A Pattern-based Development Methodology for Communication Protocols

YOUNGJOON BYUN AND BEVERLY A. SANDERS*
School of Engineering and Engineering Technology
The Pennsylvania State University at Erie
Erie, PA 16563, U.S.A.
**Department of Computer and Information Science and Engineering*
University of Florida
Gainesville, FL 32611, U.S.A.

Patterns help to improve software quality and reduce development cost by documenting the experience of experts so that good solutions to recurring problems can be reused. In this paper, we propose a pattern-based software development methodology for communication protocols, particularly focusing on the specification and validation of message interactions. For the description of communication protocols, we propose a set of patterns. A complex protocol can be obtained by composing such patterns. To provide confidence in the protocol description, we validate the pattern-based specification by using the SPIN model checker. The validation phase needs model construction for the specification and checks the desired properties of the developing protocol. To show the feasibility of our methodology, we present a case study for the development of a V.76 protocol.

Keywords: development methodology, communication protocol, design pattern, communicating extended finite state machine, SPIN model checker, V.76 protocol

1. INTRODUCTION

When a programmer develops a new software system, he or she often finds many situations similar to those that have occurred in previous developments. A design pattern is a written document providing a solution for a recurring problem in a certain context [1, 2]. A design solution that has worked well in a particular situation can be used again in similar situations in the future. Design patterns, therefore, help to improve software quality and reduce development cost through predefined solutions and their reuse.

Patterns are considered to be useful in many different types of software system developments. Research and usage of patterns in communication systems are increasing as an emerging area in the design pattern community [3]. Much of the work focuses on the structure of communicating entities and the relationship among them. In this paper, our main concern is message interactions among communicating entities because those interactions provide the principle behavior of protocol operation. To describe these interactions as well as the structures of protocols, we present a set of patterns. Then we address how to check the consistency and correctness of the pattern-based specification. Currently, many patterns are concentrated on the design and implementation of software

Received July 1, 2005; accepted November 24, 2005.
Communicated by Sung Shin.

developments with little consideration for other development phases such as requirements analysis and testing. We suggest a validation step for the design specification represented using our patterns. It is cost effective to uncover design errors as early as possible to prevent the errors from affecting later phases.

The remainder of the paper is organized as follows. In section 2, we introduce our methodology. Then, a pattern language for communication protocols is described in section 3. In section 4, we present the validation technique of pattern-based specification which includes model construction and SPIN model checking. To show the feasibility of our methodology, a case study on an ITU-T V.76 protocol is given in section 5. We conclude this paper in section 6 with a summary and further research.

2. A PATTERN-BASED DEVELOPMENT METHODOLOGY

Classic life-cycle of software engineering is composed of several phases such as requirements analysis, design, coding, testing, and maintenance. Our concern lies in the initial design for the high-level description of a protocol and its validation. Designers in that phase typically want to capture the essential functions first; for example, flows of messages and interactions with other systems. Then, they build an abstract model before the detailed design and implementation. For the specification of the abstract protocol, we propose a pattern language, a collection of patterns that work together to solve problems in a specific domain. The pattern language is categorized in two groups, structural patterns and behavioral patterns, to address overall architecture and common behavior of a protocol, respectively. Fig. 1 illustrates the pattern-based development methodology. Patterns are contained in the pattern repository for selection and composition. After analyzing the requirements of a protocol, we devise an architecture of the protocol with structural patterns. The architecture is composed of several communicating blocks along with communication paths between them. A block is an architectural building element of a developing system and can contain other blocks. At this point, blocks are considered to be black boxes. The external interfaces such as communication paths and messages are defined, but the internal details are not. The internal behavior of the abstract protocol is designed using the behavioral patterns after the architectural design. The behavioral patterns provide common interactions between blocks. They assist developers in describing the internal behavior of blocks. Each block instance has a state that may change to another state in response to an input message. The response may also trigger output messages. We use a communicating extended finite state machine (CEFSM) to formally describe the behavior [4]. Predicates and timers may be used to describe conditional behavior and timing constraints. Protocol designers complete the abstract design of a protocol system by composing the structural and behavioral patterns. It may be necessary to revise the requirements during the design step to fix any unclear requirements. As a result of the design, designers have a *system design description* document which sketches an abstract system of a protocol focusing on message interactions. The description is a combination of instantiated blocks, communication paths, messages, and finite state machines of patterns used.

From the pattern-based high-level design, we perform a validation to check the correctness and consistency of the design. In this methodology, we suggest a model

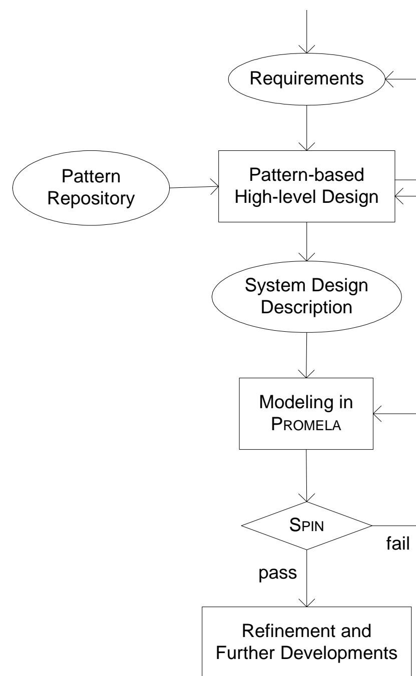

Fig. 1. Pattern-based development methodology for communication protocols.

checking technique which is an automatic technique to verify properties of a system by investigating a model of the system [5]. We selected SPIN (Simple Promela INterpreter) as our model checking tool [6]. It was developed at Bell Labs for the analysis and validation of distributed systems, especially of communication protocols. Furthermore, it is freely available from the SPIN web site, http://spinroot.com/spin/. For the SPIN model checking, a developer first builds a model of a system in PROMELA, an input language of SPIN, and identifies properties to be checked. Then, the model is simulated and verified against the properties on SPIN. Typically the construction of a model is a challenging practice in formal validation because it is crucial to the validation result and must reflect the system to be developed exactly. We provide a translation mechanism to build a model from a pattern-based specification. Because of the correspondence between pattern elements and PROMELA constructs, the translation is straightforward. To help the designer identify and correctly express the properties needed for the validation, each pattern of our pattern language provides a property specification section. These properties include the occurrence of message arrival, ordering in message interaction, and correspondence or alternativeness of messages.

3. A PATTERN LANGUAGE FOR COMMUNICATION PROTOCOLS

Patterns are rarely stand-alone. Instead, several patterns are typically related to solve problems in a specific domain. Table 1 presents our pattern language for the description of communication protocols.

Table 1. Pattern language for communication protocols.

Category	Patterns	Variants
structural	protocol layer	split protocol layer
	mux	
	dynamic handler	split dynamic handler
behavioral	basic CEFSM	predicate CEFSM
		predicate after action
		merge patterns
	timer	
	repeated events	timed repeated events
		timed retrial
	message transfer	unconfirmed sender
		unconfirmed receiver
		confirmed sender
		confirmed receiver
		timed unconfirmed receiver
		timed confirmed receiver
		timed confirmed sender
		timed retrial confirmed sender
		periodic unconfirmed sender
		periodic confirmed sender

Detailed description of each pattern is given in Byun *et al.* [7]. In this section, we present a pattern called *timed retrial confirmed sender* as an example of our pattern.

3.1 Timed Retrial Confirmed Sender

3.1.1 Context

In a communication protocol, a communicating block wants to transfer messages to its peer block at the same layer through their lower layer. The lower layer is unreliable, which means messages could be lost at the lower layer.

3.1.2 Problem

How does a communicating block transfer a message reliably to its peer through an unreliable lower layer?

3.1.3 Solution

Reliable transfer can be achieved by receiving an acknowledgement for a message and retransmitting the message if the acknowledgement is not received within a specified

time interval. We also introduce a counter to check the upper limit of repetition. Thus, the problem can be solved by composing the pattern *confirmed sender* with a timer and a counter.

Fig. 2 shows an instance of the pattern where a sending block sets a timer T with a timing value v after it receives an initial event e_1. Then, the block initializes a count c to one and conducts an action A_1. After that, the block requests a message transfer, for example using the message named *request*, to a corresponding peer block and generates any additional output messages O_1 if it is necessary. Then it moves to the next state S_2. There are three possibilities in this state. If the block receives an acknowledge message, for example, named *confirm* from the peer in the time interval set in the timer T, it resets the timer and move to the state S_3. If the timer is expired before any confirmation message, the block checks the counter. If the counter reaches the upper limit of repetition N, it generates a message named *error* and move to the state S_T. Otherwise, it retransmits the request message after increasing the counter. In the Fig. 2, note that the action list and output messages such as A_1, O_1, A_2, etc. are optional fields in a transition. One potential problem of this pattern is that if we use the same timeout value v repeatedly in the message retransmission, there is high possibility of network congestion. To solve this problem, we can use the exponential backoff technique which doubles the timeout value between each successive trial [8].

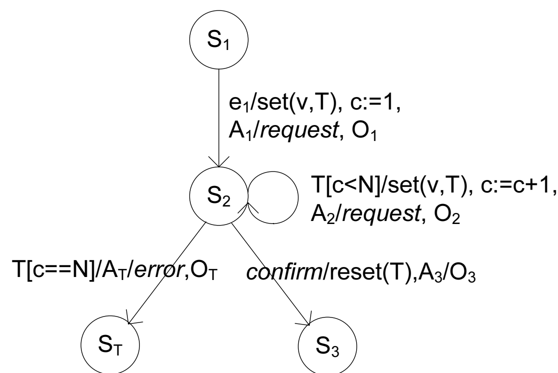

Fig. 2. Pattern timed retrial confirmed sender.

3.1.4 Examples

In the ATM signaling protocol [9], a block sets a timer *T303* for four seconds while waiting for a connection establishment message *SETUP*. The block tries at most twice for the connection establishment. The BOOTP client also uses this pattern to request its IP address and boot file information [10]. It broadcasts a packet *BOOTREQUEST* and waits for a *BOOTREPLY*. If no reply is received for a certain length of time, the client retransmits the request. The timeout interval is selected randomly and will be increased as errors continue to avoid network congestion.

3.1.5 Property specification

This section identifies the required properties that have to be met if the pattern is used in a system design. For example, the event e_1 has to occur eventually to initiate the pattern, and the message *request* has to follow the event. Moreover, the message *request* is always followed by either a confirmation message *confirm* or *N* times timeout. Total number of timeouts should be at most the upper limit *N*. These properties are represented in linear temporal logic (LTL) as follow:

- Property1: $\Diamond\, e_1$.
- Property2: $\Box\, (e_1 \rightarrow \Diamond$ request).
- Property3: $\Box$ (request $\rightarrow \Diamond\, ((\mathrm{T} \wedge (c == N)) \wedge \neg$ confirm) $\vee\, (\neg(T \wedge (c == N)) \wedge$ confirm)).
- Property4: $\Box\, (1 \leq c \leq N)$.

Temporal logic [11] is a logic for statements and reasoning that involve the notion of order in time. It is a useful formalism to specify properties of reactive and concurrent systems and provides a formal and succinct notation for desired system behavior over time. We use an LTL formula to express a property of a pattern which is used for an abstracted system model. The formula is constructed from propositions to which we apply temporal and Boolean operations [12]. Every proposition p, a statement that has either *true* or *false* Boolean value, is an LTL formula. For example, *DATA was sent by a client* is a *true* proposition if a client sent the message *DATA* sometime before. Otherwise, it has *false* value. If the propositions p and q are temporal formulae, then so are $\neg p$, $p \vee p$, $p \wedge p$, $p \rightarrow q$ which represent logical *negation*, *disjunction*, *conjunction*, and *implication*, respectively. In addition to the Boolean operators, an LTL formula contains temporal operators such as $\Box$ (*always*) and $\Diamond$ (*eventually*). For example,

$\Box$ (DATA $\rightarrow \Diamond$ (ACK)),

means that whenever the event *DATA* has occurred, the event *ACK* has to follow it eventually. A model checking tool can automatically validate if a system property represented in an LTL formula is satisfied in a system model.

3.1.6 Implementation

To show the solution in an implementation language, we exploit Specification and Description Language (SDL) that is a formal description language for communicating systems recommended by the International Telecommunication Union (ITU) [13] and is popular in design and implementation of communication protocols [4, 14]. Fig. 3 shows the SDL implementation of the solution presented in Fig. 2. Each state of a CEFSM is converted to the corresponding SDL state. A transition is represented by an input signal, tasks, and output signals of SDL. A timer in the pattern maps to an SDL timer object which is declared as an SDL variable with a timeout value.

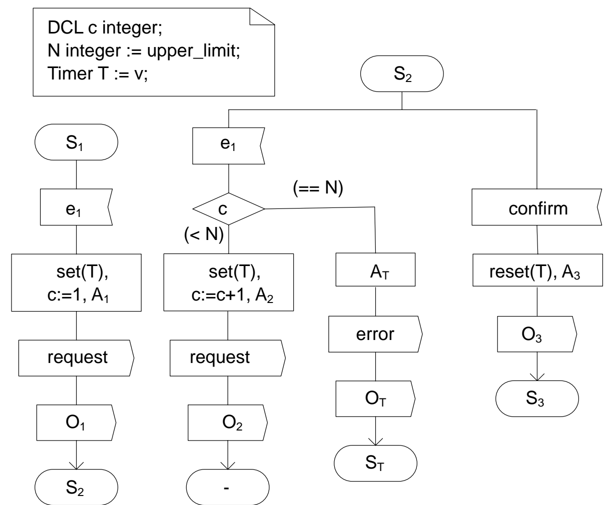

Fig. 3. SDL implementation of the pattern presented in Fig. 2.

3.1.7 See also

Faison emphasized the importance of interaction in software systems with multiple processes [15]. He found fundamental interactions between processes such as *push* and *pull* interactions to describe the way the processes relate to and communicate with each other. Gotzhein [16] suggested a set of SDL patterns such as *SendReceive*, *Blocking-RequestReply*, and *TimerControlledRepeat* to represent interactions between communicating entities. To describe the pattern, we use the CEFSM with transitions labeling with an input, predicates, actions, and outputs. The similar notation is used at Statecharts [17], a visual formalism for the specification of reactive systems, where a transition is labeled with an event, a condition, and an action.

4. VALIDATION OF PATTERN-BASED DESIGN

The pattern-based design is followed by the validation using the SPIN model checker. The basic idea of model checking is to build an abstract model of a developing system. Then it searches all possible execution states of the model to check the system correctness properties such as absence of deadlocks, non-progress cycles, unreachable code, system invariant violation, temporal properties, etc. Thus, for the SPIN model checking, we have to construct a model of a design specification in PROMELA and check the model against the properties to be satisfied in the final product.

4.1 Model Construction in PROMELA

In this section, we present a model construction mechanism for the pattern-based specification and some issues arising in the model construction. A model that is consistent with the specification is essential because any violation in the model has to exactly

reflect the same fault in the specification. The correspondence between our pattern components and PROMELA constructs makes it simple to build a consistent model of the specification. As mentioned in section 3, our patterns are composed of communicating blocks, communication paths, messages, dynamic creation of block instance, states, transitions, predicates, repetition, action list, state merge, and timers. Most pattern components have direct counterparts in PROMELA. By converting each pattern component into the corresponding PROMELA construct, we can build a model. For example, a communicating block is mapped to a PROMELA process declared in proctype and an instance of the block can be created using the run operator. A communication path between blocks is implemented with a PROMELA channel chan that carries messages and their parameters. All input and output messages of pattern are declared as symbolic constants in mtype. For instance, chan myChan = [8] of {mtype, int, byte} declares a channel named myChan that is able to store up to eight messages consisting of mtype, int, and byte fields. For the message exchange on a PROMELA channel, the send, !, and receive, ?, statements are used. For example, statements myChan!request and myChan?confirm represent the sending of request and receiving of confirm through the channel myChan, respectively. Meanwhile, a state of a pattern is converted to a label in PROMELA. A transition from a state to another state is implemented in the goto control transfer construct to jump to a label corresponding to the target state. A transition may also conduct a set of actions during the transition. In the pattern description, actions are usually composed of assignment commands and arithmetic operations. The C-like PROMELA syntax supports these actions directly. The high-level action described in plain English is commented in the PROMELA model so that the action is to be refined in later development phases. PROMELA provides bit, bool, short, int, unsigned, array, and typedef. They are usually enough for the data declaration in our patterns. A decision point with predicates is converted to the selection construct if and each predicate is converted to a corresponding guard. The repeated event is mapped to the repetition construct do. The source merge is represented using the if construct for the selection of a transition from the merged states. Other merge patterns are implemented with the goto construct.

Until now, the construction was straightforward. However, there are some issues to be considered in the model construction. First, the validation result is typically sensitive to the channel capacity. Communication via a channel is either synchronous (i.e., rendezvous) or asynchronous (i.e., buffered) depending on the channel capacity. When we specify a communication path in the design phase, we did not care about the size of message queue because we assumed that it has an unlimited size. In a PROMELA model, however, a channel can store only a finite number of messages. Furthermore, increasing the channel capacity could increase the state space dramatically. A typical approach we have used regarding the channel capacity is to check both synchronous and asynchronous communication with several buffer sizes. The results are quite different in many cases. We start with the channel size zero for synchronous communication and then increase it gradually. When the validation cannot be performed due to the state space explosion, we use other techniques such as state-vector compression and bit-state hashing [6].

A timer is an important component of our pattern language to generate a timer expiration signal when a time value assigned to the timer has been exceeded. To validate the pattern-based design, it is essential to translate the timer, timer-related operations, and timer expiration in a PROMELA model. The SPIN, however, does not provide any

timing concept. Thus to simulate the timer in our modeling, we use an abstracted timer as introduced in Bosnacki *et al*. [18] where a timer is represented in a Boolean variable initialized to false. The set operation is implemented by setting the timer value to true, while the reset operation inactivates the timer by setting the value to false. The arrival of a timer expiration signal is implemented by checking the current value of the timer variable. If it is true, the timer has been previously set and we assume that the timer signal has arrived at the time the value is checked. This is simple and abstracts the actual concrete value of a timer. Since this approach requires the protocol to work correctly for all possible values of the timer, it is sufficient for testing of the timer.

A model to be validated with SPIN should be a closed system [19]. As a result, we have to provide any missing parts of a whole system as well as the validation model itself. Frequently, the environment is much difficult than the model because it is not well defined and developers may not have enough information for the environment. Furthermore, it is necessary to identify whether an error has occurred in the model or in the environment. In our case, we identify the most common and typical execution scenario between the environment and model to initiate the validation. Then we build the minimum environment to meet the scenario and conduct the validation. After that, we augment or modify the environment, if we found any problem during the validation such as unreachable code in the model.

As an example of a SPIN model, we present a PROMELA code for the pattern presented in Fig. 2. In this model, we assume that all messages between environment and model are exchanged through the channel myChan.

```
#define SIZE   1     /* channel capacity. This value is changeable. */
#define timer bool     /* abstracted timer in Boolean value */
mtype = {e1, request, confirm, error, O1, O2, O3};     /* messages */
chan myChan = [SIZE] of {mtype};     /* channel for message interaction */
proctype timed_retrial_confirmed_sender() {
    timer T = false;     /* timer initialization */
    byte c;     /* counter for repeated attempts */
S1:myChan?e1;     /* reception of input event e1 at the state S1 */
    T = true;     /* timer set */
    c = 1;     /* counter initialized */
    /* action A1 */     /* action list which will be refined later.*/
    myChan!request;     /* sending of request message */
    myChan!O1;     /* sending of O1 message */
    goto S2;     /* transition to state S2 */
S2:if     /* There are three possibilities in the sate S2 */
   :: myChan?confirm ->     /* reception of confirm message */
      T = false;     /* timer reset */
      /* action A3 */     /* action list */
      myChan!O3;
      goto S3;
   :: T ->     /* timer expiration */
      if     /* Check if it reaches the maximum retrial limit N */
      :: (c < N) ->     /* not maximum attempts yet */
```

```
            T = true;    /* timer set again */
            c = c + 1;    /* increase the counter */
            /* action A2 */
            myChan!request;
            myChan!O2;
            goto S2    /* stay at the state S2 */
        :: (c == N) ->    /* already tried the maximum attempts */
            /* action AT */
            myChan!error;
            my_chan!OT;
            goto ST;    /* move to the state ST */
        fi;
    fi;
S3: skip;    /* No further behavior specified in this state */
ST: skip;    /* No further behavior specified in this state */
}
```

4.2 SPIN Model Checking

From the constructed PROMELA model, SPIN is able to check obvious design flaws such as system deadlocks and unreachable code. After fixing any problems found in this phase, the model is validated against system requirements. Here, one important issue is to identify the requirements or the properties of the developing system. If such properties are not provided properly, it is not possible to determine whether the system meets the required behavior. Thus the identification of the desired behavior is an important issue in the system validation. As a high-level design, the *system design description* document helps to find system's correctness properties that can be captured from the property specification section of the pattern used. They are particularly useful to identify the existence, response, and correspondence of a message. Using the SPIN, the developer can automatically check whether the model meets the properties. For example, we can check the four properties of section 3.1.5 on our sample PROMELA model.

5. CASE STUDY: V.76 PROTOCOL

The V.76 protocol describes a set of procedures to multiplex multiple streams of information between peer stations [20]. The protocol is mainly composed of a multiplex platform part and a data link connection (DLC) part. In this case study, we focus on the specification and validation of the simplified DLC part to illustrate how our methodology could be applied to the protocol. The DLC part takes charge of setting up a DLC, transfer of user information, and release of the DLC. Doldi provides an excellent description of the protocol in SDL [14].

5.1 Structural Design

A DLC entity has two distinct functions to initiate a user information transfer re-

quested from the service user (SU) and handle the information transfer request coming from the peer through the lower layer. This separation of functionality leads the entity to be designed in the pattern *split protocol layer* as shown in Fig. 4. In the architectural design, the protocol can be specified in two blocks; a connection initiator or outgoing connection (O_DLC) and a connection responder or incoming connection (I_DLC). The pattern also has an upper layer block of SU and a lower layer block of data link layer. Indeed, the pattern allows a protocol developer to consider the two functions separately. It also makes it easy for the developer to identify the communicating blocks of a developing layer and address the communication paths and messages between adjacent layers.

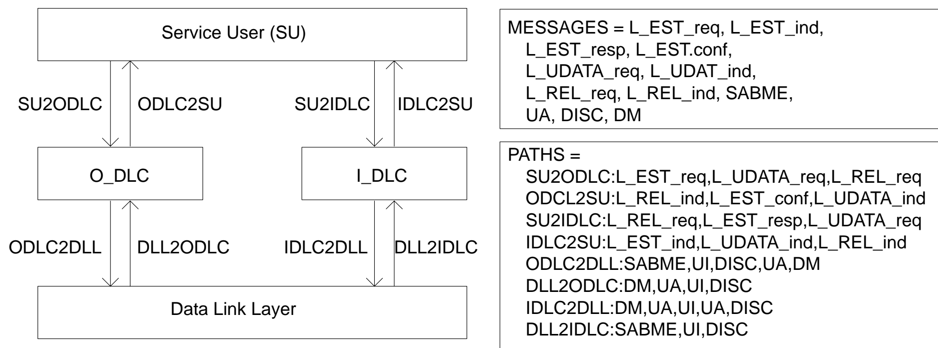

Fig. 4. V.76 protocol architecture for DLC entity using pattern *split protocol layer*.

When the SU tries a new connection, the link establishment request (L_EST_req) primitive comes to the O_DLC block from the upper layer through the path SU2ODLC. Upon receipt the primitive, the O_DLC block tries to attempt to setup a DLC. Frames between O_DLC and I_DLC flow through the communication paths ODLC2DLL, DLL2ODLC, IDLC2DLL, and DLL2IDLC via the lower layer of the protocol. The user level primitives such as L_EST_req, L_EST_ind, L_EST_resp, etc. are exchanged through the communication paths SU2ODLC, ODLC2SU, SU2IDLC, and IDLC2SU. The detailed description of protocol behavior is given in the following section.

5.2 Behavioral Design

5.2.1 Establishment of a DLC

When the O_DLC block receives an L_EST_req from SU, it initiates establishment of a DLC by sending an SABME (Set Asynchronous Balanced Mode Extended) command frame to its peer entity. The block then waits for either a UA (Unnumbered Acknowledgement) or a DM (Disconnect Mode) response depending on the situation of the peer. Before waiting for the response, the block starts the timer T401 to avoid any potential frame loss. It tries at least N400 times which are the maximum number of retransmission defined in the V.76 protocol for the response. The actual values of T401 and N400 depend on the implementation. If it receives a UA as an acknowledgement of a

successful connection, it moves to the connected state. However, if the block fails to receive any response in N400 trials, it understands this situation as a failure of the request and gives up the connection. All these behavior of the O_DLC block can be described using the pattern *timed retrial confirmed sender* as shown in Fig. 5 (a). In the figure, note that we represent the states with the same name as distinct ones for the simplification of the drawing. For example, there are three disconnected states in Fig. 5 (a), but they all must be the same state.

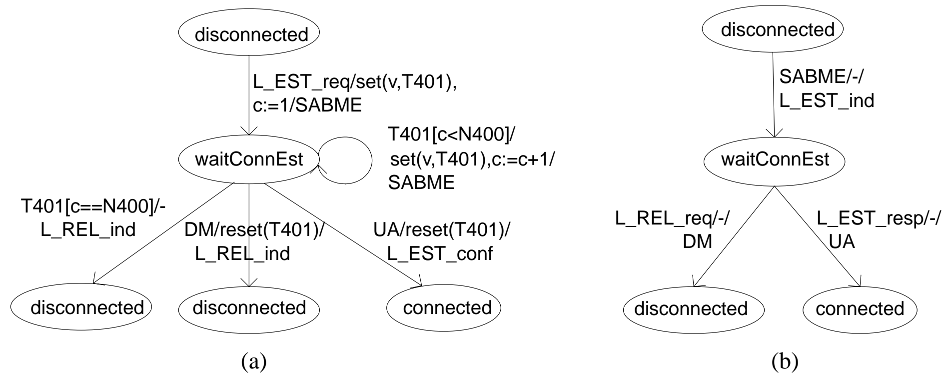

Fig. 5. Establishment of a DLC using *timed retrial confirmed sender* and *confirmed receiver*.

Meanwhile, the I_DLC block receiving the SABME command from the lower layer indicates its user of the connection request by sending the link establishment indicate (L_EST ind) primitive. The SU checks the possibility of the connection and if it could accept the request, the user responds with the link establishment response (L_EST_resp) primitive. Then, the I_DLC block acknowledges with the UA response frame to the initiator and enters the connected state. If the SU is not able to accept the request, it replies with the link release request (L_REL_req) primitive so that the I_DLC block notifies the initiator that the connection is not possible using the DM response. This behavior is well-suited for the pattern *confirmed receiver* as in Fig. 5 (b) where the SABME command is confirmed with either a UA or DM response.

5.2.2 Information transfer

After the establishment of a DLC, user information can be exchanged between O_DLC and I_DLC blocks using the patterns *unconfirmed sender* and *unconfirmed receiver* as illustrated in Fig. 6. On receipt of an L_UDATA_req primitive from the SU, a block transmits a UI (Unnumbered Information) command frame to its peer. If a block receives a UI command, it notifies the SU using an L_UDATA_ind primitive. Note that there is no confirmation to the peer block for the UI command.

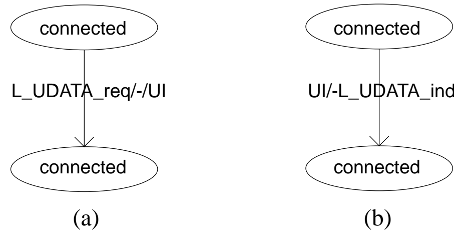

Fig. 6. User information transfer using *unconfirmed sender* and *unconfirmed receiver*.

5.2.3 Release of a DLC

Fig. 7 shows the behavior of DLC release after the information transfer. Any DLC block can initiate the connection release by sending a disconnect (DISC) command to its peer when it receives the link release request (L_REL_req) primitive from the SU. Similarly as the DLC establishment, the block starts the timer T401 and waits for a response from the peer. If the block receives either a UA or DM in the given time span, it stops the timer and notifies the SU of the release confirmation. Then, the block moves to the disconnected state. If the block has failed to receive the response after N400 attempts, it enters the disconnected state anyway and notifies it to the SU. The DLC block receiving a DISC command passes this request to its SU with the link release indication (L_REL_ind) primitive and replies to the peer with a UA response. Then it enters the disconnected state. The behaviors of DLC release are instances of the patterns *timed retrial confirmed sender* and *confirmed receiver*.

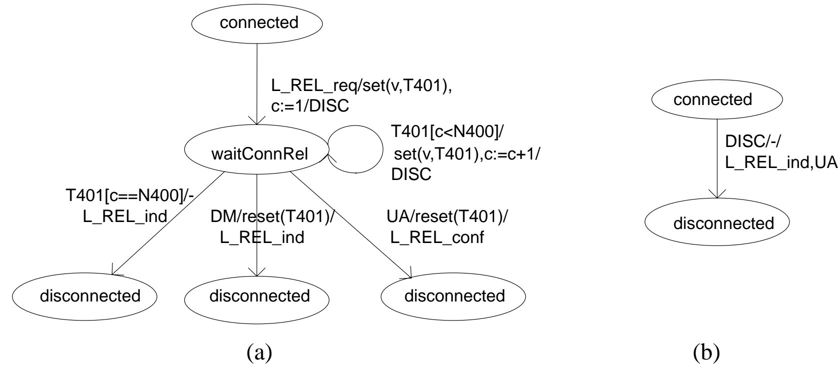

Fig. 7. Release of a DLC using *timed retrial confirmed sender* and *confirmed receiver*.

5.3 Validation of Pattern-based Design

5.3.1 Model construction in PROMELA

For the validation of pattern-based V.76 protocol, we first build a system model in PROMELA. By converting the structural design of Fig. 4 to the corresponding PROMELA code, we can obtain an outline of the validation model as follows:

```
#define BUFSZ 0     /* channel buffer size */
/* message declaration */
mtype = {L_EST_req, L_EST_ind, L_EST_resp, L_EST_conf,
         L_UDATA_req, L_UDATA_ind, L_REL_req, L_REL_ind,
         SABME, UA, DISC, DM};
/* channel declaration */
chan SU2ODLC = [BUFSZ] of {mtype};
chan ODLC2SU = [BUFSZ] of {mtype};
chan SU2IDLC = [BUFSZ] of {mtype};
chan IDLC2SU = [BUFSZ] of {mtype};
chan ODLC2DLL = [BUFSZ] of {mtype};
chan DLL2ODLC = [BUFSZ] of {mtype};
chan IDLC2DLL = [BUFSZ] of {mtype};
chan DLL2IDLC = [BUFSZ] of {mtype};
/* process declaration */
proctype OSU()   {}   /* outgoing SU process */
proctype ISU()   {}   /* incoming SU process */
proctype ODLC()  {}   /* outgoing data link connection process */
proctype IDLC()  {}   /* incoming data link connection process */
proctype DLL()   {}   /* data link layer process */
init {    /* initialization process */
    atomic {run OSU(); run ISU(); run ODLC(); run IDLC(); run DLL()}
}
```

The special process init instantiates the initial processes of the model by using the `run` operator. In this model, static instances of OSU, ISU, ODLC, IDLC, and DLL are created from the process. We use the atomic construct to create the processes atomically. For the closeness of the validation system, we have to build the environment processes corresponding to the SU and data link layer blocks. OSU, ISU, and DLL are provided for this purpose. Note that we divide the single SU block into two processes, OSU and ISU, for the simple modeling of communications with ODLC and IDLC processes. The DLL process provides the actual communication between ODLC and IDLC with the possibility of frame loss.

From the model outline, we can acquire a complete PROMELA model by converting the CEFSMs of Figs. 5, 6, and 7 to the corresponding PROMELA constructs. Appendix A presents the complete PROMELA model of the design.

5.3.2 Result of experience

As a typical procedure of SPIN validation, we first check the possibility of deadlock and the existence of unreachable code. One deadlock happens when the UA response for an SABME request is lost during the DLC establishment. See Fig. 5 (b). If the UA response was lost at the DLL, the I_DLC does not know this situation and moves to the connected state. Then it will stay there forever. To avoid the deadlock, the protocol has an inactivity check timer, T403, that represents the maximum amount of time a DLC

entity will allow to elapse without frames being exchanged on the DLC. When we designed the protocol, we missed the handling of the lost UA, and the recommendation [20] mentioned that the timer is optional without giving the reason for using it. However, the validation shows this deadlock exactly, and the problem could be solved by introducing the timer in our design. Another deadlock can happen when a responder of DLC release failed to receive the DISC command because of the consecutive frame lost. SPIN pinpointed the potential deadlock precisely, and the problem could also be fixed using the timer T403.

Identification of unreachable code is important in the system validation because the unreachable code could contain a fault. In our case, the reception of DM response during the DLC release could not be executed because there is no sender for the frame in the peer entity. Indeed the DM frame is used to report to its peer that the DLC entity is already in the disconnected state. This situation could happen if a DLC entity does not exchange any message with the peer for a while, for example, the time period set by T403. Then, the DLC entity decides that the connection is an inactive mode and moves to the disconnected state, even though the connection is still active. Since our original design did not consider the inactive mode transition, the code, which would be executed by the inactive mode transition, should be unreachable in this situation.

After fixing the above problems, we validate the system model against system properties. From the patterns used in the design, we can obtain about 14 properties of the V.76 protocol. Most of the properties are satisfied in the validation. One problem happens when we check the reception of SABME frame at the I_DLC process. The frame may never arrive if the DLL process fails to transmit the frame sufficiently often. Although this is not a design problem of the I_DLC block, developers have to consider this situation in their real system development. Regarding the properties collected in the validation phase, we would like to emphasize that the properties are not only for the validation of design, but also for the system test in the later development phase. For instance, the properties collected during the model validation could be used as a basis of test case generation.

6. CONCLUSIONS

In this paper, we suggested a software development methodology to specify and validate a communication protocol using the pattern. When a system developer designs a protocol, patterns are used for a high-level description of the protocol. The SPIN model checker is used to validate the pattern-based abstract description, which helps to find design faults in the early stage of the development. One feature of our approach is to use a communicating extended finite state machine to describe a system in visual diagrams. The finite state machine is familiar to computer scientists and engineers and easy to understand. Moreover, our methodology helps a developer obtain the required properties of a target system from the patterns used. The property section in each pattern enables the developer to capture the system correctness properties to be met in the final system.

To show the usefulness of our methodology, we have conducted several case studies, including the ITU-T's V.76 protocol. We described the protocol by using seven patterns such as *split protocol handler*, *timed retrial confirmed sender*, *confirmed receiver*, etc.

and 14 properties were identified. Through the validation, we could find some design errors and unexpected problems.

In the future, it will be desirable to supplement the current pattern language with more high-level patterns since the pattern language described in this paper is not a complete set. We can extend the language by composing and/or specializing the existing patterns, or developing new ones. In addition, tool support will make it easier to select, instantiate, and adapt patterns for the system specification. Automatic construction of a PROMELA model from the specification will also be needed and we are currently developing a prototype of such a tool using Tcl/Tk.

REFERENCES

1. 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.
2. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.
3. L. Rising, *Design Patterns in Communication Software*, Cambridge University Press, Cambridge, UK, 2001.
4. J. Ellsberger, D. Hogrefe, and A. Sarma, *SDL: Formal Object-Oriented Language for Communicating Systems*, Prentice-Hall PTR, New York, 1997.
5. E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, Cambridge, MA, 2000.
6. G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, Vol. 23, 1997, pp. 279-295.
7. Y. Byun, B. A. Sanders, and K. Chung, "A pattern language for communication protocols," in *Proceedings of 9th Conference on Pattern Languages of Programs* (*PLoP*), 2002.
8. A. S. Tanenbaum, *Computer Networks*, Prentice Hall, Upper Saddle River, New Jersey, 2002.
9. ITU-T Recommendation Q.2931 (02/95), *Broadband Integrated Service Digital Network* (*B_ISDN*) *Digital Subscriber Signaling System No. 2 – User-Network Interface* (*UNI*) *layer 3 specification for basic call/connection control*, International Telecommunication Union, 1995.
10. B. Croft and J. Gilmore, *RFC 951 – Bootstrap Protocol* (*BOOTP*), 1997.
11. A. Pnueli, "The temporal logic of programs," in *Proceedings of 18th IEEE Symposium Foundations of Computer Science*, 1977, pp. 46-57.
12. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, New York, NY, 1991.
13. ITU-T Recommendation Z.100 (11/99), *Specification and Description Language (SDL)*, International Telecommunication Union, 1999.
14. L. Doldi, *SDL Illustrated: Visually Design Executable Models*, Toulouse, France, 2001.
15. T. Faison, "Interaction patterns for communicating processes," in *Proceedings of 5th Conference on Pattern Languages of Programs* (*PLoP*), 1998.

16. J. Dorsch, A. Ek, and R. Gotzhein, "SPT − the SDL pattern tool," in *Proceedings of 4th SDL and MSC Workshop* (*SAM*), 2004, pp. 50-64.
17. D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, Vol. 8, 1987, pp. 231-274.
18. D. Bosnacki, D. Dams, L. Holenderski, and N. Sidorova, "Model checking SDL with SPIN," in *Tools and Algorithms for Construction and Analysis of Systems*, Springer, Berlin, Germany, 2000, pp. 363-377.
19. G. J. Holzmann and J. Patti, "Validating SDL specifications: an experiment," in *Proceedings of International Conference on Protocol Specification*, *Testing*, *and Verification*, 1989, pp. 317-326.
20. ITU-T Recommendation V.76 (08/96), *Generic Multiplexer using V.42 LAPM-based Procedures*, International Telecommunication Union, 1996.

APPENDIX A. PROMELA MODLE of V.76 PROTOCOL

```
#define BUFSZ 0     /* Channel buffer size. */
#define N400   2    /* Maximum number of frame retransmission. */
                    /* This number depends on implementation. */
#define timer bool

/* message declaration */
mtype = {L_EST_req, L_EST_ind, L_EST_resp, L_EST_conf, L_UDATA_req,
         L_UDATA_ind, L_REL_req, L_REL_ind, SABME, UA, UI, DISC, DM};

/* channel declaration */
chan SU2ODLC = [BUFSZ] of {mtype};
chan ODLC2SU = [BUFSZ] of {mtype};
chan SU2IDLC = [BUFSZ] of {mtype};
chan IDLC2SU = [BUFSZ] of {mtype};
chan ODLC2DLL = [BUFSZ] of {mtype};
chan DLL2ODLC = [BUFSZ] of {mtype};
chan IDLC2DLL = [BUFSZ] of {mtype};
chan DLL2IDLC = [BUFSZ] of {mtype};

proctype OSU() {    /* environment process for originating service user */
establish_DLC:
    SU2ODLC!L_EST_req;    /* Send L_EST_req to establish a DLC. */
                          /* This primitive initiates ODLC process. */
    if    /* Handle two possible responses from the peer */
    :: ODLC2SU?L_EST_conf -> goto info_transfer;    /* connection success */
    :: ODLC2SU?L_REL_ind -> goto establish_DLC;    /* connection fail */
    fi;
info_transfer:
    if    /* Consider all cases after connection establishment. */
          /* They include information transfer and connection release. */
    :: SU2ODLC!L_UDATA_req -> goto info_transfer;
```

```
        :: ODLC2SU?L_UDATA_ind -> goto info_transfer;
        :: SU2ODLC!L_REL_req -> ODLC2SU?L_REL_ind; goto establish_DLC;
        :: ODLC2SU?L_REL_ind; -> goto establish_DLC;
        fi;
}

proctype ISU() {    /* environment process for incoming service user */
establish_DLC:
        IDLC2SU?L_EST_ind; /* reception of L_EST indication */
        if    /* send either L_EST_resp or L_REL_req to simulate both cases */
        :: SU2IDLC!L_EST_resp->goto info_transfer;    /* accept the connection */
        :: SU2IDLC!L_REL_req->goto establish_DLC;     /* reject the connection */
        fi;
info_transfer:
        if    /* consider all four cases after connection establishment */
        :: SU2IDLC!L_UDATA_req -> goto info_transfer;
        :: IDLC2SU?L_UDATA_ind -> goto info_transfer;
        :: SU2IDLC!L_REL_req -> IDLC2SU?L_REL_ind; goto establish_DLC;
        :: IDLC2SU?L_REL_ind; -> goto establish_DLC;
        fi;
}

proctype ODLC() { /* ODLC process constructed from patterns */
        timer T401 = false; byte c;    /* define a timer and a counter */
disconnected:
        SU2ODLC?L_EST_req;    /* Receive L_EST_req primitive from OSU */
        T401 = true; c = 1;/* Set the timer and initialize the counter */
        ODLC2DLL!SABME;    /* Send SABME command to peer to establish a DLC */
        goto waitConnEst;
waitConnEst:    /* There are four possibilities in this state */
        if    /* Handle the timeout, reception DM, or reception UA */
        :: (T401 == true) ->    /* T401 expired */
            if    /* Compare with the maximum number of retransmission. */
            :: (c < N400) -> T401 = true; c = c + 1; ODLC2DLL!SABME; goto waitConnEst;
            :: (c == N400) -> ODLC2SU!L_REL_ind; goto disconnected;
            fi;
        :: DLL2ODLC?DM -> T401 = false; ODLC2SU!L_REL_ind; goto disconnected;
        :: DLL2ODLC?UA -> T401 = false; ODLC2SU!L_EST_conf; goto connected;
        fi;
connected:
        if    /* Handle information transfer or prepare for DLC release. */
        :: SU2ODLC?L_UDATA_req -> ODLC2DLL!UI; goto connected;
        :: DLL2ODLC?UI -> ODLC2SU!L_UDATA_ind; goto connected;
        :: SU2ODLC?L_REL_req ->T401 = true; c = 1; ODLC2DLL!DISC; goto waitConnRel;
        :: DLL2ODLC?DISC -> ODLC2SU!L_REL_ind; ODLC2DLL!UA; goto disconnected;
        fi;
waitConnRel:   /* Handle the DLC release after requesting DISC command. */
```

```
    if    /* There are four possible cases. */
    :: (T401 == true) ->    /* T401 expired */
        if    /* Compare with the maximum number of retransmission */
        :: (c < N400) -> T401 = true; c = c + 1; ODLC2DLL!DISC;
                        goto waitConnRel;
        :: (c == N400) -> ODLC2SU!L_REL_ind; goto disconnected;
        fi;
    :: DLL2ODLC?DM -> T401 = false; ODLC2SU!L_REL_ind; goto disconnected;
    :: DLL2ODLC?UA -> T401 = false; ODLC2SU!L_REL_ind; goto disconnected;
    fi;
}

proctype IDLC() {    /* IDLC process constructed from patterns */
    timer T401 = false; byte c;
disconnected:
    DLL2IDLC?SABME;    /* Reception of SABME for DLC establishment */
    IDLC2SU!L_EST_ind;    /* Indicate to ISU. */
    goto waitConnEst;
waitConnEst:
    if    /* Handle two possible responses, reject or accept, from ISU */
    :: SU2IDLC?L_REL_req -> IDLC2DLL!DM; goto disconnected;    /* reject */
    :: SU2IDLC?L_EST_resp -> IDLC2DLL!UA; goto connected;    /* accept */
    fi;
connected:
    if    /* Handle information transfer or prepare for DLC release. */
    :: DLL2IDLC?UI -> IDLC2SU!L_UDATA_ind; goto connected;
    :: SU2IDLC?L_UDATA_req -> IDLC2DLL!UI; goto connected;
    :: DLL2IDLC?DISC -> IDLC2SU!L_REL_ind; IDLC2DLL!UA; goto disconnected;
    :: SU2IDLC?L_REL_req -> T401 = true; c = 1; IDLC2DLL!DISC;
                                goto waitConnRel;
    fi;
waitConnRel:    /* Handle the DLC release after requesting DISC command. */
    if    /* There are four possible cases. */
    :: (T401 == true) ->    /* T401 expired */
        if    /* Compare with the maximum number of retransmission */
        :: (c < N400) -> T401 = true; c = c + 1; IDLC2DLL!DISC; goto waitConnRel;
        :: (c == N400) -> IDLC2SU!L_REL_ind; goto disconnected;
        fi;
    :: DLL2IDLC?DM -> T401 = false; IDLC2SU!L_REL_ind; goto disconnected;
    :: DLL2IDLC?UA -> T401 = false; IDLC2SU!L_REL_ind; goto disconnected;
    fi;
}

proctype DLL() {    /* environment process for data link layer */
    do    /* It simulates the message loss at the layer. */
    :: ODLC2DLL?SABME;
        if
```

```
            :: DLL2IDLC!SABME;     /* Correct transfer of SABME from ODLC to IDLC */
            :: skip;                /* Model the loss of SABME */
            fi;
        :: IDLC2DLL?DM;
            if
            :: DLL2ODLC!DM;     /* Correct transfer of DM from IDLC to ODLC */
            :: skip;             /* Model the loss of DM */
            fi;
        :: ODLC2DLL?UA;
            if
            :: DLL2IDLC!UA;     /* Correct transfer of UA from ODLC to IDLC */
            :: skip;            /* Model the loss of UA coming from ODLC*/
            fi;
        :: IDLC2DLL?UA;
            if
            :: DLL2ODLC!UA;     /* Correct transfer of UA from IDLC to ODLC */
            :: skip;             /* Model the loss of UA coming from IDLC */
            fi;
        :: ODLC2DLL?UI;
            if
            :: DLL2IDLC!UI;     /* Correct transfer of UI from ODLC to IDLC */
            :: skip;            /* Model the loss of UI coming from ODLC */
            fi;
        :: IDLC2DLL?UI;
            if
            :: DLL2ODLC!UI;     /* Correct transfer of UI from IDLC to ODLC */
            :: skip;            /* Model the loss of UI coming from IDLC */
            fi;
        :: ODLC2DLL?DISC;
            if
            :: DLL2IDLC!DISC;     /* Correct transfer of DISC from ODLC to IDLC */
            :: skip;              /* Model the loss of DISC coming from ODLC */
            fi;
        :: IDLC2DLL?DISC;
            if
            :: DLL2ODLC!DISC;     /* Correct transfer of DISC from IDLC to ODLC */
            :: skip;               /* Model the loss of DISC coming from IDLC */
            fi;
        od;
}

init {     /*Initialize process to start all needed processes for validation*/
        atomic {run OSU(); run ISU(); run ODLC(); run IDLC(); run DLL(); }
}
```

YoungJoon Byun received his Ph.D. degree in Computer and Information Science and Engineering from the University of Florida in 2003. Between 1993 and 1998, he worked for the Electronics and Telecommunications Research Institute, Taejon, Korea, as a member of research staff. He is currently an assistant professor in the School of Engineering and Engineering Technology at the Pennsylvania State University at Erie, PA. His research areas include design patterns for communication systems, software analysis and verification, software development environments and tools, and real-time systems.

Beverly A. Sanders received a Ph.D. from Harvard University and has held faculty positions at the University of Maryland, the Swiss Federal Institute of Technology, and is currently with the University of Florida. Her research focuses on techniques to help programmers develop high quality software including formal methods and design patterns.