Engineering Journal issn: 0125-8281

Article

An Automated Framework for BPMN Model Verification Achieving Branch Coverage

Chanon Dechsupa[a,*], Wiwat Vatanawood[b], and Arthit Thongtak[c]

Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok, Thailand
E-mail: [a,*]Chanon.d@chula.ac.th(Corresponding author), [b]Wiwat.v@chula.ac.th, [c]Arthit.t@chula.ac.th

Abstract. BPMN model is used in software development process that the procedural logics of software are described in term of graphical representation. Formal verification using colored Petri net (CPN) can be used to prove whether a designed BPMN model is frees of undesirable properties such as deadlock and unreachable task, and meets user requirements or not. Although there are many researches providing the transformation rules and frameworks for automating and verifying the CPN model, the CPN markings determination covering all execution paths is quite cumbersome. This paper proposes an automated BPMN verification framework that integrates the BPMN modeling tool and the CPN model checker together. The designed BPMN model is transformed into a CPN model and control flow graph (CFG). The CFG is used to create the execution paths and to find the interleaved activities. The interleaved activities are then considered for creating the CPN port places and markings by an applying of the branch coverage testing technique. Behaviors of the CPN model are analyzed by using a state space analysis based on the CPN model and automated markings. Our framework has been implemented as an Eclipse BPMN modeler plugin, and it is tested with the five case studies. The results show that our framework is practical. It can automate the CPN models from the BPMN model and guide the designers regarding the CPN markings determination to achieve branch coverage criteria.

Keywords: BPMN, Colored Petri net, Model checking, Formal verification, Software model.

ENGINEERING JOURNAL Volume 25 Issue 2
Received 27 May 2020
Accepted 29 January 2021
Published 28 February 2021
Online at https://engj.org/
DOI:10.4186/ej.2021.25.2.135

1. Introduction

The model-based design using BPMN primitives [1] is a graphical representation that has been widely used for portraying the software behaviors. BPMN has become the de-facto standard for specifying about operational control, signal processing and system communication. The model checking [2] using a colored Petri net [3-5] can be applied to prove the behaviors of BPMN model. The model checking procedures consist of the elaborate stages such as the CPN model abstraction, state space generation and state space analysis. Especially, the token determination or markings initiation on a CPN model requires an instruction during the verification process. It is an important stage to ensure that the designed BPMN model is exercised accordingly to the coverage criteria determined. These procedures become difficulty even if the modelers get familiar with the model checking tools. Because the isolation of BPMN modeling tool and model checker tool leads to a complicated configuration, for to agree with.

The BPMN modeling tool should be enable to design the BPMN model and to verify the model behaviors in the same tool in order to avoid the mentioned obstructions. And the tool should automate the CPN markings following the modeler needs. An automated system using meta-models and intermediate data mapping can be used for mapping the BPMN model into CPN models. For instance, control flow graph (CFG) [6] is used to be a meta-model between the BPMN model and the CPN target models because their graphs are isomorphic. In software testing disciplines, we can generate the test cases based on an execution path and execution scenario of BPMN model by using its CFG. Whereas a set of test data can be generated from the ground-truth data of an existing system.

Thus, we believe that the CFG and software testing techniques can reduce an operating cost and can make the verification easier for the BPMN model verification. The contribution of this work is to provide an alternative fashion and framework for the BPMN model verification archiving branch coverage criteria (BCC). A framework is implemented as an software extension running on Eclipse BPMN Modeler 2.0 [7] and CP4BPMN [8] APIs.

The organization of this paper is as follows. Section II describes the backgrounds of software testing technique and model checking technique. Section III reviews the related works. Section IV demonstrates the methodologies to execute each checking approach and discusses the its advantages and limitations. Section V is the study's conclusion.

2. Background

This section describes background of the BPMN model, software testing and model checking approach. We detail branch coverage testing, model checking using Petri net and colored Petri net that is the popular formal modeling languages for design and analysis a distributed system or concurrent system.

2.1. Business Process Model and Notation

Business Process Model and Notation [1, 9] or BPMN is the de-facto standard for business processes design. It provides a set of notations supporting the modeling of a business process in both the low-level design and high-level design. The notations of BPMN are extended from the elements of UML activity diagram, Flow chart, Data flow and BPM. In a low-level design or domain analysis, BPMN is used to describe the procedural logic of a system that consists of control flows, data flows, data objects. The advantages of BPMN model are a simple graphical representation and there are the BPMN modeling tools advocating an executable BPMN design. The core elements of BPMN are shown in Fig. 1, partitioned into the six groups as follows:

1) **Activities or tasks:** an activity in a BPMN model represents an action or a task that will performs or will be executed when its resources is ready. In an executable BPMN model, the task execution is controlled by an BPMN engine. The input data objects and output data objects of the task may or may not be defined, which they are the data constraints of the task.
2) **Events**: event nodes are separated into three main types: *Start*, *Intermediate*, and *End*. For representing the run-time catching or throwing of a system, the event node is determined to be the boundary event attached in a task or the grouped tasks called a sub-process.
3) **Gateways**: a gateway is used to be a control flow. The gateway can be used together with the guard condition expressions that are defined on the outgoing arcs to represent the branching and synchronization of process. The data-based gateway is that the data through the gateway, which they may come from the input or output data objects. Whereas the event-based gateway is used for branching a process only, by an event on the outgoing arcs of the branching event-based gateway wait for triggering of other tasks or performing of a time counter.
4) **Pool and lane:** a pool is used to partition a group of processes, grouped by the participants or organizations. While a lane is used for determining a scope of the process partitioned by the resource roles.
5) **Connecting elements:** a sequence of the task execution is determined by a connecting element called *Sequence flow*. The connecting element connecting between a task and data object is called *Association*, and an interaction crossing the pools is named *Massage flow*.
6) **Artifacts**: a data object is connected to the task by the association flow, representing the input and output data of task. If the communications of the processes cross the pools, the data object sent or data object received is represented by using a *Message* notation.

In the domain analysis, the procedural logic of a process is described in low-level abstraction. The resource constrains determined in the model are the input and output data objects, messages, and resource roles.

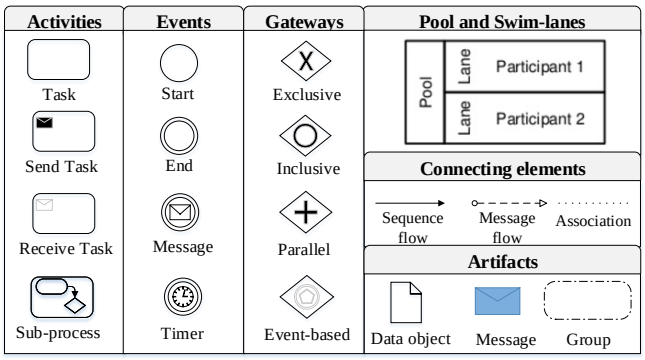


Fig. 1. Core elements of BPMN [9].

The operations of the model may rely on the variables in the resource constraints and other relevant data. Most BPMN modeling tools provide the available functionalities for simulating the model's behaviors. The executable BPMN model will be tested by running on an BPMN engine to execute the designed BPMN model and to monitor the variables during the model execution. However, there are certain limitations regarding an analysis the low-level designed model. The BPMN modeling tool provides the simple test functionalities, but they cannot find the specific part in the model such as an unreachable task and process termination. And they cannot be used to assert the model's behaviors related to the propositions qualified in term of time such as the invariant checking, loop, including the parallel execution. These behaviors can be verified in the model checking tool. Thus, there are many researchers [10-13] providing the approaches for analysis the specific properties of the BPMN model by using the computer aided verification (CAV) [14].

2.2. Properties of BPMN model

The desirable properties of BPMN model are that the model frees of anomalies causing a system crash, and the model behaviors must conform to the requirement specification. There are several mistakes that result in the unsatisfied behaviors, which are the results of the syntactic anomalies and structural anomalies.

- **Syntactic anomalies:** There are the two syntactic anomalies: incorrect usage and improper usage. The incorrect usage of BPMN elements is the main cause of syntactic anomalies. For example, the Start and End event is specified at an improper location or the model does not appear the Start event and End event. The improper usage is that the notation activities, gateways, connecting elements and pool are represented inconsistently in meaning such as the incorrect use of the User task having a sent message cross the pools. The User task must be replaced by the Sent task.
- **Structural anomalies:** the usage of incorrect gateway and uninterpretable conditions on the outgoing sequence flows of gateway is the main cause of structural anomalies. The incorrect gateway results in an undesirable property, starvation problem, deadlock, and infinite loop. The deadlock in a BPMN model occurs when an activity has more than one incoming sequence flows, but one of them does not provide a token to synchronize and perform an execution. an unreachable activity or dead activity is that the task's execution is impossible because the task is without an execution path from the Start event to itself. The incorrect use of parallel gateway may lead to a lack of synchronization as well. In this case, the BPMN model produces many process instances but they cannot complete the process execution because of the loose synchronization. The lack of synchronization may lead to deadlock problem and unsatisfied soundness property [15, 16], whereas the infinite loop is the result of the incorrect guard expression on the gateway.

2.3. Software Testing for BPMN design

Software testing can be applied to find the defects in BPMN model, which the BPMN model's behaviors are proved by an execution the BPMN model with the test cases. A test case is composed of the preconditions, test inputs, expected outputs, and postconditions. The number of the cases is based on the testing style and coverage criteria used. This paper applies white-box testing [17] with branch coverage criteria [18] to generate the token colors or markings for the CPN model.

The branch coverage or condition decision coverage is one of the test coverage metrics. The definition of branch coverage is that all outcomes of the control statements or decision nodes must be evaluated at least once, and every node in the model must be evaluated at least once as well. A sequence of the BPMN elements, the execution path is an important representation for considering whether a set of test cases relates the coverage metric used or not. The execution path can be generated from the CFG that corresponds to the given BPMN model. A control statement of the CFG is the node mapped from the BPMN gateway element.

2.4. Model checking using colored Petri net

Model checking is one of the computer aided verifications (CAV). A model checking tool usually provides both the verification mode and simulation mode for an analysis the model's property. To check the model properties, the property specification language such as Linear Temporal Logic (LTL) and Computational Temporal Logic (CTL) [19, 20] is used to describe an expected behavior. If the model holds the expected behavior, the tool reports the satisfaction. In contrast, the tool shows the violating states and their counterexamples.

Petri net [21] or classical Petri net is a mathematical modeling language for describing and checking the concurrency and distributed system. The Petri net-based languages are classified into two main groups: the low-level Petri net and high-level Petri net. A formal model described by a Petri net-based language is usually called "net model".

The core elements of Petri net are Place, Arc, Transition, and Token. A circle place represents a system's state, containing a discrete number of tokens.

DOI:10.4186/ej.2021.25.2.135

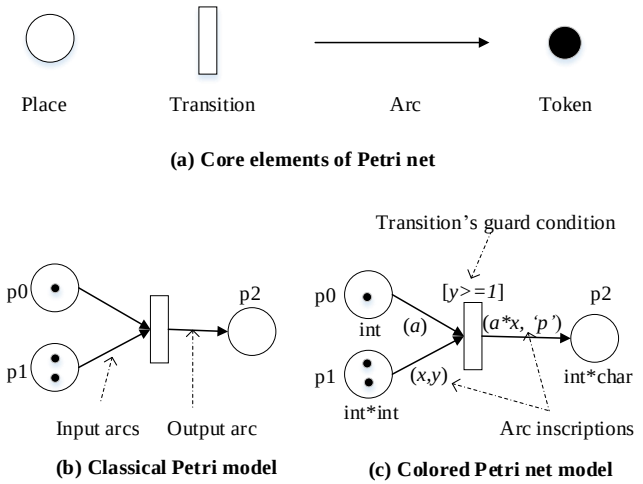


Fig. 2. The elements of the Petri net and CPN [22].

Between the place and squares transition is connected with a directed arc. The transition will fire the tokens if it has enough tokens on the input places, which the transition's firing represents a state change of a system. Figure 2 shows the core elements of Petri net and colored Petri net.

Colored Petri net or CPN [4, 22] is an enhancement of classical Petri net. It is one of high-level Petri net languages, combining a strength of Petri net and programing language. CPN preserves useful properties under the Petri net primitives and the extended formalism that allow the distinction between tokens, parameterization, and compact model. The primitive data type of CPN is called a "Color set", and the functional programming are useful attributes for a token distinction and a multiscale modeling. The color set can be determined as the place's type to be a domain constraint. It collects the token's value that is called a "Token color". The token(s) in the place must be consistent with the place type. Programming expression on the transition, arcs and data manipulation functions of the model is called a "Model inscription". The modelers can also configure the CPN model in hierarchical structure to help them to design the intricate model easier.

A labelled transition system of a net model, the reachability graph [23, 24] is a set of all possible cases derived from a performing or simulating the model's behaviors with the initial markings. It is sometimes called the state space graph that acts as the execution logs. The state space graph is used for a finding the undesirable states. The reachability graph of a CPN model is generated by the state space generator. Next, the desirable property expressed in term of the temporal logic formula is interpreted by the temporal logic parser, and the model checker finds the erroneous states in the state space graph. This technique is commonly known as the model checking using the state space analysis [25].

3. Related works

Over the past decade, many researches proposed the methods, testing tools, and frameworks for testing the BPMN model. Most researches emphasized a testing of the executable BPMN model running a BPMN engine plugin. Each research provided the alternative ways for generating the test cases based on the coverage criteria, and there are researches implementing the frameworks for monitoring the execution's results.

The work of [26], the test cases generation technique was proposed for a functional testing of the BPMN models. Test scripts are created form the execution paths of the model. This technique is the use of web-based testing tools named Selenium and Cucumber. This work does not support the low-level BPMN design because the data flows in the BPMN model are not be handled. Lübke et al. [27] proposed an applying the behavior driven development (BDD) and the process hub architecture for testing the business process model. The web service-based hub is implemented as an interaction controller. It addresses the model under test by the BPEL unit test suite. BDD is used to be a bridge communication among stakeholders by representing a BPMN process in a graphical Domain-specific language (DSL) in order to describe the context of BPMN model, and the test data and assertions of the model. Next, the context in the model was interpreted and tested in BPEL unit test suite. This is exercised with the black-box testing method, this technique is appropriate for functional testing. The works of [28, 29] proposed the automated frameworks with the BPMN suites Camunda [30] and GAmera [31] for the test case generation and test environment configuration. The frameworks reduce the time-consumption and be able to interactively monitor the model execution. These frameworks are valuable for the tester who are interested in the automated test programs.

Yotyawilai et .al [32] designed a tool for creating the test cases from a BPMN model. The obtained test cases are generated from the control flow graphs corresponding to the BPMN model, which the test cases satisfy the statement coverage criteria. Data objects and variables in the BPMN model were considered to create the test input data. The authors manually defined the boundary values of test inputs. Likewise, the work of [33] provided a tool for creating the test cases from BPMN model with BPEL model. The BPEL model acts as an implementation of a BPMN activity. All the elements of both BPMN and BPEL models are extracted, next they are used for creating the control flow graphs, test scenarios and test input data. The authors detailed and illustrated their framework with few case studies.

In the model checking area, the model checkers have been increasingly used for finding the mistakes in the BPMN model. Since the verification procedures is quite intricate, many researches provided the solutions and frameworks for reducing a complexity of each verification stage. Most of researches emphasized on the automated verification frameworks and the specific algorithms to alleviate the space explosion problem. Wang et al.[29] proposed the test case generation of the BPEL model. The authors used the CPN primitives for designing the process flows of BPMN model. The simple mapping rules of BPMN into the CPN model are provided, and the test scenario and test input data are generated from the consideration of BPMN control flow. The test scenarios

and test data are taken to be the corresponding markings of the CPN model.

Brumbulli et al. [34] provided the verification tool, a combination of OBP verification tool and the BPMN modeling tool. The message sequence chart and the property sequence chart are used to explore the goal states in the model. The model's properties are expressed in either LTL or Büchi automata form. The authors were focused on a communication verification, which the BPMN model under verification contains the pools and message flows. Flavio et al. [35] implemented a tool named BProVe for verifying the business process models. The authors extended the analysis feature for the task management and the task's state evolution, which the BPMN model is encoded and verified by using the rewriting logic with MAUDE. Although their framework is a precious strategy, it supports the checking of the control flow of the BPMN model only. The work of [8] proposed the Petri net-based verification framework for checking and analyzing the BPMN model. The authors detailed of the mapping rules of the BPMN elements into CPN models, and the state space techniques are illustrated. Their techniques and the tool support an analysis of the control flow and data flow in the BPMN model, but the tool cannot edit and simulate of the CPN model.

The work of [36] proposed the mapping rules of the BPMN model in to a formal model written in Event-B. Although Event-B with Rodin platform is a plaintext package, the expression of the BPMN structure and behavior is readable. This work handles both the control flow and data flow analysis. However, the verification using Event-B is appropriate for the modelers who expert in the mathematical proof. Yamasathien et al. [37] provided an approach to create the Promala model from the BPMN model. The mapping rules of the BPMN control flows into the formal model represented in the Promela programing language are provided. SPIN framework is used for an analysis of the BPMN model based on the business workflow patterns, but this work illustrates with few simple workflow patterns. Vincenz et al. [38] implemented the framework for verifying the control flows in the BPMN model. The framework is a colored Petri net-based application. The BPMN model is automatically translated relied on the Petri net meta-model and the mapping rules. The resource's properties and the user roles in the designed BPMN model are specified in CPN ML [39]. The CPN-ML properties can also be translated into the Promela models, and be verified by using SPIN model checker. These verification techniques are appropriate for the high-level abstraction model only because the authors did not concentrate on the data objects and data flows.

Due to the difficulty of the property expression in a temporal logic form, the work of [40] implemented the Graphical-CTL plug-in for the Eclipse-based application. The intermediate model is designed in the backend system process. The states of the BPMN model and the Graphical CTL specification are translated into the automata. The automata are taken to be the inputs of the model checking tool. The model checker returns the checking results back to the designing tool. This work is focused on an integration of the modeling tool and model checker only, and the tool does not allow the data flow verification. And there are many research studies [12, 41-44] that provided the formal model representations described in other formal languages. The representations are used in diverse verification frameworks such as UPPAAL [45], LoLA [46], ProM [47], NuSMV [48] and Eclipse BPMN modeler plugin. These works are precious strategy and viable alternative ways for the analysis of low-level abstraction BPMN model.

4. Automated framework of the BPMN model verification

Our verification framework is shown in Fig. 3. It is composed of the four core processes: 1) the transformation of the BPMN model into CPN model 2) the CFG generation and the creation of execution scenarios from the CFG 3) the generation of the CPN markings and 4) the state space generation and exploration. The formal definitions used for representing the relationships of the corresponding models are described in subsection 4.1. The details of our verification processes are in subsection 4.2 -4.6.

4.1. Formal definitions

Definition 1: a process model described by using BPMN is a tuple $BPMN^M = (Nt, At, Ft, Gw, Fw, Fd, Es, Ee, Ie, Sf, Fp, Vr, Do, Af, Lp, Ln, Mg, Mf)$ where:

Nt is a finite set of BPMN nodes.

At is a finite set of activities, $At \subseteq Nt$. For $ai \in At$, ai = (Name, Marker, refID), where ai.Name means the name of activity, and ai.Marker is the task's marker composed of Loop and Multi instance, and refID is the reference number that is automatically generated by the modeling tool.

Ft is a mapping function, Ft: $At \rightarrow$ {user task, service task, business rule task, manual task, undefined type}.

Gw is a finite set of gateways, $Gw \subseteq Nt$.

Fw is a mapping function used to indicate the gateway's type, Fw: $Gw \rightarrow$ {ex: exclusive, in: inclusive, pa: parallel, eb: event based, cx: complex}.

Fd is a mapping function used to indicate the direction of gateway, Fd: $Gw \rightarrow$ {div: divergent, con: convergent}. The gateway must be responsible either for the diverging or converging.

Es is a finite set of start events.

Ee is a finite set of end events.

Ie is a finite set of intermediate events.

Sf is a finite set of sequence flows, $(At \times At) \cup (Ie \times Ie) \cup (At \times Gw) \cup (Ie \times Gw)$. For the sequence flow that relates to the activity, the indegree and outdegree of the activity must be one.

DOI:10.4186/ej.2021.25.2.135

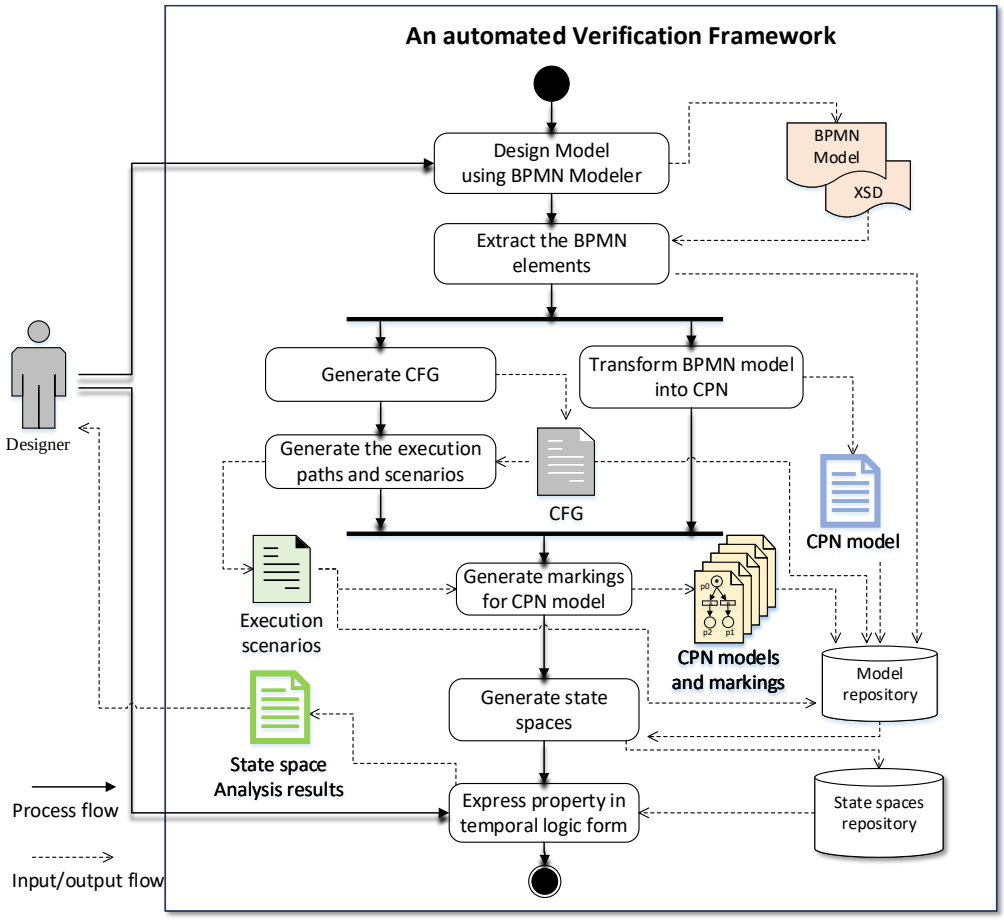


Fig. 3. Overview of our verification framework.

Fp is a labeling function that is used to indicates the condition expression on the outgoing sequence flow of a gateway, *Fp*: (*Gw* × *At*) → *"Guard label"*.

Vr is a set of variables.

Do is a set of input and output data objects of the activity.

Af is a set of association flows, (*At* × *Do*) (*At* × *Do*).

Lp is a set of pools.

Ln is a set of lanes.

Mg is a set of messages, in which the values are assigned and sent through the declared variables.

Mf is a set of message flows crossing between pools.

Fm is a mapping function that is used to indicate the message attached in the message flow, for *mg* ∈ *Mg*,

Fm: *Mf* → *mg*.

For *xi* ∈ (*At* ∪ *Gw*), the function *anc*() and *suc*(): *xi* → 2^{Nt}, which yields the ancestor nodes and the successor nodes, and the function *loc*() is used to indicate the lane and pool of the node *xi*.

Definition 2: Control flow graph is a 3-tuple, CFG^{M} = (*N*, *Ed*, *Fn*) where:

N is a finite set of nodes, for *ni* ∈ *N*, *ni* = (Name, OriginRef, SrcList, DesList, Lane, Pool), where *ni*.Name is the node's name that is mapped from the name of a BPMN activity. *ni*.OriginRef means the original BPMN reference ID, and SrcList, DesList are the list of the source and destination nodes of *ni*, while Pool and Lane indicate the node's locations, which Pool and Lane will be used in a creation of the CPN construct scope.

Ed is a set of edges, *Ed* ⊆ (*N* × *N*).

Fn: *N* → {Process-block, Decision, Conjunction, CoBegin, CoEnd}, is a node function to assign the type for each node. The CoBegin and CoEnd are additionally defined for mapping the parallel gateway and inclusive gateway.

Definition 3: CPN graph is a 9-tuple, Net^{G} = (*Pp*, *Tt*, *Aa*, *Fw*, *Cc*, *Vv*, *Fg*, *Fa*, *Fi*) where:

Pp is a finite set of places.

Tt is a finite set of transitions.

Aa is a set of arcs, (*Ss* × *Tt*) ∪ (*Tt* × *Ss*).

Fw: is a weight function to assign the weight to each arc, *Aa* → ℕ is a non-negative integer weight.

Cc is finite set of color sets.

Vv is a finite set of typed variables.

Fg is a guard labeling function used to assign the guard condition to each transition. *Fg*: *Tt* → "*conditional expression*" that the interpretation result *Fg*(*Tt*) is Boolean.

Fa is an arc expression function, *Fa*: *Aa* → "*arc expression*".

Fi is a marking initialization function, Fi: *Pp* → "*marking expression*".

4.2. Control flow graph creating

The BPMN model stored in XML format and its meta-data declared in XSD files will be extracted and transformed into a CFG and CPN model by using an XML parser and the extensions of the BPMN transformation rules that are implemented in CP4BPMN. We redesign the transformation rules and transformation process by a simultaneous automation of the CFG and the CPN model. Next, the CFG will be considered for generating the execution paths, snapshots, and scenarios. The transformation rules of the BPMN elements into CFG are as follows.

Rule No.1: If the BPMN model comprises multiple pools, the obtained CFG is the multiple sub-CFGs that are petitioned by such pools.

Rule No.2: (tasks and events transformation): For each BPMN activity $xi \in (At \cup Es \cup Ee \cup Ie)$, it is mapped to be the CFG node $ni \in N$. The name of the node xi and the node's properties comprising of the ancestor nodes, successor nodes, locations are defined by the model data that are extracted and stored in a model repository. And the type of the node ni will be determined as the Process-block.

Rule No.3: (gateways transformation): divergent exclusive gateway, event-based gateway and complex gateways are mapped into the Decision node. A convergent exclusive gateway is mapped into the Conjunction node. A divergent inclusive gateway and divergent parallel gateway are mapped into the CoBegin node. pWhile the convergent inclusive gateway and divergent parallel gateway are mapped into the CoEnd node.

Rule No.4: (sequence flow transformation): For each sequence flow $si \in Sf$, it is mapped into the edge of CFG.

Figure 4 shows an example BPMN model and the artifacts derived from our transformation process. The CFG shown in Fig. 4 (b) will be taken to explain how to automate the execution paths, scenarios, and net markings.

4.3. Execution Paths Creating

As the CFG in Fig. 4 (b), it can be observed that its structure and the BPMN graph are isomorphic. We apply the path testing technique with the consideration of DD-paths and branch coverage criteria to generate the possible execution paths from the CFG. Algorithm for generating the execution paths is listed in Algorithm 1. The algorithm supports the BPMN models designed in the single process and multiple processes, which the processes are portioned by the pools. The execution paths will be computed based on the branch coverage criteria. On line numbers 7 to 10, the core execution path is constructed by using the backward graph traversal algorithm, which the end node in a pool is retrieved at first, and the consecutive ancestor nodes of the end node be continuously traversed until the ancestor nodes is the Start node. The core execution path stored in a list is taken to be the first path for creating other paths. The decision nodes in each path are forward traversed from the start node. On line numbers 12 to 20, the decision nodes in the considered execution path are pointed out, and the node sequence from the start node

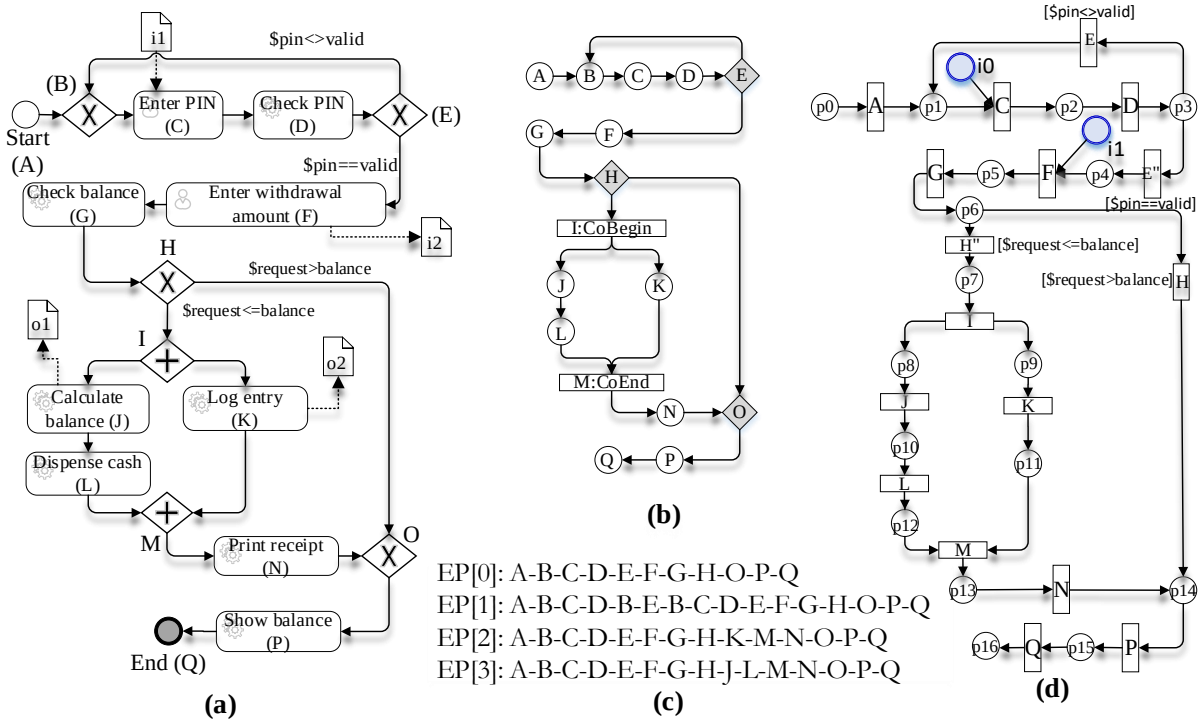


Fig. 4. An example of the ordinary BPMN model and its graphs derived from applying the transformation rules. (a) The cash withdrawal process [49] described by BPMN; (b) the CFG of the model in (a); (c) The execution paths of the CFG in (b) with the branch coverage consideration; (d) The CPN model derived from the transformation of the BPMN model in (a).

to such decision node is duplicated to be the execution sequence of the new path. On line number 14, all the outgoing edges of decision node are traversed throughout the end node. The number of execution paths will be increased if the number of the outgoing edges and successor node that does not exist in the earlier execution paths. If all the outgoing edges of decision node is considered already, the decision node must be flagged to be passed, and the next decision node will be considered. After all the decision nodes in the considered execution path are traversed, the path will be flagged to be the completely considered execution path. The execution paths of applying the algorithm to CFG in Fig.4 (b) are shown as Fig.4 (c). The node sequence in each execution path will be used to generate the CPN markings. The marking generation technique is described in subsection 4.6.

However, the derived execution paths may have the tasks that involves the parallel process block. As the execution paths EP[2] and EP[3] in Fig.4 (c). They have the parallel block containing the task J, K and L, all of them will be considered as the sub-execution path. The tasks in the sub-execution are grouped into one node. Thus, the paths EP[2] and EP[3] will be reduced in to one execution path because they have the same node sequence.

4.4. CPN model generating

The CP4BPMN tool can be used to map nearly all BPMN elements into the classical Petri net and CPN model. The details of the transformation rules of CP4BPMN are described in [8]. However, the automated verification process requires the port places and markings. They represent the interruption of the users or the outside services to drive the verification process. We extend the transformation rules by adding the input place to be the port place into a CPN construct. This technique is to represent an interleaving of the user or service task. The extended transformation rules are detailed as follows.

1) For each BPMN activity ai ∈ At and Ft(ai) → {user task, manual task}, the CPN place representing the interleaved state is added into the CPN construct. The added place is a vital place for containing the test input data or the marking to drive the model execution. We call the added place as a "port place" The data type or color set of a place will be mimicked from the XML meta-data or the input data object of task. Figure 5 illustrates the CPN construct and its port place highlighted in blue.

2) For each receive task, the port place is added and connects to the corresponding transition. The color set of the port pace is mimicked from the data object attached on the incoming message flow. While the variables in the

Algorithm 1: Generating the execution paths form CFG

```
01 Require: CFG and coverage criteria {branch-coverage}
02 Ensure: CFG is generated from a well-defined BPMN model.
03  if CFG is a single CFG then set PL: =1 End If
04   EP: = {}/* array list for collecting and pruning the execution paths*/
05     If PL > 0 then
06       For swl ∈ PL
07           While OriginRef(n) ≠ Es
08               List: = List ∪ n /*Create the core execution path by backward traversal from the End nod to Start node. */
09           End While
10            EP[0]: = List.reOrdering(Desc); EP: = EP ∪ {0};  /* descending sort order */
11           Do p:EP ≠ ∅; i=1
12              Do findUnflagDecitionNode(EP[p])≠ ∅
13                  n:= EP[p][x]   /* x is position of the Decision node */
14                   For suc(n) that is not in EP /* loop based on the number of the outgoing edges*/
15                      EP[i]: ClonedList: = getList(SC[p][0], n) /* Clone the list from the Start node to such Decision node */
16                      EP[i]: = EP[i] ∪ the successor nodes starting from decision node n to the End node.
17                      EP ∪ {i}; i++
18                    Next
19                    EP[p][n] is flagged to be a passed node, because its outgoing edges have been traversed
20                End Do
21                EP: = EP \p /*Flag the execution path is passed because all decision nodes
22                and their outgoing edges have been traversed already */
23             End Do
24             Next
25          End If
25    End If
27  Return EP
```

input data of the task and the data sent form the ancestor task will be composed and expressed on the input arc of the transition. Figure 6 shows the CPN construct derived from the transformation of the receive task.

3) To map the Service task, we apply the existing rules by replacing the output place of the corresponding transition with the port place. Generally, the output arc carries out the token colors to the output place. The replaced port place will be used for a manual manipulation the token colors, which the token colors manipulated represents the results of the service task. This technique is crucial for the dummy service. Figure 7 shows the CPN construct derived from the service task transformation.

4) The transformation rule of the Intermediate catch event is revised by the port place that represents a catching of the data objects or of the triggering signal. The data object in a message will be mimicked to be the color set of the port place. The variables of other data objects (*) that are sent from the ancestor task are extracted and expressed on the input arc of the transition. Figure 8 shows the CPN construct derived from the intermediate catch event transformation.

4.5. CPN markings creating

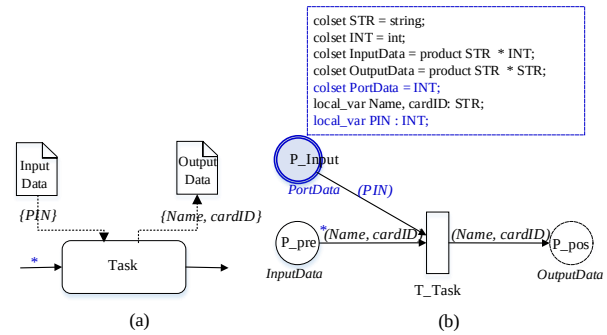
This section details about the CPN marking determination based on the obtained execution paths, CPN model, and the test data. The markings are useful for the data-driven verification. The marking will be entered into a CPN model before verification. The input data for generating the marking are usually recorded in a model repository or an excel file. We call the CPN model and its markings as the "CPN instance". Next, we use the model checking tool to generate the state space graph of each CPN instance, and integrate the obtained state space graphs together. The details of the CPN instance generation and CPN markings are composed of three steps. We exercise all these steps in each execution paths. The processes of the CPN instance generation are as follows:

1) Choose an execution path.

2) Find the segments in the chosen execution path in order to build the verification scenarios and snapshots.

3) Determine the markings on the beginning input place and the port places for each CPN instance.

Let's consider the execution paths and CPN model in Fig. 4 (c). The execution path EP[0] is selected to be the CPN instance at first, in which the activities sequence and the activity's type are used to determine the segments or snapshots. The nodes sequence in the execution path EP[0] is A-B-C-D-E-F-G-H-O-P-Q. The nodes C and F come from the BPMN user task. The nodes C and F will be determined as a cut-point of the segment. Thus, the CPN instance of the execution path EP[0] consists of three scenarios: 1) A-B, 2) C-D-E, and 3) F-G-H-O-P-Q. Next, the token colors will be assigned in the CPN constructs that possess the port places. The marking at the cut-points of each segment is called the "snapshot". Figure 9 illustrates the CPN instance and its snapshots.



Legend: 1) This rule is applied to the BPMN User task and Manual task
2) *Name, cardID* on input arc in (b) are the example data received from an ancestor task

Fig. 5. Transformation rule of the user task and manual task.

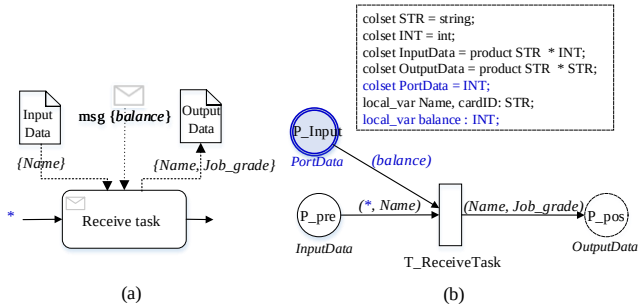
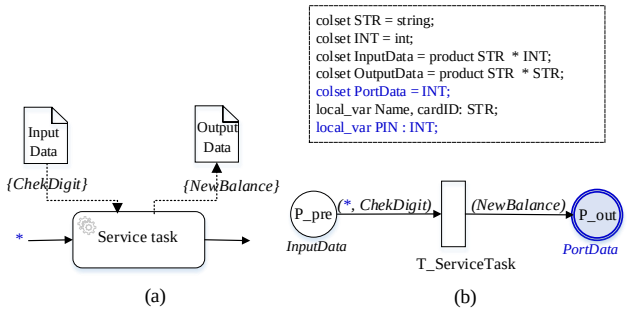


Fig. 6. Transformation rule of the receive task.



Legend: the existing rule and extended rule give the same CPN structure but the new rule sets the output place as the port place.

Fig. 7. Transformation rule of the service task.

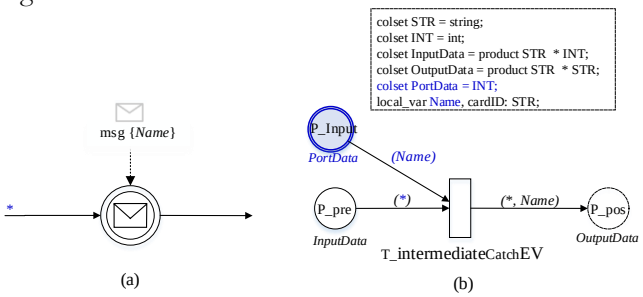


Fig. 8. Transformation rule of the intermediate catch event.

An example of a mapping the input data into a CPN marking is: the test data or input data of the BPMN user task are the customer name and his salary with the string "Jonh" and integer 15500 respectively. These information are mapped into the marking "p0:1^("JOHN", 15500);" where p0 is the CPN place containing the token, and 1^ is the number of the tokens determined by 1. The expression "JOHN", 15500 in the brackets is the token colors, which

the quotes "" is the string data type and the undefined quote is the numerical data.
As the mentioned marking, it indicates that the color set of the place p1 is the production of the primitive data types *String* × *Integer*.

As the snapshots in Fig. 9, the state space generation will be performed with the snapshot 0-1, 0-2 and 0-3 consecutively. The place p0 of the scenario 0-1 is the beginning place derived from the BPMN start event. The place p0 represents the process's initiation and the place p1 represents the idle state that waits for receiving the triggered signal or for performing the first activity. Due to the BPMN start event without the input data object configuration, the place p0 is assigned with one token by the color "unit" that is the undefined data type. The snapshot 0-2 requires one token for the place p1 which the token color is the result of the transition firing of the transition "A". The snapshot 0-2 also requires one token in the place i0 which represents the state that the input data of the BPMN task C are entered, and the task C is ready to perform an action.

To control the verification direction based on the chosen scenario, we have to interpret the guard conditions of transition on the outgoing sequence flow of the gateway. For instance, the place p3 and the transitions E and E″ is the CPN construct mapped from the branching BPMN exclusive gateway E in Fig. 4 (a). The tasks sequence in the snapshot 0-2 that the execution flow must pass the node E and the node F. So that, the token color assigned on the places p1 and i0 will drive an execution through the transition E″ with the guard condition "$pin==valid". It means that the input place p1 requires a marking that represents the entered valid PIN, because this part is the CPN construct mapped from the BPMN task named Enter PIN. While the snapshot 0-3 likes the snapshot 0-2, which the CPN construct of the node F requires one token on the places i1 and p4 (transformed from the task Enter withdrawal amount). The place i1 requests the token color in case of the entered request amount more than the balance amount. The place p4 contains one token that is the result of the transition firing of the transition E″.

The execution paths EP[2] and EP[3] in Fig. 4 (c) is composed into one execution path because they share the parallel process block. The tasks sequence in the execution paths (except the tasks in the parallel block) is in the same sequence. The tasks in the parallel process block can be verified by Petri net-based model checkers because the typical characteristics of Petri net support the concurrent process analysis. If all the execution paths and scenarios listed in Table 1 are performed completely, it guarantees that all branches and nodes in the BPMN graph in Fig. 4 (a) are executed at least once.

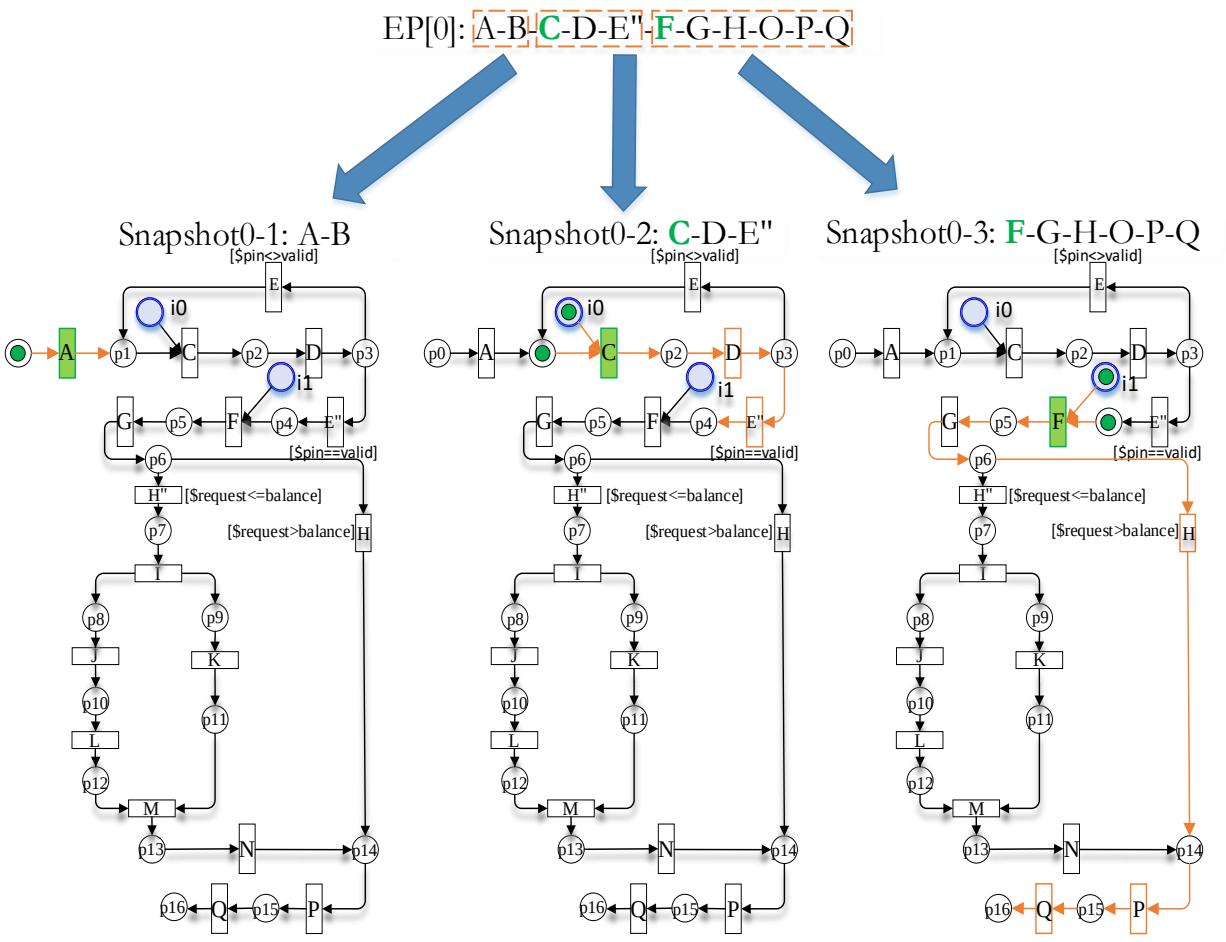


Fig. 9. The example of the CPN instance and its snapshots with markings.

Table 1. the verification scenarios and markings of the CPN model based on the execution paths listed in Fig. 4 (c).

Execution Path	Snapshots	Scenarios	Descriptions/ Constraints	Markings
EP[0]: A-B-C-D-E″-F-G-H-O-P-Q	0-1 0-2 0-3	A-B C-D-E″ F-G-H-O-P-Q	Process initiation. Enter a valid PIN. Enter a withdrawal amount more than the balance amount.	p0:1^(); p1:1^(); i0:1^("$pin"); p4:1^("*"); i1:1^("$amount");
EP[1]: A-B-C-D-B-E-B-C-D- E″-F-G-H-O-P-Q	1-1 1-2 1-3 1-4	A-B C-D-E-B C-D-E″ F-G-H-O-P-Q	Process initiation. Enter an invalid PIN. Enter a valid PIN. Enter a withdrawal amount more than the balance amount. ($amount>balance)	p0:1^(); p1:1^(); i0:1^("$pin"); p1:1^(); i0:1^("$pin"); p4:1^("*"); i1:1^("$amount");
EP[2]: A-B-C-D-B-E″- F-G-H″-I-J+-K+-L+-M-N-O-P-Q	2-1 2-2 2-3	A-B C-D-E″ F-G-H″-I-J+-K+-L+-M-N-O-P-Q	Process initiation. Enter a valid PIN. Enter a withdrawal amount lease than or equal to the balance amount. ($amount<=balance)	p0:1^(); p1:1^(); i0:1^("$pin"); p4:1^("*"); i1:1^("$amount");

The variables $pin and $amount are the token color that must be assigned by a value conforming to the place constraints. * is the expression of the token's color produced by the prior transition(s). + is the task in the parallel block, which it can be executed simultaneously with other tasks in the same parallel block.

4.6. The state space generation and exploration

The obtained CPN model, snapshots and markings will be used in the process of state space graph generation. This process uses the CPN model checker of CP4BPMN. The CPN model and its markings are read from a model repository one by one snapshot. The model checker computes all the possible states of the CPN model based on the markings determined. If the nodes in the scenario is the same sequence, the model checker computes a state space graph only once. Next, the obtained state space graphs of each snapshot will be connected to the other state space graphs into the whole state space graph. The binding element on an edge between the state space graphs represents an interleaved state of the user task that the input data is fed into such user task by the user.

Figure 11 shows the excerpt state space graph of the CPN in Fig. 4 (d). The blue state nodes are the root node of state space graph whereas the yellow nodes are the leaf nodes. The state space graph of adjacent snapshots is connected to each other. The biding element between the leaf nodes of a previous snapshot and the root node of a next snapshot are the token colors that are fed into a CPN model while the state space creation. For example, Figure 11 (e) is the connected state space graph which is a combination of the state spaces graphs derived from the snapshots 0-1, 0-2 and the snapshot 1-2. The snapshots 0-1 and 0-2 are in the execution path EP[0], while the snapshot 1-2 is in execution path EP[1]. Thus, the biding element of the state space graph of the snapshot 1-2 as Fig. 11 (d) takes place at between the state node s1 and node s10 in Fig. 11 (e). It represents the case that the user enters an invalid PIN with 000000. Likewise, the biding element connecting the state node s1 and node s2 illustrates a case the user fills in a valid PIN with 595023.

In case of the state space generator cannot proceed the state space for the next snapshots, the dead marking occurs, the state space generator process will be aborted because the obtained state space graph will cannot be integrated with other state space graphs. For example, if the transition A cannot fire the token into the place p1 that is one of the input places of the transition C in snapshot 0-2, the snapshot 0-2 not be selected paused into the state space generation process and the process is aborted.

After the whole state space graph is generated and stored in CP4BPMN, the desirable properties are expressed in CTL form to explore the goal state in the state space graph. The CP4BPMN tool provides functional commands to express the desirable properties. For instance, the command "unreach()" is used to find the unreachable task. The command "AG(q)" is used to check the invariant properties; it means that state q can persists indefinitely.

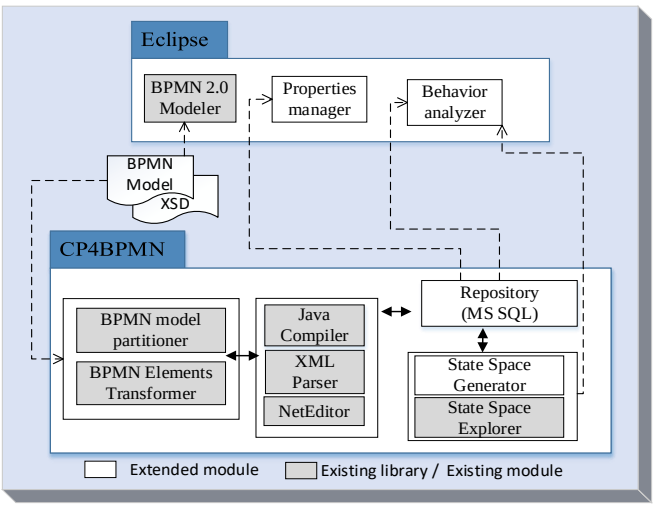


Fig. 10. An architecture of our BPMN Plugin.

DOI:10.4186/ej.2021.25.2.135

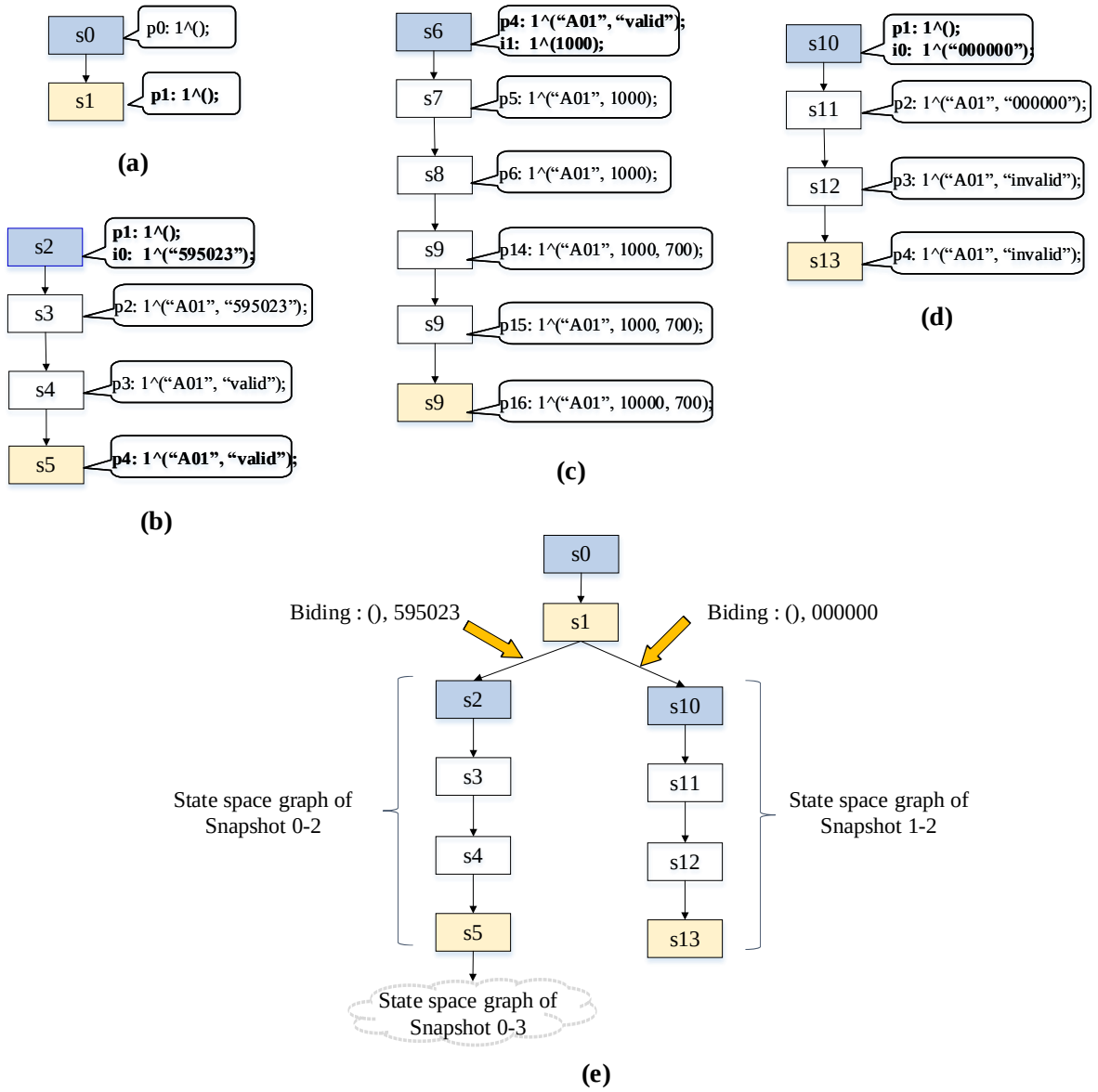


Fig. 11. The excerpt state space graph of the CPN in Fig.4 (d); (a) the state space graph of Snapshot 0-1,1-1 and 2-1; (b) the state space graph of Snapshot 0-2; (c) the state space graph of Snapshot 0-3; (d) the state space graph of Snapshot 1-2; (e) the connected state space graph.

5. Implementation and Experimental Results

We design the system architecture, and prototype the proposed framework to be the BPMN 2.0 modeler Plug-ins running in Eclipse environment. The architecture of application is shown in Fig. 10.

The modelers can design the BPMN model on Eclipse BPMN 2.0 modeler arbitrarily. Next, the XML BPMN model and meta-data declared in XSD files will be labeled with the version number. They are read by the XML parser of CP4BPMN to extract the BPMN elements in the BPMN model. All the obtained BPMN elements are stored in the model repository implemented using MS-SQL. Next, the obtained BPMN elements are transformed into the CPN models and CFGs. The property manager provides the user interfaces for determining the model's properties and managing the color sets, including the test input data for a creation of the markings of CPN model. The behavior analyzer is provided for generating and exploring the state space graph. Its communication interfaces use the Restful APIs provided by CP4BPMN. A screenshot of the BPMN Plug-in is shown in Fig. 12.

We test our framework by applying to the existing models. They are composed of five BPMN models designed in different domains as follows.

1) Hardware retail (HWR).
2) Requirements change process (RCP).
3) Real estate checking process of the mortgage loan system (ECP).
4) Disbursement web-service (DWS).
5) Incident report generator system (IRG).

The results of applying the framework are detailed in Table 2. We define the principles of correctness checking

of the BPMN model transformation, and prescribe the measurement of the tool capacity as follows:

1) Structural equivalence checking

The structural equivalence of the BPMN graphs, CFGs and CPN models are verified by an applying the directed graph isomorphism algorithm [50] (implemented by JAVA). In the consistency checking process, the number of nodes in the CFG and CPN model is verified. It is a back tracking to the source BPMN elements. If there exist in the BPMN graph but they do not appear in the CFG and CPN graph, the framework reports and points-out the cause of inconsistency problems. As the experimental results, the framework can detect the system causes and user mistakes resulting in the structural inequivalence correctly.

2) Behavioral equivalence checking

We verify the behaviors of CPN constructs that are mapped from the BPMN user task, manual task, and service task. We manually design and verify such CPN construct by using CPN tool with one-by-one mapping rule. This technique makes sure that each CPN construct is correct. Next, the overview perspective and specific properties are verified by using our Plugin. As the statistical report detailed in Table 2, our framework supports the designing of both the control flows and data flows, and can be used to verify basic behaviors including loop and parallel executions. The infinite loop and concurrent executions cannot be validated by using straightforward the ordinary software testing techniques.

We check the equivalence between CPN model and BPMN model with the specific properties by comparing their outputs. Normally, the behaviors of the CPN model must conform to behaviors of the designed BPMN. The outputs of BPMN model are derived from an execution the model by a BPMN engine, and the outputs of CPN model come from the use our framework. To configure the test environment, the deployable artifacts of BPMN model are generated and deployed in the enterprise integrator. The modelers can simulate the BPMN model running on the environment configured, and the model checker runs on our verification framework.

Let's consider the behavioral equivalence checking of the ATM process in Fig. 4 (a), the test data for executing the BPMN and the token colors of the CPN model must be determined with the same value. The test data with a valid PIN 595023 is determined at the task "Enter PIN". Whereas the CPN model must be take place by a marking on the port place of the transition mapped from the BPMN task "Enter PIN". Thus, the place i0 of the CPN model in Fig. 4 (d) must be the token color with 595023

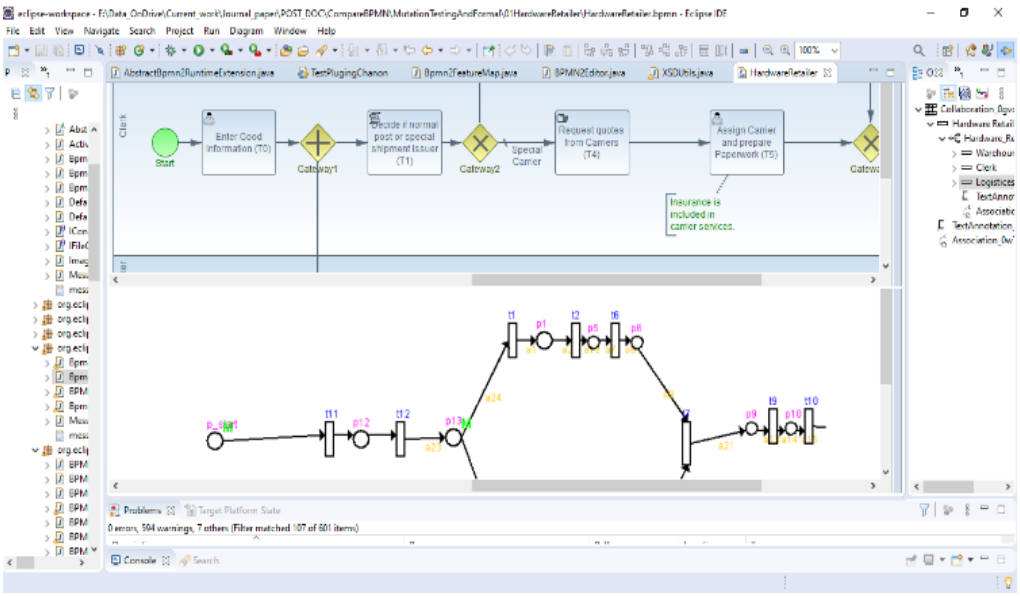


Fig. 12. Screenshot of BPMN Plugin.

Table 2. Results of applying the framework to existing models.

Model	CPN						CFG		State space		
	Col.	Pla.	Tran.	Parallel.	Loop	Time^{A}	Paths	Snapshots	Nodes	Edges	Time^{S}
HWR	7	31	24	0	1	2.32	8	27	66	59	1.02
RCP	4	48	41	2	1	5.41	14	42	73	63	3.54
ECP	6	56	46	1	2	5.01	9	24	81	75	2.48
DWS	5	52	40	1	2	5.29	11	31	78	62	2.30
IRG	12	47	39	1	1	4.37	22	33	79	69	2.12

Col. = Color sets; Pla. = Places; Tran. ='Transitions; Parallel = the number of parallel blocks in BPMN model; Loop= the number of loops in BPMN model; Time^{A} = Time (seconds) used in a process of CPN and CFG generations; Time^{S} = Time used for generating the state spaces of all snapshots including the time used in the state space integration.

as well. The outputs of both models must be the same, by the BPMN engine return $pin= "valid" and the CPN model checker must produce one token on the place p4. (the place p4 represents the state of invalid PIN). However, the simulation by using the BPMN engine still faces with the monitoring drawback. Because the BPMN modeling framework does not support the monitoring of execution paths and variables, and cannot verify the parallel execution.

3) Performance measurement

As the experimental results in Table 2, the CPN automation process is increasingly time consuming because of the extended consistency checking algorithm and graph isomorphism algorithm. Thus, the time used to generate a CPN model and CFG relates to the model size, complexity, and the number of elements and the meta-data of BPMN model

In the state space generation, we observe that the time and memory used for the state spaces integration is increasing by approximately 3 percent. As the statistical report described in Table 2, we found that the parallel block results in the time used for mapping the CPN model and generating the state space graph. The number of execution paths and snapshots do not affect the generating times if the modelers prepare the CPN markings properly, or the CPN model has a small number of the interleaved tasks.

In the state space exploration, the framework enables to express the desirable properties in CTL. The proposed Eclipse Plugin integrates the BPMN modeling tool and the model checking tool CP4BPMN together. Since the extension of the state space graph has been implemented with the same data structures. Thus, the functional commands still work well. The Plugin also enables to analyze the CPN model with the arbitrarily marking determination.

6. Conclusions and Future Work

There are numerous frameworks and tools for verifying the designed BPMN model but most of the BPMN modeling tools and verification mechanisms are isolated. The CPN primitives can be used to represent and verify the behaviors of BPMN model. Although the corresponding CPN constructs can be automatically generated from the BPMN elements, the obtained model needs the CPN markings for simulating and analysis the model's behaviors. All the execution parts in the BPMN model should be verified, and the model should free of the undesirable properties. And the markings determination should cover all the branches of the gateway in the BPMN model.

We proposed an automated CPN-based framework for verifying a design BPMN model. We extended the transformation rules of the User task, Manual task, and Service task. The control flow graph or CFG and branch coverage criteria are applied for the automation of the execution paths and the CPN's markings. The CFG is an intermediate model for considering the execution paths, snapshots, and scenarios used to determine the verification direction. The CPN markings obtained from a set of the input data can convey the verification direction achieving the branch coverage criteria. The obtained CPN markings can be ensured that the execution exercise all the execution parts. To generate the state space graph, the CPN markings are automatically fed into the CPN model based on the snapshot derived from the CFG. Next, the modelers can explore the model's behaviors from the state space graph by an expressing the goal states written in CTL form. These techniques have been implemented as a BPMN modeler plugin running on Eclipse environments. It interfaces with the CP4BPMN verification tool through APIs.

The experimental results show that our proposed technique and framework are practical. They pave the automated verification which is the integration of the BPMN modeling tool and the model checker mechanisms together. However, our framework does not support the parallel block that contains a User task or Manual task. Because they produce many CPN port places and lead to more interruptions that occur in the state space generation stage. Our ongoing work is directed towards to implement a graphic user interface for editing and animating the obtained CPN models. And we would extend the process of the CPN markings determination that accomplishes the condition coverage criteria.

Acknowledgement

C. Dechsupa would like to acknowledge Chulalongkorn University graduate school for financial support by the Postdoctoral Fellowship Ratchadaphiseksomphot Endowment Fund.

References

[1] O. M. Group, *OMG Unified Modeling Language TM (OMG UML) Version 2.5*, 2015.

[2] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT press, 2008.

[3] K. Jensen, "An introduction to the theoretical aspects of coloured Petri net," presented at *The Book Series Lecture Notes in Computer Science (LNCS)*, 2005.

[4] L. M. K. Kurt Jensen, *Coloured Petri Nets. Modelling and Validation of Concurrent Systems*. University of Aarhus, 2009.

[5] K. Jensen, L. M. Kristensen, and L. Wells, "Coloured Petri nets and CPN tools for modelling and validation of concurrent systems," *International Journal on STTT,* vol. 9, no. 3-4, pp. 213-254, 2007.

[6] A. G. Gounares and C. D. Garrett, "Control flow graph driven operating system," U.S. Patent Application No. 13/463,844, 2012.

[7] *Eclipse BPMN Modeler*. [Online]. Available: http://www.eclipse.org/bpmn

[8] C. Dechsupa, W. Vatanawood, and A. Thongtak, "Hierarchical verification for the BPMN design model using state space analysis," *IEEE Access,* vol. 7, pp. 16795-16815, 2019.

[9] Object Management Group, *Model, Business Process Notation (BPMN) version 2.0*. Object Management Group, 2011.

[10] M. Zäuram, "Business process simulation using coloured Petri nets," M.S. , Dept. Math. Com. Sci., Univ. Tartu, Tartu, Estonia, 2010.

[11] L. E. Mendoza Morales, "Business process verification: The application of model checking and timed automata," *CLEI Electron. J.,* vol. 17, p. 3, 2014.

[12] R. M. Dijkman, M. Dumas, and C. Ouyang, "Formal semantics and analysis of BPMN process models using Petri nets," Tech. Rep, Queensland Uni.of Tech., 2007.

[13] A. Krishna, P. Poizat, and G. Salaün, "VBPMN: Automated verification of BPMN processes (Tool Paper)," in *International Conference on Integrated Formal Methods*, Springer, Cham, 2017, vol. 10510, pp. 323-331.

[14] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 2014.

[15] W. M. Van Der Aalst *et al.*, "Soundness of workflow nets: classification, decidability, and analysis," *Formal Aspects of Computing,* vol. 23, no. 3, pp. 333-363, 2011.

[16] H. Groefsema and D. Bucur, "A survey of formal business process verification: From soundness to variability," in *International Symposium on Business Modeling and Software Design*, 2013, pp. 198-203.

[17] S. Nidhra and J. Dondeti, "Black box and white box testing techniques-a literature review," *International Journal of IJESA,* vol. 2, no. 2, pp. 29-50, 2012.

[18] N. Gupta, A. P. Mathur, and M. L. Soffa, "Generating test data for branch coverage," in *Proceedings ASE 2000*, IEEE, 2000, pp. 219-227.

[19] M. Fisher, *An Introduction to Practical Formal Methods Using Temporal Logic*. John Wiley & Sons, 2011.

[20] P. T. Monteiro, D. Ropers, R. Mateescu, A. T. Freitas, and H. De Jong, "Temporal logic patterns for querying dynamic models of cellular interaction networks," *Bioinformatics,* vol. 24, no. 16, 2008.

[21] W. Reisig, *A Primer in Petri Net Design*. Springer Science & Business Media, 2012.

[22] K. Jensen, "Coloured Petri nets: Basic concepts, analysis methods and practical use," *DAIMI Report Series,* vol. 37, no. 588, 2008, doi: 10.7146/dpb.v37i588.7188.

[23] X. Ye, J. Zhou, and X. Song, "On reachability graphs of Petri nets," *Computers & Electrical Engineering,* vol. 29, no. 2, pp. 263-272, 2003.

[24] P. Darondeau, S. Demri, R. Meyer, and C. Morvan, "Petri net reachability graphs: Decidability status of first order properties," *arXiv preprint,* arXiv:1210.2972, 2012.

[25] L. M. Kristensen, "State space methods for coloured Petri nets," Ph.D. Degree, Department of Computer Science, University of Aarhus, Denmark, 2000.

[26] J. L. de Moura, A. S. Charão, J. C. D. Lima, and B. de Oliveira Stein, "Test case generation from BPMN models for automated testing of Web-based BPM applications," in *2017 17th (ICCSA)*, IEEE, 2017, pp. 1-7.

[27] D. Lübke and T. van Lessen, "Modeling test cases in BPMN for behavior-driven development," *IEEE Software,* vol. 33, no. 5, pp. 15-21, 2016.

[28] C. Ngambenchawong and T. Suwannasart, "A Weak mutation testing framework for BPMN," in *Proceedings of the International MultiConference of Engineers and Computer Scientists 2019, IMECS 2019*, Hong Kong, March 13-15, 2019.

[29] Y. Wang and N. Yang, "Test case generation of web service composition based on CP-nets," *JSW,* vol. 9, no. 3, pp. 589-595, 2014.

[30] A. Fernandez, "Camunda BPM platform loan assessment process lab," Brisbane, Australia: Queensland University of Technology, 2013.

[31] J.-J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo, "GAmera: A tool for WS-BPEL composition testing using mutation analysis," in *International Conference on Web Engineering*, Springer, 2010, pp. 490-493.

[32] P. Yotyawilai and T. Suwannasart, "Design of a tool for generating test cases from BPMN," in *2014 International Conference on Data and Software Engineering (ICODSE)*, IEEE, 2014, pp. 1-6.

[33] C. Nonchot and T. Suwannasart, "A tool for generating test case from BPMN diagram with a BPEL diagram," in *Proceedings of the IMECS 2016*, 2016, vol. 1.

[34] E. Solaiman, W. Sun, and C. Molina-Jimenez, "A tool for the automatic verification of bpmn choreographies," in *2015 IEEE SCC*, IEEE, 2015, pp. 728-735.

[35] F. Corradini, F. Fornari, A. Polini, B. Re, F. Tiezzi, and A. Vandin, "BProVe: A formal verification framework for business process models," in *2017 32nd IEEE/ACM ASE*, IEEE, 2017, pp. 217-228.

[36] J. W. Bryans and W. Wei, "Formal analysis of BPMN models using Event-B," in *International Workshop on Formal Methods for Industrial Critical Systems*, Springer, 2010, pp. 33-49.

[37] S. Yamasathien and W. Vatanawood, "An approach to construct formal model of business process model from BPMN workflow patterns," in *2014 4th International Conference on DICTAP2014 (DICTAP)*, IEEE, 2014, pp. 211-215.

[38] C. Wolter, P. Miseldine, and C. Meinel, "Verification of business process entailment constraints using SPIN," in *International Symposium on Engineering Secure Software and Systems*, Springer, 2009, pp. 1-15.

[39] K. Jensen and L. M. Kristensen, "CPN ML programming," in *Coloured Petri Nets*. Springer, 2009, pp. 43-77.

[40] A. Speck, S. Witt, S. Feja, A. Lotyzc, and E. Pulvermüller, "Framework for business process verification," in *International Conference on BIS*, Springer, 2011, pp. 50-61.

[41] S. P. Silvia von Stackelberg, J. Mülle, and K. Böhm, "Detecting data-flow errors in BPMN 2.0," *OJIS,* vol. 1, no. 2, pp. 1-19, 2014.

[42] A. Suchenia, P. Wiśniewski, and A. Ligęza, "Overview of verification tools for business process models," *Annals of Computer Science and Information Systems,* vol. 13, pp. 295-302, 2017.

[43] W. M. van der Aalst, H. De Beer, and B. F. van Dongen, "Process mining and verification of properties: An approach based on temporal logic," in *OTM, Confederated International Conferences "On the Move to Meaningful Internet Systems,"* Springer, 2005, pp. 130-147.

[44] O. Allani and S. A. Ghannouchi, "Verification of BPMN 2.0 process models: an event log-based approach," *Procedia Computer Science,* vol. 100, pp. 1064-1070, 2016.

[45] *UPPAAL*. Uppsala University. Accessed: May 19, 2020. [Online]. Available: http://www.uppaal.org/

[46] K. Schmidt, "Lola a low level analyser," in *International Conference on Application and Theory of Petri Nets*, Springer, 2000, pp. 465-474.

[47] B. F. Van Dongen, A. K. A. de Medeiros, H. Verbeek, A. Weijters, and W. M. van Der Aalst, "The ProM framework: A new era in process mining tool support," in *International Conference on Application and Theory of Petri Nets*, Springer, 2005, pp. 444-454.

[48] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model verifier," in *International Conference on Computer Aided Verification*, Springer, 1999, pp. 495-499.

[49] N. Bansal and N. Singla, "Cash withdrawal from ATM machine using mobile banking," in *2016 ICCTICT*, IEEE, 2016, pp. 535-539.

[50] R. J. Ullmann, "An algorithm for subgraph isomorphism." *Journal of the ACM*, vol. 23, no. 1, pp. 31-42, 1976.

Chanon Dechsupa received his Ph.D. degree in computer engineering, Faculty of engineering, Chulalongkorn University in 2018. From 2008 to 2015, he was a database programmer and a senior system analyst with many private sectors. Currently, he is a post-doctoral researcher at department of computer engineering, Faculty of Engineering, which was supported by the Fellowship Ratchadaphiseksomphot Endowment Fund of Chulalongkorn University. His field of interest includes the formal method, software engineering and workflow design, and the data science in the software engineering context.

Wiwat Vatanawood received Ph.D. degree in computer engineering from Chulalongkorn University, Thailand. He is currently an associate professor of Computer Engineering at Faculty of Engineering, Chulalongkorn University. His research interests include formal specification methods and software architecture.

Arthit Thongtak received Dr.Eng in Electrical & Electronic Engineering from Tokyo Institute of Technology, Japan. He is currently an assistant professor at department of computer engineering, Chulalongkorn University, Thailand. His interests include Asynchronous logic design and verification, Dependable computing, and Computer architecture