A Pattern Language for Communication Protocols

YoungJoon Byun[1], Beverly Sanders[1], and KiSook Chung[2]

[1] Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611
{ybyun,sanders}@cise.ufl.edu
[2] Network Technology Lab., ETRI
161 Kajong-Dong, Yusong-Gu, Taejon 305-600, Korea
{kschung}@etri.re.kr

Abstract. In this paper, we suggest a pattern language, a collection of related design patterns, for the development of communication protocols with an emphasis on an SDL (Specification and Description Language) implementation. The patterns are grouped in two categories: structural patterns and behavioral patterns. The structural patterns are focused on the architectural aspects of communication protocols. The behavioral patterns capture common behavior of protocols modeled in communicating extended finite state machines (CEFSM).

1 Introduction

Patterns offer a way to describe a solution to a commonly occurring problem so that the solution can be reused. A pattern language is a collection of patterns that work together to solve problems in a specific domain. Table 1 shows a pattern language dedicated to communication protocols written in SDL. The patterns are grouped in two categories: structural patterns and behavior patterns.

Structural Patterns: The patterns in this category address the overall architecture of a communication protocol. The architecture is composed of several blocks along with communication paths between them. A block is an architectural building element of a developing system and can contain other blocks, resulting in a tree structure. Thus, there are two kinds of blocks in the structural patterns: a leaf block and non-leaf block. At this point, the blocks are considered to be black boxes: the external interfaces such as communication paths and messages are defined, but the internal details are not.
Behavioral Patterns: The behavioral patterns help the developer design the internal behavior of the leaf blocks identified in the structural patterns. Each block instance has a state that may change to other state in response to events such as a message input. The response to an event may also trigger additional events such as the generation of output messages. We use communicating extended finite state machines (CEFSM) to formally describe this behavior. Predicates and

timers may be used to describe conditional behavior and timing constraints. We use the event, signal, and message interchangeably in this paper.

Table 1. Pattern language for communication protocols

Category	Patterns	Variants
Structural Patterns	Protocol Layer	Split Protocol Layer
	Mux	
	Dynamic Handler	Split Dynamic Handler
Behavioral Patterns	Basic CEFSM	Predicate CEFSM
		Predicate after Action
		Source Merge
		Target Merge
		Sequential Merge
	Timer	
	Repeated Events	Timed Repeated Events
		Timed Repeated Trials
	Message Transfer	Simple Sender
		Simple Receiver
		Confirmed Sender
		Confirmed Receiver
		Timed Confirmed Sender
		Repeated Sender
		Repeated Receiver
		Repeated Confirmed Sender
		Repeated Confirmed Receiver
		Timed Repeated Trial Receiver
		Timed Repeated Trial Confirmed Sender
		Timed Repeated Trial Confirmed Receiver
		Message Transfer in Middle Layer

Design patterns have a particular form to present design problem and solution. Most forms have name, context, problem, and solution sections [4]. Our patterns use the traditional pattern template as used in [4, 9], but emphasize the SDL implementation. Table 2 shows our pattern format.

2 Structural Patterns

2.1 Protocol Layer

Context We need to design a complex system such as a communication protocol. Some parts of the system may already exist.

Problem How can we describe the structure of a communication protocol system?

Table 2. Pattern form to describe a pattern

Name	Name of the pattern. It must have a meaningful word or phrase to clearly describe the main purpose of the pattern.
Context	Situation in which the problem occurs.
Problem	Problem to be solved. It provides a general problem specification and identifies the essence of the problem.
Forces	Various viewpoints of the problem. It provides requirements that the solution must take into account, constraints to be considered, and desirable properties of the solution.
Solution	A solution that solves the problem.
Implementation	The implementation of the solution in SDL. Generally, it is obtained by an one-to-one mapping from the solution.
Example	An example that uses the pattern.
Variants	Variants or specializations of the pattern.
See also	Usages of the pattern in other systems or similar patterns.

Forces The system is too large and complex to be able to understand completely. We need techniques to help manage the complexity.

- Decomposition: We decompose the system into several subsystems that can be dealt with more-or-less independently.
- Abstraction: Each subsystem is treated as a black box and specifies the interfaces with other subsystems. Internal details are left for later.
- Reuse: Some of the subsystems may already be available and thus do not need to be designed.

Solution A communication protocol can be designed in layers. Each layer handles problems at a particular level of abstraction. A layer offers services to the higher layer and uses services from the next lower layer. This structure allows a protocol developer to design external interfaces before internal functionality [2, 11].

First, we identify the blocks belonging to a layer and determine the communication paths between the adjacent layers. The layers and communication paths are logical objects that may or may not correspond directly to physical components of network or communication links.

Second, we address the messages between layers and associate the messages with the communication paths. The communication paths show the list of message types that can be sent on the path and the direction of the message flow. Fig. 1 shows a protocol layer that includes one block, four communication paths, a

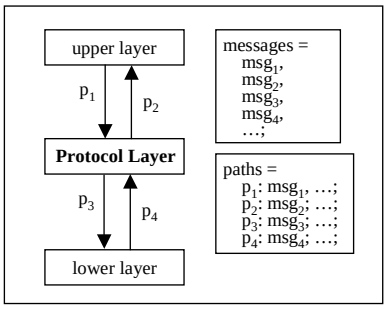

Fig. 1. A structure of *protocol layer*

message list, and the adjacent layers. The internal behavior of the protocol layer block can be designed using other patterns of this pattern language.

Variant: Split Protocol Layer Often, a communication layer can be conceptually split into two related functions. For example, sending and receiving for message transfer [3]. It may be helpful for the designer to consider the two functions separately. Fig. 2 shows a structure of the pattern *split protocol layer* where the *Outgoing* block initiates a communication requested from the upper layer and the *Incoming* block handles messages coming from the lower layer.

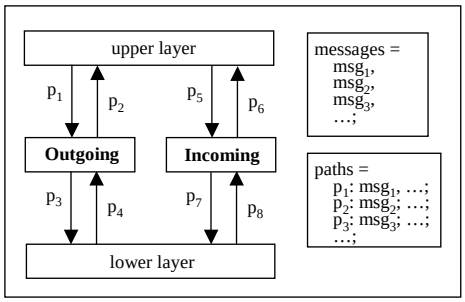

Fig. 2. A structure of *split protocol layer*

Implementation in SDL The SDL implementation can be obtained directly from the design. SDL has two constructs to describe a system structure: SDL blocks and SDL processes. SDL blocks are pure structuring mechanisms that may contain other blocks and processes while SDL processes contain the specification of behavior.

Typically, the non-leaf blocks of the structural patterns are mapped to SDL blocks and the communication paths between them are mapped to SDL channels. Two types of channels are possible in SDL. One is a delaying channel and another is a non-delaying channel. The selection of the channel is dependent on the situation. Leaf blocks may be mapped to SDL processes. In this case, the communication paths are mapped to signal routes which connect processes to other processes and to the channels of their containing block. Messages flowing on communication paths that are mapped to channels or signal routes are called signals in SDL.

Fig. 3 shows an implementation of the pattern *protocol layer* of Fig. 1. The *Protocol Layer* of Fig. 1 is mapped to an SDL block *Protocol_Layer*. Each communication path is converted to a delaying channel such as c_1, c_2, c_3, and c_4. The signals correspond to the messages flowing through the channel.

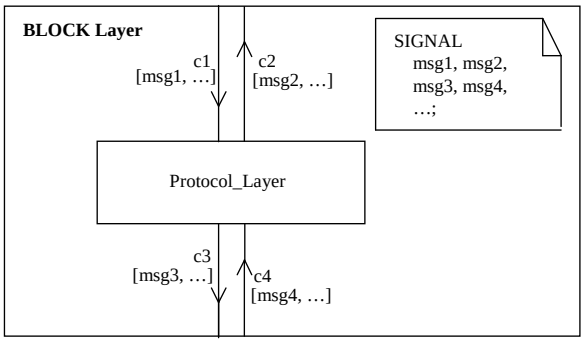

Fig. 3. SDL implementation of Fig. 1

Examples The example shows a part of the Service Specific Coordination Function (SSCF) for the User-Network Interface (UNI) in the ATM signaling system [16]. The *SSCF_UNI* layer provides a mapping function between *ATM Signaling* layer and Service Specific Connection Oriented Protocol (SSCOP) layer. The basic structure of SSCF_UNI is an example of the pattern *protocol layer* as Fig. 4.

Three kinds of messages such as *AAL_EST*, *AAL_REL*, and *AAL_DATA* are exchanged with the upper layer through the communication paths *UNI2-SSCF* and *SSCF2UNI*. *AAL_EST* is used to establish a connection from the UNI signaling layer in the form of a request and a confirmation such as *AAL_EST.req* and *AAL_EST.conf*. It also informs the upper layer that an incoming connection has been established by an indication message *AAL_EST.ind*. *AAL_REL* is used to release the connection established. *AAL_DATA* is used by the upper layer to send a data packet in a request form *AAL_DATA.req* and *SSCF_UNI* hands

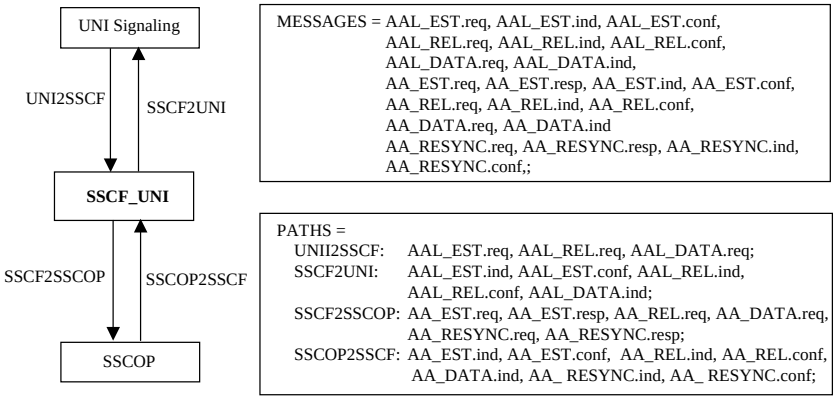

Fig. 4. Example of *protocol layer* for SSCF at UNI

out a received packet to its user in an indication form such as *AAL_DATA.ind*. The lower interface has similar messages which uses the communication paths *SSCF2SSCOP* and *SSCOP2SSCF*.

As an example of the pattern *split protocol layer*, we demonstrates a variation of alternating bit protocol (ABP) [14] which provides simple but reliable message transfer on a lossy lower layer. Fig. 5 illustrates the architecture of the protocol. The upper interface uses two messages *put* and *get* for a reliable data transmission. The lower interface uses messages *data_req*, *data_ind*, *ack_req*, and *ack_ind* to send and acknowledge messages, allowing for retransmission if necessary to deal with message loss.

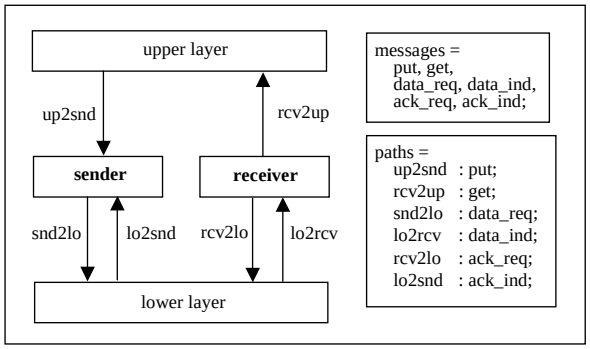

Fig. 5. Structure of ABP using *split protocol layer*

See Also *protocol conduit* [13], *layers* [4], *SDL and layered systems* [1], *Pattern Half Object + Protocol* [15]

2.2 Mux

Context In a layered design, there may be multiple blocks in an adjacent layer and resolving the destination or source of the messages is required.

Problem How can we resolve the destination or source of a message?

Solution Use a table to map an instance of a block to its address. Fig. 6 describes the structure of a mux layer where the upper layer has several blocks, while the lower layer has one block. Similarly, the pattern is applicable to the reverse situation.

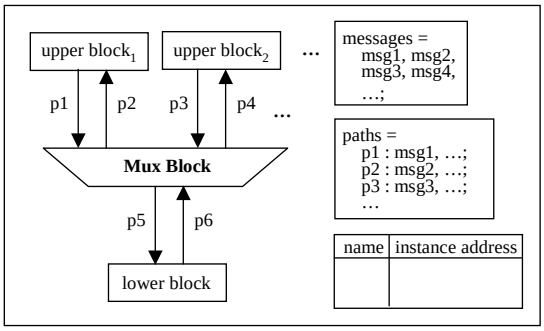

Fig. 6. A structure of *mux layer*

Implementation in SDL Fig. 7 shows an implementation of the pattern *mux* where the *Mux_Block* is an SDL process, and communication paths such as r_1, r_2, r_3 are signal routes. The process array, *Mux_Table* shows a trivial implementation of the address resolution. It stores every instance identifier, *PID*, with the key *name*. The signals enumerate the messages flowing through the signal routes. Note that the signal routes of the upper interface are joined in one point to be connected with outside channel.

See Also *mux conduit* [13]

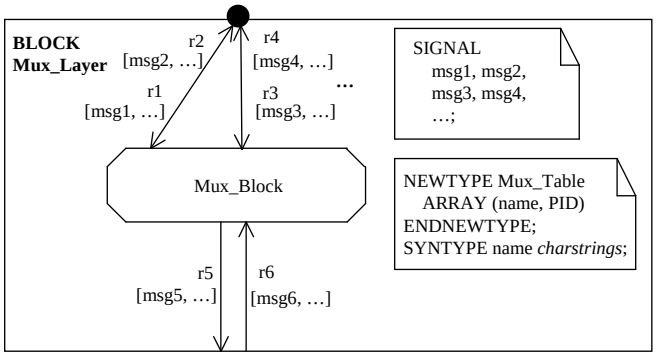

Fig. 7. An SDL implementation of *mux layer*

2.3 Dynamic Handler

Context A block needs to handle multiple communications at the same time. The expected load will be variable and the system is appropriately sized to handle it.

Problem How can a block be organized internally to service multiple communication requests?

Forces

- *Concurrent processing*: To provide a good response time, requests should be processed concurrently.
- *Capacity*: Handling concurrent requests imposes overhead for context switches, resource contention, etc. If too many requests are handled at the same time, the overhead will dominate the computation and the system performance will degrade unacceptably. Therefore the amount of concurrency should be bounded. Finding the optimal bound may be difficult to determine.
- *Static vs dynamic handler creation*: A communication request is serviced by a handler, an instance of a block servicing the request. An important design decision considers when the handler instances should be created. The static approach creates all handlers at the system start-up time and their lifetime extends until the system is shut down. When not servicing a request, the handlers are idle, but still utilizing system resources. On the other hand, an idle handler may be quickly deployed to handle a new request with little overhead.
  The dynamic approach creates a handler upon a request and its lifetime spans only as long as necessary to service the request. This approach incurs overhead for handler creation and termination, but there are no idle handlers to unnecessarily consume system resources. Thus the resources used by request handling adapt to the load.

A static approach is useful when the system load is uniformly high. A dynamic approach is better when the system is appropriately sized and the load is variable.
Hybrid approaches combining aspects of static and dynamic handling are also possible.

Solution The context of this pattern leads us to choose a dynamic approach.

The solution utilizes a single instance of a static block, *Admin*, (an example of the Singleton Pattern [7]) which is created at system startup time. The instance waits for a communication request message from adjacent layers. Upon arrival of the request message, *Admin* serves as a factory [7] and dynamically creates an instance of a *Handler* block. The *Handler* instance then services that communication. After the communication has been serviced, the instance is terminated. The maximal number of instances of *Handler* that can be created is bounded by a fixed number N. If there are more requests than N, *Admin* must either queue the message, or more typically reject the request. N is determined empirically, or by performance analysis using the expected load on the system.

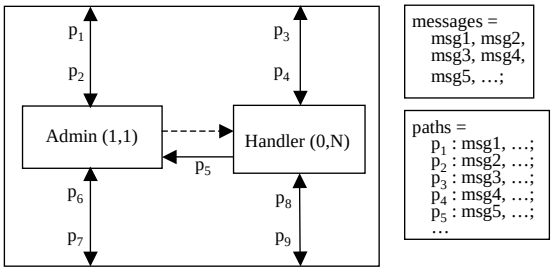

Fig. 8. A structure of *dynamic handler*

Fig. 8 shows a structure of the pattern. First, the entity *Admin* waits for a request through the communication paths either p_2 or p_6. Upon receiving a message, for instance *msg2* through p_2, the *Admin* creates an instance of *Handler* block giving the necessary information for communication, for instance, the address of the requester. The dotted line from *Admin* to *Handler* means the creation of an instance. After being created, the instance starts communication through the paths p_3, p_4, p_8, and p_9.

When the communication ends, the *Handler* instance informs the *Admin* of termination of service by sending, for example, a message *msg5* and ceases to exist.

Variant: Split Dynamic Handler The pattern *dynamic handler* considers only one type of handler. In communication systems, however, it is common to have handlers in pairs such as the pattern *split protocol layer*.

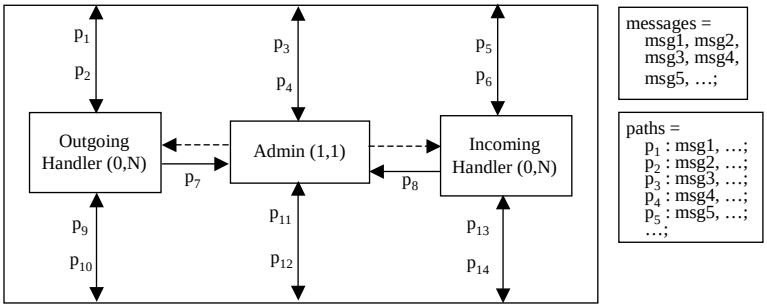

Fig. 9. A structure of *split dynamic handler*

Fig. 9 describes a dynamic creation of two type handlers, *Outgoing Handler* and *Incoming Handler*. The block *Admin* creates one of them depending on the type of a request. All other behavior is similar to the pattern *dynamic handler*. Note that the two handlers may need internal communication between them.

Implementation in SDL For the implementation of pattern *dynamic handler*, developers must consider both the structure and the behavior of the blocks in the pattern. Fig. 10 shows the structure of the pattern where two processes, *Admin* and *Handler*, exist with the initial and maximum number of instances. The process *Admin* has one instance during its life span, while the process type *Handler* has no instance at startup time and can have the maximum N instances. The processes are connected to the boundary of the block with signal routes which will interact with outside channels. The behavior of the pattern needs the creation and termination of an SDL process instance.

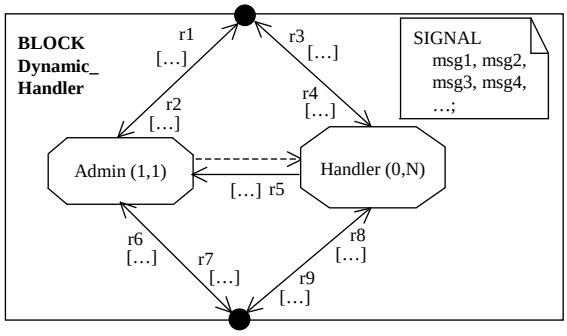

Fig. 10. SDL implementation of Fig. 8

Examples Fig. 11 shows a simplified file transfer server using the pattern *dynamic handler*. In this example, we do not present the interface with the lower layer for simplicity. The server is composed of a block *FTP_Admin* and a block *FTP_Handler*. When a user tries to download a file, an event *FTP_connect* goes to the block *FTP_Admin* indicating a file transfer trial. The block creates an instance of *FTP_Handler* to make it possible for the user to download a file from the server. The instance sends a message *FTP_connect_ok* to mean that it is ready to receive a command. For the command *get* with a file name wanted, the *FTP_Handler* provides the requested *file* with the message *success*. After getting the file, the user sends a *disconnect* message which makes the instance stop after sending the message *terminate* to the *FTP_Admin*.

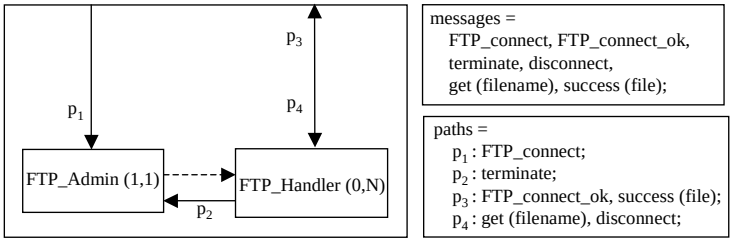

Fig. 11. A file transfer server using *dynamic handler*

As an example of *split dynamic handler*, Fig. 12 shows a simplified version of call control block composed of *Call_Admin*, *Outgoing_Handler*, and *Incoming_Handler* in a switching system. When a calling party tries a call, a message *H_init* goes to the block *Call_Admin* indicating there is a call request from a calling party. The block creates an instance of the block *Outgoing_Handler* in order to make the instance manage the calling party. After generating a message *L_init*, the instance waits for a call connection from a called party.

On the other hand, if the block *Call_Admin* receives a message *L_alert* implying there is an incoming call, it creates an instance of the block *Incoming_Handler* to setup a call connection with the called party. A message *H_alert* is used to indicate a new call is coming. When the called party answers, the messages such as *H_answer*, *L_answer*, *L_complete*, and *H_complete* are transferred.

See also *DynamicEntitySet* [8], *ConduitFactory* [13], *Pattern Half Object + Protocol* [15]

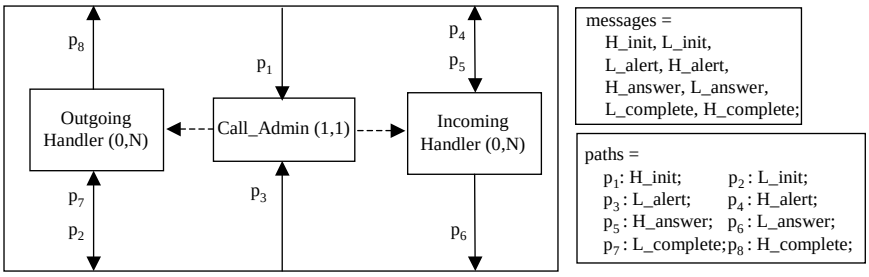

Fig. 12. A call control block using *split dynamic handler*

3 Behavioral Patterns

3.1 Communicating Extended Finite State Machine

Context Many communication systems react to events coming from outside environments. The systems can be modeled by distinct states and transitions. When a system receives an event, it moves from its current state to a new state while performing some actions and providing output signals.

Problem How can we describe the behavior of a communication system?

Forces

- *Understandability*: The notation should capture the most important aspects of the behavior of a system in a way that is convenient to express and can be easily understood by a reader.
- *Completeness*: The notation should be able to express all the important aspects of a design.
- *Definedness*: The notation should be well defined, preferably standardized. A formally defined notation allows the possibility of tool support for analysis, simulation, verification, code generation, etc.
- *Scalability*: The notation should remain tractable for systems with large numbers of states.
- *Ease of implementation*: The formalism should represent the system's behavior in a way that can be easily mapped into an implementation language.

Solution The behavior of a communication system can be described using a communicating extended finite state machine (CEFSM). CEFSMs are finite state machines extended with local variables and parameterized communication events indicating communication with another CEFSM [6, 12]. These state machines are very familiar to computer scientists and engineers and meet the criteria discussed in the forces section. In particular, the local variables help with the scalability

problem by allowing, for example, an 8-bit counter to be represented by one variable instead of 256 states. Other state based formalisms [10] may provide better support for modularizing large designs at the expense of a more complex notation. A survey of other formalisms that can be used for telecommunication systems design can be found in [5].

When an event is initiated by the environment, the system updates local variables, emits output events and transitions to a new state.

Definition 1 (CEFSM) *A CEFSM is a 5-tuple* (S, s_0, E, f, V) *where*

- S *is a set of states*
- s_0 *is an initial state*
- E *is a set of events with their parameter lists*
- f *is a state transition relation*
- V *is a set of local variables along with their types and initial values, if any.*

For a state, an input event, and a predicate composed of a subset of V*, the state transition relation* f *has a next state, a set of output events and their parameters, and an action list describing how the local variables are updated.*

As an example, see the following CEFSM:

$$
\begin{aligned}
&CEFSM = (\{S_1, S_2, S_3, S_4\}, S_1, \{init, e_1(p_1), e_2, o_1, o_2(p_2, p_3)\}, f, \{x\}),\\
&\quad \textit{where} \quad f \textit{ has the four elements}\\
&\qquad \{< S_1, init, S_2, (x := 0), \{o_1\} >,\\
&\qquad\ < S_2, e_1(p_1), S_3, (x := x + p_1), \{o_2(x, p_1)\} >,\\
&\qquad\ < S_2, e_2, S_4, (\text{“}\textit{encoding } e_2\text{”}), \{\} >,\\
&\qquad\ < S_3, e_2[x == 8], S_4, (-), \{\} >\}
\end{aligned}
$$

This CEFSM has four states, five events, and one integer variable. The input event e_1 has a parameter p_1, and the output event o_2 has two parameters, p_2 and p_3. In addition, there are four transitions among the states which are represented by the relation f. For simplicity, we do not give the types of variables and parameters.

As an internal event, we assume an *init* event to indicate the start-up signal of the CEFSM. The tuple element $< S_1, init, S_2, (x := 0), \{o_1\} >$ of relation f denotes a transition that moves from S_1 to S_2 while assigning zero to the variable x and generating the event o_1 after the initial signal *init*. The action list can include the brief activities during the transition in plain English as well as a variable update. For example, "encoding e_2" implies that the machine will encode the e_2 received. The actions may be refined in the later develop phases.

As a precondition of a transition, we introduce *predicates*, boolean valued expressions of the local variables [6]. Upon receiving an event, the CEFSM evaluates a predicate. If the predicate holds, the CEFSM executes the transition. However, if the predicate does not hold, the CEFSM ignores the event and stays at the current state. An empty predicate is defined to be shorthand for *true*. In the previous example, the transition from S_3 to S_4 has a predicate $x == 8$.

In this pattern language, we usually represent a CEFSM with a state transition diagram (STD), a directed graph whose vertices correspond to states and whose edges correspond to transitions. Fig. 13 shows an STD of the previous example. Each state is represented by a circle, and the initial state has a double circle. Each transition is labeled with an event, action list, and output events. It is denoted by *event(parameters)[predicate]/actions/outputs(parameters)*. For every transition, the event field is mandatory while predicate, actions and outputs are optional. The '–' symbol in a transition indicates that there is no corresponding field. Transitions that do not alter the state are represented by an arc that points to itself.

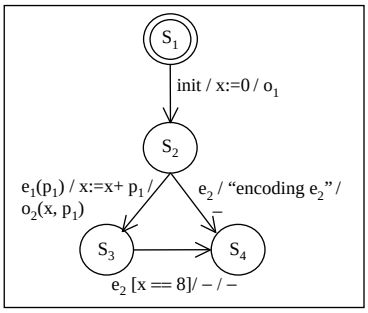

Fig. 13. STD of the example CEFSM

Typically, a complex communication system is designed with a large number of states and transitions. A CEFSM can be expanded by merging with other CEFSMs.

Definition 2 (Merge of CEFSMs) *Let $M_1 = (S_1, s_1, E_1, f_1, O_1, V_1)$ and $M_2 = (S_2, s_2, E_2, f_2, O_2, V_2)$ be two CEFSMs. $M_1 \oplus M_2$, merge of the two CEFSMs, creates a new CEFSM (S, s_0, E, f, O, V) such that*

- $S = S_1 \cup S_2$. *S is a union of* S_1 *and* S_2.
- $s_0 = s_1$, *the initial state of* M_1.
- $E = E_1 \cup E_2$
- $f = f_1 \cup f_2$
- $O = O_1 \cup O_2$
- $V = V_1 \cup V_2$

The merge is formed by using the union operation between sets.

Pattern: Basic CEFSM This is useful in the pattern language and is composed of a source state and a target state. The transition relation f has only one element such as

$$basic\ CEFSM = \{< S_s, e, S_t, A, O >\}$$

Note that the source state S_s and target state S_t could be the same state. Fig. 14 shows the STD of a *basic CEFSM*.

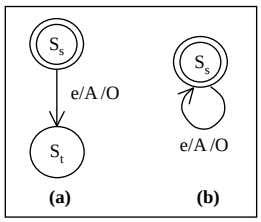

Fig. 14. STD of pattern *basic CEFSM*

Variant: Predicate CEFSM As we mentioned at the definition *CEFSM*, a CEFSM can have predicates to control the behavior of the CEFSM [6]. By adding a predicate to the pattern *basic CEFSM*, we can get the variant *predicate CEFSM* simply. However, an event usually has several predicates as a decision point. We, therefore, define the pattern having several predicates.

$$\begin{aligned} \textit{predicate CEFSM} = \{ & < S_s, e[predicate_1], S_{t1}, A_1, O_1 > \\ & < S_s, e[predicate_2], S_{t2}, A_2, O_2 > \\ & \ldots \\ & < S_s, e[predicate_n], S_{tn}, A_n, O_n > \} \end{aligned}$$

Note that the source state S_s and target states could be the same state. If the predicates are mutually exclusive, then the CEFSM is deterministic. Fig. 15 shows the STD of a *predicate CEFSM*.

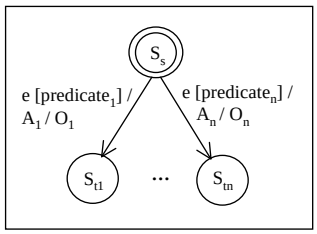

Fig. 15. STD of pattern *predicate CEFSM*

Variant: Predicate after Action At the pattern *predicate CEFSM*, an event is followed by predicates to decide the next transition. However, in some cases decisions need to be made after performing some actions. In other words, after performing a sequence of actions for an events, an instance decides its next transition based on the result of the actions. The instance therefore needs predicates after the actions.

$$
\begin{aligned}
\textit{predicate after action} = \{ & < S_s, e, S'_s, A_s, O_s > \\
& < S'_s, -[predicate_1], S_{t1}, A_1, O_1 > \\
& < S'_s, -[predicate_2], S_{t2}, A_2, O_2 > \\
& \ldots \\
& < S'_s, -[predicate_n], S_{tn}, A_n, O_n > \}
\end{aligned}
$$

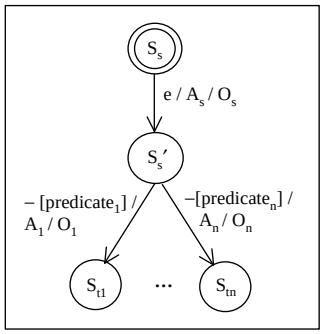

Fig. 16. STD of pattern *predicate after action*

Note that the transitions from S'_1 do not have event fields. At previous definition, we said that the event is a mandatory for a transition. This is exception of the rule. Fig. 16 shows an STD for this type of CEFSM. In fact, this pattern is a sequential merge between the transition from S_s to to S'_s and the *predicate CEFSM*. Refer to the pattern *sequential merge*.

Variant: Sequential Merge Typically, a complex communication protocol is made by composing several CEFSMs to fulfill the required functionality. There are three common types of merge: *sequential merge*, *source merge*, and *target merge*. To introduce these patterns, we classifies a state in a CEFSM into either a *terminal state* or a *nonterminal state*. A state is called a terminal state if it is not used as a source state in a transition of the CEFSM. Otherwise, it is a nonterminal state.

A *sequential merge* is a merging of a terminal state of a CEFSM with a nonterminal state of another CEFSM. Fig. 17 shows the merging of state S_2.

The combined CEFSM goes to state S_2 through the state S_1. This situation is common when a system handles a sequential inputs from environment.

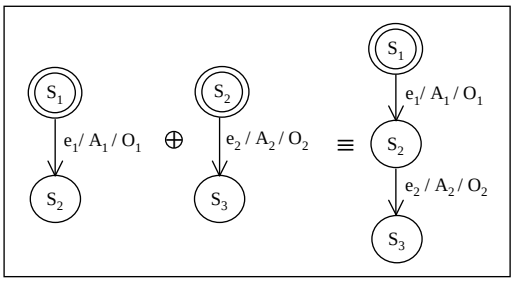

Fig. 17. Sequential merge of two CEFSMs

Variant: Source Merge Two CEFSMs can be merged by combining each nonterminal state, which is called *source merge*. In Fig. 18, the initial state, S_1, is combined, and the resulting CEFSM has three states and two transitions. This commonly occurs in a state receiving several potential input events.

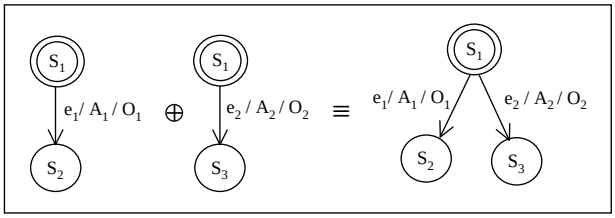

Fig. 18. Source merge of two CEFSMs

Variant: Target Merge The pattern *target merge* is obtained by combining each terminal state of a CEFSM. This is usual when two transitions want to stay at the same state after each transition. Fig. 19 shows a typical example.

Implementation in SDL The *basic CEFSM* is implemented in SDL by a mechanical one-to-one mapping from its STD. Fig. 20 shows an SDL diagram fragment for the basic CEFSM of Fig. 14 (a). Each state of the CEFSM is converted to the corresponding SDL state. A transition is represented by an

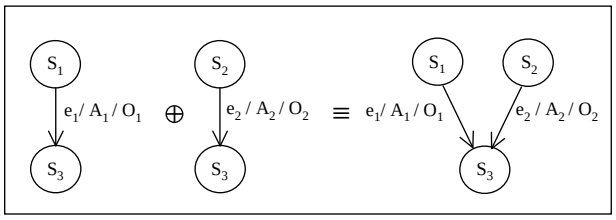

Fig. 19. Target merge of a CEFSM

input signal, a task, and an output signal of SDL. When an instance of the pattern receives an input event e with its parameters in the state S_s, it performs the task A and generates output signal O. Note that if the transition expresses an initialization with the *init* internal event, the source state has to be translated to a start symbol. The internal event is not shown at the SDL implementation.

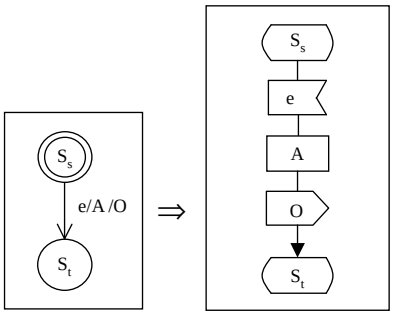

Fig. 20. SDL fragment for pattern *basic CEFSM*

The implementation of pattern *predicate CEFSM* leads to an SDL decision symbol. Fig. 21 shows an SDL diagram fragment for Fig. 15 where there are n predicates for an event e. Upon receiving an event e, the CEFSM checks the predicates and then performs actions for a true predicate. The pattern *predicate after action* are similarly implemented as Fig. 21 with a decision symbol after the actions. It it important to note that the state S'_s of Fig. 16 is not shown in the SDL diagram.

The implementation of merge patterns can be obtained by straightforward mapping of the resulting STDs. We omit the SDL implementation of the patterns.

Examples Fig. 22 shows an error detection method using a checksum procedure *checksum()* in which if the result of the procedure, *rst*, has zero, it means there

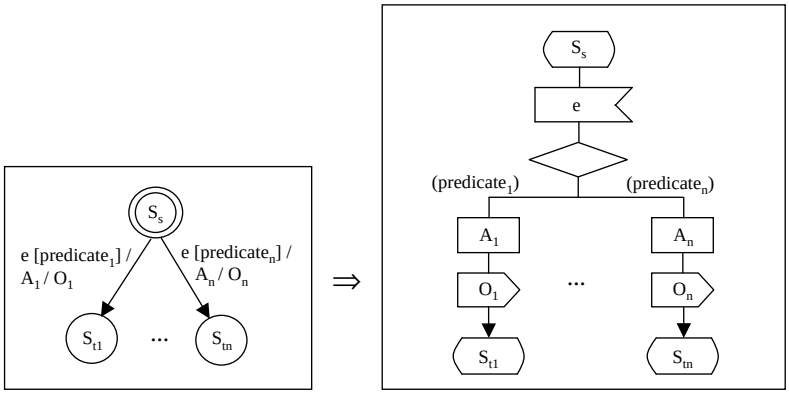

Fig. 21. SDL fragment for pattern *predicate CEFSM*

is no error in the received message. Otherwise, the message has an error, and an error notification message *nok* is sent. Note that the predicates *rst == 0* and *rst != 0* are evaluated based on the previous action *checksum()*.

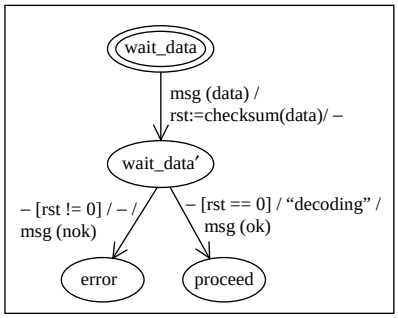

Fig. 22. Example of *predicate after action* for error detection

Suppose we are designing a system that uses connection oriented communication with other systems. The system handles connection establishment and release requests to setup a connection with a peer system and to release the connection. Fig. 23 (a) shows the connection setup scenario in *basic CEFSM*. After receiving a connection request *EST.req*, the CEFSM performs action *"connect"*. Then, it notifies the peer that the connection is set up by using the message *EST.conf*. Similarly, the disconnection step is also achieved in the pattern *basic CEFSM* as Fig. 23 (b).

In fact, it is possible to handle the two requests at one state. Note that the CEFSMs for connection setup and release have different state names. Before merging the CEFSMs, we rename both *wait_establish* and *wait_release* to *wait_msg*. The merged CEFSM is described in Fig. 23 (c), and it is an example of pattern *source merge*.

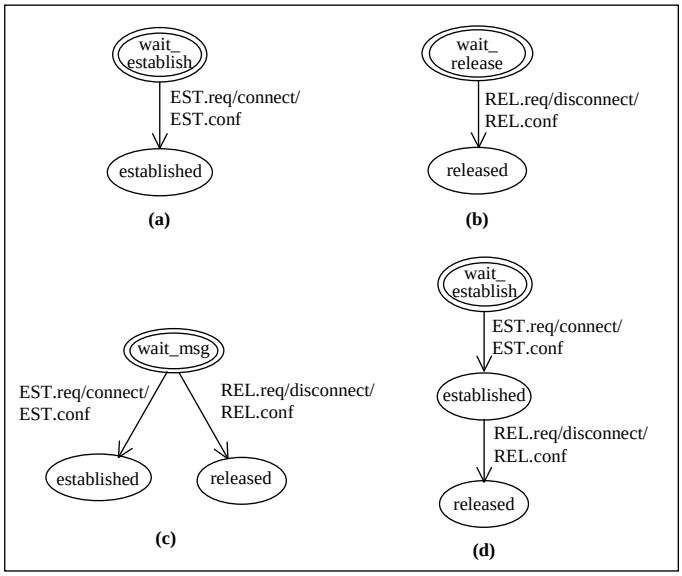

Fig. 23. Example of merge patterns

On the other hand, a connection can be disconnected only after the connection is setup. Fig. 23 (d) show that the connection CEFSM is sequentially merged with the disconnection CEFSM. The state *wait_release* was renamed to *established* before the merging. The state *released* can be reached through the state *established*.

See also To describe the CEFSM, we use STD with transitions labeling with an event, predicates, actions, and outputs where the event is mandatory and other fields are optional. The similar notation is used at Statecharts [10], a visual formalism for the specification of reactive systems, where a transition is labeled with an event, a condition, and an action: all of them are optional. Statecharts are an extension of STD to enhance the descriptive power using hierarchy of states, orthogonality, and history connectors.

3.2 Timer

Context In an event-driven system such as a communications system, an event may occur later than it is expected, or not happen at all because of transmission delay, lost message, etc. Many event-driven systems employ timing constraints where some action is taken if an expected event does not occur in a given amount of time. We are modeling the system with CEFSMs and SDL.

Problem How can we model the timing constraints to avoid unbounded waiting for an event?

Solution A CEFSM can be supplemented with a timer and timer-related operations to manipulate the timing constraints [6].

A timer T is an element of variable set V with an associated timeout value. The unit of the timeout depends on the context of the application. There are two timer operations, *set* and *reset*, which are the elements of action list A. $set(t, T)$ associates a timing value t with a timer T, and $reset(T)$ cancels the timer T. During a transition, a CEFSM can set a timer with a time value. Unless the timer is cancelled by the CEFSM, the timer will generate a timer expiration signal when the time duration is passed.

Generally, a CEFSM handles the timer expiration by either sending an error notification or requesting a resubmission of the event. When an event wanted by an CEFSM occurs before the timer expiration, the CEFSM cancels the timer and proceeds normally. Note that all these time concepts come from SDL [6].

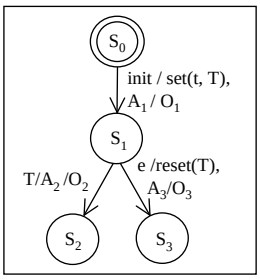

Fig. 24. A pattern *timer* with an expected event e

Fig. 24 shows an STD of the pattern *timer*. First, a timer T should be set before using it. In the diagram, the transition happens upon the internal event *init*. On timeout for the timer T, the CEFSM moves to the state S_2. If the CEFSM receives the expected event e, it resets the timer and performs the remaining transition.

Implementation in SDL A timer variable in the CEFSM maps to an SDL Timer object. The Timer object must be declared like any other variable in SDL. In addition, the timeout value is given as part of the declaration. The operations on the Timer object are set(Timer) and reset(Timer). Since the timeout value is given in the declaration, it is not specified in the set operation. Figure 25 shows the implementation of the typical timer pattern discussed above in SDL.

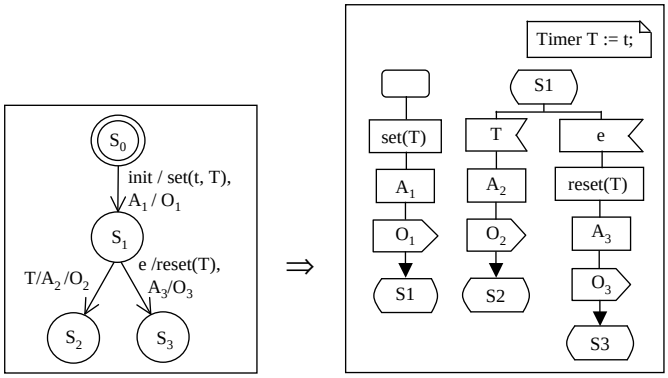

Fig. 25. SDL implementation of pattern *timer*

3.3 Repeated Events

Context In an event-driven system, a single event may not be sufficient to initiate a reaction: an event must occur several times. For example, if a message is transmitted in several packets, all packets containing message fragments must arrive before the message is encoded and handled further.

Problem How does a communication system handle repeated events?

Forces

- *Timing constraints*: It is necessary to impose timing constraints for the repeated events. There are two kinds of timing constraints: a timing constraint for each event, and a timing constraint for all events.
- *Number of repetitions*: In some cases, the number of repetitions is known in advance. In other cases, it is not. Even when the number is not known in advance, there still may be an upper bound on the number of repetitions.

Solution This pattern is a specialization of the pattern *predicate CEFSM*. First, the CEFSM has an integer variable c, which is initialized to one, to count the number of occurrences of an event. Predicates $c < N$ and $c == N$, where N is the number of times the event should be repeated, determine the next transition. If $c < N$ then c is incremented and the state unchanged. Otherwise, the state changes and any specified output events are generated. Fig. 26 shows the solution and its SDL implementation.

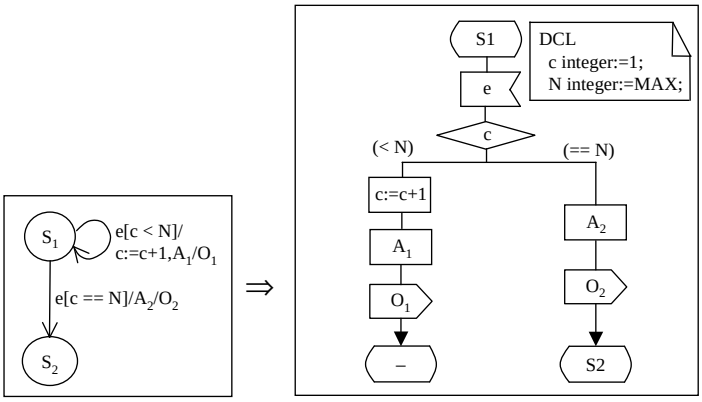

Fig. 26. STD of pattern *repeated events* and its SDL implementation

Variant: Timed Repeated Events In this variant, we add timers to enforce timing constraints. This pattern can thus be considered a combination of the *repeated events* and *timers* patterns. Two timers are used. Timer T_1 is used for the individual occurrences of event e, while T_2 is used for all of the events together.

Fig. 27 shows the resulting CEFSM and its SDL implementation. At the state S_1, the CEFSM receives the event e until it has all events needed. If the number of events is less than N, the machine increases the c and sets the timer T_1 again. If the timer T_1 or T_2 expires, the machine moves to a state to handle this exceptional case. Note that either timer could be omitted. Then only the total time or single time would be checked.

Variant: Timed Repeated Trials In the previous patterns, the number of repetitions of an event before the system moved to a different state was known in advance. This variant considers the case where the number of repetitions is not known in advance (and may be 0), but the maximum number is bounded.

A typical scenario would be where the repeated event is the expiration of a timer set. For example, a message is transmitted and the timer is set. If an

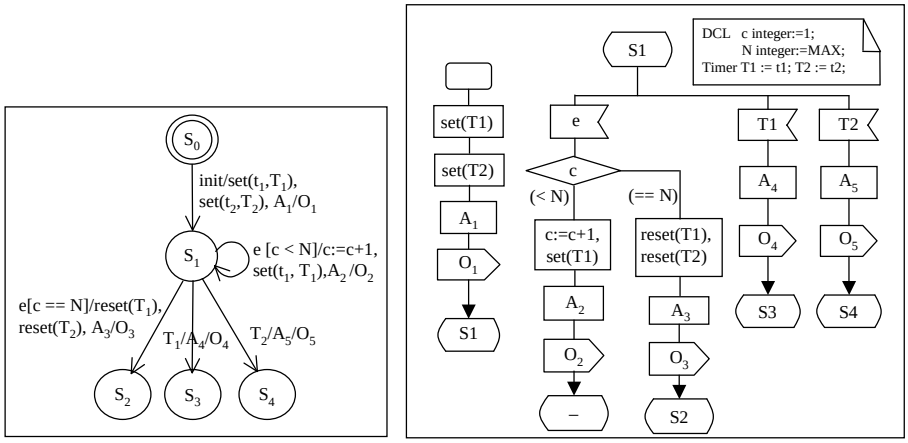

Fig. 27. STD and SDL implementation of pattern *timed repeated events*

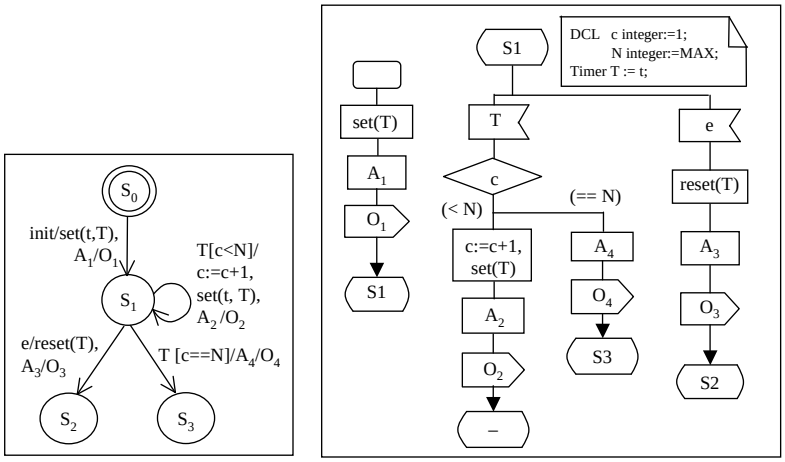

Fig. 28. STD and SDL implementation of pattern *timed repeated trials*

acknowledgment is not received before the timer expires, the message is retransmitted. The message will be transmitted a maximum of N times before the system moves to an error state.

Fig. 28 shows the solution where the repeated event is the expiration of the timer.

Example Fig. 29 describes a part of a call setup software with an STD and its SDL implementation. The software reads a telephone number of nine digits from a calling party. The CEFSM initializes its local variable *c* to one and generates *dialing_tone* to initiate a dialing tone. In the state *dialing*, it receives an event *dial* with a parameter *digit*, which means that the digit is pushed by a caller. The machine receives eight digits at the state. After receiving the ninth digit, it moves to the next state *connecting*. Moving to the state, the machine generates an output *c_req* to request a call connection with the callee.

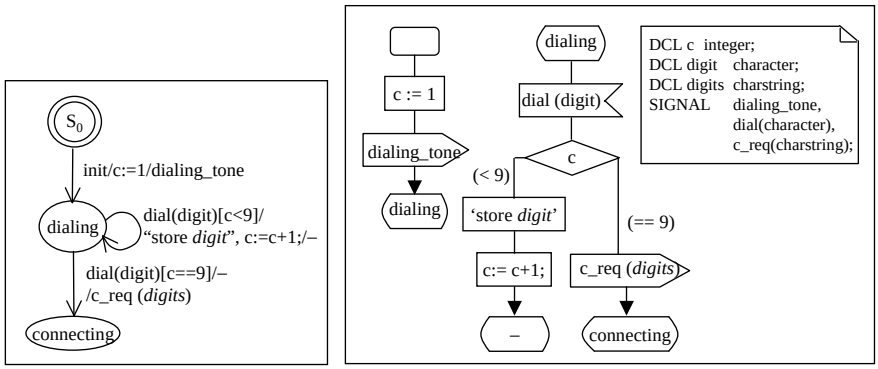

Fig. 29. Nine-digit dialing using *repeated events*

The example can be expanded with a timer T to avoid unlimited waiting. If a caller does not push a digit in a given time bound, the machine notifies the caller that time is over by giving a special beep. This case is implemented by adding a transition for the timer expiration.

See also Pattern *TimerControlledRepeat* [8] repeats a message transmission to avoid message loss during data transfer. If a sender entity does not receive an expected acknowledgment in the given expiration time from a receiver entity, the message is repeated by the sender. This pattern is considered as an instantiation of the pattern *timed repeated trials*.

3.4 Message Transfer

Context In a communication network with a layered protocol, two peers want to exchange messages.

Problem How do two peers communicate in a layered protocol?

Solution Conceptually, a layer provides a set of services to its users. Thus, the users consider the lower layer as a service provider [2, 17]. They communicate with the service provider through service access point (SAP). The service provider coordinates and manages communications between users by using four types of primitives, *request*, *indication*, *response*, and *confirm* as shown in Fig. 30.

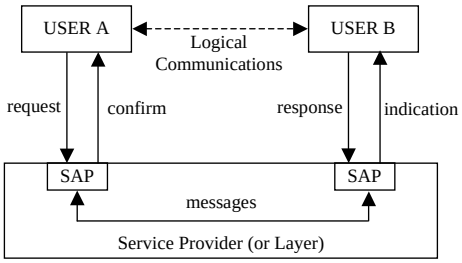

Fig. 30. Protocol layer as a service provider

A user *A* sends a request message to a peer by invoking the primitive *request* to a lower layer. The peer of the requester is informed by the lower layer using the primitive *indication*. The peer responds to the indication by invoking the primitive *response* to the lower layer. The lower layer notifies the original requester of the reply from the peer using the primitive *confirm*.

In general, there are two kinds of communication between peers : *confirmed transfer* and *unconfirmed transfer*. If a requester needs an acknowledgment from a peer, it uses a confirmed transfer with the four primitives. But if this is not the case, the requester sends a message without expecting an acknowledgment. For example, a connection setup phase always uses a confirmed transfer because a peer must agree to establish a connection with its requester. A data transfer phase, on the other hand, uses either confirmed or unconfirmed transfer depending on the protocol [17].

Variant: Simple Sender/Receiver Two primitives *request* and *indication* are used for simple but unconfirmed communication. Depending on the protocol, the names of the primitives might be different. Fig. 31 shows two CEFSMs for an

unconfirmed message transfer: one is to send a message in *request* form and another is to receive a message in *indication* form. It is important to consider the two CEFSMs together because they are used at the same time. As the figure indicates, the two CEFSMs are the specialization of the pattern *basic CEFSM*. The *sender* has a transition with a primitive *request* after performing an action A_1. The CEFSM may have other output messages such as O_1. For the primitive *request*, the *receiver* is notified by a corresponding event *indication*.

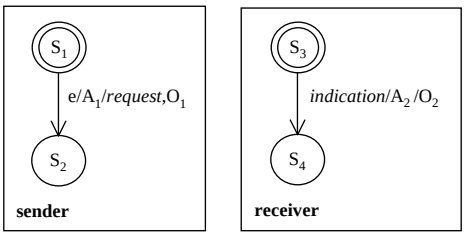

Fig. 31. Pattern for unconfirmed simple message transfer

Variant: Confirmed Sender/Receiver For a confirmed message transfer, two peers use four primitives, *request*, *indication*, *response*, and *confirm*. As Fig. 32 shows, the *requester* asks a confirmed message transfer with a primitive *request*, and then waits for a confirmation from its peer *replier*. When the *replier* knows that there is a request from a peer through an event *indication*, it performs a_3 for the event and replies with the primitive *response*. The CEFSM *requester* is a specialization of the pattern *sequential merge* of the two *basic CEFSM*s.

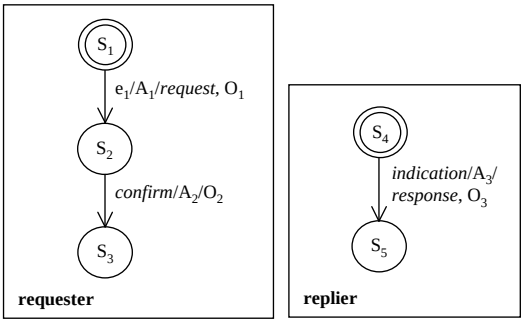

Fig. 32. Pattern for confirmed message transfer

It is not always necessary to have the four primitives for confirmed transfer. For example, in locally confirmed transfer, the primitive *confirm* is provided by the lower layer, not by a peer [2]. Thus we do not need the primitive *response* from the *replier*.

Based on the patterns *simple sender*, *simple receiver*, *confirmed sender*, and *confirmed receiver*, it is possible to extend the patterns by combining them with the previous patterns *predicate CEFSM*, *timer*, *repeated events*, etc. For example, in some situations the pattern *confirmed sender* might wait for the primitive *confirm* for a longer time than it expected or indefinitely because of message loss, transmission delay, heavy traffic, etc. To restrict the waiting time for the primitive *confirm*, the pattern *confirmed sender* can be improved with the pattern *timer*. The resulting pattern *timed confirmed sender* shown in Fig. 33 adds timing constraints to the pattern *confirmed sender*. Similarly, we can make new patterns such as *repeated sender*, *repeated receiver*, *repeated confirmed sender*, *repeated confirmed receiver*, *timed repeated trial receiver*, *timed repeated trial confirmed sender*, and *timed repeated trial confirmed receiver* from the combination of patterns.

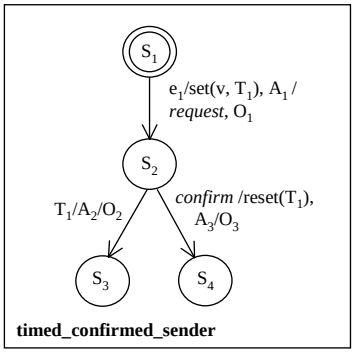

Fig. 33. STD of pattern *timed confirmed sender*

Variant: Message Transfer in Middle Layer The previous message transfer patterns such as *simple sender*, *simple receiver*, *confirmed sender*, etc. describe the interface between a layer and its lower layer. In a layered protocol, a layer may also be a service provider of the next upper layer. In other words, if a layer is between two layers, it must provide a set of services to the upper layer. From the middle layer's point of view, the services of the layer are initiated from the upper layer entity, and they are achieved by using the services of the lower layer. The pattern *message transfer in middle layer* presents the message transfer for the two interfaces. Fig. 34 shows one case of the pattern where the upper interface uses the pattern *simple sender/receiver*, while the lower interface

uses the pattern *confirmed sender/receiver*. It is important to note that the figure is only one instance of several potential combinations between upper and lower interfaces.

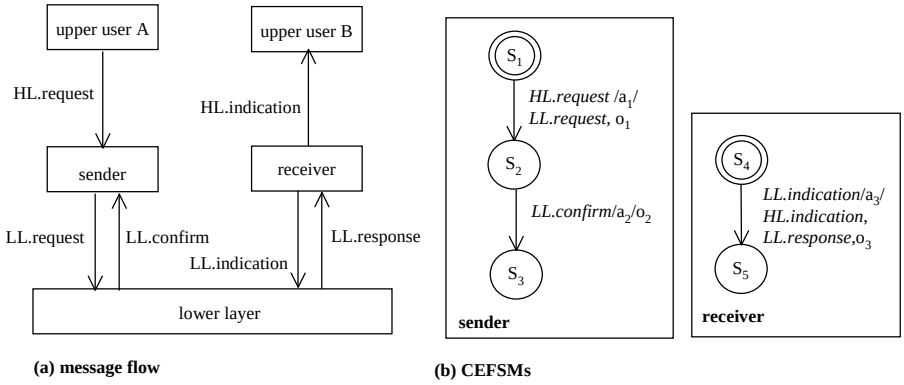

Fig. 34. An instance of pattern *message transfer in middle layer*

Implementation in SDL The patterns presented in this subsection are the combination of the *basic CEFSM*, *predicate CEFSM*, *timer*, and *repeated events*. We can get the SDL implementation of each pattern from these patterns.

Example In this example, we present a connection setup phase for Signaling ATM Adaptation Layer (SAAL). The SSCF for the UNI shown in Fig. 4 is almost null layer and provides a simple mapping between SSCOP and UNI signaling layer 3. Fig. 35 (a) shows a signal flow for an SAAL connection establishment [3].

On the side where the SSCF user requests the establishment of the connection, the *AAL_EST.req* is directly mapped to an *AA_EST.req*. The confirmation from SSCOP is mapped accordingly. On the incoming side, the SSCF answers directly with an *AA_EST.resp* to SSCOP for the *AA_EST.ind*. Note that the user receives only an indication. The connection setup can be achieved by the pattern *message transfer in middle layer* where both the upper interface and lower interface use the pattern *confirmed sender/receiver*. Fig. 35 (b) shows the STDs of the CEFSMs.

See also *SimpleSend*, *BlockingRequestReply*, *TimerControlledRepeat* [8]

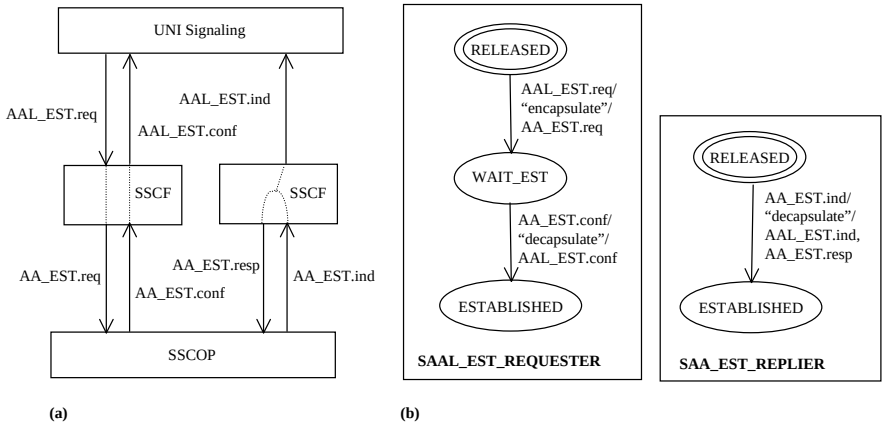

Fig. 35. SAAL connection setup using *message transfer in middle layer*

4 Conclusion

A pattern is powerful artifact for design reuse. It describes a solution to a problem that recurs in the design and implementation of a software system. It helps to document an experience, capture well-proven design discipline, and promote design practice. The solution is able to provide higher-level abstractions to be used several times in similar situations.

The main contribution of this paper is to present a pattern language for the development of communication protocols. In addition, we used CEFSM and its STD to describe the behavior of patterns and presented the SDL implementation of each pattern. We expect that the implementation in a specific language will be helpful for direct usage in applications written in the language.

The pattern language is composed of structural and behavioral patterns. The structural patterns describe the common architectural aspects of communication systems. The structures are represented by communicating bodies (boxes) and communication paths (lines). It also shows the dynamic creation of the communicating instances. The behavioral patterns describe the recurring behavior of a protocol system. The CEFSM represented in STD is easy to understand because it clearly describes states, events, and actions in a diagram.

The patterns are a set of building blocks. They can be stored in a pattern repository so that a new system could be composed with them as [9]. Traditional software development has several phases. When a system engineer develops a system, the pattern repository can be used for the higher-level description of the system. The engineer can devise an architecture of the system with structural patterns, and then the execution of the system can be obtained with the behavioral patterns. It is easy to get an SDL skeleton code from the pattern-based system. The designer can refine the design from the SDL skeleton.

As a further research, the language could be supplemented with more patterns. The patterns presented in the pattern language are not complete set. We can extend the language by either composing the patterns or specializing the existing ones as well as developing new ones. However, we believe that the categories will be still useful for further patterns.

Acknowledgments

We would like to thank two shepherds who have provided many helpful and insightful comments on the patterns described in this paper: Luigi Guadagno, our shepherd for this paper, and Michael Wu who shepherded a previous paper where earlier versions of some of these patterns were described. Chang-Sup Keum participated in the development of the patterns in the earlier paper. We also thank Electronics and Telecommunications Research Institute (ETRI) for their financial support.

References

1. R. Arthaud. SDL and layered systems: Proposed extensions to SDL to better support the design of layered systems. In R. Reed and J. Reed, editors, *10th International SDL Forum (SDL 2001)*, number 2078 in Lecture Notes in Computer Science. Springer-Verlag, 2001.
2. U. Black. *OSI: A Model for Computer Communications Standards*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1991.
3. H. Brandt and C. Hapke. *ATM Signalling: Protocols and Practice*. John Wiley & Sons, 2001.
4. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, New York, 1996.
5. F. Dietrich and J.-P. Hubaux. Formal methods for communication services. Technical Report SSC/1999/023, Institute for Computer Communications and Applications, Swiss Federal Institute of Technology, CH-1015 Lausanne, 1999.
6. J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL: Formal Object-oriented Language for Communicating Systems*. Prentice Hall, 1997.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
8. B. Geppert and F. Rößler. Pattern-based configuring of a customized resource reservation protocol with sdl. Technical Report 19/96, Computer Science Department, University of Kaiserslautern, Germany, 1996.
9. B. Geppert and F. Rößler. The SDL pattern approach – a reuse-driven sdl design methodology. *Computer Networks*, 35(6):627–645, 2001.
10. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
11. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
12. J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

13. H. Huni, R. E. Johnson, and R. Engel. A framework for network protocol software. In *Proceedings OOPSLA '95, ACM SIGPLAN Notices*, 1995.
14. W. Lynch. Reliable full-duplex file transmission over half-duplex telephone lines. *Communications of the ACM*, 11(6):407–410, 1969.
15. G. Meszaros. Pattern: Half-object + protocol. In D. Schmidt J. Coplien, editor, *Pattern Languages of Program Design*. Addison-Wesley, Reading, Massachusetts, 1994.
16. ITU-T Recommendation Q.2130. *B-ISDN ATM Adaptation Layer - Service Specific Coordination Function for Support of Signaling at the User-Network Interface (SSCF at UNI)*. International Telecommunication Union (ITU), July 1994.
17. A. S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, 1996.