agent will always be notified of the interaction and the target exclusion in one or another way. The list of assumptions is omitted, as it is very similar to the assertions of previous design steps.[8]

4.2.6 Design step VI

The assumption that the underlying basic service be reliable does not always hold with existing networks. We therefore extend the protocol specification to additionally cope with message loss and disruption by applying the *TimerControlledRepeat* pattern several times.

Let us consider the corresponding configuration for the OPEN stream operation. As the CONNECT and ACCEPT/REFUSE messages can get lost by an unreliable basic service, we first upgrade this confirmed message exchange by embedding a *TimerControlledRepeat* pattern instance. The pattern-related parts of the resulting specification are shaded in Appendix A, step VI. During pattern adaptation the generic parameter *'maxNoOfRepeats'* is set to 0, so that the corresponding decision *'noOfRepeats'* > *'maxNoOfRepeats'* can be completely left out. As a consequence, the CONNECT message is not retransmitted, but an error handling immediately initiated. In the course of the error handling, the origin agent first informs the application of the failure through an OPEN.refuse_ind(TargList). Then the agent tries to undo the operation for the already reached targets by sending a DISCONNECT(TargList) message downstream (Figure 17 a).

As a stream usually traverses several unreliable networks and contains many hops, it can take a lot of time to uncover a transfer failure. In order to make the protocol faster and less sensible w.r.t. message losses the message passing between neighbouring ST agents is also controlled. For this purpose each correctly received control message is additionally acknowledged by a special ACK message. The resulting confirmed message exchanges are introduced by applying *MultipleRequests-MultipleReplies* and *BlockingRequestReply* for multicast (e.g. ADD) and unicast operations (e.g. LEAVE), respectively. Then the retransmission strategy is again provided through the *TimerControlledRepeat* pattern. After a certain number of retransmissions, an error handling is started depend-



a.) *b.)*

Fig.17: MSC diagram: design step VI, OPEN operation

8. Note that we do not apply new patterns.

ing on the kind of message being lost. For instance, a lost CONNECT message is followed by a REFUSE and DISCONNECT message in upstream and downstream direction, respectively, in order to undo the active operation w.r.t. the involved targets (Figure 17 b). SDL symbols related to the *TimerControlledRepeat* pattern are shaded in Appendix A, Step VI.

The underlying basic service is also expected to disrupt messages. This makes a checksum mechanism indispensable. Instead of simply discarding disrupted messages, the receiving ST agent returns an ERROR message that causes the sending agent to repeat the message immediately without awaiting the corresponding timeout. Actually, a negative acknowledgement is handled in the same way as an expiration of the associated timer (Appendix A, Step VI).

Note that we must also deal with duplicate messages when applying retransmission strategies. This is realized by applying the *DuplicateHandle* pattern, shaded in Appendix A, Step VI.

According to the semantic properties of the applied patterns, assumption (A3.2) has to be replaced by the new assumptions listed below.

(A3.1)+(A3.3) + (A4.1)-(A4.4)
(A6.1): The applied timer intervals for error control are greater than the maximal round trip time of the controlled message (e.g., CONNECT PDU) and the corresponding replies (e.g., ACCEPT PDU).
(A6.2): Incoming messages can be uniquely identified and are logged.

⇩

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

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

4.2.7 Design step VII

Finally, we complete the protocol specification by allowing several streams to be active at the same time. The *DynamicEntitySet* pattern is capable to manage several streams, so that in each traversed network node a stream is handled by its own protocol machine.

We are going to distinguish between two kinds of protocol agents: router agents and host agents. Router agents take on the functionality of an *IntAgent* and are instantiated for each single stream that traverses the node. The same holds for host agents, which include the functionality of an *OrigAgent* and *TargAgent*.

Both host and router agents have a similar structure, which is prescribed by the *DynamicEntitySet* pattern. The *StreamAdm* component manages all traversing streams by creating origin, target, or intermediate agents when needed and forwarding stream-related messages to the corresponding processes (Appendix A, Step VII). As the *StreamAdm* process must be able to assign incoming messages to a particular stream, each PDU and service primitive (except OPEN.req) contains an SID (Stream IDentifier) parameter, which identifies the stream globally. In order to be unique the SID is constructed by concatenation of the origin's 32-bit IP address and a locally unique stream identifier at the origin node. As a result, it follows from the pattern's semantic properties ((A7.1) and (P7.1)) that previous considerations are still valid for each stream.

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

⇩

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

It is worth mentioning, that incoming messages with unknown SIDs (excluding OPEN.req and CONNECT) are handled differently by host and router agents. Whereas host agents simply discard unrecognized messages, routers handle certain of them as follows. The JOIN operation described in design step V defines JOIN and JOIN_REJECT messages, which were assumed to be somehow transmitted between an intermediate agent of the stream and the new target agent. Actually, these messages are forwarded by routers, which are not part of the stream. Therefore, a router's *StreamAdm* component forwards unrecognized messages to a special *NoStream* process, which transmits[9] JOIN and JOIN_REJECT messages to their destinations. The details are specified by refining the *DynamicEntitySet* pattern instance (Appendix A, Step VII) and do not affect property (P7.1).

5 Conclusion

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

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

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

9. For transportation of these messages over an unreliable basic service the same mechanisms are applied as introduced in design step VI.

References

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

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

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

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

[5] B. Geppert, R. Gotzhein, and F. Rößler, *Configuring Communication Protocols Using SDL Patterns*, SDL'97 Time for Testing - SDL, MSC and Trends, Proceedings of the 8th SDL Forum, France, 1997

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

[7] B. Geppert and F. Rößler, *Combining SDL and Pattern-based Design for the Customization of Communication Subsystems*, in: A. Wolisz, I. Schieferdecker, A. Rennoch (Ed.): Formale Beschreibungstechniken für verteilte Systeme, GI/ITG-Fachgespräch, ISBN 3-88457-315-2, 1997

[8] B. Geppert and F. Rößler, *Generic Engineering of Communication Protocols - Current Experience and Future Issues*, 1st IEEE International Conference on Formal Engineering Methods, ICFEM'97, Hiroshima, Japan, 1997

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

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

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

[12] P. Schaible, *Pattern-based Configuring of the Reservation Protocol ST2+ and Development of an SDL-Framework for Resource Reservation Protocol*s (in german), diploma thesis, University of Kaiserslautern, 1997

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

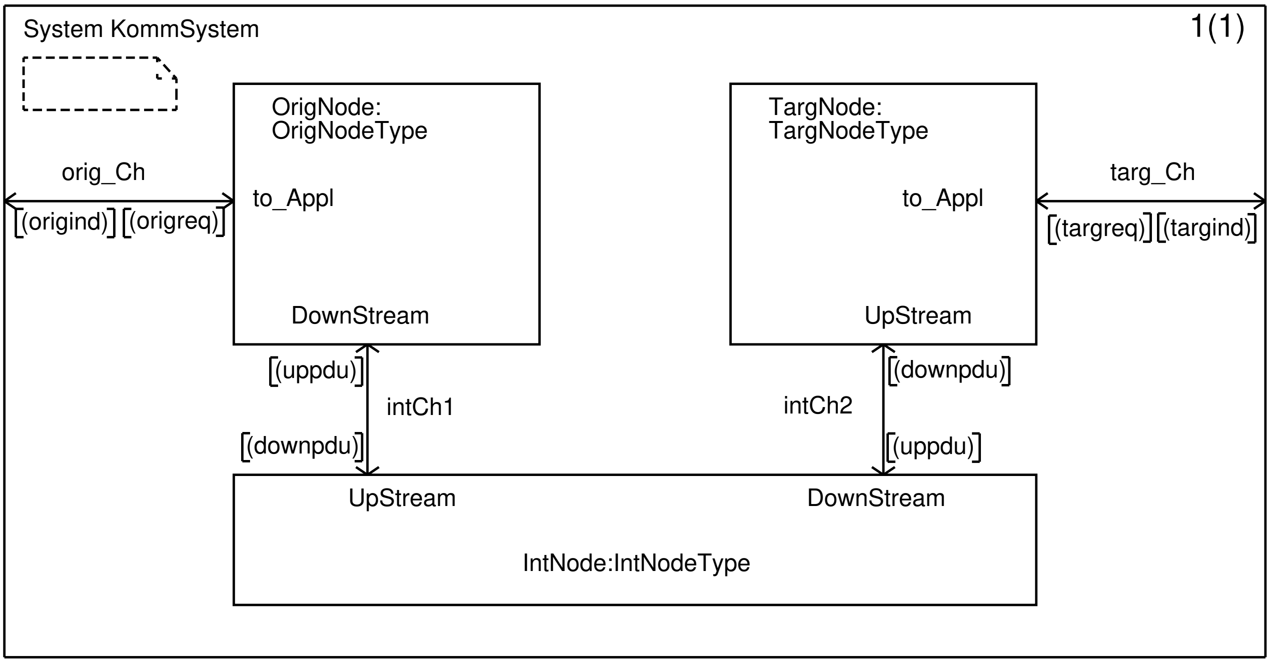
[14] D. Witaszek, E. Holz, M. Wasowski, S. Lau, and J. Fischer, *A Development Method for SDL-92 Specifications Based on OMT*, SDL'95 with MSC in CASE, Proceedings of the 7th SDL Forum, 1995

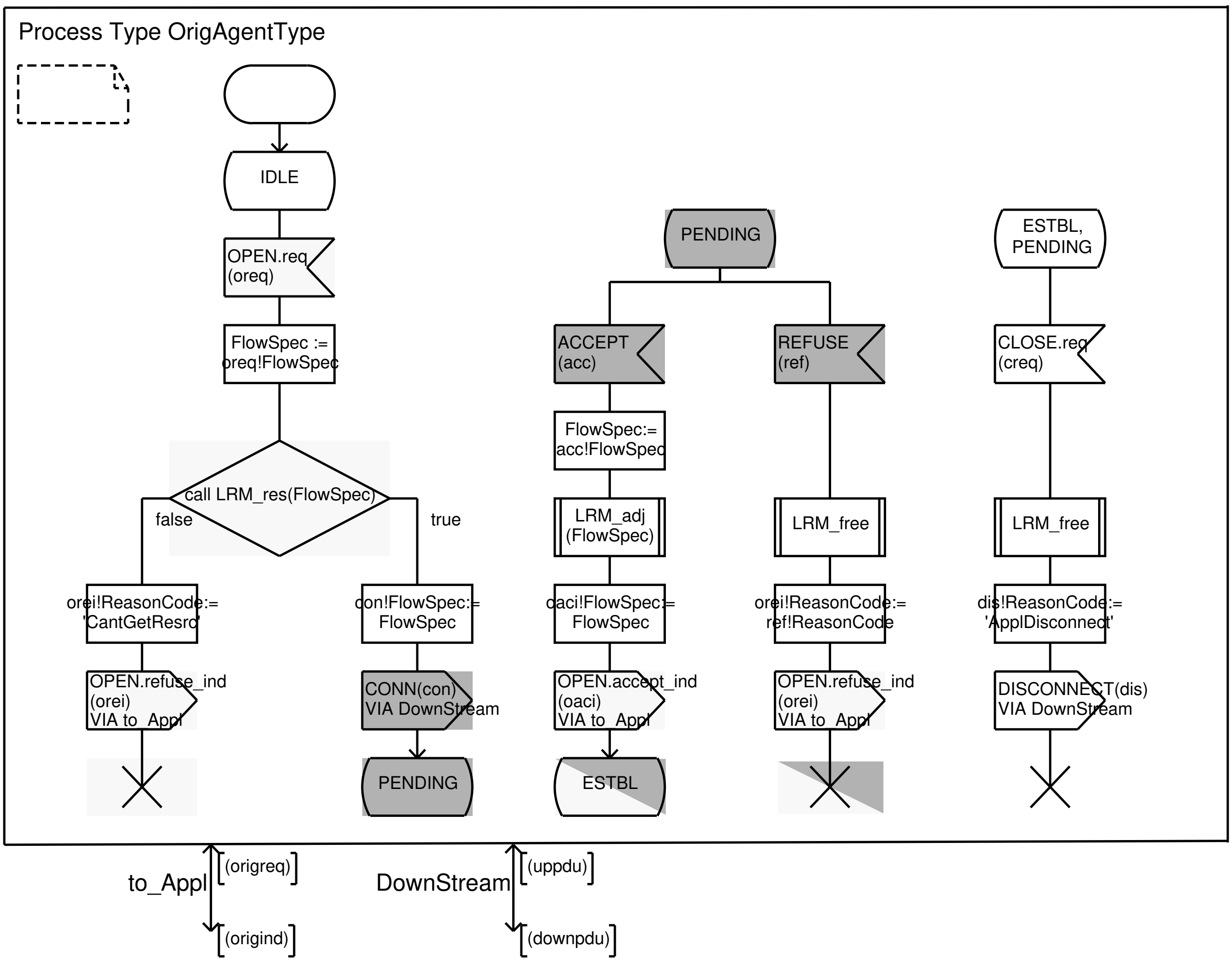
[15] Z.100 (03/93) *CCITT Specification and Description Language (SDL)*, ITU-T, 1994

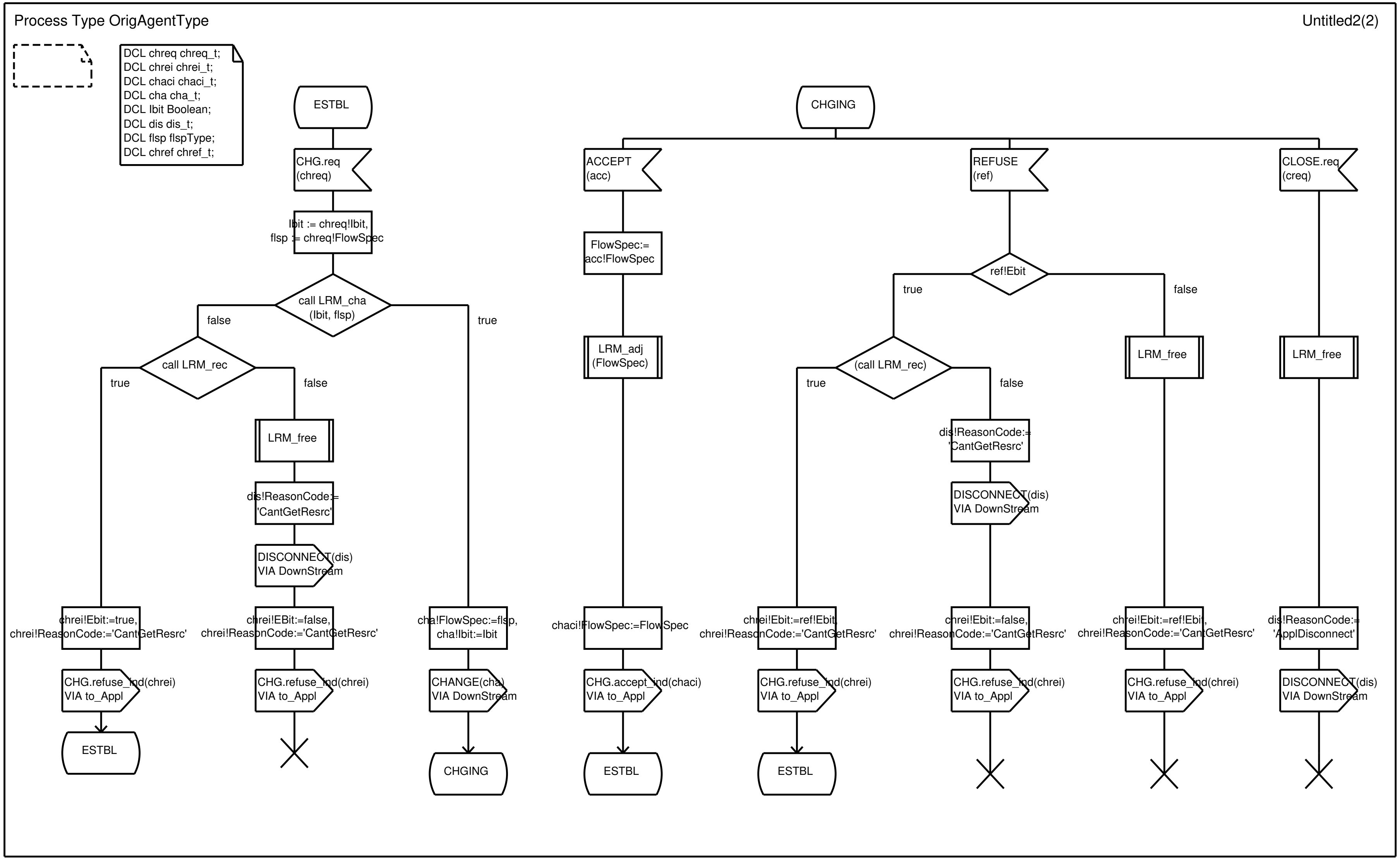
[16] Z.120 (10/96) *Message Sequence Chart (MSC)*, ITU-T, 1996

Appendix A: SDL Specification

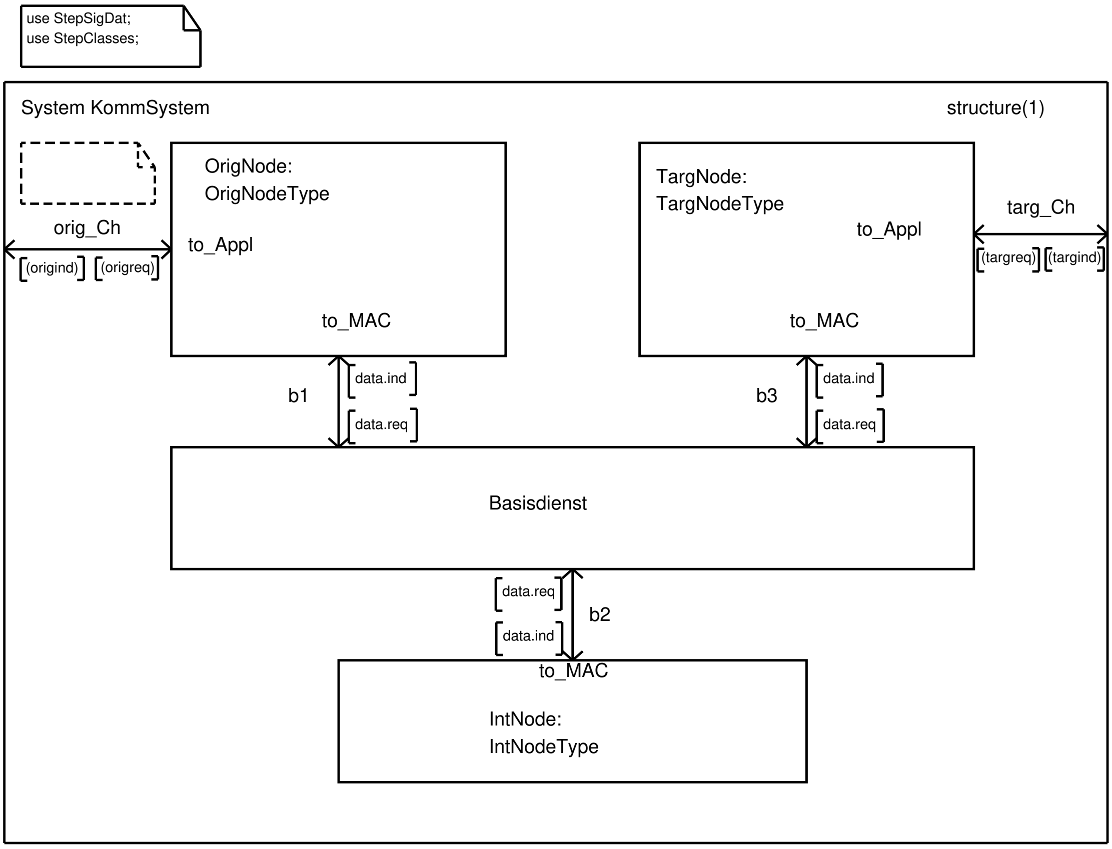
STEP I

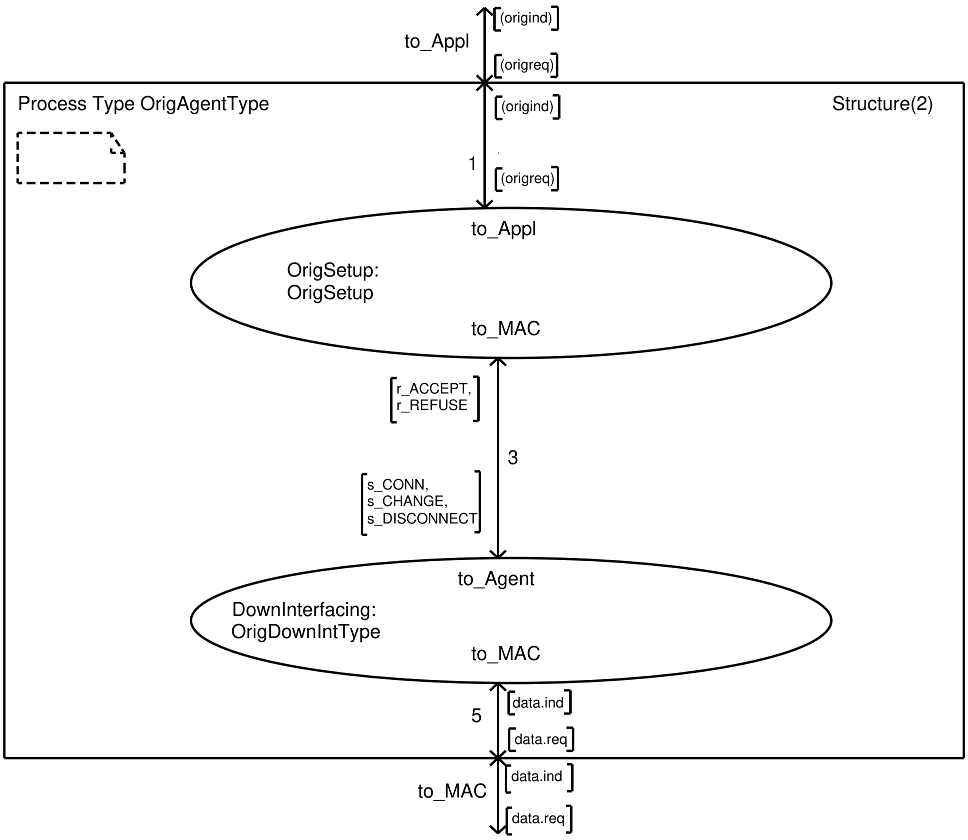


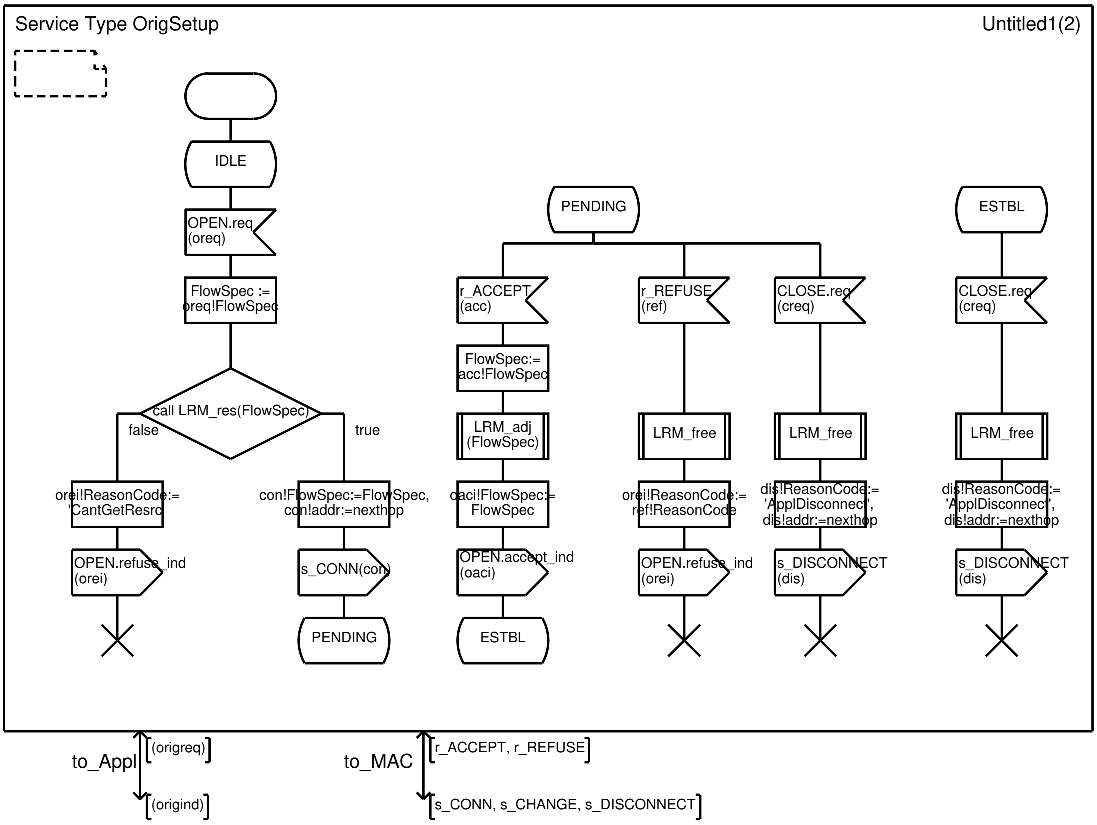


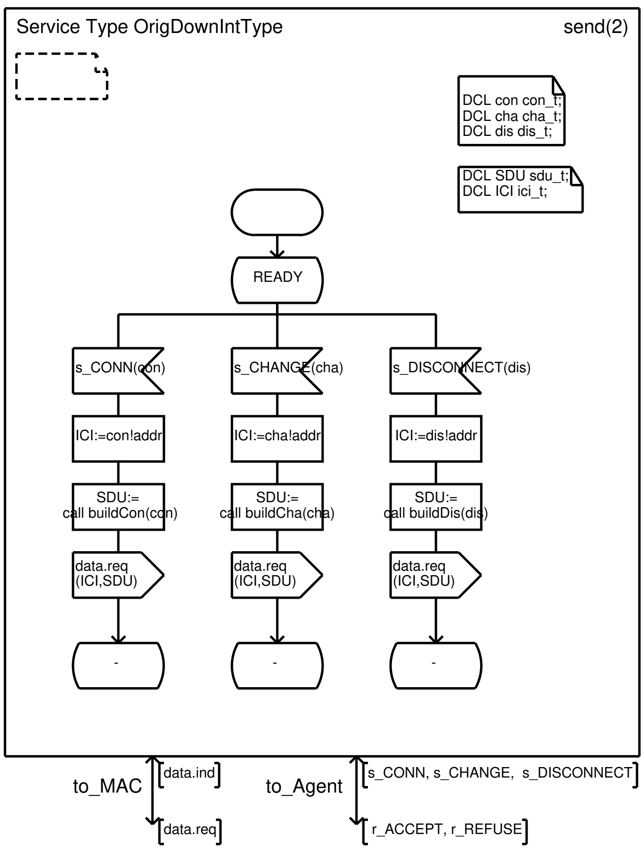

Process Type OrigAgentType
Untitled2(2)
DCL chreq chreq_t;
DCL chrei chrei_t;
DCL chaci chaci_t;
DCL cha cha_t;
DCL lbit Boolean;
DCL dis dis_t;
DCL flsp flspType;
DCL chref chref_t;
ESTBL
CHG.req (chreq)
lbit := chreq!lbit, flsp := chreq!FlowSpec
call LRM_cha (lbit, flsp)
false
true
call LRM_rec
true
false
LRM_free
dis!ReasonCode:= 'CantGetResrc'
DISCONNECT(dis) VIA DownStream
chrei!Ebit:=true, chrei!ReasonCode:='CantGetResrc'
chrei!EBit:=false, chrei!ReasonCode:='CantGetResrc'
cha!FlowSpec:=flsp, cha!lbit:=lbit
CHG.refuse_ind(chrei) VIA to_Appl
CHANGE(cha) VIA DownStream
ESTBL
CHGING
CHGING
ACCEPT (acc)
REFUSE (ref)
CLOSE.req (creq)
FlowSpec:= acc!FlowSpec
LRM_adj (FlowSpec)
chaci!FlowSpec:=FlowSpec
CHG.accept_ind(chaci) VIA to_Appl
ESTBL
ref!Ebit
true
false
(call LRM_rec)
true
false
dis!ReasonCode:= 'CantGetResrc'
DISCONNECT(dis) VIA DownStream
chrei!Ebit:=ref!Ebit, chrei!ReasonCode:='CantGetResrc'
chrei!Ebit:=false, chrei!ReasonCode:='CantGetResrc'
CHG.refuse_ind(chrei) VIA to_Appl
ESTBL
LRM_free
chrei!Ebit:=ref!Ebit, chrei!ReasonCode:='CantGetResrc'
CHG.refuse_ind(chrei) VIA to_Appl
LRM_free
dis!ReasonCode:= 'ApplDisconnect'
DISCONNECT(dis) VIA DownStream

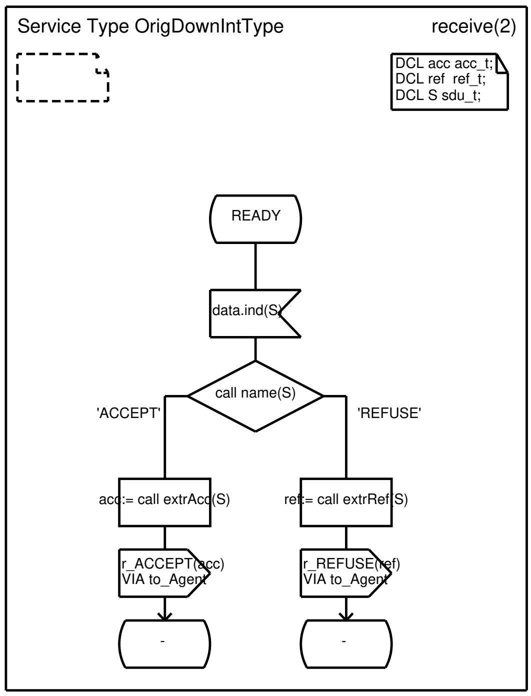
STEP III

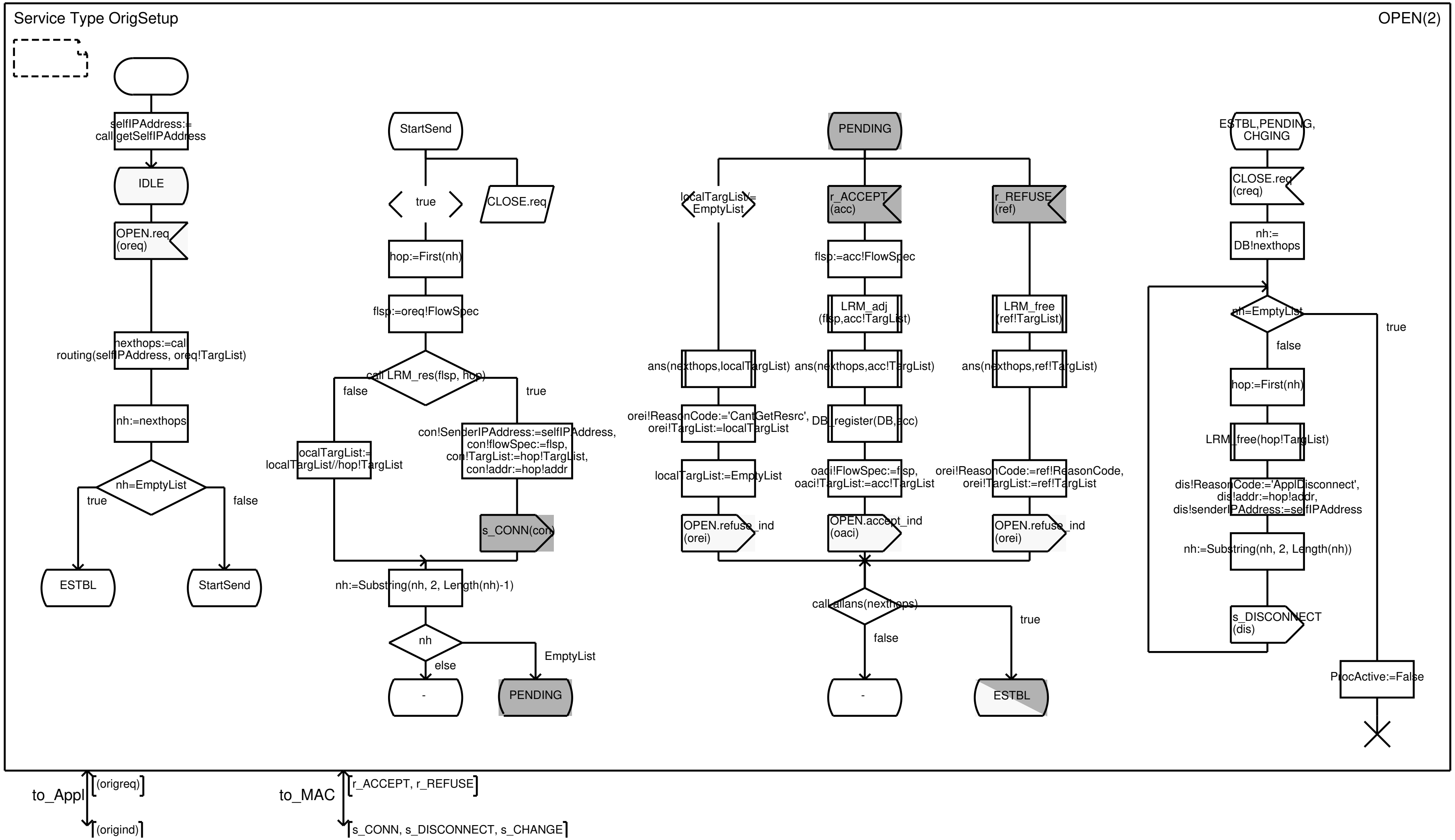

use StepSigDat;
use StepClasses;
System KommSystem
structure(1)
orig_Ch
(origind)
(origreq)
OrigNode:
OrigNodeType
to_Appl
to_MAC
TargNode:
TargNodeType
to_Appl
to_MAC
targ_Ch
(targreq)
(targind)
b1
data.ind
data.req
b3
data.ind
data.req
Basisdienst
data.req
data.ind
b2
to_MAC
IntNode:
IntNodeType


to_Appl
(origind)
(origreq)
Process Type OrigAgentType
Structure(2)
(origind)
1
(origreq)
to_Appl
OrigSetup:
OrigSetup
to_MAC
r_ACCEPT,
r_REFUSE
3
s_CONN,
s_CHANGE,
s_DISCONNECT
to_Agent
DownInterfacing:
OrigDownIntType
to_MAC
data.ind
5
data.req
data.ind
to_MAC
data.req

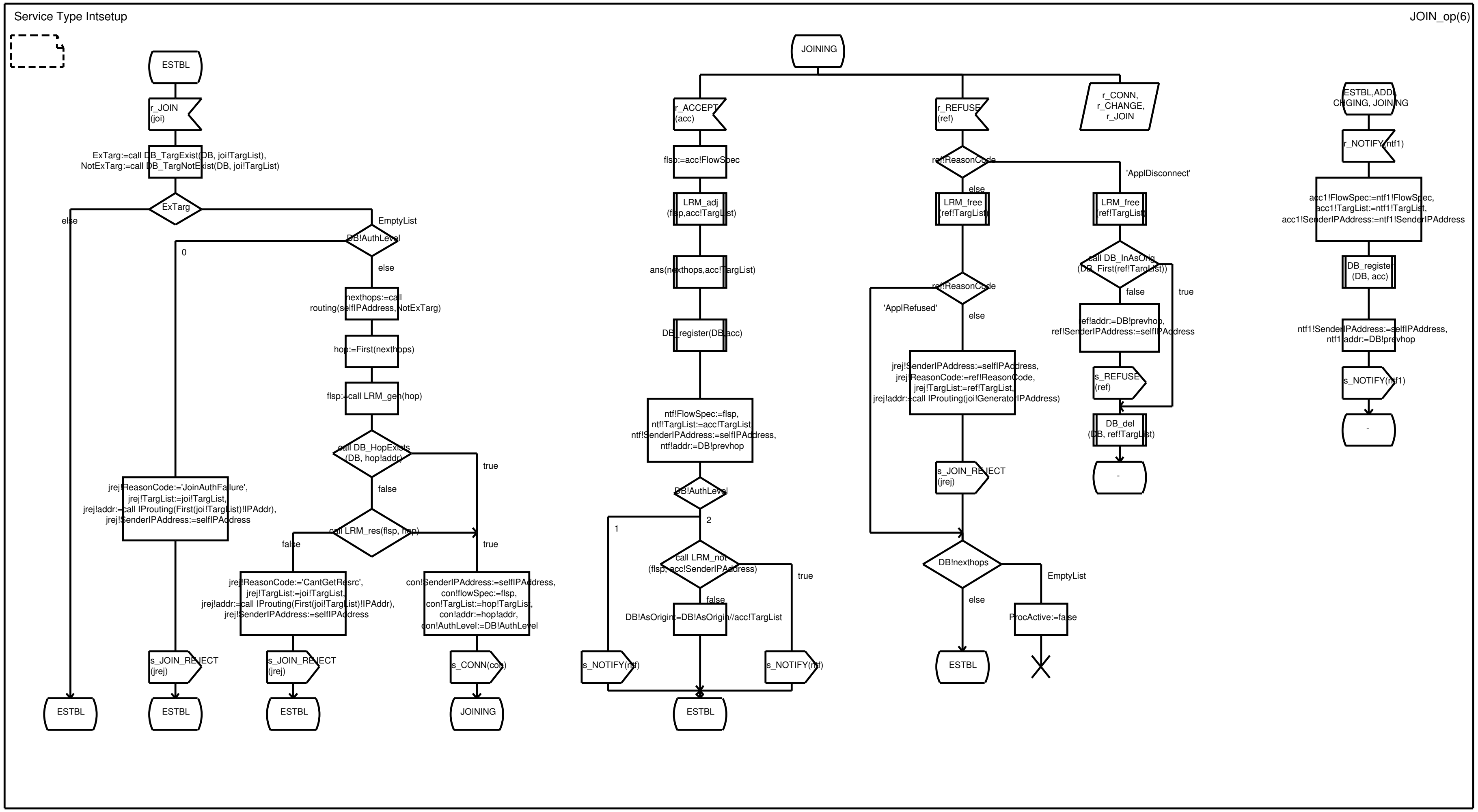

Service Type OrigSetup
Untitled1(2)
IDLE
OPEN.req
(oreq)
FlowSpec :=
oreq!FlowSpec
call LRM_res(FlowSpec)
false
true
orei!ReasonCode:=
CantGetResrc
con!FlowSpec:=FlowSpec,
con!addr:=nexthop
OPEN.refuse_ind
(orei)
s_CONN(con)
PENDING
PENDING
r_ACCEPT
(acc)
FlowSpec:=
acc!FlowSpec
LRM_adj
(FlowSpec)
oaci!FlowSpec:=
FlowSpec
OPEN.accept_ind
(oaci)
ESTBL
r_REFUSE
(ref)
LRM_free
orei!ReasonCode:=
ref!ReasonCode
OPEN.refuse_ind
(orei)
CLOSE.req
(creq)
LRM_free
dis!ReasonCode:=
'ApplDisconnect',
dis!addr:=nexthop
s_DISCONNECT
(dis)
ESTBL
CLOSE.req
(creq)
LRM_free
dis!ReasonCode:=
'ApplDisconnect',
dis!addr:=nexthop
s_DISCONNECT
(dis)
to_Appl
(origreq)
(origind)
to_MAC
r_ACCEPT, r_REFUSE
s_CONN, s_CHANGE, s_DISCONNECT

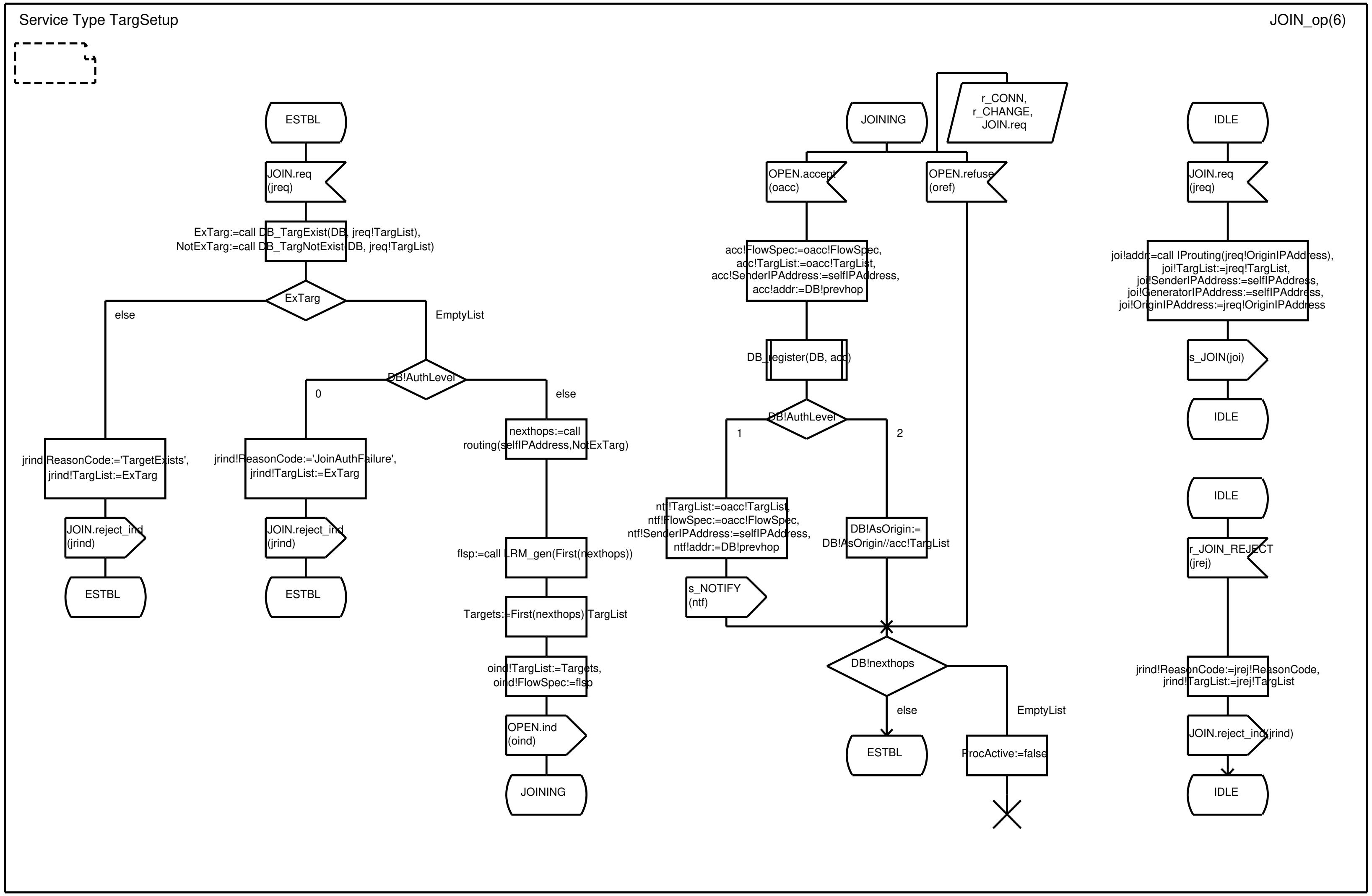

Service Type OrigDownIntType
send(2)
DCL con con_t;
DCL cha cha_t;
DCL dis dis_t;
DCL SDU sdu_t;
DCL ICI ici_t;
READY
s_CONN(con)
s_CHANGE(cha)
s_DISCONNECT(dis)
ICI:=con!addr
ICI:=cha!addr
ICI:=dis!addr
SDU:=
call buildCon(con)
SDU:=
call buildCha(cha)
SDU:=
call buildDis(dis)
data.req
(ICI,SDU)
data.req
(ICI,SDU)
data.req
(ICI,SDU)
-
-
-
to_MAC
data.ind
data.req
to_Agent
s_CONN, s_CHANGE, s_DISCONNECT
r_ACCEPT, r_REFUSE

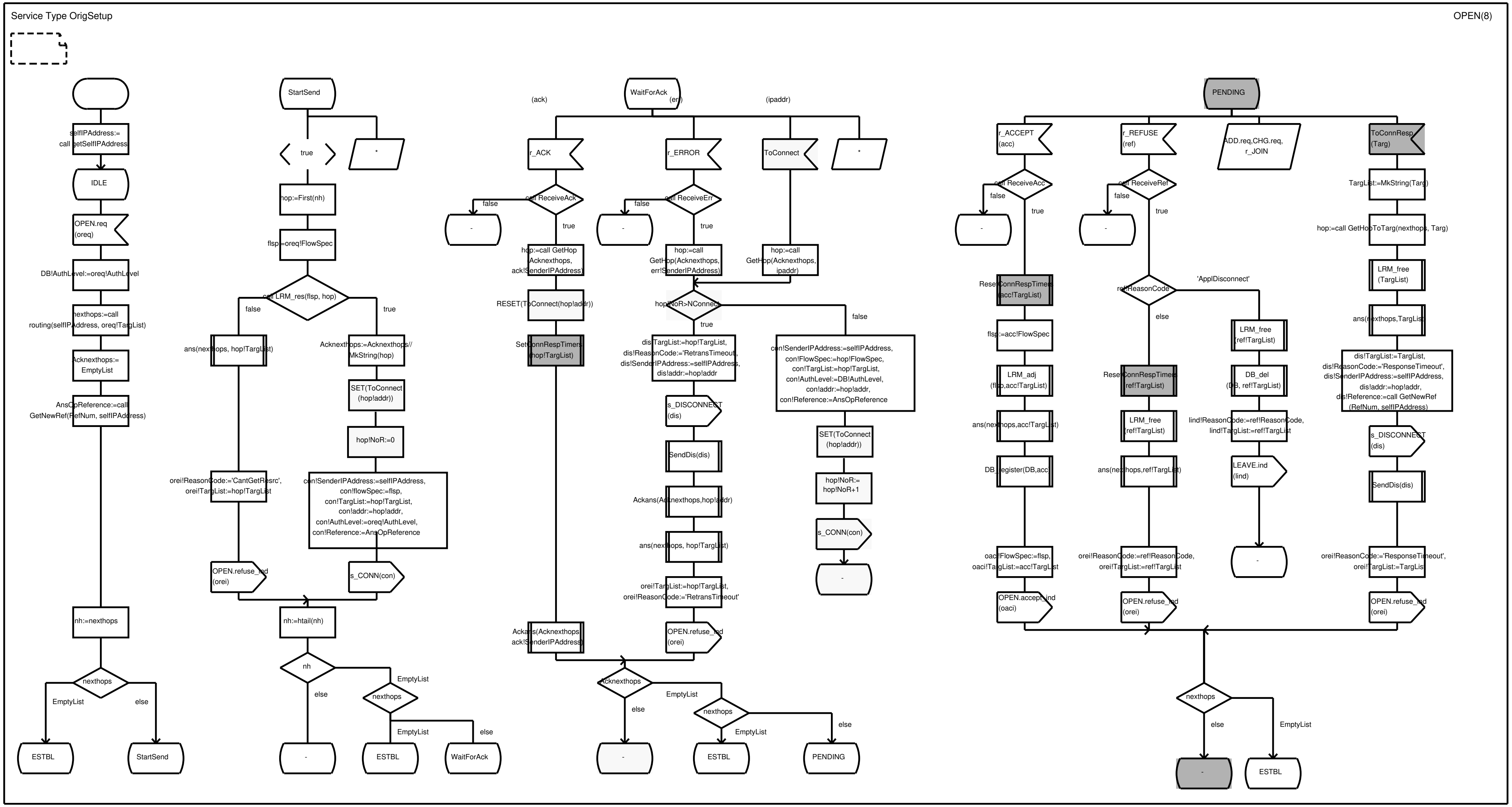

Service Type OrigDownIntType
receive(2)
DCL acc acc_t;
DCL ref ref_t;
DCL S sdu_t;
READY
data.ind(S)
call name(S)
'ACCEPT'
'REFUSE'
acc:= call extrAcc(S)
ref:= call extrRef(S)
r_ACCEPT(acc)
VIA to_Agent
r_REFUSE(ref)
VIA to_Agent
-
-

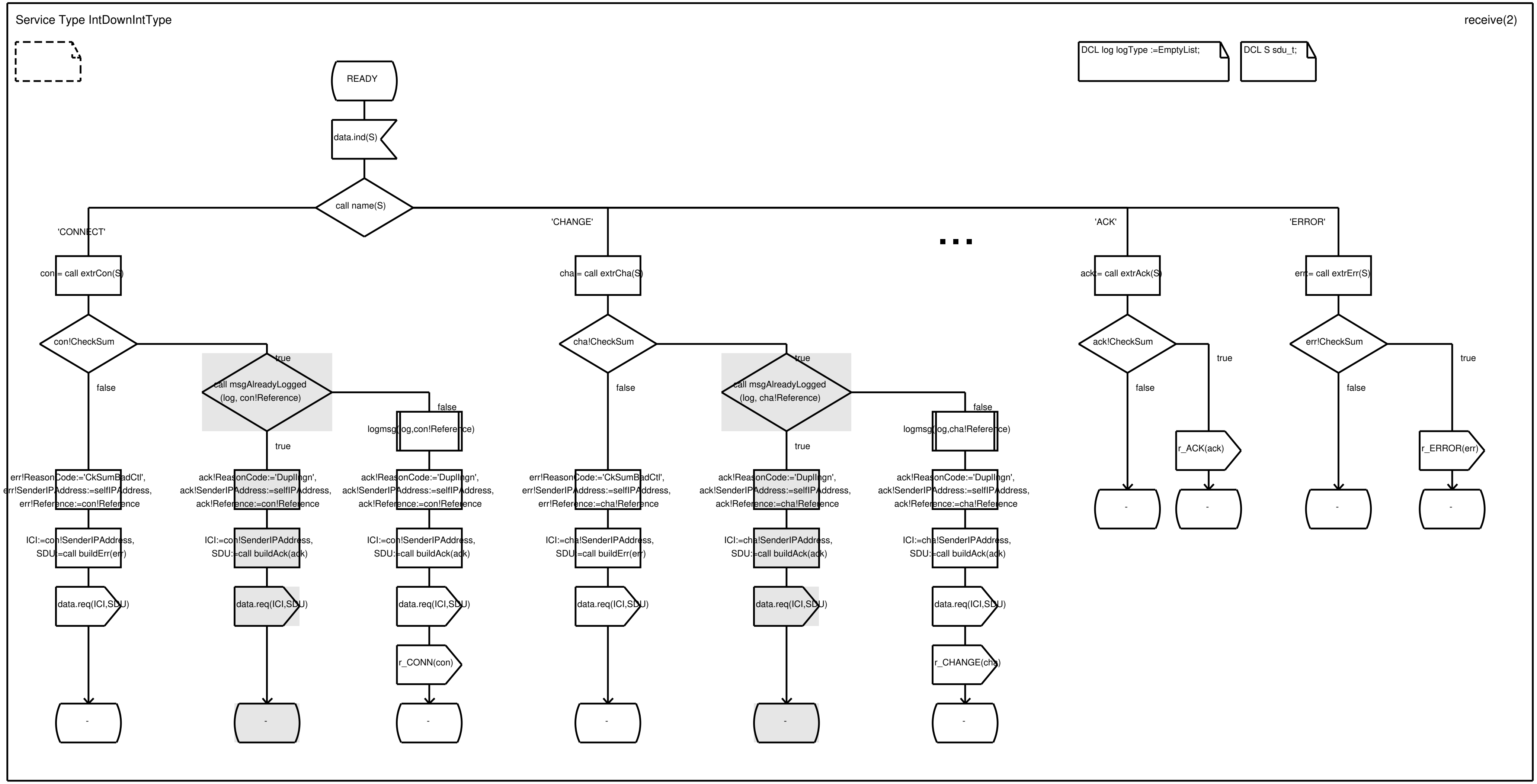


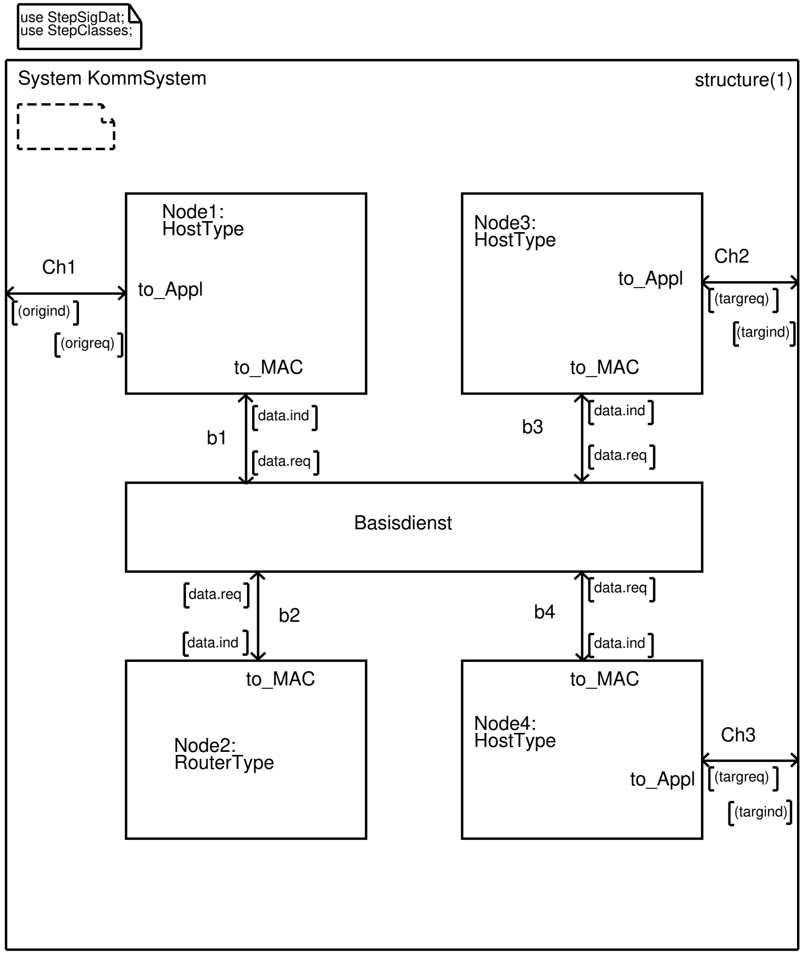
Service Type OrigSetup
OPEN(2)
selfIPAddress:=
call getSelfIPAddress
IDLE
OPEN.req
(oreq)
nexthops:=call
routing(selfIPAddress, oreq!TargList)
nh:=nexthops
nh=EmptyList
true
false
ESTBL
StartSend
StartSend
true
CLOSE.req
hop:=First(nh)
flsp:=oreq!FlowSpec
call LRM_res(flsp, hop)
false
true
localTargList:=
localTargList//hop!TargList
con!SenderIPAddress:=selfIPAddress,
con!flowSpec:=flsp,
con!TargList:=hop!TargList,
con!addr:=hop!addr
s_CONN(con)
nh:=Substring(nh, 2, Length(nh)-1)
nh
else
EmptyList
-
PENDING
PENDING
localTargList/=
EmptyList
r_ACCEPT
(acc)
r_REFUSE
(ref)
flsp:=acc!FlowSpec
LRM_adj
(flsp,acc!TargList)
LRM_free
(ref!TargList)
ans(nexthops,localTargList)
ans(nexthops,acc!TargList)
ans(nexthops,ref!TargList)
orei!ReasonCode:='CantGetResrc',
orei!TargList:=localTargList
DB_register(DB,acc)
localTargList:=EmptyList
oaci!FlowSpec:=flsp,
oaci!TargList:=acc!TargList
orei!ReasonCode:=ref!ReasonCode,
orei!TargList:=ref!TargList
OPEN.refuse_ind
(orei)
OPEN.accept_ind
(oaci)
OPEN.refuse_ind
(orei)
call allans(nexthops)
false
true
-
ESTBL
ESTBL,PENDING,
CHGING
CLOSE.req
(creq)
nh:=
DB!nexthops
nh=EmptyList
true
false
hop:=First(nh)
LRM_free(hop!TargList)
dis!ReasonCode:='ApplDisconnect',
dis!addr:=hop!addr,
dis!senderIPAddress:=selfIPAddress
nh:=Substring(nh, 2, Length(nh))
s_DISCONNECT
(dis)
ProcActive:=False
to_Appl
(origreq)
(origind)
to_MAC
r_ACCEPT, r_REFUSE
s_CONN, s_DISCONNECT, s_CHANGE

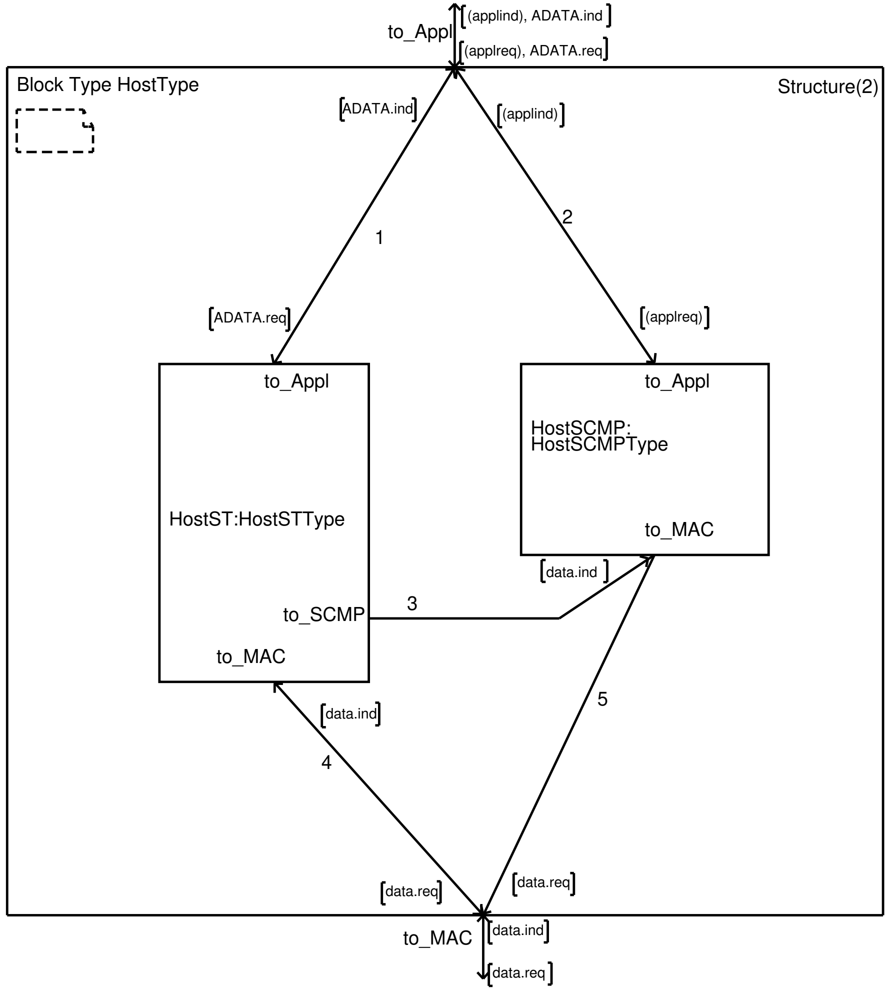

Service Type Intsetup
JOIN_op(6)
ESTBL
r_JOIN
(joi)
ExTarg:=call DB_TargExist(DB, joi!TargList),
NotExTarg:=call DB_TargNotExist(DB, joi!TargList)
ExTarg
else
EmptyList
DB!AuthLevel
0
else
nexthops:=call
routing(selfIPAddress,NotExTarg)
hop:=First(nexthops)
flsp:=call LRM_gen(hop)
call DB_HopExists
(DB, hop!addr)
true
false
call LRM_res(flsp, hop)
false
true
jrej!ReasonCode:='JoinAuthFailure',
jrej!TargList:=joi!TargList,
jrej!addr:=call IProuting(First(joi!TargList)!IPAddr),
jrej!SenderIPAddress:=selfIPAddress
jrej!ReasonCode:='CantGetResrc',
jrej!TargList:=joi!TargList,
jrej!addr:=call IProuting(First(joi!TargList)!IPAddr),
jrej!SenderIPAddress:=selfIPAddress
con!SenderIPAddress:=selfIPAddress,
con!flowSpec:=flsp,
con!TargList:=hop!TargList,
con!addr:=hop!addr,
con!AuthLevel:=DB!AuthLevel
s_JOIN_REJECT
(jrej)
s_JOIN_REJECT
(jrej)
s_CONN(con)
ESTBL
ESTBL
ESTBL
ESTBL
JOINING
JOINING
r_ACCEPT
(acc)
flsp:=acc!FlowSpec
LRM_adj
(flsp,acc!TargList)
ans(nexthops,acc!TargList)
DB_register(DB,acc)
ntf!FlowSpec:=flsp,
ntf!TargList:=acc!TargList,
ntf!SenderIPAddress:=selfIPAddress,
ntf!addr:=DB!prevhop
DB!AuthLevel
1
2
call LRM_not
(flsp, acc!SenderIPAddress)
true
false
DB!AsOrigin:=DB!AsOrigin//acc!TargList
s_NOTIFY(ntf)
s_NOTIFY(ntf)
ESTBL
r_REFUSE
(ref)
ref!ReasonCode
'ApplDisconnect'
else
LRM_free
(ref!TargList)
LRM_free
(ref!TargList)
ref!ReasonCode
'ApplRefused'
else
jrej!SenderIPAddress:=selfIPAddress,
jrej!ReasonCode:=ref!ReasonCode,
jrej!TargList:=ref!TargList,
jrej!addr:=call IProuting(joi!GeneratorIPAddress)
s_JOIN_REJECT
(jrej)
DB!nexthops
EmptyList
else
ProcActive:=false
ESTBL
call DB_InAsOrig
(DB_First(ref!TargList))
false
true
ref!addr:=DB!prevhop,
ref!SenderIPAddress:=selfIPAddress
s_REFUSE
(ref)
DB_del
(DB, ref!TargList)
-
r_CONN,
r_CHANGE,
r_JOIN
ESTBL,ADD,
CHGING, JOINING
r_NOTIFY(ntf1)
acc1!FlowSpec:=ntf1!FlowSpec,
acc1!TargList:=ntf1!TargList,
acc1!SenderIPAddress:=ntf1!SenderIPAddress
DB_register
(DB, acc)
ntf1!SenderIPAddress:=selfIPAddress,
ntf1!addr:=DB!prevhop
s_NOTIFY(ntf1)
-

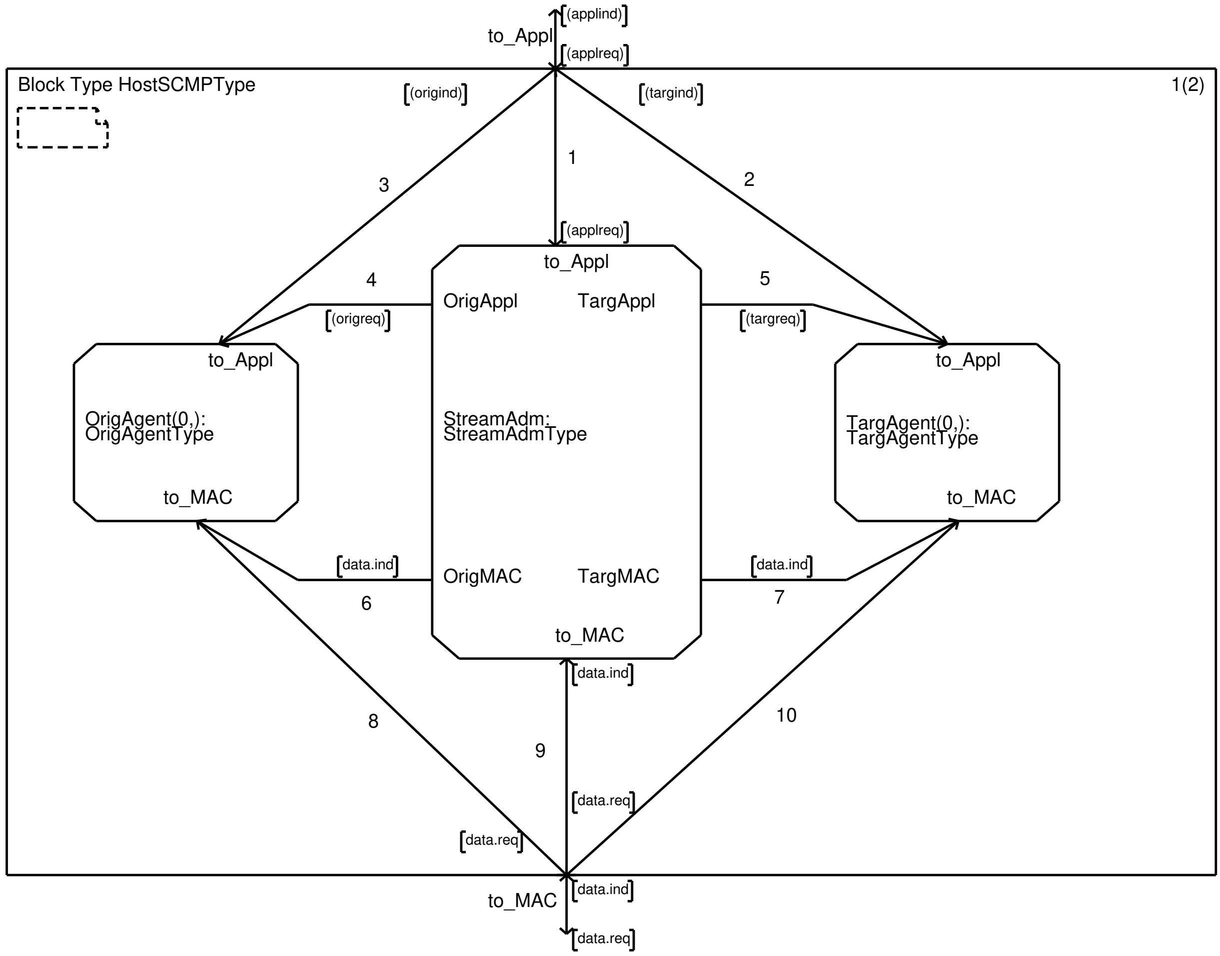

Service Type TargSetup
JOIN_op(6)
ESTBL
JOIN.req
(jreq)
ExTarg:=call DB_TargExist(DB, jreq!TargList),
NotExTarg:=call DB_TargNotExist(DB, jreq!TargList)
ExTarg
else
EmptyList
DB!AuthLevel
0
else
jrind!ReasonCode:='TargetExists',
jrind!TargList:=ExTarg
JOIN.reject_ind
(jrind)
ESTBL
jrind!ReasonCode:='JoinAuthFailure',
jrind!TargList:=ExTarg
JOIN.reject_ind
(jrind)
ESTBL
nexthops:=call
routing(selfIPAddress,NotExTarg)
flsp:=call LRM_gen(First(nexthops))
Targets:=First(nexthops)!TargList
oind!TargList:=Targets,
oind!FlowSpec:=flsp
OPEN.ind
(oind)
JOINING
JOINING
r_CONN,
r_CHANGE,
JOIN.req
OPEN.accept
(oacc)
OPEN.refuse
(oref)
acc!FlowSpec:=oacc!FlowSpec,
acc!TargList:=oacc!TargList,
acc!SenderIPAddress:=selfIPAddress,
acc!addr:=DB!prevhop
DB_register(DB, acc)
DB!AuthLevel
1
2
ntf!TargList:=oacc!TargList,
ntf!FlowSpec:=oacc!FlowSpec,
ntf!SenderIPAddress:=selfIPAddress,
ntf!addr:=DB!prevhop
DB!AsOrigin:=
DB!AsOrigin//acc!TargList
s_NOTIFY
(ntf)
DB!nexthops
else
EmptyList
ESTBL
ProcActive:=false
IDLE
JOIN.req
(jreq)
joi!addr:=call IProuting(jreq!OriginIPAddress),
joi!TargList:=jreq!TargList,
joi!SenderIPAddress:=selfIPAddress,
joi!GeneratorIPAddress:=selfIPAddress,
joi!OriginIPAddress:=jreq!OriginIPAddress
s_JOIN(joi)
IDLE
IDLE
r_JOIN_REJECT
(jrej)
jrind!ReasonCode:=jrej!ReasonCode,
jrind!TargList:=jrej!TargList
JOIN.reject_ind(jrind)
IDLE

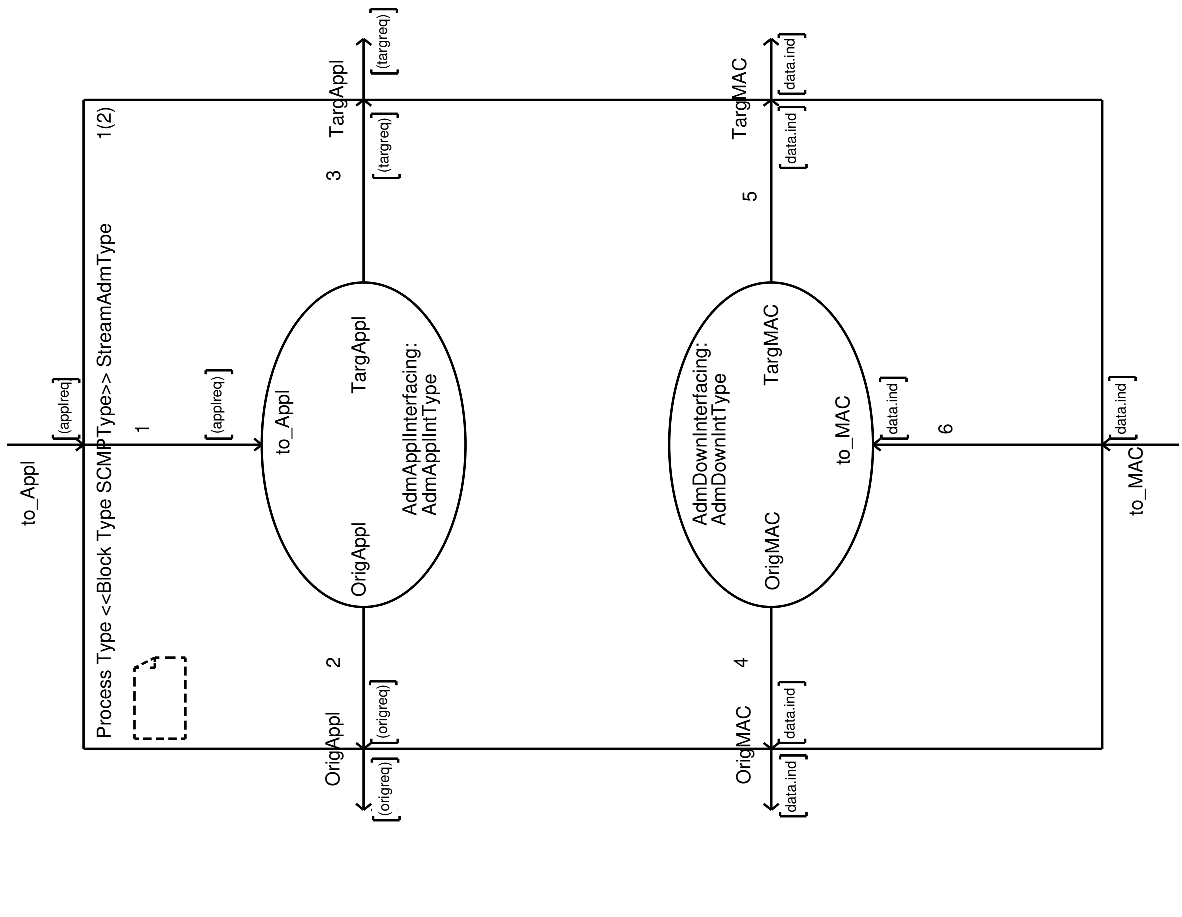

Service Type OrigSetup
OPEN(8)
selfIPAddress:= call getSelfIPAddress
IDLE
OPEN.req (oreq)
DB!AuthLevel:=oreq!AuthLevel
nexthops:=call routing(selfIPAddress, oreq!TargList)
Acknexthops:= EmptyList
AnsOpReference:=call GetNewRef(RefNum, selfIPAddress)
nh:=nexthops
nexthops
EmptyList
else
ESTBL
StartSend
StartSend
true
*
hop:=First(nh)
flsp:=oreq!FlowSpec
call LRM_res(flsp, hop)
false
true
ans(nexthops, hop!TargList)
Acknexthops:=Acknexthops// MkString(hop)
SET(ToConnect (hop!addr))
hop!NoR:=0
orei!ReasonCode:='CantGetResrc', orei!TargList:=hop!TargList
con!SenderIPAddress:=selfIPAddress, con!flowSpec:=flsp, con!TargList:=hop!TargList, con!addr:=hop!addr, con!AuthLevel:=oreq!AuthLevel, con!Reference:=AnsOpReference
OPEN.refuse_ind (orei)
s_CONN(con)
nh:=htail(nh)
nh
else
EmptyList
nexthops
EmptyList
else
-
ESTBL
WaitForAck
WaitForAck
(ack)
(err)
(ipaddr)
r_ACK
r_ERROR
ToConnect
*
call ReceiveAck
false
true
-
hop:=call GetHop (Acknexthops, ack!SenderIPAddress)
RESET(ToConnect(hop!addr))
SetConnRespTimers (hop!TargList)
Ackans(Acknexthops, ack!SenderIPAddress)
call ReceiveErr
false
true
-
hop:=call GetHop(Acknexthops, err!SenderIPAddress)
hop:=call GetHop(Acknexthops, ipaddr)
hop!NoR>NConnect
true
false
dis!TargList:=hop!TargList, dis!ReasonCode:='RetransTimeout', dis!SenderIPAddress:=selfIPAddress, dis!addr:=hop!addr
s_DISCONNECT (dis)
SendDis(dis)
Ackans(Acknexthops,hop!addr)
ans(nexthops, hop!TargList)
orei!TargList:=hop!TargList, orei!ReasonCode:='RetransTimeout'
OPEN.refuse_ind (orei)
con!SenderIPAddress:=selfIPAddress, con!FlowSpec:=hop!FlowSpec, con!TargList:=hop!TargList, con!AuthLevel:=DB!AuthLevel, con!addr:=hop!addr, con!Reference:=AnsOpReference
SET(ToConnect (hop!addr))
hop!NoR:= hop!NoR+1
s_CONN(con)
-
Acknexthops
else
EmptyList
nexthops
EmptyList
else
-
ESTBL
PENDING
PENDING
r_ACCEPT (acc)
r_REFUSE (ref)
ADD.req,CHG.req, r_JOIN
ToConnResp (Targ)
call ReceiveAcc
false
true
-
ResetConnRespTimers (acc!TargList)
flsp:=acc!FlowSpec
LRM_adj (flsp,acc!TargList)
ans(nexthops,acc!TargList)
DB_register(DB,acc)
oaci!FlowSpec:=flsp, oaci!TargList:=acc!TargList
OPEN.accept_ind (oaci)
call ReceiveRef
false
true
-
ref!ReasonCode
'ApplDisconnect'
else
ResetConnRespTimers (ref!TargList)
LRM_free (ref!TargList)
ans(nexthops,ref!TargList)
orei!ReasonCode:=ref!ReasonCode, orei!TargList:=ref!TargList
OPEN.refuse_ind (orei)
LRM_free (ref!TargList)
DB_del (DB, ref!TargList)
lind!ReasonCode:=ref!ReasonCode, lind!TargList:=ref!TargList
LEAVE.ind (lind)
-
TargList:=MkString(Targ)
hop:=call GetHopToTarg(nexthops, Targ)
LRM_free (TargList)
ans(nexthops,TargList)
dis!TargList:=TargList, dis!ReasonCode:='ResponseTimeout', dis!SenderIPAddress:=selfIPAddress, dis!addr:=hop!addr, dis!Reference:=call GetNewRef (RefNum, selfIPAddress)
s_DISCONNECT (dis)
SendDis(dis)
orei!ReasonCode:='ResponseTimeout', orei!TargList:=TargList
OPEN.refuse_ind (orei)
nexthops
else
EmptyList
-
ESTBL

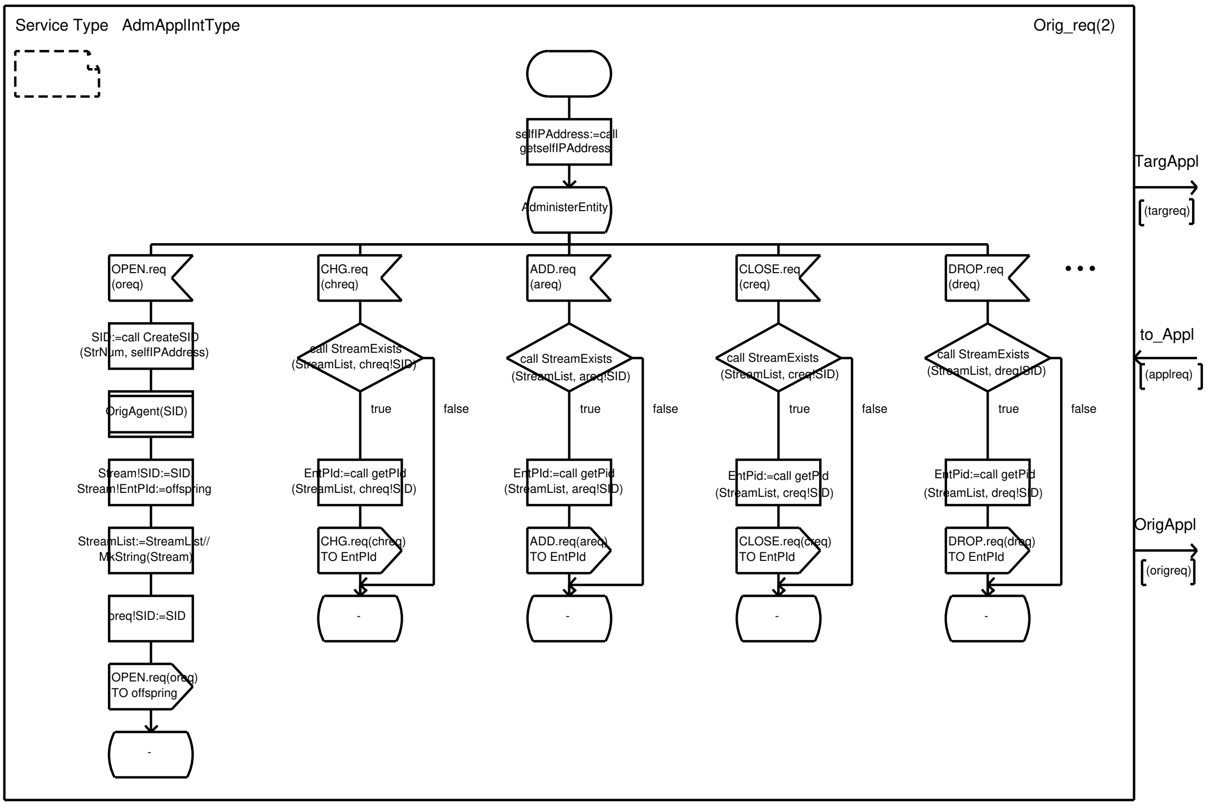

Service Type IntDownIntType
receive(2)
DCL log logType :=EmptyList;
DCL S sdu_t;
READY
data.ind(S)
call name(S)
'CONNECT'
'CHANGE'
'ACK'
'ERROR'
con:= call extrCon(S)
cha:= call extrCha(S)
ack:= call extrAck(S)
err:= call extrErr(S)
con!CheckSum
cha!CheckSum
ack!CheckSum
err!CheckSum
true
false
call msgAlreadyLogged (log, con!Reference)
call msgAlreadyLogged (log, cha!Reference)
logmsg(log,con!Reference)
logmsg(log,cha!Reference)
err!ReasonCode:='CkSumBadCtl', err!SenderIPAddress:=selfIPAddress, err!Reference:=con!Reference
ack!ReasonCode:='DuplIngn', ack!SenderIPAddress:=selfIPAddress, ack!Reference:=con!Reference
err!ReasonCode:='CkSumBadCtl', err!SenderIPAddress:=selfIPAddress, err!Reference:=cha!Reference
ack!ReasonCode:='DuplIngn', ack!SenderIPAddress:=selfIPAddress, ack!Reference:=cha!Reference
ICI:=con!SenderIPAddress, SDU:=call buildErr(err)
ICI:=con!SenderIPAddress, SDU:=call buildAck(ack)
ICI:=cha!SenderIPAddress, SDU:=call buildErr(err)
ICI:=cha!SenderIPAddress, SDU:=call buildAck(ack)
data.req(ICI,SDU)
r_CONN(con)
r_CHANGE(cha)
r_ACK(ack)
r_ERROR(err)
-

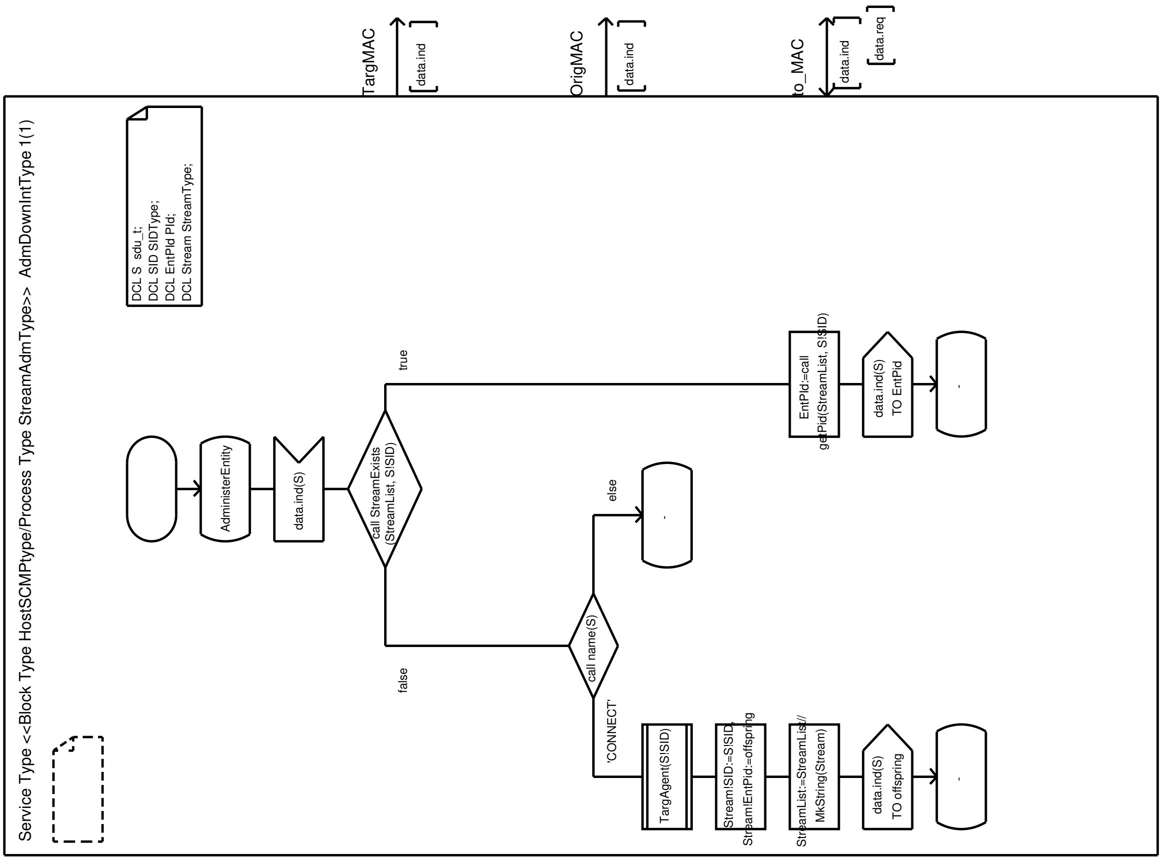

use StepSigDat;
use StepClasses;
System KommSystem
structure(1)
Node1:
HostType
to_Appl
to_MAC
Node3:
HostType
to_Appl
to_MAC
Ch1
(origind)
(origreq)
Ch2
(targreq)
(targind)
b1
data.ind
data.req
b3
data.ind
data.req
Basisdienst
data.req
data.ind
b2
b4
data.req
data.ind
to_MAC
Node2:
RouterType
to_MAC
Node4:
HostType
to_Appl
Ch3
(targreq)
(targind)

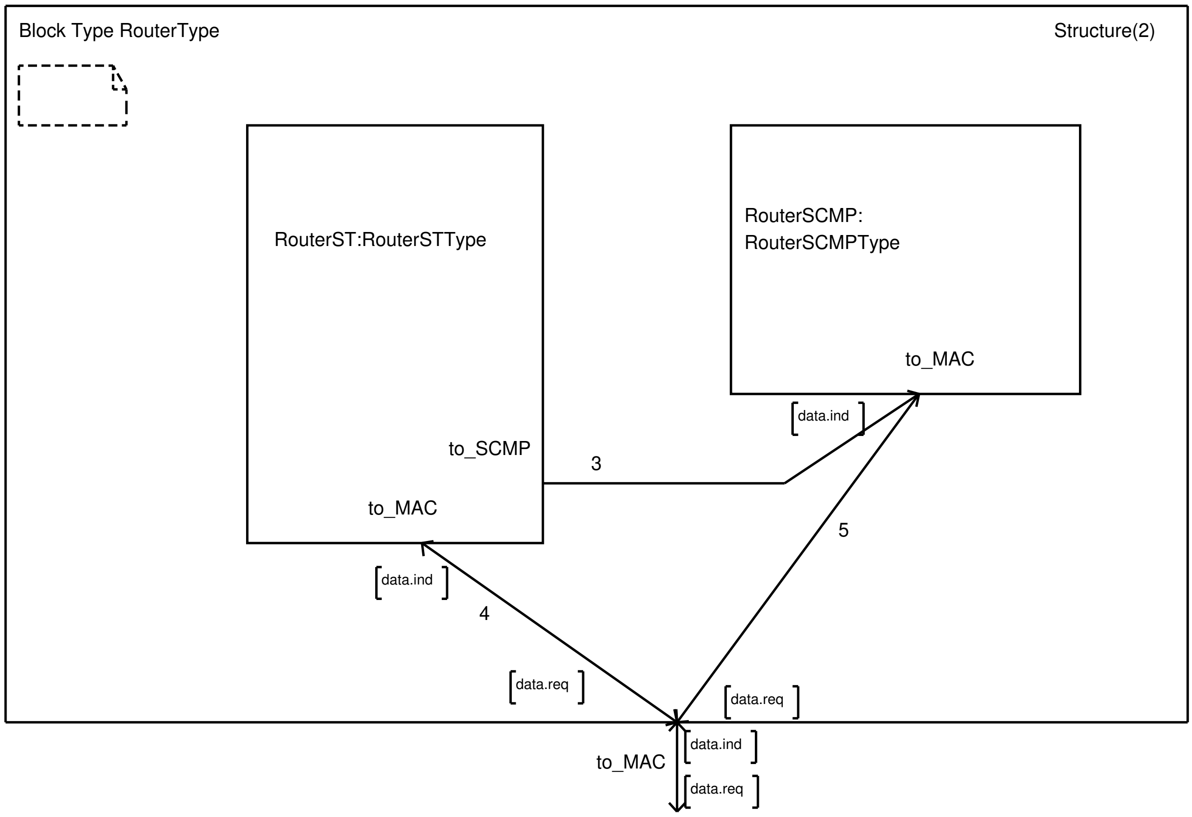

to_Appl
(applind), ADATA.ind
(applreq), ADATA.req
Block Type HostType
Structure(2)
ADATA.ind
(applind)
1
2
ADATA.req
(applreq)
to_Appl
HostST:HostSTType
to_SCMP
to_MAC
to_Appl
HostSCMP:
HostSCMPType
to_MAC
data.ind
3
data.ind
4
5
data.req
data.req
to_MAC
data.ind
data.req

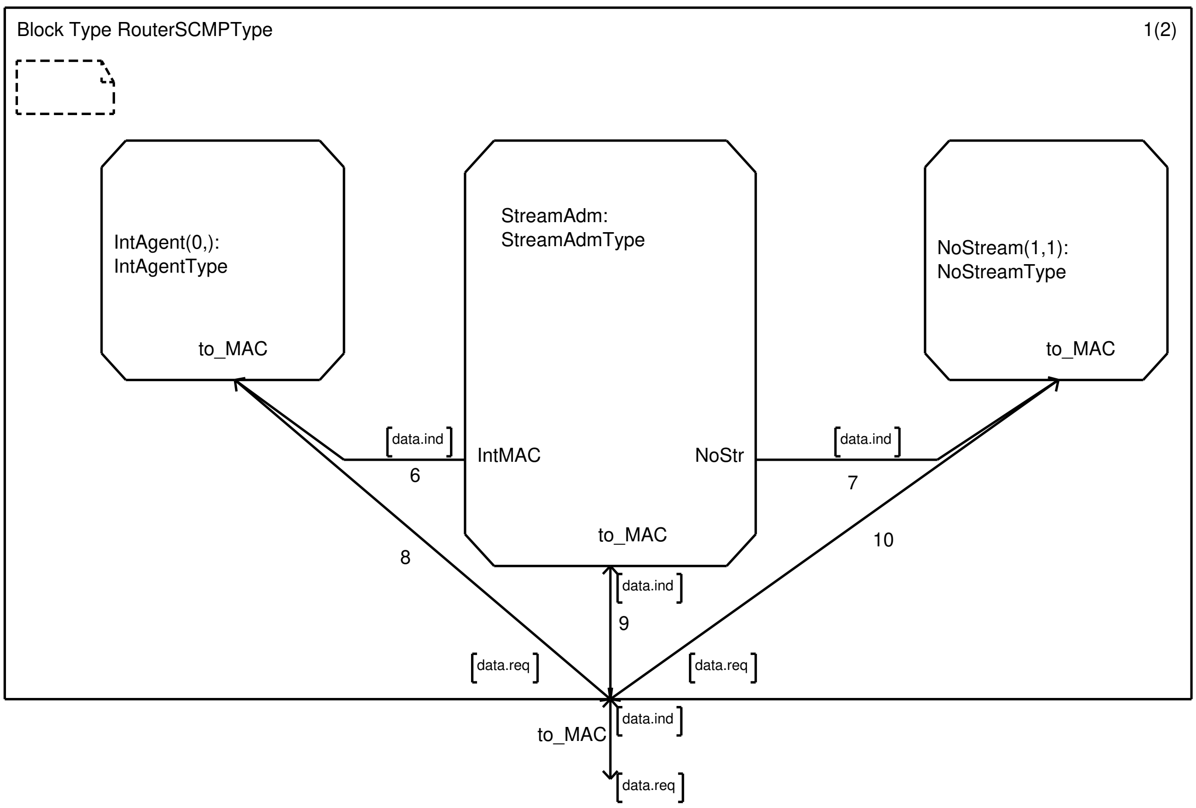

Block Type HostSCMPType
1(2)
to_Appl
(applind)
(applreq)
(origind)
(targind)
1
2
3
4
5
to_Appl
OrigAppl
TargAppl
(origreq)
(targreq)
StreamAdm:
StreamAdmType
OrigMAC
TargMAC
to_MAC
to_Appl
OrigAgent(0,):
OrigAgentType
to_MAC
to_Appl
TargAgent(0,):
TargAgentType
to_MAC
data.ind
6
7
8
9
10
data.req
to_MAC

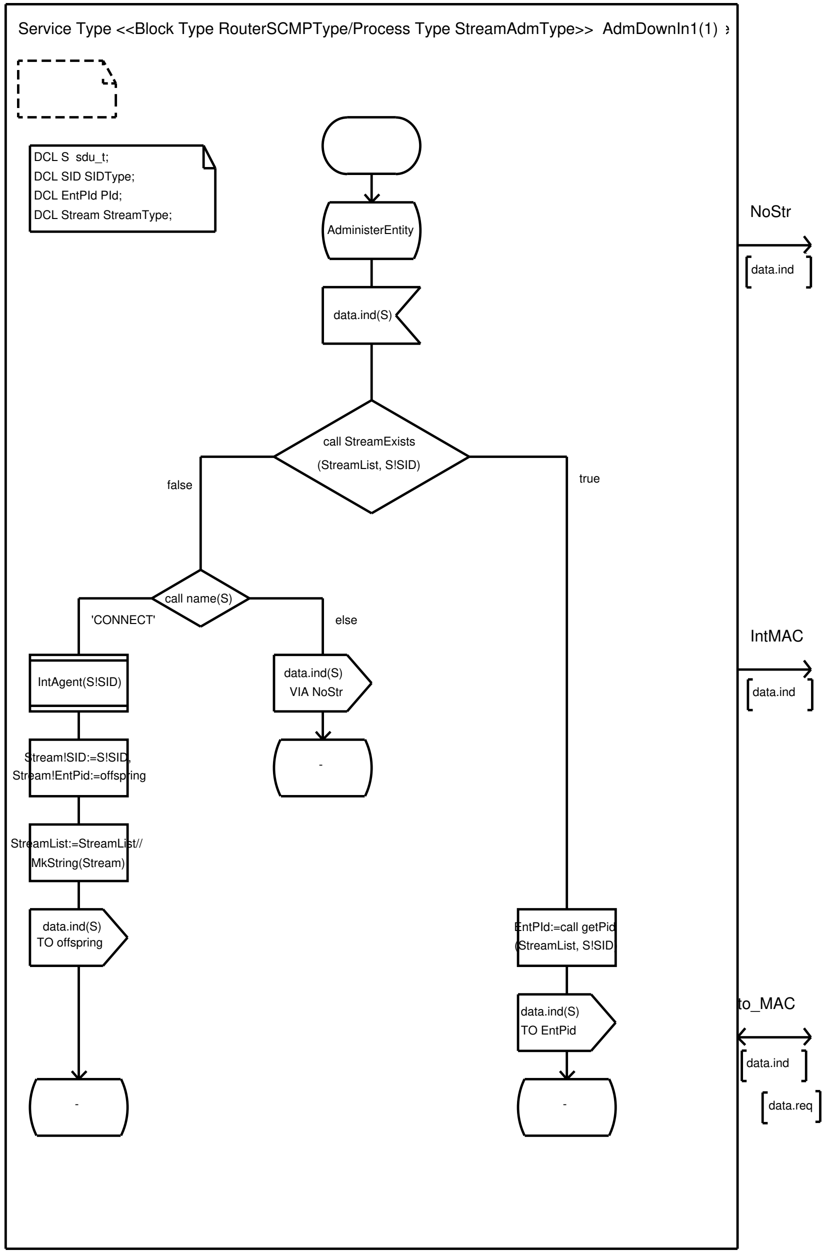

Process Type <<Block Type SCMPType>> StreamAdmType
1(2)
to_Appl
(applreq)
1
to_Appl
TargAppl
OrigAppl
AdmApplInterfacing:
AdmApplIntType
TargAppl
OrigAppl
(targreq)
(origreq)
3
2
AdmDownInterfacing:
AdmDownIntType
TargMAC
OrigMAC
to_MAC
data.ind
4
5
6
to_MAC

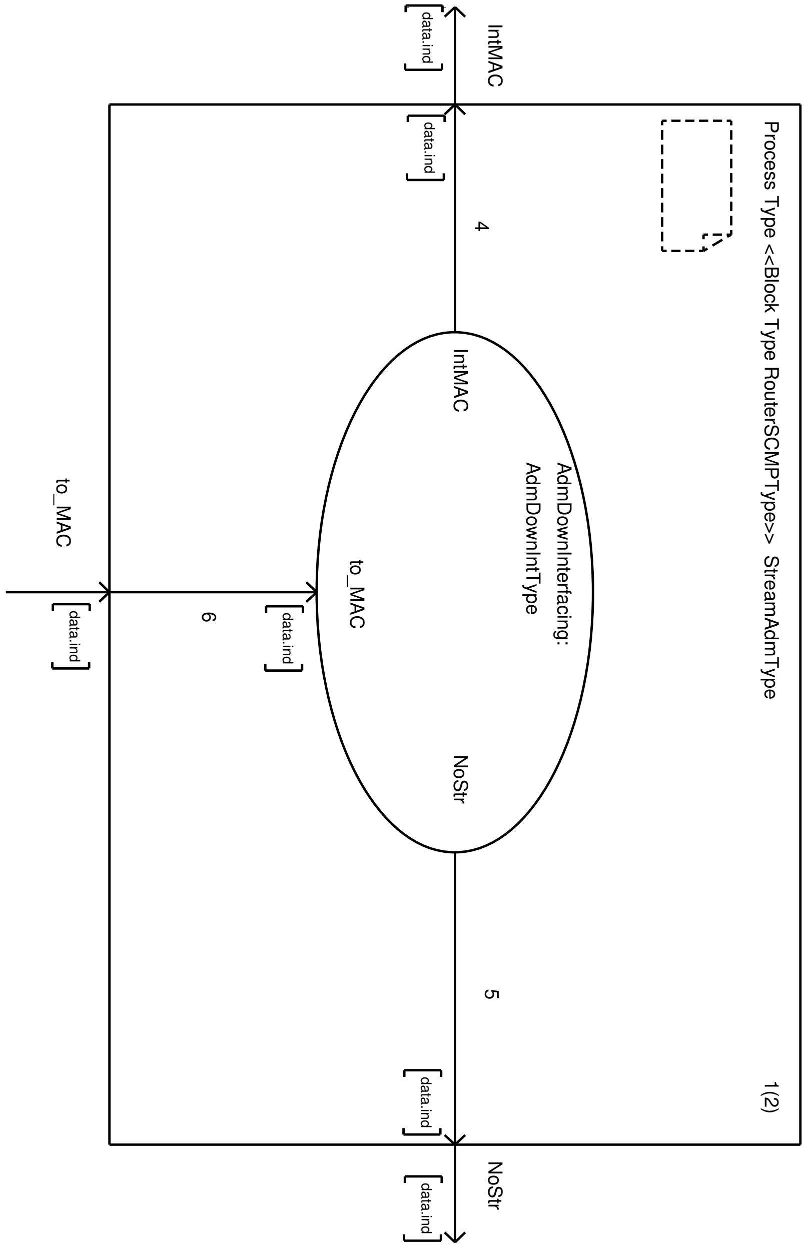

Service Type AdmApplIntType
Orig_req(2)
selfIPAddress:=call getselfIPAddress
AdministerEntity
OPEN.req (oreq)
CHG.req (chreq)
ADD.req (areq)
CLOSE.req (creq)
DROP.req (dreq)
SID:=call CreateSID (StrNum, selfIPAddress)
OrigAgent(SID)
Stream!SID:=SID Stream!EntPId:=offspring
StreamList:=StreamList// MkString(Stream)
oreq!SID:=SID
OPEN.req(oreq) TO offspring
call StreamExists (StreamList, chreq!SID)
call StreamExists (StreamList, areq!SID)
call StreamExists (StreamList, creq!SID)
call StreamExists (StreamList, dreq!SID)
true
false
EntPId:=call getPId (StreamList, chreq!SID)
EntPId:=call getPId (StreamList, areq!SID)
EntPid:=call getPId (StreamList, creq!SID)
EntPid:=call getPId (StreamList, dreq!SID)
CHG.req(chreq) TO EntPId
ADD.req(areq) TO EntPId
CLOSE.req(creq) TO EntPId
DROP.req(dreq) TO EntPId
TargAppl
(targreq)
to_Appl
(applreq)
OrigAppl
(origreq)


Service Type <<Block Type HostSCMPtype/Process Type StreamAdmType>> AdmDownIntType 1(1)
DCL S sdu_t;
DCL SID SIDType;
DCL EntPId PId;
DCL Stream StreamType;
TargMAC
data.ind
OrigMAC
data.ind
to_MAC
data.ind
data.req
AdministerEntity
data.ind(S)
call StreamExists (StreamList, S!SID)
true
false
EntPId:=call getPId(StreamList, S!SID)
data.ind(S) TO EntPId
call name(S)
else
'CONNECT'
TargAgent(S!SID)
Stream!SID:=S!SID Stream!EntPId:=offspring
StreamList:=StreamList// MkString(Stream)
data.ind(S) TO offspring


Block Type RouterType
Structure(2)
RouterST:RouterSTType
to_SCMP
to_MAC
RouterSCMP:
RouterSCMPType
to_MAC
data.ind
3
5
data.ind
4
data.req
data.req
data.ind
to_MAC
data.req


Block Type RouterSCMPType
1(2)
IntAgent(0,):
IntAgentType
to_MAC
StreamAdm:
StreamAdmType
IntMAC
NoStr
to_MAC
NoStream(1,1):
NoStreamType
to_MAC
data.ind
6
data.ind
7
8
10
data.ind
9
data.req
data.req
data.ind
to_MAC
data.req


Service Type <<Block Type RouterSCMPType/Process Type StreamAdmType>> AdmDownIn1(1)
DCL S sdu_t;
DCL SID SIDType;
DCL EntPId PId;
DCL Stream StreamType;
AdministerEntity
data.ind(S)
call StreamExists
(StreamList, S!SID)
false
true
call name(S)
'CONNECT'
else
IntAgent(S!SID)
data.ind(S)
VIA NoStr
Stream!SID:=S!SID,
Stream!EntPid:=offspring
StreamList:=StreamList//
MkString(Stream)
data.ind(S)
TO offspring
EntPId:=call getPid
(StreamList, S!SID)
data.ind(S)
TO EntPid
NoStr
data.ind
IntMAC
data.ind
to_MAC
data.ind
data.req


Process Type <<Block Type RouterSCMPType>> StreamAdmType
1(2)
AdmDownInterfacing:
AdmDownIntType
IntMAC
data.ind
data.ind
4
IntMAC
to_MAC
data.ind
6
data.ind
to_MAC
NoStr
5
data.ind
NoStr
data.ind