SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •
Special Focus on Software Engineering for Internet Computing

August 2016, Vol. 59 080105:1–080105:25
doi: 10.1007/s11432-016-5598-5

Petri net based test case generation for evolved specification

Zuohua DING[1*], Mingyue JIANG[1], Haibo CHEN[1], Zhi JIN[2] & Mengchu ZHOU[3]

[1]*School of Information Science, Zhejiang Sci-Tech University, Hangzhou* 310018*, China;*
[2]*Software Engineering Institute, Peking University, Beijing* 100871*, China;*
[3]*Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ* 07102*, USA*

Received April 25, 2016; accepted May 29, 2016; published online July 18, 2016

Abstract Model-based testing can use a model to test a concrete program's implementation. When the model is changed due to the evolution of the specification, it is important to maintain the test suites up to date, such that it can be used for regression testing. A complete regeneration of the whole test suite from the new model, although inefficient, is still frequently used in practice. To address this problem effectively, we propose a test case reusability analysis technique to identify reusable test cases of the original test suite based on graph analysis, such that we can generate new test cases to cover only the change-related parts of the new model. The Market Information System (MIS) is employed to demonstrate the feasibility and effectiveness of the proposed method. Our experimental results show that the use of our method saves about 31.5% test case generation cost.

Keywords test case generation, regression testing, evolved specification, Petri net, reachability graph

Citation Ding Z H, Jiang M Y, Chen H B, et al. Petri net based test case generation for evolved specification. Sci China Inf Sci, 2016, 59(8): 080105, doi: 10.1007/s11432-016-5598-5

1 Introduction

In model-based testing, a model is used to describe the specification of a system. We can use this model to test concrete implementation. Since the model is an abstraction to the system, it is easier to grasp. Therefore, once the model is considered valid, we can use it for automatic test case generation.

Since specification may evolve during the lifetime of a software application when requirements are added, corrected, and removed, the model will also be updated to reflect such requirement changes. Since the test suite generated from the original model may not attest to the new specification, it is important to maintain the test suite to reflect the new model effectively. To do this, a straightforward approach is to regenerate a new test suite from the new model, which is often used in practice. This approach, however, is time-consuming, especially for complex models.

In the existing work, Tahat et al. [1] propose selective regression test case generation techniques to generate regression test cases to test the modified parts of the model. Korel et al. [2] propose to define the model change as a set of elementary modifications. These studies fail to consider obsolete test cases and reuse test cases. Jiang et al. [3] propose a method trying to solve this issue. Their idea is to match every

* Corresponding author (email: zouhuading@hotmail.com)

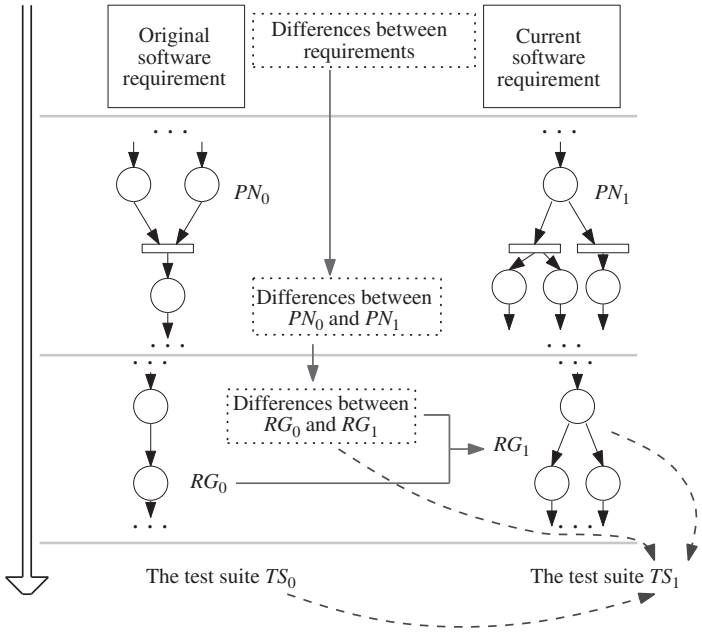


Figure 1 The framework of constructing test cases for evolving software requirements.

invocation/event sequence of the original model in the new model graph by means of graph analysis. If a sequence can be successfully matched, then its corresponding test case is reusable; otherwise, it is obsolete. However, if the number of test cases is big, this becomes hard. Especially if a test case is very long, the identification of the nodes and edges between two graphs is equivalent to the subgraph isomorphism problem, which is NP-complete [4].

Generally, to generate test cases as specification has evolved, the following three problems need to be addressed.

- How to reflect the requirement changes in a model;
- How to use the changed part of a model to generate test cases;
- How to determine reusable test cases.

In this paper, we propose a new way to handle these problems in order to construct test cases that are up to date. The key idea of the proposed approach is shown in Figure 1. We use a Petri net to model requirements, and thus the evolution of software specification is reflected in Petri net models. With the Petri net model of a software system, we can generate test cases from its Reachability Graph (RG). Given a software system, suppose that there exists a test suite TS_0 corresponding to the PN model PN_0, which is the test suite for testing the previous implementation of the system. After the software requirement changes, we have to construct a new test suite, namely, TS_1 corresponding to the new PN model PN_1, in order to test the latest software implementation. By considering (a) there may be multiple duplications between TS_0 and TS_1, and (b) considerable efforts are required to build the new reachability graph RG_1 and construct TS_1, we construct TS_1 in such a way that the complete reconstruction of requirement models as well as TS_1 is avoided. As a first step, we map the requirements differences into the distinctions between two Petri net models, i.e., PN_0 and PN_1. We further use these distinctions to guide the construction of RG_1 and TS_1. On one hand, to construct RG_1, we leverage the difference between PN_0 and PN_1, as well as the original reachability graph RG_0, such that RG_1 is constructed by revising RG_0 only. On the other hand, TS_1 is generated by applying reusable test cases in TS_0 and further generating new test cases suggested by the differences between RG_0 and RG_1. Finally, TS_1 consists of reusable old test cases that are constructed from old model (describing unchanged parts of the requirements), and new test cases that are constructed from the added parts of the new model (describing evolved parts of the requirements).

The rest of this paper is structured as follows. Section 2 briefly describes how to model textual

requirements and the changes with Petri nets. Section 3 shows how to rebuild Reachability Graph based on the changes only. Section 4 generates test cases for changed requirements. Section 5 is a case study. Section 6 talks about the threats to validity. Section 7 discusses the related work. Section 8 is the conclusion of the paper.

2 Modeling software specification and variation

2.1 Modeling software specification

Petri nets are used to model system requirements. With a tool developed in our previous work [5], we can automatically map the textual use cases to a Petri net.

This work chooses the format [6] to write use cases. Their primary elements are listed as follows:

Use Case | System under Discussion |
Primary actor | Scope |
Main scenario: 1. step 1, 2. step 2, ..., n. step n |
Variation | Extension

In the main scenario, extension and variation, the sentence in every step describes only one activity of the actor (primary actor, SuD). The sentence can be specified in the following structures.

- subject+verb+[object]
- subject+verb1+object1+to+verb2+[object2]
- subject+verb+object1+to/from+object2
- subject+verb+object+adjective
- subject+verb+object+present participle
- subject+verb+object+past participle

where '+' is a connector used between two elements, which helps to build a sentence format. Each type of sentences corresponds to a syntax tree.

Our net model is a subclass of Petri nets, in which each subsystem is described by a Petri-net-state-machine, called a closed process net, and the interactions among the subsystems are through a rendezvous mechanism. The following is the sketch of the process. From a model-based point of view, the tool automatically transforms textual use cases to Petri net-based behavioral models. We first construct Source Metamodel from textual use cases, and Behavior Target Metamodel for the target Petri net, and then define rules to transform a source model to a target one. The model transformation relies on the model transformation platform ADT (ATL Development Tools) provided by Eclipse[1)].

Our source model conforms to the source metamodel and is generated by configuring the attributes in the metamodel. The required values for the configuration can be obtained from the set of use cases. The Target Model can be obtained by configuring the target metamodel with the values obtained from the model transformation. The model transformation is through executing a set of rules written in ATL (Atlas Transformation Language) in the platform provided by Eclipse.

Every SuD or User in a source metamodel is transformed to a Closed Process Net in the behavior model, and an activity is transformed to a Transition. Besides, an activity also generates an outgoing internal place to the transition, which represents the new state caused by the activity. A use case corresponds to a branch net in the closed process net. The corresponding Petri net structure is sketched in Figure 2.

Since each SuD is trigged by a user action, and interacts with other SuDs to complete a service, the first transition of each branch net is an Input Communication transition. The process structure can be designed as one sState place followed by several branches, where a branch is executed if its first transition is enabled and fired.

The branch nets can be merged to form sequential or choice order based on the preConditions and postConditions of their corresponding use cases.

1) http://www.eclipse.com

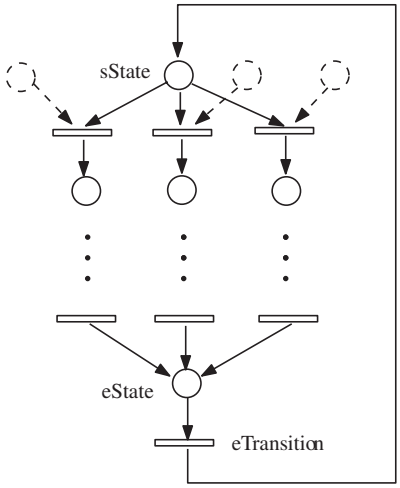


Figure 2 A process generated from a SuD.

Remark 1. Note that there is a study that maps a textual use case with restricted natural language to Petri nets by Somé [7]. But it is manual and with no requirement checking based on the model. In our work [8], we build a chain to automatically transform textual cases to a source model, from a source model to a target model, from a target model to a Petri net. By using a net checking tool INA, we can check the properties of requirements such as consistence, completeness and correctness. There are some other studies to build a formal behavior model from use cases, for examples, Sinha et al. [9] convert a combination of use case diagrams and UML class diagrams to an extended finite state machine model. Singh et al. [10] present a domain independent tool that automates the process of extraction, analysis and classification of events from textual requirements.

2.2 Modeling specification variations

Once the requirements change, the model needs to be updated to reflect such changes. We use the term of an original Petri net model to denote the model representing the software requirement before its change, and use the term of a new Petri net model or resultant one interchangeably to denote the one that is consistent with the current software requirement. Hence, once the software requirements vary, we need a new Petri net model for test case generation.

In this paper, we focus on three typical categories of requirement variations: addition/deletion of requirements, and correction of some requirements. Addition of a requirement leads to inserting transitions, links and places into the original model to construct a new model. Deletion of a requirement results in the deletion of some transitions, links and places from the original model so as to obtain a new one. Correction of a requirement can be regarded as a combination of deletion and addition.

To reflect the requirement changes in a Petri net model, we define four basic operations: *Deletion of Transition*, *Insertion of Transition*, *Deletion of Link* and *Insertion of Link*. Note that, a modification to a Petri net model may require more than one basic operation, i.e., a combination of these four basic operations.

As the deletion of a transition or link exactly preforms reverse actions to the insertion of a transition or link, we only clarify operations that insert an element into the original Petri net. Next, we will explain in detail how these insertion operations change a Petri net model.

2.2.1 *Insertion of a transition*

We first focus on the operation that inserts a new transition into the original Petri net model. Insertion of a transition changes the structure of a Petri net model, as it always introduces a new transition and links into the original petri net model. It further brings in new places for many cases. More

importantly, inserting a transition at a different location leads to different resulting models, because diverse relationships between the inserted transition and existing transitions may be constructed.

To demonstrate different categories of insertion operations, we apply a basic Petri net model, which is shown in the first column of Table 1, as the original model. We use t_i to denote the inserted transition, and use p_{*i} and p_{i^*} to denote its input and output places, respectively. We add t_i with reference to transition t_1, one of the transitions in the original model, and then show the resultant model after performing the insertion operation.

We summarize four general categories of insertion operations below, and details of their resultant models are displayed in Table 1. Specifically, varying elements in the resultant model are tagged with different colors, those with red color are inserted parts, and those with blue color may be either new elements or old ones.

• [InsertionT-1] t_i and t_1 have the same input place. In this case, p_{*i} already exists ($p_{*i} = p_1$), while p_{i*} can be either an existing place of the original model, or a new place to be introduced into the resultant model.

• [InsertionT-2] t_i and t_1 have the same output place. In this case, p_{i*} already exists ($p_{i*} = p_2$), while p_{*i} can be either an existing place of the original model, or a new place to be introduced into the resultant model.

• [InsertionT-3] The input place of t_i is the output place of t_1. In this case, p_{*i} already exists ($p_{*i} = p_2$), but p_{i*} is a new place to be introduced into the resultant model.

• [InsertionT-4] The output place of t_i is the input place of t_1. In this case, p_{i*} already exists ($p_{i*} = p_1$), but p_{*i} is a new place to be introduced into the resultant model.

2.2.2 *Insertion of a link*

In a Petri net model, links connect transitions and places. There are two categories of links: (1) links pointing to places, and (2) links pointing to transitions. The insertion of different categories of links brings varying effects to the model.

To clarify how the insertion of a link influences a Petri net model, we insert different categories of links into an original model (displayed in the first column of Table 2) to obtain the corresponding resultant models, based on which we clarify the effects taken by inserting links. Here, we apply transition t_1 of the original model as a reference transition, which will be the starting point or ending point of the inserted link, and we use p_l to denote the place related to the inserted link. We depict two insertion operations below, and their illustrative description are shown in Table 2.

• [InsertionL-1] The inserted link points to p_l. In this case, the set of output places of t_1 changes, as p_l is appended into it.

• [InsertionL-2] The inserted link points to t_1. In this case, the set of input places of t_1 changes, as p_l is appended into it.

Note that, in both cases, p_l may be either an existing place in the original model or a new place introduced by the operation of inserting a link.

Remark 2. Note that here we do not consider the operations: inserting/deleting places. The reasons are the following. On one hand, a place can not stand alone in the net, otherwise it represents a dead local state. Since a place is always connected to a link, so if we remove it from the net, the connected link will be removed, otherwise the net misses a local state, which is not reasonable. If we add a place to the net, we need to add a link at the same time, otherwise the place is separated from the net, and it is a dead local state. On the other hand, the insertion of a transition will add a new transition and links into the petri net model, meanwhile it brings in new places. The insertion of a link will pint to a place or a transition. In either way, the link has a connected place. Since the deletion of a transition or link is the reverse actions to the insertion of a transition or link, it will remove the corresponding places.

Table 1 Insertion of a Transition: PN changes vs RG changes

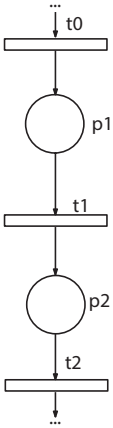
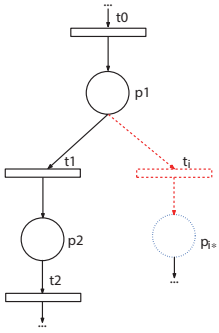
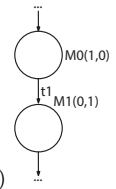
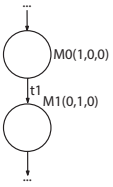
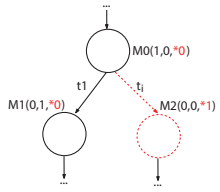
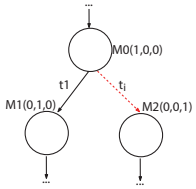
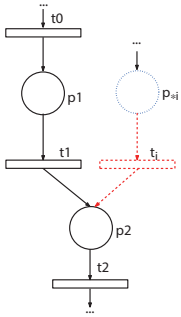
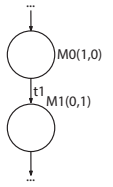
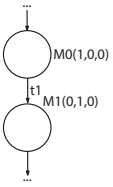
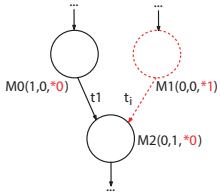
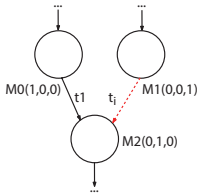
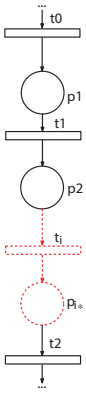
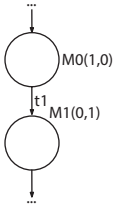
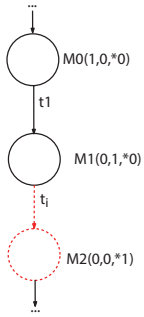
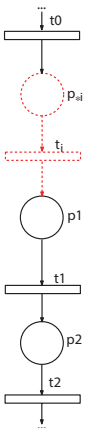
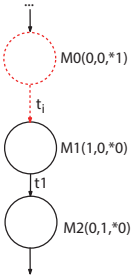
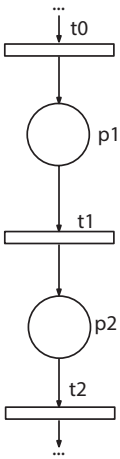
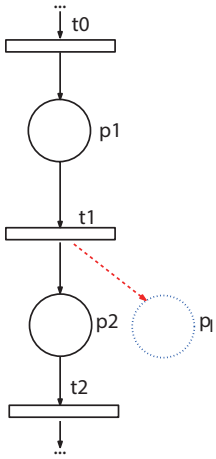
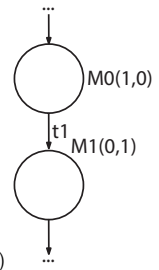
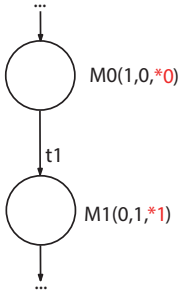
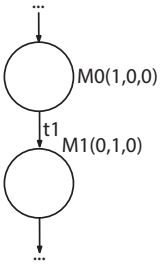
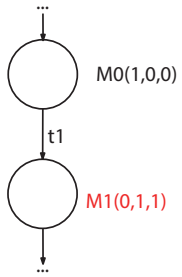
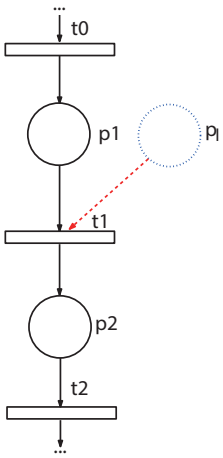
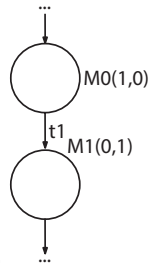
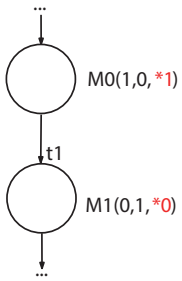
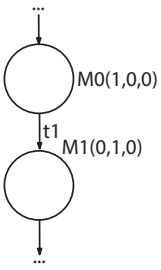
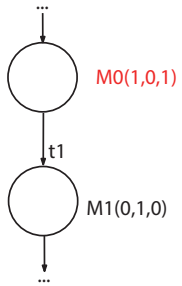
PN_0	PN changes	PN_1	RG_0	RG changes	RG_1
	InsertionT-1		 (a)  (b)	RIT-1	 
	InsertionT-2		 (c)  (d)	RIT-2	 
	InsertionT-3		 (e)	RIT-3	
	InsertionT-4			RIT-4	

Table 2 Insertion of a Link: PN changes vs RG changes

PN_0	PN changes	PN_1	RG_0	RG changes	RG_1
	InsertionL-1			RIL-1	
					
	InsertionL-2			RIL-2	
					

3 Updating reachability graph

When a Petri net model changes, its reachability graph (RG) changes. RG is characterized by its marking vector and edge set. Each edge connects two marking vectors, and thus the state transformation is accomplished. Hence, RG evolution is reflected by its marking vector and edge set.

Since our net is a subclass of Petri nets, we can use some tools such as INA, MATLAB and HiPS to build the reachability graph directly from the current Petri net model. However, if the changes bring us some selections or loops, we may experience a state explosion problem when we build RG.

Can we rebuild RG by considering the changes only? We intend to answer this question by defining rules for rebuilding RG by just modifying RG_0 of the original model. In this section, we first clarify the way RG changes with respect to its Petri net's variations. Accordingly, we mainly focus on RG's changes caused by the insertion of transitions and links, because the changes of deleting transitions and links just lead to opposite changes to RG. Based on this analysis, we develop an algorithm for automatically

constructing a new RG reflecting current software requirements by using the old RG and PN variations.

Since the insertion (deletion) of transitions/links is associated with the insertion (deletion) of places, the marking vector in RG_1 has attributes different from RG_0. To tell the difference, we add a 0^* to the old vector if a new place with no token is added to the net, and 1^* if it has a token.

3.1 Reachability graph changes caused by insertion of transition

We have defined four categories of insertion operations of a transition, each of which brings distinct changes to the original Petri net model and thus leads to specific variations in its RG. We next elaborate RG variations resulting from different insertion operations.

For the original models displayed in Table 1, its RG contains two states, M_0 and M_1, and one edge connecting these two nodes, which is labeled with t_1. Particularly, there may be two different marking vectors for the RG. In the case that the relevant place of the inserted transition t_i, namely, p_{*i} or p_{i*}, does not belong to the original model, the initial marking vector only contains two elements, denoting the token values of p_1 and p_2, i.e., $\mathrm{RG}_0(a)$ in Table 1. However, if p_{*i} or p_{i*} is a place of the original model, then the marking vector contains three elements, which represent the token values of places p_1, p_2 and p_{*i} (or p_{i*}), , i.e., $\mathrm{RG}_0(b)$ in Table 1.

The RG update is described below according to the above four categories of insertion operations of a transition, and the corresponding illustrative examples are shown in Table 1.

RIT-1: Inserting t_i by performing the operation [InsertionT-1]. In this case, a new edge fired by t_i, which is an alternative to the directed edge fired by t_1, is inserted into RG_0, in order for RG_1 to reflect this change. As a result, both directed edges have the same starting node M_0 but different ending nodes. Note that, in $\mathrm{RG}_0(a)$, the marking vector has two attributes, while in RG_1, the marking vector has three attributes since a new state is added to the net.

RIT-2: Inserting t_i by performing the operation [InsertionT-2]. In this case, a new edge fired by t_i is added to RG, which is an alternative to the existing edges fired by t_1 in RG. In this case, both edges have the same ending node. The difference between the marking vectors of RG_0 and RG_1 also depends on whether case (c) or case (b) is considered.

RIT-3: Inserting t_i by performing the operation [InsertionT-3]. In this case, a new edge fired by t_1 is added to RG, which is just after the the edge fired by t. Note that in RG_0, the marking vector has two attributes, while in the RG_1, the marking vector has three attributes since a new place has been added to the net. If we remove the nodes whose new attribute is 1^* and remove 0^* from other nodes, then RG_1 is the same as RG_0.

RIT-4: Inserting t_i by performing the operation [InsertionT-4]. In this case, a new edge fired by t_1 is added to RG, which is just before the the edge fired by t. Note that, in RG_0, the marking vector has two attributes, while in RG_1, the marking vector has three attributes because a new place has been added to the net. If we remove the nodes whose new attribute is 1^* and remove 0^* from other nodes, then RG_1 is the same as RG_0.

3.2 Reachability graph changes caused by insertion of link

According to the above two categories of operations of inserting links, their impacts on RG are clarified below.

Generally, if the insertion of a link introduces a new place to PN_0, this operation changes the marking vectors of RG_0. That is, a new attribute is added into the old marking vector such that the newly added place's marking can be recorded. However, if the insertion of a link does not introduce any new place, the marking vectors of RG_0 are the same as those of RG_1, but their marking values at their respective nodes may be different.

We present RG changes resulting from the insertion of links in Table 2. Two reasonable RG_0 models are considered, according to the above two situations. $\mathrm{RG}_0(a)$ indicates that the place related to the added link does not exist in PN_0, while $\mathrm{RG}_0(b)$ represents another case.

3.3 Construction of new reachability graph using PN variations

Based on RG_0 and the changes of a Petri net, we derive RG_1 reflecting current software requirements. The proposed method is expected to be more cost-effective, because only parts of RG_0 are revised for constructing RG_1.

Algorithm 1 presents the way of constructing RG_1 with reference to the changed Petri net. The inputs to the algorithm are RG_0, PN_1, which is the resultant Petri net model after applying changes to PN_0, and some sets of Petri net changes. The outputs of Algorithm 1 include RG_1), which is the expected RG of PN_1, and variations between RG_0 and RG_1. We use four sets of varied elements of the Petri net model to represent Petri net changes: ${}^{+}T$ and ${}^{-}T$ denoting inserted and deleted transitions, and ${}^{+}L$ and ${}^{-}L$ denoting inserted and deleted links. We use ${}^{+}E$ and ${}^{-}E$ to store the differences between RG_0 and RG_1, that is, the inserted and deleted edges in RG_1, respectively.

For each changed element, the algorithm modifies RG_0 in order to reflect this change. After all elements are processed, RG_1 is reported as the RG corresponds to PN_1 (Algorithm 1: lines 13–26). Updating RG_0 with respect to Petri net changes is accomplished by four different functions, which respectively handle four different types of Petri net changes. i.e., *InsertionTUpdate* (*DeletionTUpdate*) for handling inserting (deleting) a transition, *InsertionLUpdate* (*DeletionLUpdate*) for handling inserting (deleting) a link. We next clarify *InsertionTUpdate* and *InsertionLUpdate* in detail in Algorithm 1, and *DeletionTUpdate* and *DeletionLUpdate* perform exactly opposite procedures as their corresponding insertion operation.

Given an inserted transition t, PN_1 and RG_0 of the Petri net before inserting t, the function *InsertionTUpdate* refines RG according to t as follows. The algorithm first checks the type of an insertion operation related to t (line 30), according to the four different categories of operations discussed in Subsection 2.1. If t belongs to the first category (*InsertionT-1*) (*type*==1), with respect to our explanation in Subsections 2.1 and 3.1, we have to further confirm whether its insertion brings in a new place or not. That is, whether t's post place is a new place introduced by t or an existing one (lines 32, 33). For the former, RG's marking vectors must have one more element, and must contain one more state than before. Thus, the algorithm first modifies the marking vector of RG (*AddAnElementIntoMarkingVector*), then adds a state (*AddAState*) into the existing RG. If the insertion of t is an *InsertionT-2* type operation, the algorithm handles it following a procedure similar to that of *InsertionT-1*. The only difference is that its pre-place determines whether a new place is introduced or not. For the above two categories of insertion operations, if they do not introduce any new place, both marking vectors and states of RG keep unchanged. For the other two cases, a new place is introduced, so the marking vectors and states of RG_0 are both refined. To change RG_0 with respect to t, the last step is to identify the relevant edge introduced by inserting t, that is, the edge e returned by *ComputeAddedEdge*. e is further added into RG, and meanwhile, it is also recorded in *inEDG* (lines 48,19).

For an inserted link l, PN_1 and RG_0 before inserting l, the algorithm updates RG by using changes reflected by inserting l. As discussed in Subsections 2.2 and 3.2, inserting a link into a Petri net yields changes of marking values rather than changing the number of RG nodes. The algorithm analyzes l, PN and RG to determine whether inserting l introduces a new place. If a new place is accompanied with l, a new element is added into RG's marking vectors; otherwise, the marking vectors keep unchanged. The insertion of l changes the marking values of nodes relating to the transition which is the source or destination of l. Eventually, the algorithm identifies these places and changes their marking value (*ChangeMarkingValue*).

The complexity of Algorithm 1 is computed as the following. Suppose that PN_1 contains x_1 places, and RG_0 contains x_2 edges and x_3 states, respectively. Let N_1 denote the complexity to search a place in a Petri net, $1 \leqslant N_1 \leqslant x_1$, and N_2 denote the complexity to compute an edge that relates to a given transition, $1 \leqslant N_2 \leqslant x_2$. We also need the following notations. n_1: the number of InsertionT-1 transitions; n_2: the number of InsertionT-2 transitions; n_3: the number of InsertionT-3 and InsertionT-4 transitions; n_4: the number of DeletionT-1 transitions; n_5: the number of DeletionT-2 transitions; n_6: the number of DeletionT-3 and DeletionT-4 transitions. Note that, $n_1 + n_2 + n_3 = |inT|$, and $n_4 + n_5 + n_6 = |deT|$.

Algorithm 1: Construction of new Reachability Graph

```
Input:
PN1: The updated Petri net
inT: A set of inserted transitions,
deT: A set of deleted transitions,
inL: A set of inserted links),
deL: A set of deleted links),
RG0: The original Reachability Graph.
Output:
RG1: The new reachability Graph.
inEDG: a set of inserted edges for constructing RG1
deEDG: a set of deleted edges for constructing RG1
Begin
    RG1 = RG0
    inEDG = ∅
    deEDG = ∅
    for ci : inT
      RG1 = InsertionTUpdate(ci, RG1, PN1, inEDG )
    endfor
    for ci : deT
      RG1 = DeletionTUpdate(ci, RG1, PN1, deEDG)
    endfor
    for ci : inL
      RG1 = InsertionLUpdate(ci, RG1, PN1)
    endfor
    for ci : deL
      RG1 = DeletionLUpdate(ci, RG1, PN1)
    endfor
End
```

```
function InsertionTUpdate(t, RG, PN, AEdge)
    type = CheckInsertionTType(c)
    if type == 1 or then //performing InsertionT-1
      postP = GetPostPlace( t, PN)
      if CheckPlaceExistence( postP, t, PN ) == 0 then
        RG = AddAnElementIntoMarkingVector( RG )
        RG = AddAState( RG, t, PN)
      endif
    else if type == 2 then //performing InsertionT-2
      preP = GetPrePlace( t, PN)
      if CheckPlaceExistence( preP, t, PN ) == 0 then
        RG = AddAnElementIntoMarkingVector( RG )
        RG = AddAState( RG, t, PN)
      endif
    else //performing InsertionT-3 or InsertionT-4
      RG = AddAnElementIntoMarkingVector( RG )
      RG = AddAState( RG, t, PN)
    endif
    e = ComputeTheAddedEdge( t, PN, RG, type )
    RG = AddAnEdge( e, RG)
    AEdge = AEdge ⋃ {e}
    return RG
end
```

```
function DeletionTUpdate(t, RG, PN, DEdge)
    type = CheckDeletionTType(c)
    if type == 1 or then //performing DeletionT-1
      postP = GetPostPlace( t, PN)
      if CheckPlaceExistence( postP, t, PN ) == 0 then
        RG = DeleteAnElementFromMarkingVector( RG )
        RG = DeleteAState( RG, t, PN)
      endif
    else if type == 2 then //performing DeletionT-2
      preP = GetPrePlace( t, PN)
      if CheckPlaceExistence( preP, t, PN ) == 0 then
        RG = DeleteAnElementFromMarkingVector( RG )
        RG = DeleteAState( RG, t, PN)
      endif
    else //performing DeletionT-3 or DeletionT-4
      RG = DeleteAnElementFromMarkingVector( RG )
      RG = DeleteAState( RG, t, PN)
    endif
    e = ComputeTheDeletedEdge( t, PN, RG, type )
    RG = DeleteAnEdge( e, RG)
    DEdge = DEdge ⋃ {e}
    return RG
end
```

```
function InsertionLUpdate(l, RG, PN)
    place = GetLinkRelatedPlace( l, PN)
    if CheckPlaceExistence( place, l, PN ) == 0 then
      RG = AddAnElementIntoMarkingVector( RG )
    endif
    RG = ChangeMarkingValue( RG, l, PN )
  return RG
end
```

```
function DeletionLUpdate(l, RG, PN)
    place = GetLinkRelatedPlace( l, PN)
    if CheckPlaceExistence( place, l, PN ) == 0 then
      RG = DeleteAnElementFromMarkingVector( RG )
    endif
    RG = ChangeMarkingValue( RG, l, PN )
  return RG
end
```

The time complexity of Algorithm 1 is

$$\begin{aligned}
&O((n_1+n_2)\times(N_1+N_2+x_3)+n_3\times(N_2+x_3)+(n_4+n_5)\times(N_1+N_2+x_3)\\
&+n_6\times(N_2+x_3)+(|inT|+|deT|)\times(N_1+x_3))\\
<\ &O((|inT)\times(x_1+x_2+x_3)+|inT|\times(x_2+x_3)+(deT)\times(x_1+x_2+x_3)+|deT|\times(x_2+x_3)\\
&+(|inT|+|deT|)\times(x_1+x_2+x3))\\
<\ &O((|inT|+|deT|)\times(x_1+x_2+x_3)).
\end{aligned}$$

4 Generation of test cases for changed requirements

Based on the covering criterion of white-box testing [11], we generate test cases that can cover all nodes and edges of a reachability graph.

Definition 1. A trace is a path started from starting node passing through other nodes and edges and ended with the starting node or an end node. If a trace has a loop, then the loop can not repeat more than twice in a row. A test case is a trace.

4.1 Generating test cases from RG

Algorithm 2 is an algorithm to generate test cases. In the algorithm, we assume that the length of a test case cannot be larger than 60 nodes due to computer memory limitation.

Algorithm 2: Generate Test Cases

```
Input: Reachability Graph
Output: Test Cases
Begin
   boolean[][] rg = getReachabilityGraphModel();
   List<List<Integer>> paths = new List<List<Integer>>();
   int visitNode(int p,List<Integer>paths,
            List<List<Integer>> results,
            int loopnum,int depth){
   if(Util.isDualLoop(paths))
     return loopnum;
   for(int i = 0; i < rg.length; i++){
    if(!rg[p][i])
      continue;
    List<Integer> clonepaths = new ArrayList<Integer>();
    clonepaths.addAll(paths);
    loopnum = visitAdjacentNodeForLoop(i,
       clonepaths,results,loopnum,++depth);
   }
   Return paths;
End
```

In Algorithm 2, we use recursion to generate test cases.

• Function getReachabilityGraphModel() reads a file containing a reachability graph, and constructs a matrix to express it.

• The collected paths are stored as the test cases.

• Function visitNode(int p, List<Integer> paths, List<List<Integer>> results, int loopnum, int depth) is a recursion to visit the reachable nodes.

• Function isDualLoop(List<Integer> paths) checks the number of times that a loop occurs continuously in a path. If the number is bigger than or equal to 2, the path is thrown away.

• If there are more than two reachable nodes from node p, then the paths to p will be cloned for those reachable nodes.

The complexity of Algorithm 3 is computed as the following. Let n_1 and n_2 denote the number of states and the number of edges of RG, respectively. From the complexity of depth first algorithm, the time complexity of Algorithm 2 is $O(n_1 + n_2)$.

4.2 Generating test cases from changed Petri net

By applying Algorithm 2, we can construct a test suite TS_0 from RG_0, which is considered to be a basic test suite, because it contains most of the typical test cases for testing the modeled system. After system requirements change, most parts of TS_0 may still be applicable to the new requirements. Therefore, instead of completely reconstructing new test cases for the system's new requirements, we here adopt a more cost-effective way of obtaining such test cases. That is, collecting reusable test cases of TS_0 and further constructing additional test cases for extra newly added requirements. We present our test cases construction strategy in Algorithm 3.

The first step of Algorithm 3 is to reuse available test cases of TS_0. For any test case t_i in TS_0, the algorithm confirms its reusability by conducting two kinds of checking (from line 11 to line 21) to see: (1) whether t_i contains edges in *deEDG*; and (2) whether t_i contains states that are relevant to edges in *inEDG*. If for both checking, the answers are "NO", then t_i is a reusable test case and is pushed into TS_1 (line 16). Specifically, for the case of checking (1), if t_i contains any edges belonging to *deEDG*, it is not reusable because those edges do not exist in RG_1. For the case of checking (2), if there exists some state of t_i, which is related to an edge of *inEDG* (that is, this state relates to either the starting or the ending state of the edge), this indicates that the trace represented by t_i has been changed during the generation of RG_1. In other words, in RG_1, there should be a trace that is similar, but not the same, to the trace represented by t_i. Therefore, such a t_i cannot be directly reused. However, regarding to the relationships between t_i and the relevant edges of *inEDG*, t_i can be adjusted to generate a new test case that is consistent with RG_1. Therefore, for such a set of test cases (denoted by T_2 in Algorithm 3), we further modify them to get their corresponding test cases relating to RG_1 (from line 22 to line 28), which are also pushed into TS_1 (line 25). Note that, after these operations, *inEDG* is also updated since the involved edges have been covered by test cases and thus no test cases need to be further generated for them (these edges are removed from *inEDG* in line 26).

Those unprocessed extra requirements are now reflected through the updated *inEDG*. The next important task of Algorithm 3 is to construct test cases for those additional requirements. Algorithm 3 describes this procedure through function *GenExtraTCs*. For each edge e_i that is inserted into RG_1, the algorithm uses RG_1 to collect all paths started from the start node, passing through e_i and ended with the ending node (*GenTCUsingAnEdge*). All those paths related to e_i constitute a test suite TS_t. Finally, the union of those test suites for all inserted edges, namely, $\mathrm{TS}_{\mathrm{add}}$, becomes one part of the final test suite TS_1.

As an illustration of the newly designed mechanism, i.e., the handling of t_i that contains states relating to the edges in *inEDG*, we consider the original RG(e) and the resulting RGs from conducting InsertionT-3 in Table 1. Assume a test case is $t_0 = M_0(1,0) \rightarrow M_1(0,1)$. With the reference to operation InsertionT-3, we can get $inEDG = \{e\}$. Specifically, the starting state and ending state of e are $M_1'(0,1,0)$ and $M_2'(0,0,1)$, respectively. Firstly, we identify that the state $M_1(0,1)$ relates to the starting state of e, namely, $M_1'(0,1,0)$. The reason for this is explained as follows: the operation InsertionT-3 suggests that the marking vector of G_1 has an additional attribute than that of G_0, and thus only need to compare the former two attributes of states of G_0 and G_1 when the starting node of e is analyzed. Obviously, M_1 matches M_1' in this situation. With the above knowledge, the test case t_0 can be refined by incorporating the additional edge and its ending state, which is $t_0' = M_0'(1,0,0) \rightarrow M_1'(0,1,0) \rightarrow M_2'(0,0,1)$.

The complexity of Algorithm 3 is computed as the following. Let $n_1 = |inEDG|$, $n_2 = |deEDG|$, $n_3 = |\mathrm{TS}_1|$, $n_4 = |\mathrm{TS}_2|$, where $n_3 + n_4 = |\mathrm{TS}_0|$. Suppose that a test case averagely contains x_1 states and x_2 edges, and then the complexity for the first kind of checking is $O(x_2 \times n_2)$, and the complexity

Algorithm 3: Generation of Test Cases From Changed Petri Net

```
1 Input:
2 TS0: A test suite constructed from RG0, the reachability graph of the original Petri net
3 RG1: The new reachability graph referring to changed Petri net
4 inEDG: A set of inserted edges in RG1
5 deEDG: A set of deleted edges from RG0
6 Output:
7 TS1: A test suite for testing Changed requirements
8 Begin
9    TS1 = ∅
/* to store reusable test cases of TS0 */
10   TS2 = ∅
/* to store test cases that are not reusable but related to edges of inEDG */
11   for t_i in TS0
12     if containsDeletedEdges( t_i, deEDG) == true then
13       continue
14     else /* containsDeletedEdges( t_i, deEDG) == false */
15       if isRelatedToInsertedEdges( t_i, inEDG) == false then
16         TS1 = TS1 ⋃ {t_i}
17       else
18         TS2 = TS2 ⋃ {t_i}
19       endif
20     endif
21   endfor
22   for t_i in TS2
23     gs = getRelatedsEdge( t_i, inEDG)
24     t' = updateTestCase( t_i, gs)
25     TS1 = TS1 ⋃ {t'}
26     inEDG = removeEdges(inEDG, gs)
27     endif
28   endfor
// generate new test cases for testing extra system requirements
29   TS_add = GenExtraTCs ( inEDG, RG1 )
30   TS1 = TS1 ⋃ TS_add
31 End

32 function GenExtraTCs (inEDG, RG)
33  TS = ∅
34  for e_i in inEDG
35    TS_t = GenTCUsingAnEdge( e_i, RG)
36    TS = TS ⋃ TS_t
37  endfor
38  return TS
39 End
```

for the second kind of checking is $O(x_1 \times n_1)$. The time complexity of Algorithm 3 is $O(n_3 \times (x_1 \times n_1 + x_2 \times n_2) + n_4 \times (x_1 \times n_1) + (n_1 - n_4) \times (x_1 + x_2)) < O(|\mathrm{TS}_0| \times (x_1 \times n_1 + x_2 \times n_2))$.

4.3 Illustration with an example

We check our approach by a simple example. We modify a Petri net PN_0 as shown in Figure 3. Its RG_0 is in Figure 4. In Figure 4, the blue lines and circles are caused by inserting operations in PN_0, and the red lines and circles are caused by deleting operations in Petri net.

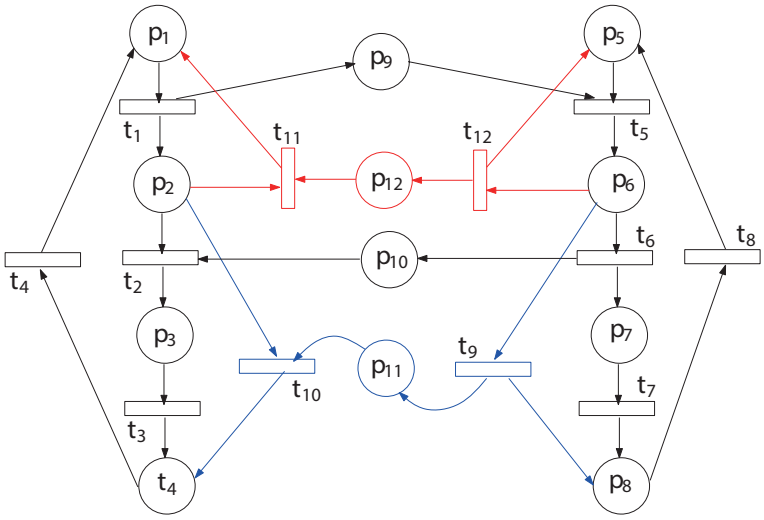


Figure 3 The modification of Petri net.

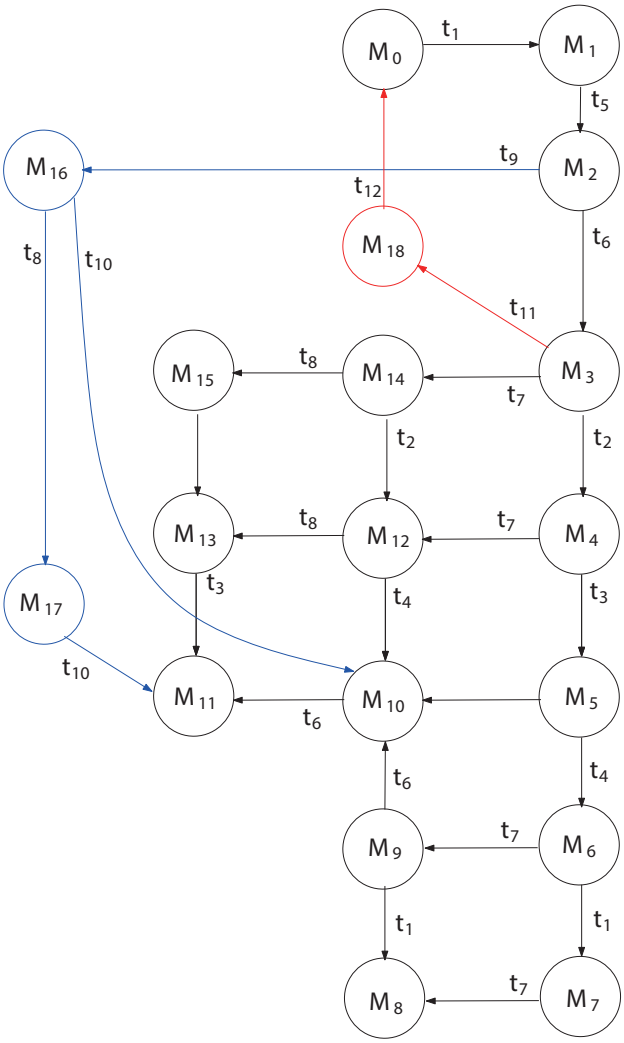


Figure 4 The modification of Reachability Graph.

Use Case: #7 Buyer makes a purchase
Scope: Marketplace
SuD: Marketplace Information System
Primary Actor: Buyer
Supporting Actor: Seller, Credit Verification Agency
Main success scenario specification:
1. Buyer enters basic search criteria to System.
2. System responds with the list of matches.
3. Buyer requests the complete listing of a selected offer to System.
4. System responds with the requested information.
5. Buyer chooses to accept a selected offer to Clerk.
6. System requests the billing information and payment method.
7. Buyer enters billing information, selects a payment method and provides the payment details to Clerk.
8. System responds to the buyer with a uniquely identified authorization number.
Alternatives:
5a Rejection available.
5a1 The Buyer decides not to accept the offer to Clerk.
5a2 Use case ends here.
6a System failed to validate the offer.
6a1 Use case aborted.
Sub-variations:
2b The amount of matches is too high.
2b1 Buyer narrows the search results with additional criteria to System.
2b2 Resume with step 2.

Use Case: CL#2
Buyer to Clerk
Scope: Marketplace Information System
SuD: Clerk
Primary Actor: Buyer
Supporting Actor: Computer System
Main success scenario specification:
1. Buyer submits to the Clerk a reference to a chosen offer.
2. Clerk submits the reference to the System.
3. Clerk reports the System response to the Buyer.
4. Clerk requests billing and shipping information, payment method and payment details to Buyer.
5. Clerk enters the billing and shipping information, payment method and payment details to System.
6. Clerk reports the system response (with the unique acknowledgement) to the Buyer.
Extensions:
3a Clerk sends failure of validation to Buyer.
3a1 Use case aborted.

Use Case: CS#3 Clerk buys a selected item on behalf of a Buyer
Scope: Marketplace Information System
SuD: Computer System
Primary Actor: Clerk (ultimate: Buyer)
Supporting Actor: Seller, Credit Verification Agency
Main success scenario specification:
1. System matches the search criteria.
2. System responds with the list of matches to Buyer.
3. Buyer requests the complete list of a selected offer.
4. System responds with the requested information to Buyer.
5. Clerk is contacted by a Buyer who has decided to accept a selected offer.
6. System validates the offer.
7. System requests the billing information and payment method to Buyer.
8. Clerk enters the billing information, selects a payment method and provides the necessary details.
9.System validates the Buyers information with the Credit Verification Agency.
10. System performs the sale.
11. System informs the seller that the offer has been accepted and provides the shipping information.
12. System transfers the payment to the sellers account.
13. System responds to the Buyer with a uniquely identified authorization number.
Extensions:
6a System failed to validate the offer to Clerk.
6a1 Use case aborted.
Sub-variations:
2b the amount of matches is too high.
2b1 Buyer narrows the search results with additional criteria.
2b2 Resume with step 2.

Figure 5 The textual use cases of MIS.

There are 8283 original test cases. For the modified Petri net PN_1, based on our algorithms, 7152 test cases can be reused, and there are 27829 new test cases. As a comparison, we apply Algorithm 2 directly to the new RG in Figure 4 to compute test cases, and we obtained 43981 test cases. After comparing with 8283 original test cases, we get exactly 7152 same test cases, which means that they can be reused.

5 Case study

We use the Market Information System (MIS) [12] to illustrate our approach. Figure 5 gives three textual use cases. The use case Buyer executes the operations of "making a purchase" by communicating with Clerk and System. "Buyer makes a purchase" can be summarized by two events: Buyer searches for an offer and Buyer buys a selected item. The extension use cases contain that Buyer is not login and Buyer is not registered.

The corresponding Petri net is shown in Figure 6.

5.1 Test cases of MIS before modification

The RG_0 of the Petri net is shown in Figure 7, and there are 23602 original test cases. Parts of the test cases are shown in Table 3.

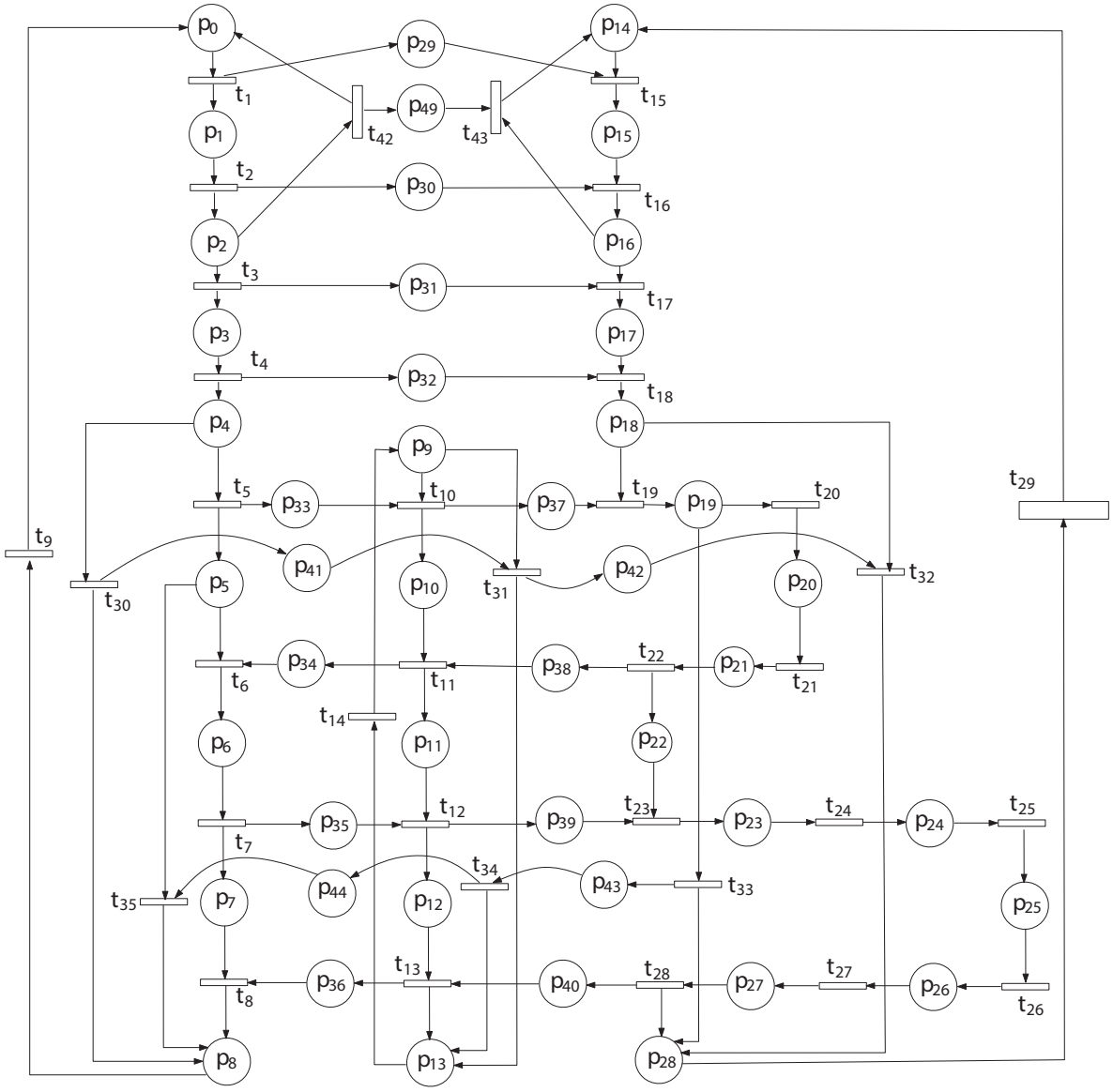


Figure 6 The Petri net before modification.

5.2 Changes of requirements

For "Buyer makes a purchase", in order to add a function for shopping cart into which Buyer can put several offers, we modify step 5, from 'Buyer chooses to accept a selected offer' to 'Buyer adds the selected offer to shopping cart', and add step 5.1—Buyer pays for the offers in shopping cart after step 5 in Buyer's use case. Beyond that, a favorite function which allows customers to store the favorite offers, is implemented by adding a branch in Alternatives. In order to protect the user's information, the timeout page is added, e.g., step 9a and step 9a1. The added steps 8b and 8b1 are used to make the way of user experience friendly by invoking the user's operation history. Given that there are more alternative payment methods, the system should allow users to change the payment method in the process of purchase. Therefore, steps 11b, 11b1 and 11b2 are added. In the meantime of modifying Buyer's use case, the related communicating information from the Buyer has to be transferred into Clerk, and then Clerk submits the corresponding message to System.

Modification 1: In order to delete the rejected operations, the use case activities shown in Table 4 are deleted from the original use case.

Modification 2: In order to insert the function of changing a payment method, the use case activities

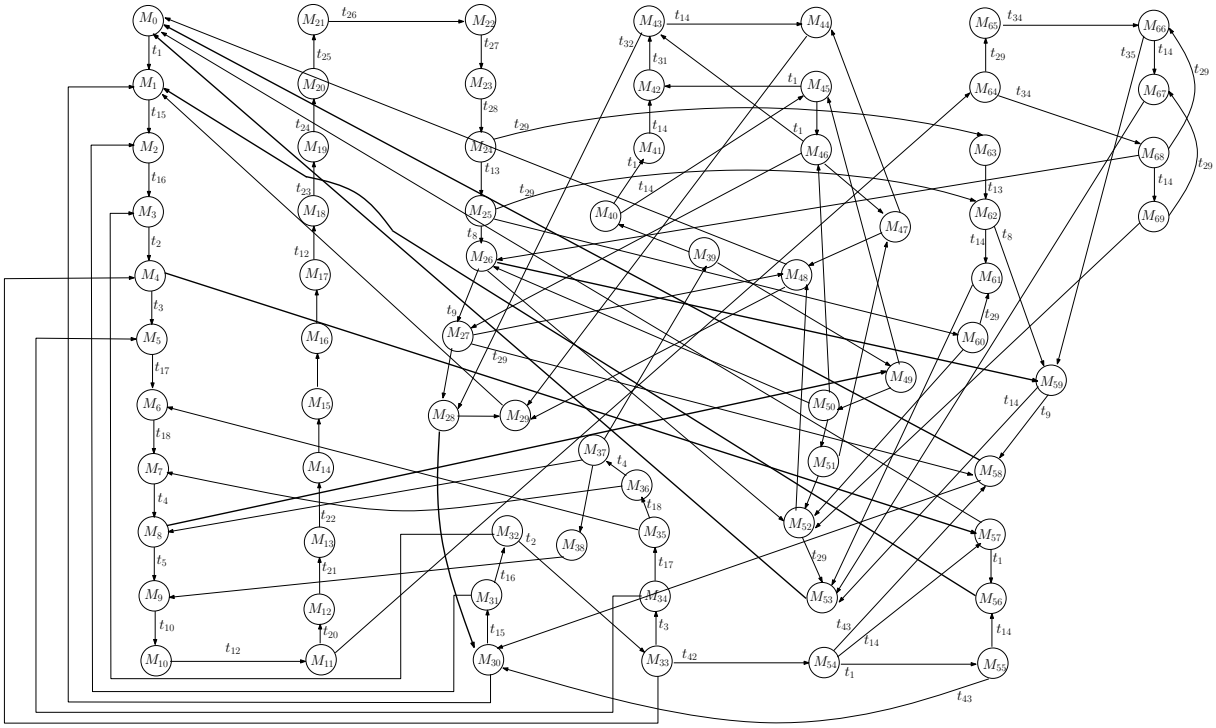


Figure 7 RG_0 for the net in Figure 6.

Table 3 Part of test cases

Original test cases
trace1: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,17,18,19,20,21,22,23,24,25,26,27,48,0
trace2: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,17,18,19,20,21,22,23,24,25,26,52,48,0
trace3: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,17,18,19,20,21,22,23,24,25,60,52,48,0
trace4: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,17,18,19,20,21,22,23,24,63,62,59,53,0
trace5: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,64,65,66,59,53,0
trace6: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,64,65,66,59,58,0
...
trace23601**:** 0,1,2,3,4,57,56,1,2,3,4,5,6,7,8,49,50,51,52,53,0
trace23602**:** 0,1,2,3,4,57,56,1,2,3,4,57,0

shown in Table 5 are inserted to the original use case.

Modification 3: In order to insert the function of Timeout, the use case activities shown in Table 6 are inserted to the original use case.

5.3 New Petri net model after modification

According to the deleted and inserted activities, we modify the Petri net, and the new net model is shown in Figure 8.

Table 4 Delete reject operations

Use case: Buyer makes a purchase
5**a** rejection available.
5**a**1 The Buyer decides not to accept the offer to Clerk.
5**a**2 Use case ends here.
Use case: Buyer to Clerk
5**a** rejection available.
1**a**1 The Buyer decides not to accept the offer to System.
1**a**2 Use case ends here.
Use case: Clerk buys a selected item on behalf of a Buyer
5**a** rejection available.
5**a**1 The Buyer decides not to accept the offer.
5**a**2 Use case ends here.

Table 5 Insert the function of changing payment

Use case: Buyer makes a purchase
9**a** no enter after 60 seconds
7**a**1 System displays a login timeout page.
7**a**2 Use case ends here.
Use case: Buyer to Clerk
9**a** no enter after 60 seconds
3**a**1 Clerk displays a login timeout page Buyer.
3**a**2 Use case ends here.
Use case: Clerk buys a selected item on behalf of a Buyer
9**a** no enter after 60 seconds
9**a**1 System displays a login timeout page to Clerk.
9**a**2 Use case ends here.

Table 6 Insert the function of Timeout

Use case: Buyer makes a purchase
11**a** payment method changes available.
8**a**1 The Buyer changes the payment method to Clerk.
8**a**2 Step to 9
Use case: Buyer to Clerk
11**a** payment method changes available.
4**a**1 The Buyer changes the payment method to System.
4**a**2 Step to 9
Use case: Clerk buys a selected item on behalf of a Buyer
11**a** payment method changes available.
12**a**1 The Buyer changes the payment method.
12**a**2 Step to 9

5.4 New test cases for new requirements

We generate new test cases step by step based on the order of changes.

New Test Cases For Change 1:

Since blue part is deleted from the original Petri net in Figure 8, we get the sets of deletion of transitions and links as ^{-}T={t30, t31, t32}, ^{-}L={t30-p41-t31, t31-p42-t32}. Based on the rules, we generate RG_1 as shown in Figure 9. The DEDGE is DEDGE={M37-t30-M39, M39-t9-M40, M39-t14-M49, M40-t1-M41, M40-t14-M45, M41-t14-M42, M42-t14-M43, M45-t1-M42, M43-t32-M28, M43-t14-M44, M46-

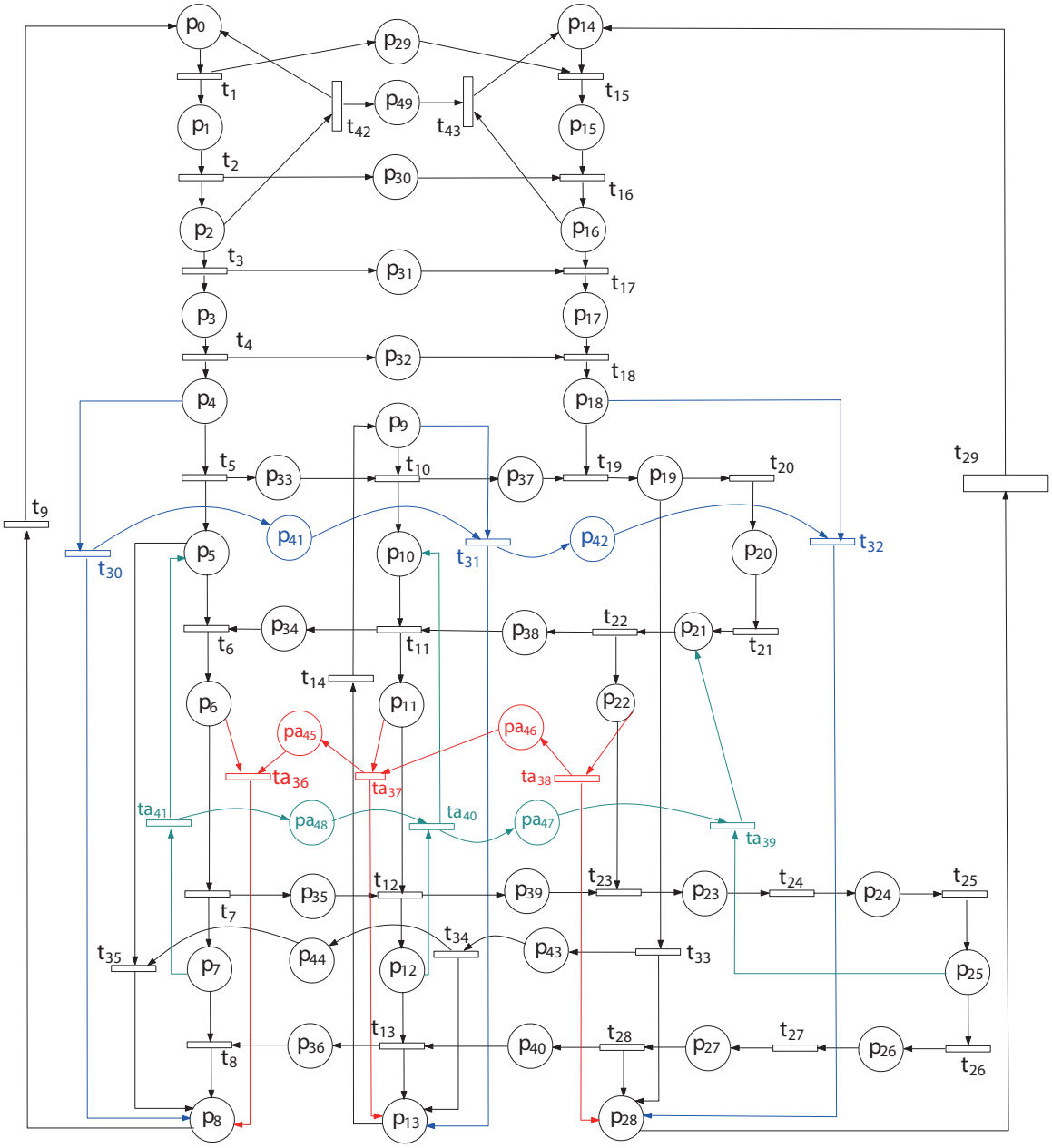


Figure 8 Petri net model after modification.

t1-M43, M44-t32-M29, M47-t1-M44, M45-t31-M46, M49-t9-M45, M46-t14-M47, M46-t32-M27, M50-t9-M46, M47-t32-M48, M51-t9-M47, M52-t9-M48, M49-t31-M50, M8-t30-M49, M50-t32-M2, M50-t14-M51, M51-t32-M52}. Algorithm 3 outputs the test cases that cover the deleted nodes, which means these test cases can not be reused. There are new test cases. Finally, we have 8585 reused test cases. The whole test cases is 8585.

New Test Cases For Change 2:

Since the red part is inserted into PN_0 in Figure 8, we get the sets of insertion of transitions and links as ${}^{+}T$={t36, t37, t38}, and ${}^{+}L$={p45, p46}. Based on rules, we generate RG_1 as shown in Figure 10. IEDGE = {M21-t36-M52, M52-t22-M53, M52-t40-M56, M53-t40-M54, M21-t41-M54, M54-t11-M55, M54-t14-M14, M55-t28-M28, M56-t22-M54, M56-t41-M13}. Algorithm 4 outputs the test cases that cover the inserted edge. All of the test cases (i.e., 8585 test cases) in the previous step can be reused as shown in Table 7. The number of new test cases is 14651 shown in Table 8. These test cases correspond to the function of Timeout. The number of whole test cases is 23236.

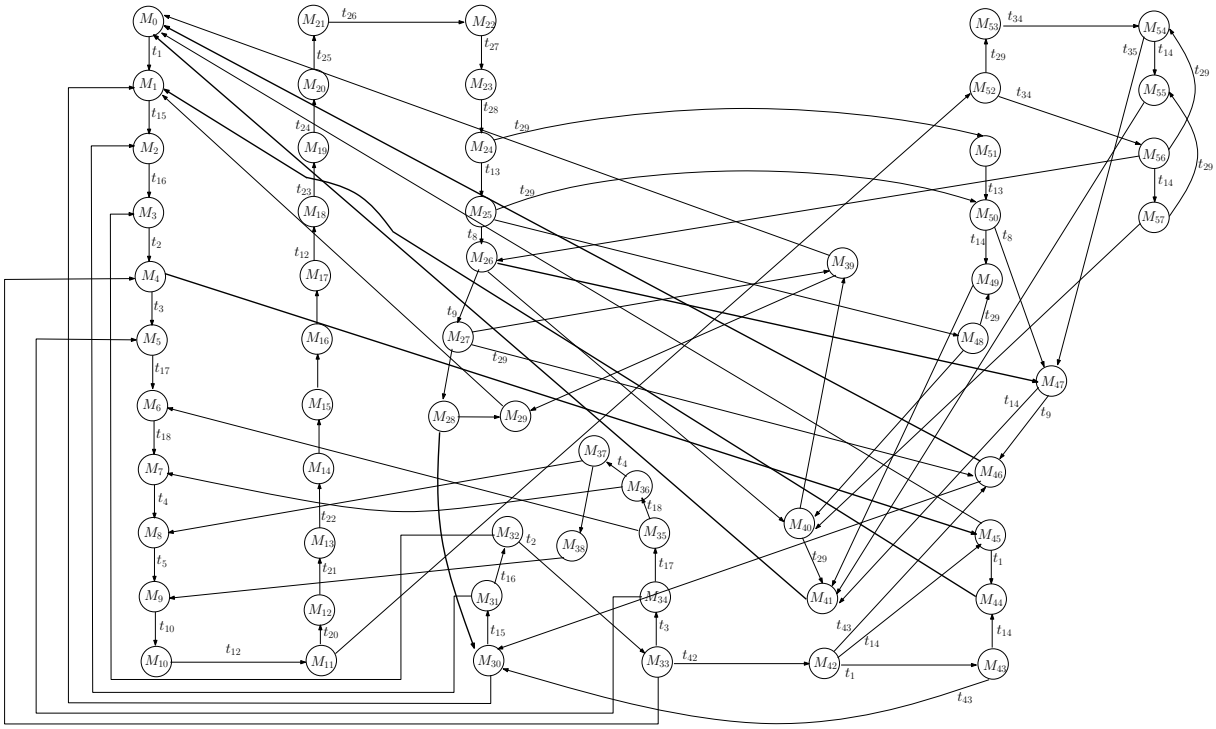


Figure 9 Step1: Reachability Graph.

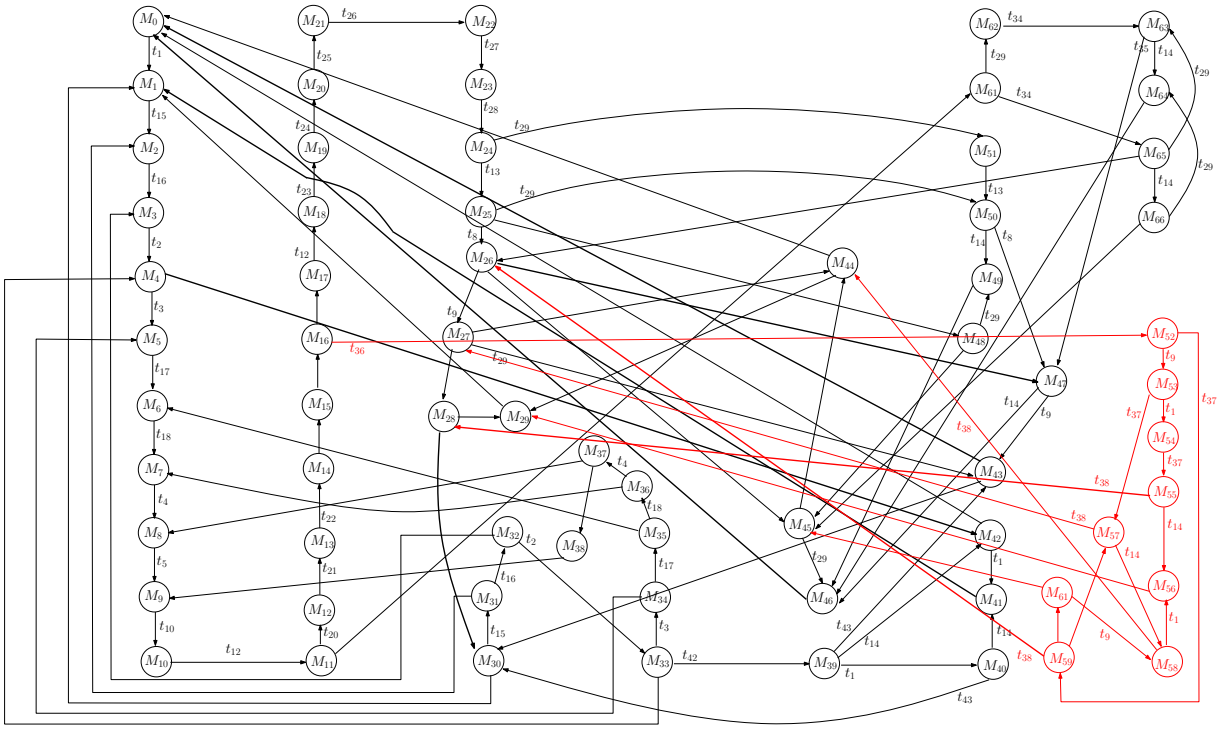


Figure 10 (Color online) Step2: Reachability Graph.

New Test Cases For Change 3:

Since green parts are inserted into the net as shown in Figure 8, we get the sets of insertion of transitions and links as ${}^{+}T$={t39, t40, t41}, ${}^{+}L$={p47, p48}. Based on rules, we generate RG_2 as is shown in Figure 11. IEDGE = {M21-t39-M52, M52-t9-M53, M52-t37-M59, M53-t1-M54, M53-t37-M57, M54-t37-M55, M55-t28-M28, M55-t14-M56, M57-t1-M55, M56-t39-M29, M58-t39-M56, M59-t32-M57,

Table 7 Reused test cases.

Reuseable test cases
trace1: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,17,18,19,20,21,22,23,24,25,26,27,43,0
trace2: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,17,18,19,20,21,22,23,24,25,26,45,44,0
trace3: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,17,18,19,20,21,22,23,24,25,48,45,44,0
trace4: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
17,18,19,20,21,22,23,24,51,50,47,43,0
trace5: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,52,53,54,47,43,0
trace6: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,52,53,54,47,
43,30,1,2,3,4,42,0
. . .
trace8584**:**0,1,2,3,4,42,41,1,2,3,4,5,6,7,8,9,10,11,52,56,57,55,
46,0
trace8585**:** 0,1,2,3,4,42,41,1,2,3,4,42,0

Table 8 New test cases.

New test cases of step2
trace1 **:** 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,52,53,54,55,28,29,1,2,3,4,42,0
trace2 **:** 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,52,53,57,27,43,0
trace3 **:** 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,52,53,57,27,44,0
trace4 **:** 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,52,53,57,58,44,0
trace5 **:** 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,52,59,26,27,43,0
trace6 **:** 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,22,23,24,25,26,27,28,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
16,52,59,26,27,44,0
. . .
trace14650 **:** 0,1,2,3,4,42,41,1,2,3,4,5,6,7,8,9,10,11,61,65,
66,45,44,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,52,59,60,45,46,0
trace14651 **:** 0,1,2,3,4,42,41,1,2,3,4,5,6,7,8,9,10,11,61,
65,66,45,44,29,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,52,59,60,58,44,0

M57-t14-M58, M57-t38-M27, M58-t38-M48, M61-t9-M58, M59-t38-M26, M59-t14-M61, M61-t38-M45}. Algorithm 4 outputs the test cases that cover the inserted edges. All of the test cases in the previous step can be reused (i.e., 23236 test cases). The number of new test cases is 3976 as shown in Table 9. These test cases correspond to the function of changing payment. The number of whole test cases is 27212.

5.5 Analysis results

The number of original test cases is 23602. After the first modification, there are 8585 test cases left. After the second modification, 14651 new test cases are added, and we get 23236 test cases. After the

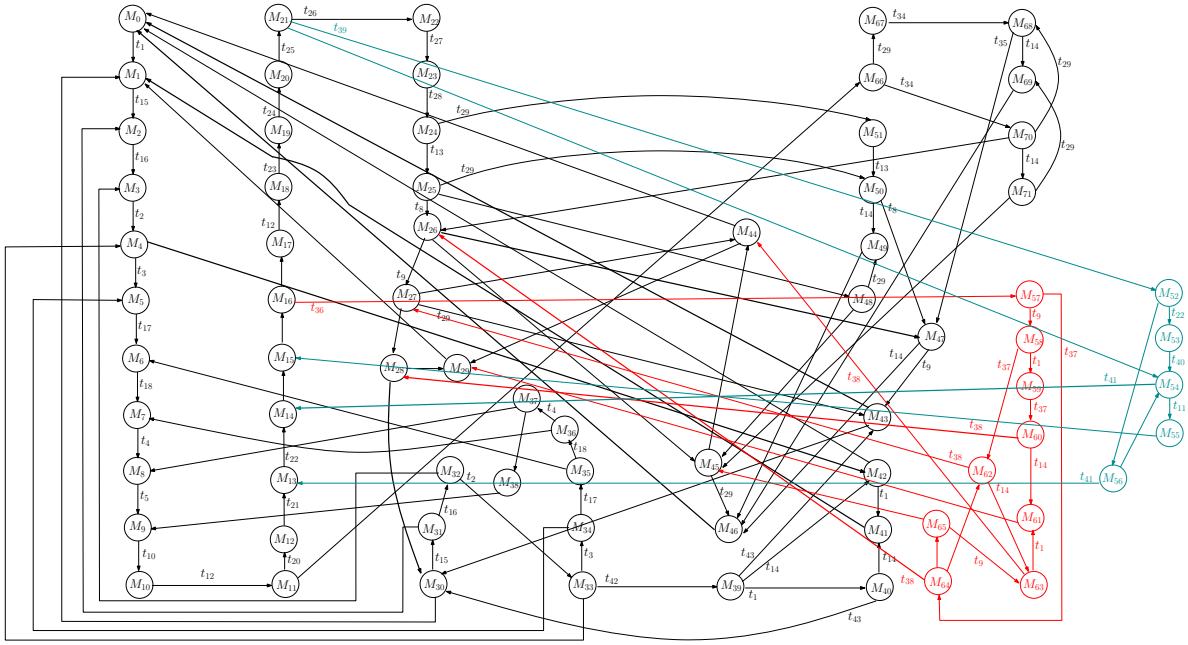


Figure 11 (Color online) Step3: Reachability Graph.

third modification, we add 3976 new test cases and the total number of test cases is 27212. Thus, after modification, using our method, we only need to construct test oracles for $14651 + 3796 = 18447$ new test cases, rather than the overall 27212 test cases. Therefore, by using the proposed method, we saved $\frac{27212-18447}{27212} = 31.5\%$ efforts related to the construction of test oracles.

Remark 3. MIS contains 19 use cases, which are adopted as an example to illustrate our technique. However, it can be applied to other big and complex systems. The reasons are located in following two aspects. (1) our model is scalable since each subsystem is described by a Closed Process Net and subsystems are communicated through rendezvous communication. Thus if a new component is added, we just add a closed process net. If we add or modify a test case, we only need to perform the operations of addition/insertion /deletion of transitions/links. (2) The complexity to generate test cases from modified Petri net is $O(|\mathrm{TS}_0|(x_1 \times n_1 + x_2 \times n_2))$, where $n_1 = |inEDG|$, $n_2 = |deEDG|$, x_1 and x_2 represent the average states and edges for a use case. We will apply our technique to more real systems later. In fact, in our other work [12], with the same language, we have modeled Requirements Specification of European Integrated Railway Radio Enhanced Network (EIRENE)[2)].

6 Threats to validity

External (projects and tools): Our results may not generalize to projects beyond the scope of those used in our evaluation. The reason is that the proposed Petri net model is applied to a class of systems with message changing to connect subsystems. For example, it can not model unbounded number of processes like Erlang. To mitigate this threat, we have also modeled the Functional Requirements Specification of European Integrated Railway [8]. We have used a tool developed by us[3)] to automatically model textual use cases with Petri net. Although we can extract all the required values for the use case model, it is possible that we may lose some information that is not considered in the use case model, such as those represented by fuzzy words. Another threat is that we measure the test cases by the paths that are not longer than 60 nodes because of the limitation of hardware configuration, which may not be true for other projects.

2) https://www.transportstyrelsen.se/globalassets/global/jarnvag/vagledning/godkannande/eirene_srsv15.pdf
3) https://sourceforge.net/projects/texturaluc2pn/

Table 9 New test cases.

New test cases of step3
trace1: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,52,53,54,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,1,
2,3,4,5,6,7,8,9,10,11,66,67,68,47,43,0
trace2: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,52,53,54,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,1,
2,3,4,42,0
trace3: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,52,53,54,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,1,
2,3,4,42,41,1,2,3,4,42,0
trace4: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,52,53,54,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,30,1,
2,3,4,42,0
trace5: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,52,53,54,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,30,1,
2,3,4,42,41,1,2,3,4,42,0
trace6: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
19,20,21,52,53,54,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,30,31,
2,3,4,42,0
...
trace3975**:** 0,1,2,3,4,42,41,1,2,3,4,5,6,7,8,9,10,11,12,
13,14,15,16,17,18,19,20,21,52,56,54,55,15,16,57,64,65,63,44,29,1,2,
3,4,42,0
trace3976**:** 0,1,2,3,4,42,41,1,2,3,4,5,6,7,8,9,10,11,12,13,
14,15,16,17,18,19,20,21,52,56,54,55,15,16,57,64,65,63,61,29,1,2,3,4,42,0

Internal (correctness of our implementation): To the best of our knowledge, there is no algorithm to generate reachability graph based on the operations to a Petri net. Thus there are no implementations for such an algorithm. Therefore, we implement those algorithms ourselves. To ensure the correctness of our implementation, we adopt metamorphic testing (MT) strategy. MT is effective in alleviating the test oracle problems and has been applied to many problems. It tells whether the program is faulty or not by checking the violation of some properties each of which is known as a metamorphic relation (MR) of the given program instead of checking the single output of one given test input.

Construct (metrics and versions): From this experiment, we see that once we add a transition or link to the net, we get new test cases. But this is not always true. Sometimes, when we add them, the number of nodes in RG may decrease, and thus the test case count may decrease. Thus we need to know what kind of structure will bring us such a situation. Another threat comes from the impact of the changed requirements to the system. We need to design a metric to measure such impacting.

7 Relate work

There has been a lot of work on regression testing in general, as surveyed in two reviews [13–15]. Model-Based Testing (MBT) uses high level artifacts as basis for test generation. It is regarded to be complementary to code-based testing (CBT). Models can be built from source code [16–18], but they are generally too expensive when applied to large programs [19]. Next we will discuss the approaches on the models from high level analysis.

State based: Korel et al. [2] present Extended Finite State Machine (EFSM) based approach to reduce regression test suites. Their approach can identify the difference between the original and the new models as a set of elementary model modifications. For each elementary modification, they perform regression test reduction to reduce the regression test suite based on EFSM dependence analysis. Our model is

based on Petri net, taking care of local states and communications between subsystems. Chakrabarti and Srikant [20] extract a finite state model based on state space enumeration to compute test sequences to verify subsequent versions of the system. Our target is to maximally reuse the original test suite based on the new model rather than generating all new test cases from scratch. Jiang et al. [3] use nondeterministic Action Machines to model systems. To determine whether a test case is obsolete with respect to the new model, they match every invocation/event sequence of the original model in the new model graph by means of graph analysis. If we can successfully match a sequence, then the corresponding test case is reusable; otherwise, it is obsolete. Our model is Petri net, which can well describe concurrent systems and is famous for its ability to model synchronization. Our test cases reduction is based on graph analysis for reachability graph.

UML based: Farooq et al. [21] use UML state machines and class diagrams for regression test case selection. Based on the impact of changes of class diagrams to state machines, they find new test cases. Naslavsky et al. [22] adopt UML class and sequence diagrams as its modeling perspective. They compare two versions of models to create a differencing model and transform sequence diagrams into modified models, then use the differencing model and traceability models to do a pairwise graph traversal of original and modified models, finally use selected dangerous entities identified to classify test cases. Chen et al. [23] propose two regression test selection techniques. The first one targets at assuring important customer features to be adequately retested. The second one uses risk analysis to ensure that potentially high risk problem areas are properly exercised. Their work is based on UML activity diagrams. Briand et al. [24] propose a technique to classify regression test cases as obsolete, re-testable, and re-usable ones. Their work is based on UML sequence diagrams and class diagrams. Our technique is based on a model that can well describe the behavior of concurrent systems. The test cases selection is based on graph analysis. We can maximally use the old test cases, and generate new test cases to cover those parts of the new model not covered by any original test cases.

Component based: Tao et al. [25] conduct regression testing of component-based software from API view, interaction view, and state-based view. They identify diverse changes made to components, then perform change impact analysis, and finally refresh regression test suite using a state-based testing practice. Orso et al. [26] propose an approach for the regression testing of COTS components based on component metacontents consisting of metadata and metamethods. They need to identify the metacontents, information about the coverage achieved by the test suite on the original version of the software, and the changes made to the set of components, and then use metacontents for regression test selection. Zheng et al. [27] propose an approach to identify component change for COTS (Commercial-off-the-shelf) software. Regression test selection is based on the glue code that interfaces with the sections of the changed component. Our approach is based on behavior model and graph analysis, focusing on test case reusing from old test suite and new test case generation for changed parts.

8 Conclusion

This paper presents a method to improve regression testing. We use Petri nets to model requirements. We identify reusable test cases of the original test suite based on reachability graph analysis. Therefore, we can generate new test cases to cover only the change-related parts of the new model.

There are two issues we need to consider in the future. The first one is how to guarantee that the changes of requirements can be automatically reflected in the model. The second one is to remove the constraints from the definition of a test case. After solving the first issue, the test case regeneration can be automated. After solving the second issue, our technique can be readily applied to large scale system development.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 61210004, 61572441).

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Tahat L H, Bader A, Vaysburg B, et al. Requirement based automated black-box test generation. In: Proceedings of the 25th Annual International Computer Software and Applications Conference, Chicago, 2001. 489–495
2 Korel B, Tahat L H, Vaysburg B. Model based regression test reduction using dependence analysis. In: Proceedings of the IEEE International Conference on Software Maintenance. Los Alamitos: IEEE, 2002. 214–223
3 Jiang B, Tse T H, Grieskamp W, et al. Regression testing process improvement for specification evolution of real-world protocol software. In: Proceedings of the 10th International Conference on Quality Software, Zhangjiajie, 2010. 62–71
4 Garey M R, Johnson D S. Computers and Intractability: a Guide to the Theory of NP-completeness. New York: W. H. Freeman, 1979
5 Ding Z H, Jiang M Y. Model driven synthesis of behavioral models from textual use cases. Inform Control Automat Robot, 2011, 132: 713–717
6 Cockburn A. Writing Effective Use Cases. Upper Saddle River: Addison-Wesley, 2001
7 Somé S. Petri nets based formalization of textual use cases. Technical Report TR-2007-11. University of Ottawa, 2007
8 Ding Z H, Jiang M Y, Zhou M C. Generating Petri net-based behavioral models from textual use cases and application in railway networks. IEEE Trans Intell Transp Syst, 2016, doi: 10.1109/TITS.2016.2518745
9 Sinha A, Amit P, Clay W. On generating EFSM models from use cases, scenarios and state machines. In: Proceedings of the 6th International Workshop on Scenarios and State Machines. Washington, DC: IEEE, 2007. 1
10 Singh S K, Gupta R, Sabharwal S, et al. Automatic extraction of events from textual requirements specification. In: Proceedings of the World Congress on Nature & Biologically Inspired Computing, Coimbatore, 2009. 415–420
11 Myers G J. The Art of Software Testing. 3rd ed. Hoboken: John Wiley & Sons, 2011
12 Plasil F, Vladimir M. Use cases: assembling "whole picture" behavior. TR 02/11. Department of Computer Science, University of New Hampshire, 2011
13 Biswas S, Mall R, Satpathy M, et al. Regression test selection techniques: a survey. Informatica, 2011, 35: 289–321
14 Engstrom E, Runeson P, Skoglund M. A systematic review on regression test selection techniques. Inform Softw Technol, 2010, 52: 14–30
15 Yoo S, Harman M. Regression testing minimization, selection and prioritization: a survey. Softw Test Verif Reliab, 2012, 22: 67–120
16 Panigrahi C R, Mall R. Model-based regression test case prioritization. ACM SIGSOFT Softw Eng Notes, 2010, 35: 6
17 Rothermel G, Harrold M J. A safe, efficient regression test selection technique. ACM Trans Softw Eng Methodol, 1997, 6: 173–210
18 von Mayrhauser A, Zhang N. Automated regression testing using DBT and Sleuth. J Softw Maint, 1999, 11: 2
19 Graves T L, Harrold M J, Kim J-M, et al. An empirical study of regression test selection techniques. ACM Trans Softw Eng Methodol, 2001, 10: 184–208
20 Chakrabarti S K, Srikant Y N. Specification based regression testing using explicit state space enumeration. In: Proceedings of the International Conference on Software Engineering Advances, Tahiti, 2006. 20
21 Farooq Q, Iqbal M, Malik Z I, et al. An approach for selective state machine based regression testing. In: Proceedings of the 3rd International Workshop on Advances in Model-Based Testing. New York: ACM, 2007. 44–52
22 Naslavsky L, Ziv H, Richardson D J. A model-based regression test selection technique. In: Proceedings of International Conference on Software Maintenance, Edmonton, 2009. 515–518
23 Chen Y P, Probert R L, Sims D P. Specification-based regression test selection with risk analysis. In: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, Indianapolis, 2002. 1
24 Briand L C, Labiche Y, He S. Automating regression test selection based on UML designs. Inf Softw Technol, 2009, 51: 16–30
25 Tao C Q, Li B X, Gao J. A systematic state-based approach to regression testing of component software. J Softw, 2013, 8: 560–571
26 Orso A, Harrold M J, Rosenblum D, et al. Using component metacontents to support the regression testing of component-based software. In: Proceedings of the IEEE International Conference on Software Maintenance, Florence, 2001. 716–725
27 Zheng J, Robinson B, Williams L, et al. Applying regression test selection for COTS-based applications. In: Proceedings of the 28th IEEE International Conference on Software Engineering, Shanghai, 2006. 512–521