

A Tool Support for Design and Validation of Communication Protocol using State Transition Diagrams and Patterns

YoungJoon Byun

California State University – Monterey Bay, Seaside, USA

Email: youngjoon_byun@csumb.edu

Abstract—In this paper, we introduce a software tool to assist design and validation of a communication protocol specified in state transition diagrams and patterns. When protocol developers start development of a new system, they tend to describe the developing system with several high-level description elements such as communicating blocks, communication paths, messages, and finite state machines. Then, they want to validate the correctness of their design as early as possible to find out any faults in the design. In this paper, we propose a software tool with which protocol developers can specify structural architectures and behavioral main operations of a protocol system through the graphical user interface. Then, the tool generates PROMELA code from the design specification to make it possible for the developers to validate the specification using the SPIN model checker. Meanwhile, we can specify a protocol system with more high-level abstraction using a pattern, a combination of several basic elements of the protocol. A pattern is a software reuse mechanism to apply a well-known solution of a recurring problem to the similar software developments repeatedly. In the paper, we also propose the usage of patterns in our tool. Using the pattern support, it will be possible for the tool to provide automatic pattern selection, instantiation, and composition. As results of this tool, protocol developers can describe a communication protocol more efficiently and reduce the development cost. Furthermore, they can have the confidence for the specification at the early stage of software development.

Index Terms—software tool, state transition diagram, communication protocols, SPIN, PROMELA, patterns

I. INTRODUCTION

Your goal is to simulate the usual appearance of papers in a Journal of the Academy Publisher. We are requesting that you follow these guidelines as closely as possible. Classic life-cycle of software engineering is composed of several phases such as requirements analysis, design, coding, testing, and maintenance. In this paper, our main

concern is high-level design of a communication protocol at the initial design phase. Developers in that phase typically want to capture essential functions of a developing system before the detailed design and implementation. Essential functions in the protocol system may include high-level communicating blocks, message flows, interactions with other outside systems, etc.

As proposed at the pattern-based software development methodology [2], protocol developers typically devise system architecture of a protocol with structural elements such as communicating blocks along with communication paths between them. A block is an architectural building element of the developing protocol 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 each block is specified using the behavioral elements that provide interactions between blocks. In our methodology, we use finite state machines to describe the behavior of each communicating block.

In this paper, we present a tool that helps the high-level design of a protocol system through a graphical user interface (GUI). By using this tool, developers can address overall architecture and behavior of a protocol with several elements, including communication blocks, communication paths, messages, states, transitions between states, and timers. By selecting and composing these elements, designers can build an abstract system of a target system. In addition to the high-level design of a system, the tool assists the validation of design using the SPIN model checker [4]. The idea is that in many cases developers want to check the correctness and consistency of the design immediately after when they finished the design. Thus, they can fix any design faults as early as possible. For the SPIN model checking, developers should build a model of a developing system in PROMELA, a modeling language of SPIN. Then, the model is simulated and verified on SPIN. To help the construction of PROMELA code, our tool automatically generates the PROMELA code from the design. Through this validation phase, developers can have the confidence for the specification at the early stage of software development.

Based on “A tool support for design and validation of communication protocol using state transition diagrams”, by YoungJoon Byun which appeared in the Proceedings of the Fourth International Conference on Information Technology: New Generations, Las Vegas, USA, April 2007. © 2007 IEEE.

Then the developers can move to the detailed design and implementation phases.

As another issue of protocol design, we can specify the communication protocol in more abstract and high-level using a pattern. To address common architecture and behavior of a protocol, we proposed a set of patterns [9] which are categorized in two groups, structural patterns and behavioral patterns. The basic idea of a pattern is that when a software developer creates a new system, the developer 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 [10][11]. A design solution that has worked well in a particular situation can be used again in similar situations in the future. In our case, a pattern is a combination of several core elements of a protocol such as communication blocks, communication paths, messages, states, transitions between states, and timers. In this paper, we propose a function of our tool to support the pattern-based development methodology. The tool will provide pattern selection, instantiation, and composition for the protocol specification through a GUI.

The remainder of the paper is organized as follows. In Section 2, we present related work as background of our tool. Then, the behavioral description of a block using a state transition diagram (STD) is given in Section 3. In Section 4, we describe the tool in detail, focusing on GUI, tool functionalities, and PROMELA code generation. Our design for the pattern support in the tool is given in Section 5, and we conclude this paper in Section 6 with summary and further research.

II. RELATED WORK

A. SPIN and PROMELA

Model checking is an automated technique to validate correctness of a system by investigating a finite state model of the system [4]. This technique is especially useful for reactive and distributed systems that are characterized by many interactions among processes. A model checker explores all states reachable from an initial state and validates a set of correctness properties on the model.

Model checking typically starts with the construction of a model and identification of properties to be checked. Then, it validates the properties with an appropriate model checker. A system is usually modeled using a state based description such as communication automata, CSP (Communicating Sequential Processes), Petri Nets, etc. A model must be made as closely as possible to a system so that the validation result for the model reflects the system's execution exactly. The properties a model checker validates in a model include the reachability of a certain state, safety and liveness of a system, and relative order of events in a system. Several formalisms are used to precisely express the properties.

Model checking has been successfully used in hardware validation and communication software. There are many model checkers applicable today and we select

SPIN (Simple Promela INterpreter) as our validation tool. It was developed at Bell Labs for the analysis and validation of distributed systems, especially of communication protocols. Furthermore, it is freely available at the SPIN web site, www.spinroot.com. A system is described in a modeling language called PROMELA (PROcess MEta LAnguage). Given a system model specified in PROMELA, SPIN can simulate the system's execution. It is also possible to perform an exhaustive exploration of a system's state space using a depth-first search algorithm. During the simulation and validation, SPIN can check for absence of deadlocks, non-progress cycles, unspecified message receptions, unexecutable code, etc. The model is also proven the correctness of system invariant claims and temporal claims. If a system model violates a correctness property, SPIN recognizes this, and a trace of violation is generated.

PROMELA is a validation modeling language for SPIN, focusing on the abstraction of message exchange in a system. It has C-like syntax with features from Dijkstra's guarded commands and communication primitives from Hoare's CSP. The important constructs of a PROMELA program are process, message channel, and variable. Processes execute asynchronously, which means there are no assumptions on the relative speed of processes and only one process is executed at a particular point. Each statement in a process is either executable or blocked. If a statement is blocked for some reason, the statement halts until it becomes executable. Processes interact either by message passing via channels or memory sharing of global variables. Communication via a message channel is either synchronous (i.e., rendezvous) or asynchronous (i.e., buffered).

B. Visual Notation Using State Transition Diagrams

Our design provides visual description of a system with several elements. In fact, visual specification such as state transition diagram is a common way to describe system abstraction. Leue and Holzmann [7] suggested v-PROMELA, a visual and object-oriented modeling language for SPIN. The language is designed to present abstraction and hierarchical layering. The visual notation is able to express both structure and behavior of a reactive concurrent system to acquire software architecture at the early life-cycle stages. They use UML for Real-Time notation for structural description and adapt many important ideas from Realtime Object-Oriented Modeling (ROOM). Behavior is specified using hierarchical communicating extended finite state machines. This paper also suggests a translation mechanism from the visual constructs to a PROMELA program. The visual notation is supported by a graphical tool, VIP. On the other hand, Mikk et al. [8] have translated Statecharts into PROMELA using extended hierarchical automata as an intermediate format. Their technique allows a system described in Statecharts to be validated in a linear temporal logic model checker.

III. STATE TRANSITION DIAGRAM

Many communication systems react to messages coming from outside environments. A communicating extended finite state machine (CEFSM) is a finite state machine extended with local variables and parameterized messages to describe the behavior of a communication system [3]. When a system receives an input message, it performs some actions for the message and emitting output messages. Then, it may move from its current state to a new state.

Definition. A CEFSM is a 5-tuple (S, s_0, M, f, V) where

- S is a set of states.
- s_0 is an initial state.
- M is a set of messages with their parameter list.
- 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 message, and a predicate composed of a subset of V , the state transition relation f has a next state, a set of output messages, and an action list presenting local variables updates and/or abstract description of behavior to be performed at this transition.

As a precondition of a transition, we introduce a predicate, a Boolean-valued expression of the local variables [3]. Upon receiving a message, a CEFSM evaluates the predicate. If the predicate holds, the CEFSM executes the transition. However, if the predicate does not hold, the CEFSM ignores the message and stays at the current state. An empty predicate is defined to be shorthand for true.

Meanwhile, in a message-driven system such as a communications protocol, a message may occur later than it is expected, or not happen at all because of transmission delay, message loss, etc. Many communication protocols employ timing constraints where timing violation action is taken if an expected message does not occur in a given amount of time. To manipulate timing constraints, our CEFSM is supplemented with a timer and timer-related operations [3]. A timer T is an element of the variable set with an associated time value v . It has two modes, active and inactive. Initially, a timer is inactive. The operation $set(v, T)$ allocates the time value v to the timer T and makes the timer be in the active mode. Once a timer is set, the timer generates a timer expiration signal after passing the time value unless it has been cancelled by the reset operation. The operation $reset(T)$ stops the timer T and changes it to the initial inactive mode.

As an example of CEFSM, see the following CEFSM that includes a part of V.76 protocol behavior [6].

```
{disconnected, waitConn, connected}, disconnected,
{est_req(type), conn(type), disc, ack, est_conf,
rel_ind}, f, {type, trial}),
where f is
{<disconnected, est_req(type), waitConn, (trial=1), {conn(type)}>,
<waitConn, disc[trial<3], waitConn, (trial++), {conn(type)}>,
<waitConn, ack, connected, ("reserve resource"), {est_conf}>,
<waitConn, disc[trial==3], disconnected, (), {rel_ind}>}
```

This CEFSM has three states (*disconnected*, *waitConn*, *connected*), an initial state *disconnected*, five messages (*est_req*, *conn*, *disc*, *ack*, *est_conf*, *rel_ind*), and two variables (*type*, *trial*). The output messages *est_req* and *conn* have a parameter *type*. For simplicity, we do not include the type of variables in this example. Transitions are represented by the relation f . For example, the transition $\langle \text{disconnected}, \text{est_req}(\text{type}), \text{waitConn}, (\text{trial}=1), \{\text{conn}(\text{type})\} \rangle$ indicates that the protocol moves from the state *disconnected* to the state *waitConn* when a communicating block of the protocol receives the input message *est_req* with a parameter in *type*. During this transition, the block assigns number one to the variable *trial* and generate the output message *conn* with the argument *type*. Actually, action list can include brief activities of a protocol in plain English as well as variable updates and assignments. For instance, “reserve resource” at the transition from *waitConn* to *connected* implies that the CEFSM needs to reserve resource when the block receives an input message *ack* at the state *waitConn*. This action will be refined in later development phases. Note that the transition from *waitConn* to *disconnected* has a predicate, $[\text{trial}=3]$, that asks whether the variable *trial* is equal to three.

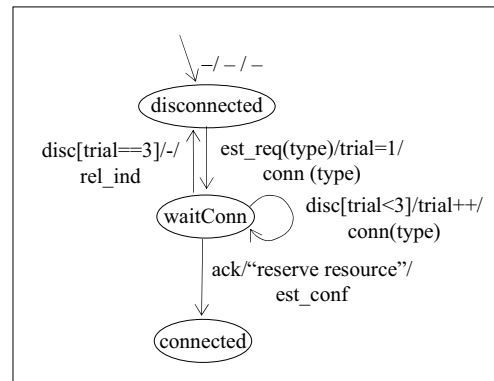


Figure 1. State transition diagram of a sample V.76 protocol

In many cases, the original definition of CEFSM with 5-tuple is difficult to recognize the meaning of protocol behavior. Thus, we represent the finite state machine in an STD, a directed graph whose vertices correspond to states and whose edges correspond to transitions. Figure 1 shows the STD of the example CEFSM. Each state is represented by a circle, and the initial state is indicated by the coming transition without any source state. In this example, the state *disconnected* is the initial state. Each transition is labeled with an input message, action list, and output messages that is denoted by *input (parameters) [predicate] / actions / outputs (parameters)*. The ‘-’ symbol in a transition indicates that there is no corresponding field. A transition that does not alter the state is represented by an arc that points to itself.

IV. TOOL DEVELOPMENT

A. Protocol Specification Using the Tool

Our tool, *FSM2SPIN*, is designed to provide the description of structural and behavioral aspects of a

protocol through the GUI. It also helps to validate the design specification by generating PROMELA code from the specification. Figure 2 shows the main user interface of the tool that provides the design of architectural aspect with the structural elements such as communicating blocks, communication paths, and message definition.

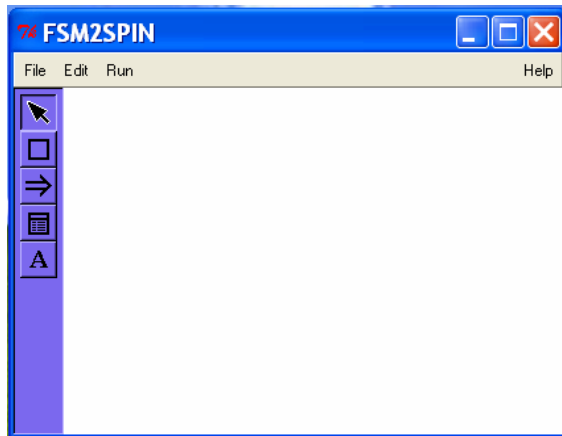


Figure 2. Main window of FSM2SPIN

From the menus located on the top, a user can start main functions of the tool. *File* menu provides *Open*, *Save*, *Print*, and *Exit* sub-menus to open an exist specification, save the current specification, print the specification, and exit the tool, respectively. *Edit* menu also has sub-menus to assist the editing of the specification. Using the *Run* menu, a user can generate the PROMELA code from a specification so that the user can conduct model checking for the specification with SPIN. *Help* menu provides useful information about the tool.

The buttons on the left side of the GUI represent the elements available for the design of structural aspect. The *Block* button represented by a square box makes a user to draw a communicating block in the main window as a placeholder to specify protocol behavior. The *Path* button denoted by an arrow line makes it possible for a user to connect two communicating blocks with a communication path and present messages flowing through it. A user can use *Message* button to define messages to be exchanged between blocks through communication paths.

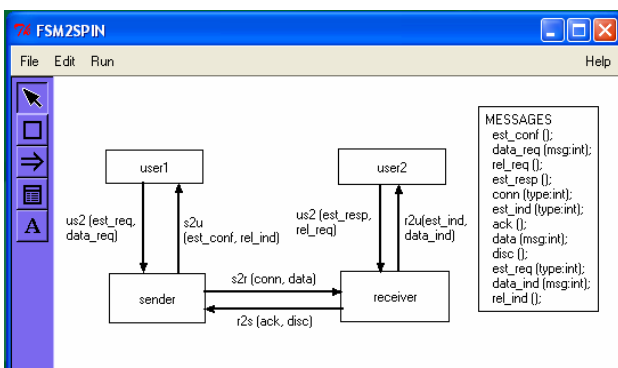


Figure 3. Specification of structural design of sample V.76 protocol

Using these elements, user can specify the architecture of a protocol. For example, Figure 3 represents a sample structural design of the V.76 protocol in which the protocol has 4 blocks (*user1*, *user2*, *sender*, *receiver*), 6 paths (*u2s*, *s2u*, *u2r*, etc), and 12 messages (*est_conf*, *data_req*, *rel_req*, etc).

After finishing the architectural design, a user can specify the behavioral aspect of each block by double-clicking a communicating block from the main window. Then, the tool creates a new window for the behavior specification. From the window, a user can specify an STD of a block using the *State* and *Transition* buttons. Figure 4 represents the behavior window of the block *sender* that is specified in Figure 3. Actually the behavioral design is based on the sample finite state machine of Section 3.

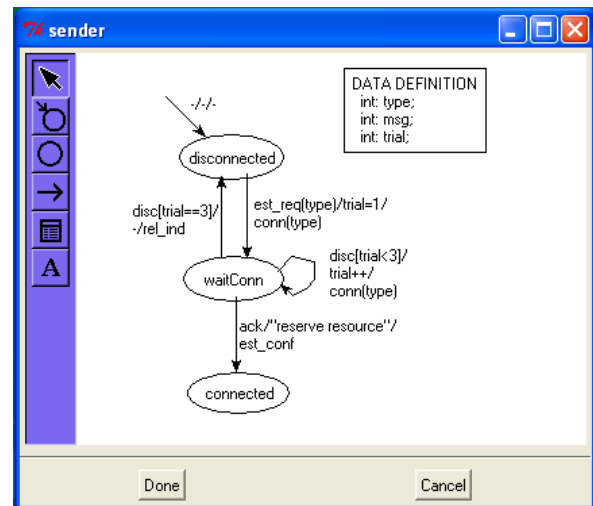


Figure 4. Specification of behavioral design of *sender* block

B. PROMELA code generation

As we already mentioned in the paper, our design specification is composed of several core elements. To store these elements in a design document, we use an internal format that is represented by the following grammar:

```

system → block+ path* message+
block → block_name FSM
path → path_name src_block dst_block
message_name*
message → message_name (argument_name :
type)*
FSM → initial_state state* transition* variable*
timer*
transition → input_message [guard] / action /
output_message*
guard → Boolean operation
action → arithmetic operation | brief description of
action in English
variable → variable_name : type
type → int | char | bool
timer → timer_name expiration_period
    
```

In this grammar, the star symbol ‘*’ indicates zero or more times repetition of the proceeding grammar element. A vertical bar ‘|’ and a square bracket ‘[’ and ‘]’ indicate a choice and an option, respectively. For example, a system can be composed of several communicating blocks, zero or more communication paths, and several messages as indicated at the first line of the grammar. Each block includes a block name and an FSM that is composed of an initial state, several states, transitions, variables, and timers.

From this internal representation of a design, it is relatively straightforward for our tool to generate the PROMELA code because both our specification and PROMELA have the similar theoretical background of a finite state machine. In other words, a block of our specification can be mapped to a process of PROMELA and an instance of the block can be created dynamically using the run operator of PROMELA. A communication path can be implemented in a PROMELA channel that carries messages with parameters. All input and output messages can be defined by using symbolic constants in PROMELA.

Meanwhile, recall that the STD used in our design description has a set of states and transitions. Actually, each state can be converted to a label in PROMELA code. Transition from a state to another state can be implemented in goto control transfer construct in PROMELA. For the message exchange on a communication path, the send and receive statements are used in PROMELA. A decision point with predicates is converted to the selection construct of PROMELA and each predicate is converted to a corresponding guard. The following is a fragment of sample PROMELA code generated from the sample design of Figure 3 and 4:

```
#define BUFSIZE 0

mtype = {est_conf, data_req, rel_req,
est_resp, conn, est_ind, ack, data, disc,
est_req, data_ind, rel_ind};

chan u2s = [BUFSIZE] of {mtype, int};
chan s2u = [BUFSIZE] of {mtype};
chan u2r = [BUFSIZE] of {mtype};
chan r2u = [BUFSIZE] of {mtype, int};
chan s2r = [BUFSIZE] of {mtype, int};
chan r2s = [BUFSIZE] of {mtype};

active proctype sender ()
{
    int type;
    int msg;
    int trial;

    goto disconnected;

disconnected:
    if
    :: u2s?est_req(type);
       trial=1;
       s2r!conn(type);
       goto waitConn;
    fi;
}
```

disconnected:
...

V. TOOL SUPPORT FOR PATTERNS

A. Patterns for Communication Protocols

Patterns are rarely stand-alone. Instead, several patterns are typically related to solve problems in a specific domain, which is called a pattern language. Table 1 presents our pattern language for the description of communication protocols. Detailed description of each pattern can be found at Byun et al. [9]. In this section, we present a pattern called *timed retrial confirmed sender* in a simplified form to give the general idea of our patterns.

TABLE I. 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

A.1. Timed Retrial Confirmed Sender

In this pattern, a communication block wants to transfer a message to a remote block through a lower layer. The problem of this pattern is that the lower layer is unreliable. Thus the message could be lost at the lower layer. Reliable transfer of a message over an unreliable lower layer can be achieved by using an acknowledgement for a message. To avoid unlimited waiting for a lost message or acknowledgement, we introduce a timer. Thus, if an acknowledgement is not received within a specified time period, the message is retransmitted. The pattern also needs a counter to check the upper limit of repetition.

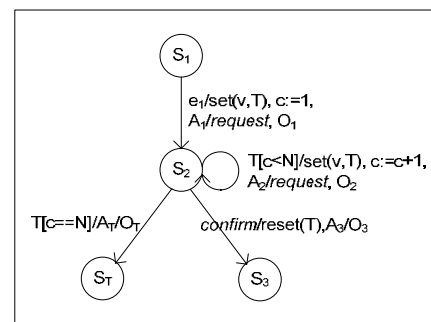


Figure 5. Pattern *timed retrial confirmed sender*.

Figure 5 shows a solution to the problem in a CEFSM. In the figure, 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 using a message named *request* to a corresponding peer block and generates any additional output messages O_1 if it exists. 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, for example, O_T to indicate the transmission failure and move to the state S_7 . Otherwise, it retransmits the request message after increasing the counter. In Figure 5, note that the action list and output messages such as A_1 , O_1 , A_2 , etc. are optional. 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 [12].

As an example of the pattern in a real protocol, a sending block of ATM signaling protocol [13] uses a timer $T303$ for four seconds while waiting for a connection establishment message *SETUP*. The block tries at most twice for the connection establishment.

B. Tool support for patterns

To make it possible for a protocol developer to specify a communication protocol using patterns, our tool will provide pattern selection, pattern instantiation, and pattern composition through a GUI. For the pattern selection, we will provide a menu to the design windows, for example Figure 3 and 4, to hold the list of patterns available. When a user selects a pattern from the menu, a new window called *pattern instantiation window* will be created for a developer to instantiate the selected pattern with specific values. For instance, if a user clicked the *timed retrial confirmed sender button*, an instantiate window will show up and ask the user for the specific data such as a real request message name, a confirm message name, a timer name, timing period, and a maximum number of repetition. All these data must be given for the pattern instantiation. When the user finishes the instantiation, the user will close the window. Then, the instantiated pattern will be displayed on the original design window. Indeed the pattern-based specification is nothing but the combination of several instantiated patterns to describe the architecture and behavior of the protocol. Because a user cannot design a system with the provided patterns solely in many cases, the user typically needs a delicate mechanism to fill the gap of coarse patterns. Thus the buttons of block, link, state, and transition in the design window will help the fine editing and composition to finalize the design.

B.1. Pattern Instantiation

For the adaptation of a pattern in a specific situation, the pattern must obtain concrete names and values for all pattern elements. Instantiation of a structural pattern is straightforward with a specific name for each communicating block, communication path, and message. However, instantiation of a behavioral pattern need some consideration. Basically, our behavioral patterns are defined using the CEFSM. For example, our pattern of Figure 5 can be described as $(\{S_1, S_2, S_3, S_7\}, S_1, \{e_1, T, confirm, request, error, O_1, O_2, O_3, O_T\}, f, \{c, v\})$, where f has $\langle S_1, e_1, S_2, (set(v,T), c:=1, A_1), \{request, O_1\} \rangle, \langle S_2, T[c<N], S_2, (set(v,T), c:=c+1, A_2), \{request, O_2\} \rangle, \langle S_2, T[c==N], S_6, (A_T), \{error, O_T\} \rangle, \langle S_2, confirm, S_3, (reset(T), A_3), \{O_3\} \rangle$. For an instantiation of the pattern, a concrete timer name, time period, number of repetition should be given by a user. Moreover, the number of transitions for the confirmation should also be provided. Even though there are only two types of confirmation *timeout* ($T[c==N]$) and *confirm* in Figure 5, the number of responses is not typically fixed. Thus this number is also decided by the user.

For the proper instantiation, a skeleton of each pattern will be stored at the tool with an internal representation. For instance, the pattern *timed retrial confirmed sender* has the following format:

```
?T
?v
?N
S1, S2, {e1}, [true], {set(v,T)[M], c:=1, A1},
{request[M], O1}
S2, S2, {T[M]}, [c<N[M]], {c:=c+1, set(v,T)[M],
A2}, {request[M], O2}
S2, ST, {T[M]}, [c==N[M]], {AT}, {OT}
?n
S2, S3_n, {confirm_n[M]}, [true], {reset(T)[M],
A3_n}, {O3_n}
```

A line beginning with a character '?' needs user's direct input. For example, T , v , N , and n should be provided by a user. Additionally, a user should instantiate remaining fields such as the state name and transition. Indeed the internal representation is a placeholder for the concrete name and value. Note that a field with $[M]$ indicates that it is a mandatory one to make the pattern meaningful.

Figure 6 shows an example of pattern instantiation window using the pattern *timed retrial confirmed sender* for a V76 protocol connection set up [6] where T , v , N , and n has $T401$, 2.0 , 2 , and 3 , respectively. In Figure 6, when a block receives a message L_EST_req from an SU (service user), it initiates establishment of a DLC (data link connection) 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 2 times which is the maximum number of retransmission for the response. In

this instantiation, the actual value of $T401$ has 2.0. If it receives a UA as an acknowledgement of a successful connection, it moves to the state *connected*. However, if the block fails to receive any response in 2 trials or receives a DM response, it understands this situation as a failure of the request and gives up the connection.

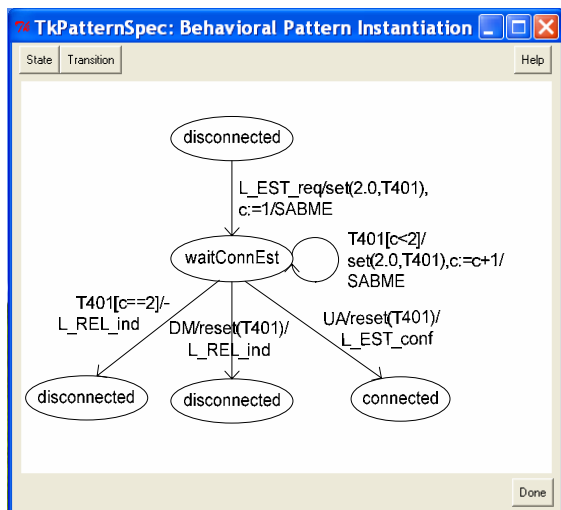


Figure 6. An example of pattern instantiation window for timed retransmission confirmed sender.

B.2. Pattern Composition

After a pattern instantiation, it may be necessary for a user to combine two or more CEFSMs into a single CEFSM. For example, once a DLC is established at the V.76 protocol as Figure 6, a user data can be received from a remote block using a patterns *unconfirmed receiver* at the state *connected*. Note that the two patterns have the common state *connected*. We can compose these patterns' instantiations by merging states. Thus a user can merge two states either by giving the same state name or by overlapping a state to another state.

V. CONCLUSIONS

In this paper, we presented a software tool to help the design specification of a communication protocol. When protocol developers design a protocol system, core elements such as communicating blocks, communication paths, messages, states, and transitions can be selected from the tool and then they are instantiated to describe the protocol structure and behavior. To fix any design faults in the early stage, the tool generates PROMELA code for the SPIN based model checking. Note that it is cost effective to uncover design errors in the early phase to prevent the errors from affecting later phases. One feature of our tool is to visualize the design in state transition diagrams. The finite state machine is familiar to computer scientists and engineers and easy to identify the behavior of an interactive system. A user can combine each element through an interactive user interface. Meanwhile, PROMELA model construction is typically a challenging practice in formal validation because the model is crucial to the validation result and must reflect the system to be developed correctly. Because of the tight correspondence

between our description elements and PROMELA, our tool can automatically generate PROMELA code from a specification. This prototype has been designed and implemented using Tcl and Tk. In addition to the element-level design of a communication protocol, we proposed a tool function to help the specification in pattern-level. When a developer designs a protocol, patterns can be selected, instantiated, and composed to describe the protocol in more high-level.

As further research, we will implement the pattern support part in our tool. We also believe that the current prototype tool should be improved to product-level with enhanced functions and reliability. For example, current prototype does not support a timer that is one of the key elements in protocol specification. In addition to the design and validation aspects of a protocol development, we are considering SDL (Specification and Description Language) as our implementation language [5]. To make it easy for developers to implement the specification, we have a plan to generate a skeleton SDL code from the design specification in the tool.

In our tool, it will also be desirable for the tool to provide a mechanism to manage the pattern language which will make it possible to create a new pattern and store the pattern in a hierarchical structure. Indeed, 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. It is worth noting that the current patterns could be used as a foundation for other domains. For instance, suppose we are going to develop a reactive system using a pattern. Then, a pattern which is specific to the reactive system can be constructed on top of the current patterns after analyzing any recurring situation in the domain.

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, NY, 1996.
- [2] Y. Byun and B. A. Sanders, "A pattern-based development methodology for communication protocols," in *Proceedings of the 20th Annual ACM Symposium on Applied Computing*, Sante Fe, NM, March 2005, pp. 1524-1528.
- [3] J. Ellsberger, D. Hogrefe, and A. Sarma, *SDL: Formal Object-Oriented Language for Communicating Systems*, Prentice-Hall PTR, New York, NY, 1997.
- [4] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, 1997, pp. 279-295.
- [5] ITU-T Recommendation Z.100 (11/99), *Specification and Description Language (SDL)*, International Telecommunication Union, 1999.
- [6] ITU-T Recommendation V.76 (08/96), *Generic Multiplexer using V.42 LAPM-based Procedures*, International Telecommunication Union, 1996.
- [7] S. Leue and G. Holzmann, "v-Promela: A visual, object-oriented language for Spin," in *Proceedings of the Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Saint Malo, France, May 1999.

- [8] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann, "Implementing statecharts in Promela/Spin," in *Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, Boca Raton, Florida, October 1998.
- [9] Y. Byun, B. A. Sanders, and K. Chung, "A pattern language for communication protocols," in *Proceedings of the 9th Conference on Pattern Languages of Programs (PLoP)*, 2002.
- [10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, New York, NY, 1996.
- [11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.
- [12] A. S. Tanenbaum, *Computer Networks*, Prentice Hall, Upper Saddle River, NJ, 2002.
- [13] 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.

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, Taejeon, Korea, as a member of research staff. He is currently an assistant professor of the School of Information Technology and Communication Design at California State University – Monterey Bay, CA. His research areas include design patterns for communication systems, software analysis and verification, software development environments and tools, and real-time systems.