Modeling, validation, and verification of PCEP using the IF language[*]

Iksoon Hwang[1], Mounir Lallali[1], Ana Cavalli[1], and Dominique Verchere[2]

[1] TELECOM & Management SudParis, 9 Rue Charles Fourier, 91011 Évry Cedex, FRANCE
{Iksoon.Hwang, Mounir.Lallali, Ana.Cavalli}@it-sudparis.eu
[2] Alcatel-Lucent R&I, Route de Villejust, 91620 Nozay, FRANCE
Dominique.Verchere@alcatel-lucent.com

Abstract. In this paper, we present the modeling, validation, and verification of an industrial protocol for constraint-based path computation, called PCEP. From the PCEP specification defined by IETF, we divide the functionalities of PCEP into two parts: application and protocol. The protocol part of PCEP is then described in the IF language which is based on communicating timed automata. A number of basic requirements are identified from the PCEP specification and then described as properties in the IF language. Based on these properties, the validation and verification of the formal specification are carried out using the IF toolset. Test cases are generated by using an automatic test generation tool, called TestGen-IF, which uses partial state space exploration guided by test purposes. As a result of the modeling, validation, and verification, some errors and ambiguities are found in the PCEP specification. Also a number of test cases are obtained which will be used for testing implementations.

1 Introduction

Formal methods are mathematically rigorous techniques that can be used to describe and analyze the behavior of systems. A number of advantages arise from the use of formal methods during the software development procedure. Less ambiguous specifications are provided and these can be used in model checking and model-based testing. Validation is the process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements [1]. Model checking and model-based testing have been widely used for validation and verification of systems. Recently, there have been a number of industrial case studies that use formal methods in validation and verification. Bozga *et al.* [2] presented the verification and test generation for the SSCOP protocol and Jia and Graf [3] performed the verification experiments on the MASCARA protocol using IF (Intermediate Format) [4]. Hessel and Pettersson [5] provides model-based testing of a WAP gateway using UPPAAL [6].

[*] This work has been supported by the French competitiveness cluster SYSTEM@TIC, through CARRIOCAS project.

The CARRIOCAS project [7] aims at providing a distributed pilot network for industrial applications with high complexity, scope, and scale. A number of hardware and software components are developed in the project in order to provide the connectivity services for such large-scale distributed, data, and computing intensive applications. One of the important activities of the CARRIOCAS project is the validation experiment on the proposed pilot network. As a part of the validation activities, a communication protocol for constraint-based path computation which is called Path Computation Element Communication Protocol (PCEP) [8] is chosen for validation and verification.

In this paper, we present the modeling, validation, and verification of PCEP, which are carried out in the CARRIOCAS project. From the PCEP specification defined by IETF (Internet Engineering Task Force), we divide the functionalities of PCEP into two parts: application and protocol. The protocol part is then described in the IF language. A number of basic requirements are identified from the PCEP specification and then described as properties in IF. Based on these properties, the validation and verification of the formal specification are carried out using the IF toolset [9]. From the basic requirements, a number of test purposes are defined and test cases are generated by using an automatic test generation tool, called TestGen-IF [10]. As a result of the modeling, validation, and verification, we found some errors and ambiguities in the PCEP specification. Also we obtained a number of test cases which will be used for testing implementations in the CARRIOCAS project.

The paper is organized as follows. In Section 2, we explain briefly about the CARRIOCAS project and PCEP. The IF language and the IF toolset are explained in Section 3. In Section 4, we describe how to model PCEP in the IF language and the validation of the formal specification is carried out in Section 5. The test generation methods and results are discussed in Section 6 and finally Section 7 concludes the paper.

2 CARRIOCAS project and PCEP

2.1 CARRIOCAS project

Large scale distributed applications (often termed as Grid applications) challenge the performance of the existing telecom network infrastructures. In the CARRIOCAS project, a number of research and industrial applications are considered such as car design with crash simulations for safety analysis and energy production with atomic reactor models for central problem simulations. These applications require ultra-high performance computers to execute their simulation application workflows during the life-cycle of the project and also need exchange of massive amounts of data in order to enable local and distant groups of engineers to work collaboratively while viewing and analyzing their results.

The purpose of the CARRIOCAS project is to design and develop the components of high-throughput capacity system and flexible network architectures that can adapt its connectivity services dynamically for the data-intensive and delay sensitive distributed applications. The pilot network aggregates the Ethernet data flows issued from client networks to be transported by carrier grade Ethernet Virtual Circuits. The CARRIOCAS project is attempting to develop a common service management component based on the Scheduling, Reconfiguration, and Virtualization (SRV) functions to allow the compositions of different connection service elements from a network infrastructure. The SRV functions can be extended above different types of infrastructures including computational servers and data storage centers to deliver bundles of service elements. These extensions require advanced Network Management capabilities based on Path Computation Element (PCE) functions which provide the routes of the connection services, e.g. the routes on GMPLS (Generalized Multi-Protocol Label Switching)-capable carrier grade Ethernet switches.

In addition to the design and development of the network, one of the important activities of the CARRIOCAS project is the validation experiment on the proposed pilot network. As a part of the validation activities, a communication protocol for constraint-based path computation which is called PCEP is chosen for validation and verification.

2.2 Path Computation Element Communication Protocol

In large scale and multi-domain networks, path computation can be complex and may require specific computational components and cooperation between elements in different nodes. In order to address these problems, an architecture based on PCE model has been proposed [8]. In this PCE-based architecture, a PCE is an entity that computes a network path based on a network graph and computational constraints and a Path Computation Client (PCC) is any kind of client application requesting a path computation to be performed by a PCE. PCEP is a communication protocol between a PCC and a PCE, or between two PCEs in order to exchange path computation requests and path computation replies as well as notifications of specific events related to the use of a PCE. PCEP operates over TCP [11] which provides reliable messaging and flow control. The following PCEP messages are defined:

- *Open* message is used to initiate and negotiate a PCEP session.
- *Keepalive* message is used to establish and maintain a PCEP session.
- *PCReq* message is sent to request a path computation.
- *PCRep* message is sent in reply to a path computation request.
- *PCNtf* message is sent to notify a specific event.
- *PCErr* message is sent upon the occurrence of a protocol error condition.
- *Close* message is used to close a PCEP session.

A PCC may have PCEP sessions with more than one PCE and similarly a PCE may have PCEP sessions with multiple PCCs. Once the TCP connection is established between a PCC and a PCE, the PCC and the PCE (also referred to as “PCEP peers”) initiate PCEP session establishment. Various session parameters including the Keepalive timer, the Deadtimer, other detailed capabilities, and policy rules are carried within *Open* messages. If the session parameters are agreed, *Keepalive* messages are used to acknowledge *Open* messages. Once the PCEP session has been successfully established, *Keepalive* messages may be exchanged between PCEP peers to ensure the liveness of the PCEP session. If the session parameters are not acceptable but negotiable, session negotiation can be performed where the proposed session parameters are contained within *PCErr* messages. If the PCEP peers disagree on the session parameters or one of the PCEP peers does not answer after the expiration of the establishment timer, the TCP connection is immediately closed. Figure 1 shows the scenario of PCEP session establishment after negotiation.

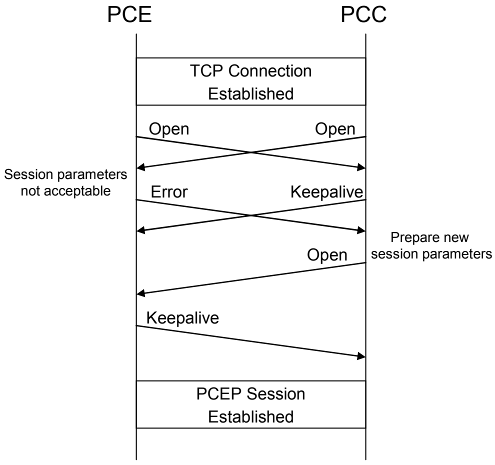
PCE
PCC
TCP Connection Established
Open
Open
Session parameters not acceptable
Error
Keepalive
Prepare new session parameters
Open
Keepalive
PCEP Session Established

Fig. 1. PCEP session establishment after negotiation

After establishment of a PCEP session, when an event is triggered that requires the computation of a set of paths, the PCC sends a *PCReq* message which contains a set of constraints and attributes for the path for computing. Upon receiving a path computation request from a PCC, the PCE triggers a path computation, the results are sent back to the PCC in a *PCRep* message where they can be either positive (one or more computed paths) or negative (no path found). When a PCE wants to notify a specific event to PCCs such as possible unacceptable delay because of overload, it sends a *PCNtf* message to PCCs. Similarly, a PCC may desire to notify a PCE of a particular event such as the cancelation of pending requests. A *PCErr* message is sent in several situations: when a protocol error condition is met or when the request is not compliant with the PCEP specification, e.g. reception of malformed messages or unexpected messages. When one of the PCEP peers desires to terminate a PCEP session, it first sends a *Close* message and then closes the TCP connection. When the PCEP session is terminated, the PCC and the PCE cancel all pending operations and clear corresponding resources.

3 IF language

3.1 IF model

IF [4] is a formal method based on communicating timed automata in order to model asynchronous communicating real-time systems. In IF, a system is expressed by a set of parallel processes communicating asynchronously through a set of buffers. A process instance can be created and destroyed dynamically during system execution. An IF process is described as a timed automaton extended with discrete data variables. A process has a set of control states and a private buffer for input messages, and can have local data such as discrete variables and clocks. There are two types of control states: stable states and unstable states. An unstable state is a temporary state where no interleaving between processes is possible. In other words, if a process moves to an unstable state by an action, the atomicity of the execution is guaranteed until it reaches a stable state.

Transitions describe the behavior of a process on stimuli. A transition can be triggered either by (timed) guards or by an input message where an urgency attribute (*eager*, *delayable* or *lazy*) defines the priority of the transition over time progress. When an *eager* transition is executable, time progress is blocked until the transition is executed. If there is an executable *delayable* transition, time can progress as long as the transition is executable. If time progress makes the *delayable* transition non-executable, time progress is blocked until the transition is executed. For *lazy* transitions, time can progress although the transitions become non-executable. The action of a transition may include sending output messages, setting/resetting clocks, assignment of variable values, and creation/destruction of processes.

3.2 IF toolset

The IF toolset [9] provides an environment for modeling and validation of an IF specification. The core components of the toolset are the IF static analyzer and the IF exploration platform. The IF static analyzer transforms an IF specification into an abstract syntax tree which is a collection of C++ objects. The IF exploration platform performs the simulation of process executions by using the abstract syntax trees. A set of APIs is provided by the IF exploration platform, which allows implementation of user-specific exploration. Through these APIs, CADP [12], a tool for validation of LTS models and TGV [13], for test case generation using on-the-fly technique can be connected to the IF toolset.

In the IF toolset, it is possible to check if given properties hold for an IF specification by using observers. Once a property is described in the IF language using a specific syntax for observers, e.g. monitoring of events and cutting off generation of irrelevant states, it is executed in parallel with the target system. The communication between the system and the observer process is synchronous and the observer process has always the highest priority during exploration so that monitoring of an event is triggered immediately when the event occurs.

4 Formal description of PCEP

4.1 Overall architecture

In our experiments, we divide the functionalities of PCEP into two parts: application and protocol. The functionalities of the application part include session initiation, session parameter negotiation, request/reply of path computation, notification of specific events, closing the session, etc. The functionalities of the protocol part include the handling of finite state machines including local variables and timers, collision resolution procedure, keeping the current session by exchanging *Keepalive* messages, etc. In our formal specification of PCEP, the protocol part is modeled by a system. PCEP applications and the lower layer (TCP) are, therefore, considered to be an environment. A complete set of service primitives are defined between PCEP applications and the system and between the system and the lower layer. As a PCEP application can communicate with more than one peer PCEP applications, it is necessary to model multiple instances of the PCEP protocol. The system consists of a main process and multiple instances of a child process where each instance of the child process handles a PCEP session. The Figure 2 shows the overall architecture.

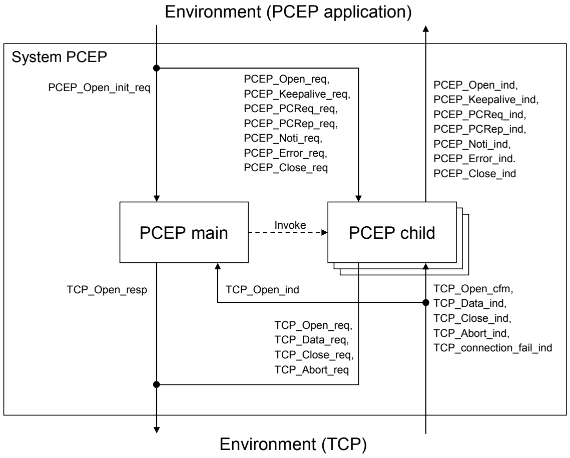

Fig. 2. Overall architecture of the formal specification of PCEP

In TCP, function interfaces are defined to provide a certain minimum set of services to guarantee that all TCP implementations can support the same protocol hierarchy [11]. The service primitives between the system and TCP are based on those function interfaces. For exchange of user data, i.e. exchange of the PCEP messages through Send and Receive user calls, we introduce parameterized service primitives, TCP_Data_PCEP_xxx_req and TCP_Data_PCEP_xxx_ind. The TCP_Data_PCEP_Unknown_ind service primitive represents an unknown PCEP message from its peer and the TCP_connection_fail_ind service primitive represents TCP connection failures such as the failure of sending a message. The service primitives between the system and PCEP applications are based on the PCEP messages. In order to model session initiation request from a PCEP application, the PCEP_Open_init_req service primitive is introduced. When the main process receives a session initiation request by receiving either the TCP_Open_ind message from a PCEP peer or the PCEP_Open_init_req message from a PCEP application, it creates a child process which manages the PCEP session.

4.2 States, internal variables, and timers

As mentioned in Section 4.1, we have two kinds of processes in our model: a main process and a child process. Since the purpose of the main process is to create instances of a child process when there are session initiation requests, the main process has only one stable state, Idle. In a child process the following four stable states are defined based on the PCEP specification: TCPPending, OpenWait, KeepWait, and SessionUP. In addition to the stable states, we introduce a number of unstable states in order to branch off the control flow of a process. If the behavior is decided by internal variable values or clock values, each case can be represented by a transition. If the decision should be made by the parameter values of an input message, however, it is necessary to have more than one transition, one to receive a message and the others to check the parameter values. In this case, the parameter values are checked in unstable states in order to guarantee the atomicity of the behavior. The following shows an example.

```
state TCPPending;
  deadline lazy;
  input TCP_Open_cfm(tcpConnectResult);
    nextstate TCPPending_TCP_Open_cfm_decision;
  ...
endstate;

state TCPPending_TCP_Open_cfm_decision #unstable ;
  provided (tcpConnectResult = ConnectSuccess);
    ...
      nextstate OpenWait;

  provided (tcpConnectResult = ConnectFail) and
             (tcpConnectRetry < TCPConnectMaxRetry);
    ...
      nextstate TCPPending;
  ...
endstate;
```

In our model, four internal variables and five timers are defined based on the PCEP specification: tcpConnectRetry, pcepOpenRetry, remoteOK, and localOK for internal variables, and tcpConnectTimer, pcepOpenWaitTimer, pcepKeepWaitTimer, pcepKeepaliveTimer, and pcepDeadTimer for timers. In addition to the above internal variables, the main process manages a childInfoTable which contains information on the current ongoing sessions in order to manage the number of active sessions and duplicated session initiation requests. Also the childInfoTable is used for collision resolution procedure when there are simultaneous session initiation requests between PCEP peers.

As mentioned in Section 4.1, a decision on session parameter negotiation is carried out by PCEP applications in our model. When a child process in OpenWait state receives an *Open* message from its peer, it sends the received information to its PCEP application and then waits for a reply from the application. If there is no reply for a given time, the child process should release the corresponding PCEP resources and close the TCP connection. A new timer, internalKeepWaitTimer is introduced for that purpose. Similarly, internalOpenWaitTimer is introduced for waiting a reply from applications when a *PCErr* message is received from its peer for session negotiation. The SyncTimer is not included in our model because the cancelation of path computation request is considered to be the functionality of PCEP applications.

4.3 Abstraction of information

In PCEP, each PCEP message has a common header and may have a number of PCEP objects. In our service primitives, the number of parameters is minimized in order to reduce the problem size of the model. A minimum set of parameters to decide the behavior of the system is defined for each service primitive as follows: Keepalive and Deadtimer for an *Open* message, a list of pairs of errorType and errorValue for a *PCErr* message, the existence of rpObject and endPointObject for a *PCReq* message, and the existence of rpObject for a *PCRep* message. The Keepalive parameter value is used to send a *Keepalive* message periodically in SessionUP state. A PCEP session is closed if there is no PCEP message from its peer during the time given in the Deadtimer parameter. The errorType and errorValue are necessary since the behavior of the system can be different according to these values. The existence of rpObject and endPointObject is used to send proper *PCErr* messages when these mandatory parameters are missing.

In our model, we assume that the PCEP messages received from PCEP peers may have errors such as missing mandatory objects and unknown objects. In order to model any errors in the received PCEP message which cannot be modeled by other parameters, a boolean type parameter, errorInPCEPMessage is defined in TCP_Data_PCEP_XXX_ind service primitives. For example, receiving an unknown object is modeled by "errorInPCEPMessage=true" while missing rpObject by "rpObjectExist=false". Since the errorInPCEPMessage parameter represents most erroneous messages, some cases are missing in our model, e.g. the system should send a *PCErr* message with "errorType=3" when it receives an unknown object from its peer. When there is a TCP connection request from its peer, the decision whether accept it or not is usually made by system calls without interaction with its applications. Therefore, the tcpConnectResult parameter, which contains the result of this decision, is included in TCP_Open_ind service primitive.

4.4 Remarks

In this section, we explain two issues that we have faced when modeling.

How to make time progress in a system? In IF, it is assumed that time does not progress during execution of transitions, i.e. the time spent during execution of transitions is always zero. Time can progress only in stable configurations as long as there is no executable *eager* transition. In our model, there is always at least one executable transition enabled by an input message from environment at any stable state as we designed the system in such a way that the interaction between the system and environment is possible at any instance. Therefore, if the transitions enabled by an input message from environment have *eager* deadline, time in the system will never progress. In order to solve this problem, we used *lazy* deadline for every transition enabled by an input message from environment.

Is the first incoming message served first in a process? In IF, the communication between two processes is asynchronous, i.e. messages from other processes are stored in a buffer before being consumed. However, messages from environment are handled in a different way. The Figure 3 shows how incoming messages are handled in a process.

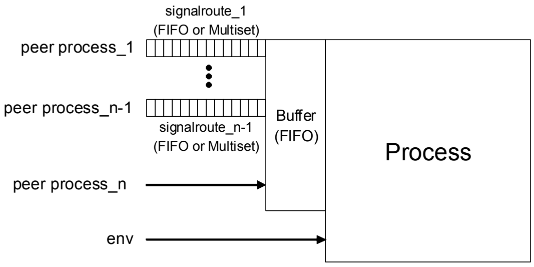

Fig. 3. Handling of incoming messages in a process

As shown in the figure, there is a FIFO buffer for each process. The messages sent from other processes arrive at the buffer either through a signalroute or directly from the peer process. Each signalroute has its own property such as fifo/multiset, reliable/lossy, etc. When a message is passed to a receiver process from a signalroute, it is stored in the buffer. The order of passing messages from signalroute to the buffer can be either FIFO or random according to the property of the signalroute. When a process sends a message to a receiver process directly by using the destination process ID, it is stored in the buffer immediately. When a message is sent from environment to a process, the message is not stored in the buffer and is consumed immediately by the process although there are messages in the buffer waiting for being consumed. Therefore, the communication between environment and a process is synchronous while the communication between two processes is asynchronous.

In our model, in the beginning, there was an internal communication between the main process and a child process. When an instance of a child process is to terminate, it sends a *done* message to the main process and the main process clears all internal information related to that child process. In order to keep the consistency of resources, the main process should consume the *done* message as soon as it receives. However, this cannot be guaranteed since the main process can receive messages from environment at any time. As a result, we had resource mismatch problem, i.e. the child information was not cleared in the main process although the child process had already been terminated. In order to solve this problem, we removed the internal communication and the internal information related to a child process is cleared by the child process itself. Note that our solution is a temporary one and only applicable to limited cases. A general solution is required for the above problem.

5 Validation of the formal description

5.1 Validation with IF observers

IF observers can be used to check if given properties hold for an IF specification. Properties are based on observable actions such as input and output messages, and also include checking variable values and clock values. Once a property is described in the IF language, an exhaustive state space exploration using either breadth first search or depth first search is carried out by the IF simulator. During the simulation, the observer process checks if it can observe the expected behavior. In our experiments, first, we define the following three general properties.

- Property G1: The specification must be deadlock-free.
- Property G2: In any state, there must be at least one active timer.
- Property G3: In any state, the value of any active timer must be no greater than its maximum.

The purpose of the property G2 is to avoid infinite time progress waiting for some events[1]. The property G3 checks if the system stops timers appropriately either by timeout or by cancelation. Second, the properties specific to PCEP are identified from the PCEP specification. In order to facilitate the modeling, we define a state transition table that describes the behavior of the system for a given set of state, input, and conditions. Table 1 shows an example.

According to Table 1, if the system is in KeepWait state where the Keepalive timer is active and remoteOK=0 and it receives a TCP_Data_PCEP_Keepalive_ind message without error, it should send a PCEP_Keepalive_ind message and move to OpenWait state. The above information represents an atomic behavior that the system should follow and it can be considered to be a requirement. In our experiments, 101 basic requirements are identified from a complete state transition table and then 66 requirements are described as properties in the IF language. For each requirement, it is considered that the property holds for the formal

[1] In the case of PCEP, the property G2 may not hold if the Keepalive timer and the Deadtimer have zero values.

Table 1. State transition table

Input	KeepWait			
	Index	Condition	Output	Next state
TCP_Data_PCEP_Keepalive_ind	56	Keepalive timer active, No error in msg, remoteOK=0	PCEP_Keepalive_ind	OpenWait

specification if the system sends the expected output messages and moves to the expected next state for the given set of state, input, and conditions.

In order to reduce the problem size during the validation, first, the timer values and retry numbers are limited. The maximum values of the tcpConnectTimer, the pcepOpenWaitTimer, and the pcepKeepWaitTimer are limited to 2 instead of 60 seconds[2]. The maximum values of the pcepKeepaliveTimer and the pcepDeadTimer are limited to 2 and 8 respectively as the recommended value for the pcepDeadTimer is four times the value of the pcepKeepaliveTimer used by the remote peer. The tcpConnectMaxRetry is limited to 1 instead of 5. Second, we limit the range of parameter values. The values of the Keepalive and the Deadtimer parameters in an *Open* message are fixed to have 2 and 8, respectively. The number of error objects that can be carried within a *PCErr* message is limited to 1 and the values of errorType and errorValue parameters are limited to 1 and from 1 to 6 respectively.

5.2 Validation results

With the limited timer values, retry numbers, and parameter values, the system with one instance of a child process was completely explored with 410 states and 12010 transitions. When we allowed all possible values for the input parameters while the timer values and retry numbers were limited, we had extremely large number of transitions. The system was completely explored with 3095 states and 55355305 transitions. When we used the timer values and retry numbers as given in the PCEP specification, e.g. 60 seconds for the tcpConnectTimer, while the parameter values were limited, we could explore the system with 85750 states and 3945670 transitions. In the case of two instances of a child process with the limited timer values, retry numbers, and parameter values, the simulator completed state space exploration with 74476 states and 4294788 transitions. For most properties, the validation is carried out where the timer values, retry numbers, and parameter values are limited. If it is necessary to have other parameter values, the ranges of those parameters are changed appropriately in the formal specification. For the property 29, the validation is carried out with two instances of a child process as it checks if the current PCEP connection is released by the collision resolution procedure. Table 2 shows the validation results.

[2] It is possible to consider that timer values are abstracted such that 0 represents no time progress, 1 time progress up to 59 seconds, and 2 represents 60 seconds.

Table 2. Validation results

Properties	# of states	# of trans	Time (hh:mm:ss)	Results
Prop. G1	85514	2855179	12:26:09	Interrupted (No failure)
Prop. G2	1568	54981	6	Validated
Prop. G3	51923	1050744	3:47	Validated
Prop. 18	2516	55929	9	Failed (No success)
Prop. 29	108111	2629793	9:58:28	Interrupted (Success, No failure)
Others	< 33500	< 867000	< 2:25	Validated

Among 69 properties (three general and 66 specific to PCEP), 66 properties were successfully validated where most simulations were terminated within 15 seconds. For two properties (property G1 and property 29), the simulations were interrupted after around 10 hours because of state explosion problem[3]. Although we observed the expected behavior and no failure was found for those two properties, we cannot say that the validation was successful since we could not explore all possible state space. We found that the property 18 does not hold for our formal specification. Although we explored all possible state space, we could not observe the expected behavior. This was due to the problem that was found in the original PCEP specification. There exists a case in the PCEP specification that never happens.

5.3 Remarks

One of the difficulties during the validation was the large size of state space. In our model, some internal variables, e.g. internal variables for receiving parameter values, are used temporarily. During the validation, we found that a number of redundant states were generated due to those internal variables. In order to remove those redundant states, we initialized those internal variables by the end of each transition. For example, once we check the value of the tcpConnectResult variable in TCPPending_TCP_Open_cfm_decision state as presented in Section 4.2, the variable is initialized as it is not used in other states. As a result, the numbers of states and transitions explored after exhaustive state space exploration reduced from 30329 states and 941313 transitions to 410 states and 12010 transitions.

In IF observers, internal variable values and the parameter values of messages can be checked after the execution of each transition. Therefore, if a property includes checking variable values or parameter values while those values are initialized after the execution of the transition, e.g. by going back to Idle state, it is not possible to check the property. The main reason why only 66 requirements

[3] In our experiments, it is considered to have the state explosion problem either when the simulation proceeds very slowly where the memory usage reaches almost its maximum or when the simulator crashes due to lack of memory.

are described as properties while 101 basic requirements are identified is due to this limitation.

6 Test generation of PCEP

6.1 Testgen-IF

The TestGen-IF generates timed test cases from an IF specification and a set of test purposes. Partial state space exploration guided by test purposes is carried out, which is called the Hit-or-Jump algorithm [14]. A test purpose is a set of (ordered) conditions. A condition is a conjunction of a process instance constraint, state constraints, action constraints, variable constraints, and clock constraints. A process instant constraint indicates the identifier of a process instance. A state constraint indicates source state or target state of a transition. Action constraints describe observable actions such as sending or receiving messages as well as non-observable actions such as informal statements. A variable constraint gives conditions on variable values and a clock constraint conditions on either clock values or status of clocks, e.g. active/inactive. The following shows the test purpose which corresponds to the requirement given in Table 1.

$$\begin{array}{ll}
tp_{56} & = \{cond_1\} \\
cond_1 & = constraint_1 \wedge constraint_2 \wedge \ldots \wedge constraint_7 \\
constraint_1 & = \text{“}process : instance = \{\texttt{PCEPChild}\}\texttt{0}\text{”} \\
constraint_2 & = \text{“}state : source = \texttt{KeepWait}\text{”} \\
constraint_3 & = \text{“}state : target = \texttt{OpenWait}\text{”} \\
constraint_4 & = \text{“}action : \texttt{input TCP\_Data\_PCEP\_Keepalive\_ind(f)}\text{”} \\
constraint_5 & = \text{“}action : \texttt{output PCEP\_Keepalive\_ind()}\text{”} \\
constraint_6 & = \text{“}variable : \texttt{remoteOK = false}\text{”} \\
constraint_7 & = \text{“}clock : \texttt{pcepKeepWaitTimer is active}\text{”}
\end{array}$$

The state exploration starts from a given state s_i, which is the initial state in the beginning, using breadth first search with a given *depth limit*. Initially, all conditions in a test purpose are unmarked. If an unmarked condition is satisfied in a transition during exploration where the target state is s_j, which is called a *Hit*, the condition is marked, the path from s_i to s_j is stored, the buffer that stores visited state information is cleared, and then the exploration starts again from s_j. If no unmarked condition is satisfied during the exploration until the given *depth limit*, which is called a *Jump*, one of leaf nodes, e.g. the state s'_j is chosen for the start state, the path from s_i to s'_j is stored, the buffer that stores visited state information is cleared, and then exploration starts again from s'_j. The state space exploration terminates either when all conditions are marked or when all state space is explored within the *depth limit*. Once all conditions are satisfied during the exploration, i.e. all conditions are marked, the path from the initial state to the target state of the transition where the last condition is satisfied becomes a test sequence for the test purpose. A test sequence consists of observable actions such as input and output messages and delays which represent time intervals between observable actions.

6.2 Test generation and results

As mentioned in Section 5.1, 101 basic requirements are identified from the PCEP specification. In our experiments, for simplicity, we generate a test case for each requirement. Among 101 requirements, 98 requirements are described as test purposes. Three requirements are missing because of the following reasons. As explained in Section 5.2, there is a requirement that our formal specification does not meet (property 18). For the other two requirements, they are not considered because the atomicity of the behavior cannot be guaranteed in our description of test purposes since two processes are involved in the behavior.

Similar to the case of validation, the timer values, retry numbers, and parameter values are limited for most test purposes. If it is necessary to have other parameter values, the ranges of those parameters are changed appropriately. The following shows an example test sequence which is generated by the test purpose given in Section 6.1

```
?TCP_Open_ind{1,ConnectSuccess} !TCP_Open_resp{ConnectSuccess}
!TCP_Data_PCEP_Open_req{{{2,8}}}
?TCP_Data_PCEP_Open_ind{f,{{2,8}}} !PCEP_Open_ind{{{2,8}}}
?PCEP_Error_req{{1,{{1,4},}}} !TCP_Data_PCEP_Error_req{{1,{{1,4},}}}
?TCP_Data_PCEP_Keepalive_ind{f} !PCEP_Keepalive_ind{}
```

For all 98 test purposes, the test sequences are generated successfully. After deleting the test sequences which are the prefix of another one, we finally obtain 90 test sequences where the total number of test inputs is 353. The generated test sequences will be used for testing PCEP implementation developed by one of the partners of the CARRIOCAS project.

7 Conclusions

In this paper, we presented the modeling, validation, and verification of PCEP which is a protocol for constraint-based path computation defined by IETF. The protocol part of PCEP is described in the IF language. A number of basic requirements are identified from the PCEP specification and then described as properties in the IF language. Based on these properties, the validation of the formal specification is carried out by using the IF toolset. From the basic requirements, a number of test purposes are defined and test cases are generated by using the TestGen-IF. The obtained test cases will be used for testing implementations developed by one of the partners of the CARRIOCAS project.

Our experiments showed very promising results concerning the use of formal methods for modeling, validation, and verification. A number of errors and ambiguities were found from the original specification including a wrong sentence that misleads the behavior of the protocol, a non-executable transition (related to the property 18 as explained in Section 5.2), and unclear descriptions such as when a timer should be started. It should be noted that most of these errors were found during the modeling phase. Therefore, we can conclude that if we describe specifications using formal methods, we can obtain higher quality of specifications, i.e. with less errors and ambiguities even before validation of the formal specifications.

References

1. Software Engineering Institute/Carnegie Mellon: Capability Maturity Model Integration (CMMISM) for Software Engineering Version 1.1 (2002)
2. Bozga, M., Fernandez, J.C., Ghirvu, L., Jard, C., Jron, T., Kerbrat, A., Morel, P., Mounier, L.: Verification and test generation for the SSCOP protocol. Journal of Science of Computer Programming **36** (2000) 27–52
3. Jia, G., Graf, S.: Verification experiments on the MASCARA protocol. In: Proceedings of the 8th International SPIN Workshop on Model Checking of Software. Volume 2057 of LNCS., Springer-Verlag (2001) 123–142
4. Bozga, M., Graf, S., Mounier, L.: IF-2.0: A validation environment for component-based real-time systems. In: Proceedings of CAV'02, London, UK, Springer-Verlag (2002) 343–348
5. Hessel, A., Pettersson, P.: Model-based testing of a WAP gateway: An industrial case-study. In: Proceedings of 11th International Workshop on Formal Methods for Industrial Critical Systems. Volume 4346 of LNCS. Springer-Verlag (2007) 116–131
6. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a Nutshell. Int. Journal on Software Tools for Technology Transfer **1** (1997) 134–152
7. Audouin, O., Cavalli, A., Chiosi, A., Leclerc, O., Mouton, C., Oksman, J., Pasin, M., Rodrigues, D., Thual, L.: CARRIOCAS project: an experimental high bit rate optical network tailored for computing and data intensive distributed applications. In: Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series. (2007)
8. Vasseur, J.P., Le Roux, J.L.: Path Computation Element (PCE) Communication Protocol (Nov. 2008) IETF Internet draft, draft-ietf-pce-pcep-19.txt, work in progress.
9. Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The IF toolset. In: SFM-04. Volume 3185 of LNCS. Springer-Verlag (2004) 237–267
10. Cavalli, A.R., Montes De Oca, E., Mallouli, W., Lallali, M.: Two complementary tools for the formal testing of distributed systems with time constraints. In: Proceedings of the 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications, Vancouver, Canada (2008) 315–318
11. Postel, J.: Transmission Control Protocol. RFC 793 (Standard) (1981) Updated by RFCs 1122, 3168.
12. Fernandez, J.C., Garavel, H., Kerbrat, A., Mounier, L., Mateescu, R., Sighireanu, M.: CADP - A protocol validation and verification toolbox. In: CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification, London, UK, Springer-Verlag (1996) 437–440
13. Fernandez, J.C., Jard, C., Jéron, T., Viho, C.: Using on-the-fly verification techniques for the generation of test suites. In: CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification, London, UK, Springer-Verlag (1996) 348–359
14. Cavalli, A.R., Lee, D., Rinderknecht, C., Zaïdi, F.: Hit-or-Jump: An algorithm for embedded testing with applications to in services. In: Proceedings of FORTE XII / PSTV XIX '99, Deventer, The Netherlands, Kluwer, B.V. (1999) 41–56