Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 329 (2016) 123–148

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Test Scenario Generation from Natural Language Requirements Descriptions based on Petri-Nets

Edgar Sarmiento [1,2] Julio C. S. P. Leite [3]

Departament of Informatics
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil

Eduardo Almentero [4]

Mathematics Department
Universidade Federal Rural do Rio de Janeiro
Rio de Janeiro, Brazil

Guina Sotomayor Alzamora [5]

Instituto de Matemática y Ciencias Afines
Universidad Nacional de Ingeniería
Lima, Peru

Abstract

Test generation from functional requirements in natural language (NL) is often time-consuming and error prone, especially in complex projects. In this context, formal representations like Petri-Nets are increasingly used as input for automated test scenario generation. However, formal representations are not trivial, and it requires a strong knowledge on formal modeling. In this paper we propose an approach to generate test scenarios that takes as input a Restricted-form of Natural Language (RNL) requirements specification. This approach translates automatically RNL requirements specified as Scenarios into executable Petri-Net models; these Petri-Nets are used as input model for test scenario generation. Our approach checks the quality of the input models and aims to decrease the time and the effort with respect to test scenario generation process. Demonstration of the feasibility of the proposed approach is based on an example of use that describes the operation of the approach.

Keywords: scenario, requirements, petri-nets, testing, test scenario.

[1] Thanks to CAPES funding agency for financial support

[2] Email: ecalisaya@inf.puc-rio.br

[3] Email: julio@inf.puc-rio.br

[4] Email: almentero@ufrrj.br

[5] Email: guinas@gmail.com

http://dx.doi.org/10.1016/j.entcs.2016.12.008

1 Introduction

Software testing is one of the validation techniques most commonly used, this approach improves the quality of the final product, particularly checking that the software behavior meets its requirements. However, test generation and execution tasks are still quite expensive and usually done manually, then the automation of testing process is a challenging topic.

The Model-based Testing (MBT) is an alternative to the automation of these tasks, in which tests are derived from system specifications. Thus, the expected system behavior is described using formal specification notations [2]. MBT refers to black-box testing method in which test scenarios and oracle are automatically generated from a formal and functional model of a System Under Test (SUT). An important benefit of MBT, among other things, is automatically generate test scenarios from a model of a system under test and the automatically validate these test scenarios by executing the system under test and comparing their results against the expected results. The main shortcomings of MBT are the model construction and selection of suitable formal notations [24], often most of the proposed approaches require manual intervention or the creation of additional complex behavioral models. According to [25], this significantly hinders their applicability in practice.

In most of existing approaches for generating test scenarios for model-based systems, testing practitioners usually decompose the system in different use scenarios, then, formal representations are created for each identified scenario. The test scenarios are derived from these intermediate formal representations. According to [2], the quality of these specifications is crucial for an effective testing campaign; thus, it is desirable to describe the expected system behavior via some (semi-)formal notations. Examples of formal notation are Petri-Nets [17] or Communicating Sequential Processes (CSP) [19].

The use of (semi-)formal notations facilitates the process of test automation. However, this practice is expensive and not widely used in industrial practice. On the other hand, in order to allow for an easy communication between clients and developers, natural language-based representations are frequently used in Requirements Engineering. In this context, functional requirements are represented as scenarios and described by specific flows of events, which are based on user perspective. The use of scenarios helps understanding a specific situation in an application, prioritizing their behavior [14]. Some of the most prominent languages to write scenarios are restricted-form of use case descriptions [5], [9]; scenario representation [14]; UML dynamic behavior diagrams; and Message Sequence Charts [1]. Although some of these languages provide an accessible visualization of models, they lack formal semantics to support further analysis or test generation.

In this context, *scenario specifications* are usually informal or semi-formal, and due to natural language ambiguity, they cannot be used directly for MBT activities. In order to perform an automated MBT from these scenarios, it is necessary: (i) to detect and fix defects within scenarios; (ii) to translate them from informal to formal representations, like Petri-Nets; and (iii) to derive testing from formal

representations. Petri-Nets [17] are formal models based on strict mathematical theories, they provide a mathematical simplicity for the modeling and simulation of concurrent systems and the analysis of properties by the reachability tree. The translation of textual scenarios into Petri-Net models and the generation of test scenarios from these formal models are challenging topics, because the tasks involved in these processes are costly especially if we consider large systems.

We propose here an automated approach to generate test scenarios from natural language requirements specifications; our approach uses a RNL to describe scenarios (conforming to a metamodel) as input, derives an equivalent Petri-Net model (conforming to a restricted Petri-Net metamodel) and generates test scenarios as output (sequence of event transitions and guard conditions). The proposed approach is composed of a scenario verification module (detect defects), a model transformation method (defined as mapping rules) and criteria to generate test scenarios traversing the reachability tree of Petri-Nets. These phases are being implemented in the C&L [4] prototype tool.

This paper is organized as follows. Section 2 describes an overview of the approach. Section 3 presents the language used to write scenarios. Section 4 describes the transformation from scenario into Petri-Net. Section 5 presents the test scenario generation strategy. Finally, Section 6 presents some related work to our proposal, the conclusions, limitation and some suggestions for future work.

2 Strategy Overview

For practical reasons, and in order to allow for an easy communication with stakeholders, informal or semiformal representations are widely used by user-oriented approaches. User-oriented approaches are dominant during Requirements Engineering activities in industry; and, one of the key elements in this perspective is the notion of *scenarios*.

In literature, the term *scenario* is used with different meanings in different contexts, and there is no clear distinction between scenarios and use cases. While some authors consider that each scenario corresponds to one use case [8], others define a scenario as sequences of use case steps that represent different paths through a use case [5]. According to Glinz [8], a scenario may comprise a concrete sequence of interaction steps (*instance scenario*) or a set of possible interaction steps (*type scenario*). The most common components used to detail scenarios are: Title/Name, Goal, Pre-condition, Post-condition, Actors, Episodes/Main Flow and Exceptions/Alternative Flows.

The proposed approach focuses on *scenario specifications* because these are used as input in other development activities such as design, coding and testing. They constitute an essential part of the software development process. Therefore, we need to ensure that the different situations in the application are described into clear and well-defined *scenario* representations [14].

So, in our approach: **First**, requirements engineers start to describe the different functionalities, services or situations of the system by using the scenario language

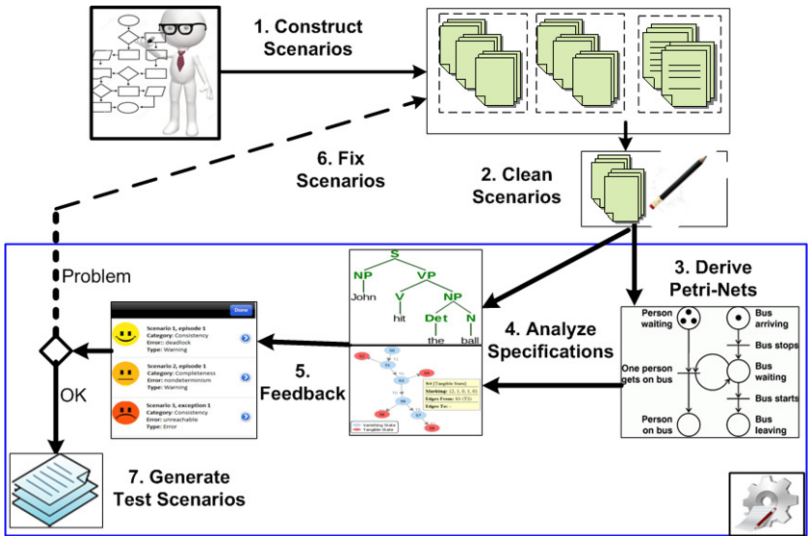

Fig. 1. Overview of the proposed approach.

presented in previous works [22] and [20]. **Second**, irrelevant information within scenario elements are removed. **Third**, in order to perform an automated test scenarios generation from scenario representations, an initial system design is derived by translating these *scenarios* into *Place/Transition* Petri-Nets, and synthesizing them into a consistent whole Petri-Net. **Fourth**, *scenarios* and their resulting Petri-Nets are automatically analyzed to detect and fix defects that hurt *Unambiguity*, *Completeness*, *Consistency* and *Correctness* properties. **Fifth**, the analysis outcome is formatted and returned to the requirements engineers. **Sixth**, if defects are found, the analysis feedback is used to improve the scenario descriptions, since the identified problems can be traced to the scenarios. **Seventh**, if no defects are found, the resulting Petri-Nets are used to derive test scenarios. Figure 1 depicts an overview of our approach. The different phases of our approach are being implemented in the C&L [4] tool. The steps one to six are fully implemented and detailed in [20] and [22].

In our proposal, each scenario sentence (imperative or declarative) is translated into a Petri-Net node (transition or place). These Petri-Net nodes are linked by arcs giving rise to a Petri-Net model. Each translated scenario defines part of the system formal specification.

Based on the generated Petri-Net model, the different sequence of events –paths of the system– can be validated by simulation. The available Petri-Net tools [16] enable the requirements engineers to simulate the scenario behavior using the equivalent Petri-Net model. They also allow detailing scenario's sentences, by rewriting them as more concrete ones, in order to ease the understanding of its behavior. Moreover, these tools also enable the verification of structural and behavioral properties (reachability, boundedness and liveness); with the feedback provided by these tools, the requirements engineer can improve the scenario descriptions and then start the process again. If no problem is found, these *scenarios* and *Petri-Net*

models can be used in next activities of the software development process like MBT.

By formally representing a system; Petri-Net models can be used as an input to the model-based testing process, automating some repetitive and time-consuming tasks such as *test scenario generation* or *test execution*. An important benefit of MBT is automatically generating test scenarios from a formal model of a SUT [18]. The automated translation of formal Petri-Net models from RNL scenario specifications overcomes the main shortcoming of MBT, which is the formal model construction.

In our proposal, *Test Scenario Generation* process can be carried out automatically using the generated Petri-Net models from *scenario specifications*; the goal is to translate every sequence of events or transitions into atomic steps, enabling its implementation. Consequently, this will allow the automatic execution of the *test scenarios*.

3 Writing RNL Scenarios

As mentioned before, the language used to write *scenarios* is a RNL. Using RNL it is possible to write *imperative* and *declarative* sentences. An imperative sentence describes actor events and a declarative sentence describes actor or resource states. Thus, software requirements specifications can be described as clear and well-defined scenario descriptions.

The use of RNL restricts the vocabulary used to write scenarios and prevents the introduction of ambiguous sentences in the scenario specification, contributing to the quality of documentation. RNL is also necessary to define syntax rules for sentences construction. Moreover, it helps the automatic transformation of textual scenarios into formal executable models.

The natural language based-scenario used in this work is an adaptation of [14] and proposed in [20].

3.1 Scenario

Scenario specifications capture system behaviors or situations in the domain [14] and, it helps the understanding of the requirements by the developers and other stakeholders. The following scenario definition enables a further transformation:

Definition 3.1 A *scenario* is a collection of partially ordered *event occurrences*, each guarded by a set of *conditions* (*pre-condition* and *post-condition*) or restricted by *constraints* [20]. An *event* is an actor operation or an interaction involving users, system, environment, or system's components. A *condition* is an actor/resource/system state (e.g. the availability of some resource). An *actor* can be a user, a device, the system, system's components or agents in the environment; they have a role in the scenario or influence the system.

Figure 2 presents an abstract conceptual model for the scenario language used in this work, using a class diagram. According to this conceptual model, the scenario

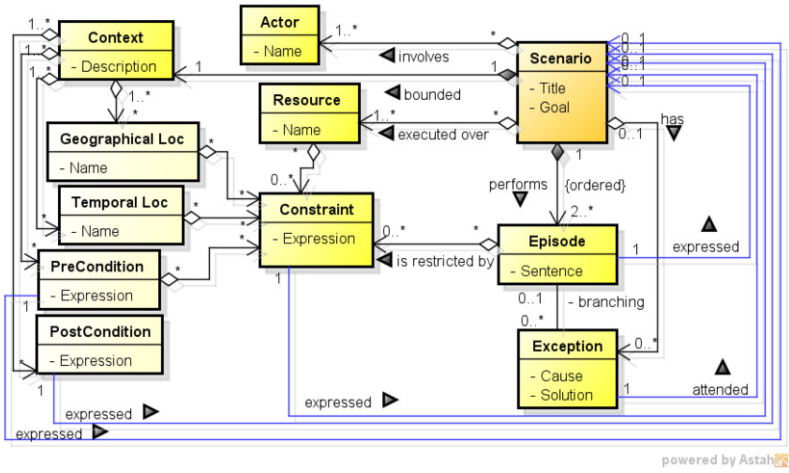

Fig. 2. Scenario conceptual model [20]

language is composed of the main entity Scenario, and the Context, Resource, Actor, Episode, Exception and Constraint entities.

A *scenario* starts in an initial state (Context) with all necessary Resources and Actors, and must satisfy a Goal that is reached by performing its Episodes. The *episodes* describe the operational behavior of the *situation*, which includes the main course of action and possible alternatives. An Exception can arise during the execution of *episodes*, and indicates that there exists an obstacle to satisfy the *goal*. The treatment to this exception does not need to satisfy the scenario *goal*. A *scenario*, *pre-condition*, *post-condition*, *constraint*, *episode* or *exception* can be expressed by another *scenario*.

Table 1 shows the template for writing a scenario based on the RNL scenario language proposed in [20]. It explains what each scenario element means and how it should be filled.

Episodes represent the main flow of actions, which is a sequence of steps where everything works as expected. Exceptions are situations that prevent the proper course of the scenario. Its treatment should be described using a sentence. It also represents the alternative/exceptional flows. Constraints are non-functional aspects described as declarative sentences that restrict the quality with witch the *goal* is achieved, *resources* are needed and *episodes* are performed. It is an attribute of resources, episodes or context's sub-components.

The *episode sentences* are simple, conditional, optional or loop. Simple episodes are those necessary to complete the scenario. Conditional episodes are those whose occurrence depends on internal or external condition. Optional episodes are those that may or may not take place depending on conditions that cannot be detailed. Loop episodes can be used as repetition structures whose occurrence depends on internal or external condition. Internal conditions may be due to alternative pre-conditions, actors or resources constraints and previous episodes. External conditions may be provided by external actors or another scenario.

A sequence of episodes implies a precedence order, but a non-sequential order

Table 1
Template for writing scenarios.

Element	Description	
Title	<Identifies the scenario and must be unique>	
Goal	<Describe the purpose of the scenario>	
Context	<Must be described through at least one of these options: > <Describes the scenario initial state>	
	Geographical location	<represents the physical set of the scenario>
	Temporal location	<is the time specification for the scenario development>
	Pre-condition	<the initial state of the scenario or the name of other scenario>
	Post-condition	<the final state of the scenario or the name of other scenario>
Resources	<enumeration of relevant physical elements or information used by the scenario to achieve its goal> <Must appear in at least one of the episodes>	
Actors	<enumeration of persons, devices or organization structures directly involved with the situation> <Must appear in at least one of the episodes>	
Episodes	Id	1
	Sentence	<performs an action that can use/modify resources and be executed by actors> <or the name of other scenario>
	Pre-condition (*)	<that we expect are already satisfied before the episode is performed>
	Post-condition (*)	<that we expect will be achieved after the episode occurs>
	Constraint (*)	<restrict the quality with witch the episode is performed>
Exception	Id	1.1
	Cause	<caused by invalid input data or the lack or malfunction of a necessary resource>
	Solution	<is treated by an imperative sentence> <or the name of other scenario>
	Post-condition (*)	<may generate effects on the resource/system states, or simply produce a message>

(*) optional

can be bounded by the symbol "#" allowing the grouping of two or more episodes. This is used to describe indistinct sequential order, concurrent or parallel episodes (#<Episode Series>#).

Context's sub-components, Episode's conditions and Exception's causes may be expressed by one or more simple sentences linked by the logical connectors AND or OR.

Table 2 and Table 3 present scenario examples. These examples themselves are explained later. The "Submit Order" scenario described in Table 2 describes the interactions between the "Online Broker System" and its partner services, **Local Supplier** and **International Supplier** (Table 3). These examples were initially presented as use case descriptions in the "Online Broker System" case study by Somé [23].

3.2 Restricted-form of Natural Language (RNL)

In order to reduce ambiguity in natural language requirements and facilitate the model transformation, in [20] is defined a *scenario grammar* for writing sentences in accordance to its conceptual model (Figure 2).

Using this grammar, it is possible to write imperative (Title, Episode sentence or Exception solution) and declarative (Conditions or States) sentences. An imperative sentence describes Actor events; and a declarative sentence describes Actor or Resource states (or Conditions).

A sentence in scenario grammar is basically defined according to the format "**Subject+Verb+Predicate**", where *subject*, *verb* and *predicate* represent the

Table 2
Description of scenario "Submit Order" in the Broker System [20].

TITLE: **Submit Order**
GOAL: Allow customers to find the best supplier for a given order.
CONTEXT:
PRE-CONDITION: The Broker System is online AND the Broker System welcome page is being displayed
ACTOR: Customer, Broker System
RESOURCES: Login page, Login information, Order
EPISODES
1. The Customer loads the login page
2. The Broker System asks for the Customers login information
3. The Customer enters her login information
4. The Broker System checks the provided login information
5. The Broker System displays an order page
6. The Customer creates a new Order
7. DO the Customer adds an item to the Order WHILE the Customer has more items to add to the order
8. The Customer submits the Order
9. The Broker System broadcast the Order to the Suppliers
10. # LOCAL SUPPLIER BID FOR ORDER
11. INTERNATIONAL SUPPLIER BID FOR ORDER #
12. PROCESS BIDS
EXCEPTIONS
1.1 IF Customer is not registered THEN REGISTER CUSTOMER
2.1 IF after 60 seconds THEN The Broker System displays a login timeout page
4.1 IF the Customer login information is not accurate THEN The Broker System displays an alert message

subject, main verb and objects affected by the main verb, respectively. Therefore, the sentence construction is centered on the main verb.

Table 4 shows the grammar for writing scenario elements using partial Extended-BNF. According to this grammar, a Scenario must be described by the attributes: Title, Goal, Context, Resource, Actor, Episodes and Exception.

In Table 4, + means composition, $\{x\}$ means 0 or more occurrences of x, $\{x\}_1^N$ means 1 or more occurrences of x, () is used for grouping, | stands for OR and $[x]$ denotes that x is optional. The following words contain only terminal symbols: Phrase, Verb, Predicate, Name, Action-Verb, Linking-Verb, Letter, and Digit. The following words and phrases are terminal symbols: TITLE, GOAL, CONTEXT, RESOURCE, ACTOR, EPISODES, EXCEPTION, GEOGRAPHICAL LOCATION, TEMPORAL LOCATION, PRE-CONDITION, POST-CONDITION, CONSTRAINT, IF, THEN, WHILE, DO, AND, OR, MUST, NOT, "[" and "]".

A *episode sentence* and *exception solution* are declared according to the format "[Actor | Resource] + Action-Verb + [Direct-Object-Predicate]", where "Action-Verb" express action, and "Direct-Object-Predicate" refers to an object affected by the action. In the sentence "The Customer submits the Order", the word "submits" is an Action-Verb and the word "Order" is the Direct-Object-Predicate.

A *condition* (episode condition or exception cause) may be formally defined as a logical sentence declared according to the format "(Actor | Resource) + Linking-Verb + Predicate". A "Linking-Verb" (copular verb) is a word used to link the Subject (Actor or Resource) of a sentence with a Predicate (a subject complement), such as the word "is" in the sentence "Feeder area is available". Linking verbs are not followed by objects. Instead, they are followed by phrases which give extra

Table 3
Scenarios of the "Online Broker System" [20].

TITLE: **Local Supplier bid for order**
GOAL: Submit a bid
CONTEXT: Create a bid for an Order
PRE-CONDITION: An Order has been broadcasted
POST-CONDITION: Local Supplier has bidden
ACTOR: Local Supplier, Broker System
RESOURCES: Order, bid
EPISODES
1. Local Supplier receives the Order and examines it
2. Local Supplier determines the applicable taxes to the order and creates a bid
3. Local Supplier submits a bid for the Order
4. The Broker System sends the Bid to the Customer
EXCEPTIONS
1.1 IF Local Supplier can not satisfy the Order THEN Local Supplier passes on the Order
TITLE: **International Supplier bid for order**
GOAL: Process a bid
CONTEXT: Process a bid for an Order
PRE-CONDITION: An Order has been broadcasted
POST-CONDITION: International Supplier has bidden
ACTOR: International Supplier, Broker System
RESOURCES: Order, bid
EPISODES
1. International Supplier receives the Order and examines it
2. International Supplier submits a Bid for de Order
3. The Broker System sends the Bid to the Customer
EXCEPTIONS
1.1 IF The Order includes items restricted for exportation THEN International Supplier passes on the Order
1.2 IF International Supplier can not satisfy the Order THEN International Supplier passes on the Order
TITLE: **Process bids**
GOAL: Process a bid
CONTEXT: Process a bid for an Order
PRE-CONDITION: Local Supplier has bidden OR International Supplier has bidden
ACTOR: Customer, Broker System
RESOURCES: Order, bid
EPISODES
1. Customer examines the bid
2. Customer signals the system to proceed with bid
3. HANDLE PAYMENT
4. System put an order with the selected bidder

information about the *subject* (e.g. noun phrases, adjective phrases, adverb phrases or prepositional phrases). Linking verbs include the conjugated form of limited number of verbs.

Like *condition*, a State (*pre-condition* and *post-condition*) may be formally defined as a sentence declared according to the format "(Actor | Resource) + State-Verb + Predicate". A "State-Verb" expresses a state which is relatively static. They include verbs of perception, cognition, the senses, emotion and state of being. In the sentence "The buffer is empty", the word "is" is a State-Verb and the word "empty" is the Predicate. State verbs are not normally used in continuous forms. Some examples of linking-verbs and state-verbs are included in [20].

3.3 Scenario Relationship-based Modularity

The main benefit of using the *Scenario Language* is that scenario supports the composition of various scenarios using scenario relationships. This feature provides

Table 4
Scenario Grammar [20].

Type	Description
<Scenario>	TITLE: <Title> + GOAL: <Goal> + CONTEXT: <Context> + RESOURCE: {<Resource>}$_1^N$+ ACTOR: {<Actor>}$_1^N$+ EPISODES: <Episodes> + EXCEPTION: {<Exception>}
< Title >	([Actor | Resource]+ Action-Verb + Predicate) | Phrase
< Goal >	[Actor | Resource]+ Verb + Predicate
< Context >	[GEOGRAPHICAL LOCATION: <Geographical Location>]+ [TEMPORAL LOCATION: <Temporal Location>]+ [PRE-CONDITION: <Pre-condition>]+ [POST-CONDITION: <Post-condition>]
<Geographical Location>	Name +[CONSTRAINT: {<Constraint>}]| <Geographical Location><connective>< Geographical Location >
<Temporal Location>	Name +[CONSTRAINT: {<Constraint>}]| <Temporal Location>< connective><Temporal Location>
<Pre-condition>	<expression> | <Title> | <Pre-condition><connective><Pre-condition>
<Post-condition>	<expression> | <Title> | <Post-condition><connective><Post-condition>
<expression>	((Actor | Resource) + State-Verb + Predicate) | Phrase
<connective>	AND | OR
<Resource>	Name +[CONSTRAINT: {<Constraint>}]
<Actor>	Name
<Episodes>	<Group> | <Episodes><Group>
<Group>	<Sequential Group> | <Non-Sequential Group>
<Sequential Group>	<Episode><Episode> | <Sequential Group><Episode>
<Non-Sequential Group>	{<Episode>}# <Episode Series> #{<Episode>}
<Episode Series>	<Episode><Episode> | <Episode Series><Episode>
<Episode>	<Simple Episode> | <Conditional Episode> | <Optional Episode> | <Loop Episode>
<Simple Episode>	<Id><Episode Sentence> +[PRE-CONDITION: <Pre-condition>]+ [POST-CONDITION: <Post-condition>] + [CONSTRAINT: {< Constraint>}]
<Conditional Episode>	<Id> IF <Condition> THEN <Episode Sentence> +[PRE-CONDITION: <Pre-condition>]+ [POST-CONDITION: <Post-condition>] + [CONSTRAINT: {<Constraint>}]
<Optional Episode>	<Id> "[" <Episode Sentence> "]"+[PRE-CONDITION: <Pre-condition>]+ [POST-CONDITION: <Post-condition >] + [CONSTRAINT: {<Constraint>}]
<Loop Episode>	<Id> DO <Episode Sentence> WHILE <Condition> +[PRE-CONDITION: <Pre-condition >]+ [POST-CONDITION: <Post-condition>] + [CONSTRAINT: {<Constraint>}]
<Id>	<id-chair> { (. | , | ; | :) + <id-char>} + [. | , | ; | :]
<id-char>	Letter | Digit
<Episode Sentence>	([Actor | Resource]+ Action-Verb +[Direct-Object-Predicate]) | <Title>
<Condition>	<atomic sentence> | <Condition><connective><Condition>
<atomic sentence>	((Actor | Resource) + Linking-Verb + Predicate)| Phrase
<Exception>	<Id> IF <Cause> THEN <Solution> +[POST-CONDITION: <Post-condition>]
<Cause>	<atomic sentence> | <expression> | <Cause><connective><Cause>
<Solution>	([Actor | Resource]+ Action-Verb +[Direct-Object-Predicate]) | <Title>
<Constraint>	([Actor | Resource] + [MUST] + [NOT]+ Predicate) | <Title> | Phrase

modularity through the inter-connectivity among related scenarios. Modularity is considered a mechanism to deal with the scenario explosion problem [14], [18].

Scenarios are related to other scenario by sequential relationships. In a scenario description, if we include the title of another scenario (UPPERCASE sentence) within the context (*pre-condition* or *post-condition*), an *episode* (*sentence*), an *exception* (solution) or a *constraint*; this *context*, *episode*, *exception* or *constraint* will be treated by this last scenario. Scenarios are sequentially connected to other *scenarios* by:

- **Pre-condition** is a relationship defined within the context element of a scenario. A scenario that is pre-condition to other must be executed first.

- **Post-condition** is a relationship defined within the context element of a scenario. A scenario that is post-condition of other must be executed last.
- **Sub-scenario** is defined when an episode of a scenario can be described by another scenario. This allows the decomposition of complex scenarios.
- **Exception** relationship is defined when a scenario is used to detail the exceptional behavior (solution) of another. The exceptional scenario is executed when the exception is triggered in the main scenario.
- **Constraint** relationship is defined when a scenario is used to detail non-functional aspects that qualify/restrict the proper execution of another, which also give us an order among the scenarios.

Scenarios also interact by non-sequential and non-explicit relationships. Explicit non-sequential relationships among scenarios are described using the structure for grouping non-sequential episodes (#<episodes series>#); i.e., if a set of episodes inside a non-sequential group are detailed in another scenarios (sub-scenario relationships), then these sub-scenarios are executed in an indistinct order or concurrently.

In the scenario description of Table 2, the *episodes* 10 and 11 of the main execution flow reference *sub-scenarios* described in Table 3. These *sub-scenarios* are explicitly described to be executed concurrently.

In some cases, the given scenarios could interact by non-explicit relationships, what can lead to erroneous situations such as deadlocks. An heuristic for finding non-explicit relationships is shown in [20]; two or more scenarios are likely related when they share common portions in their descriptions, i.e., they involve the participation of common actors, they access shared resources or they are executed in the same context. Two or more *scenarios* could interact concurrently through the following relationships:

- **Non-determinism**, it compares *pre-conditions*. When a set of pre-conditions described within a scenario S_i appears like pre-conditions in another scenario S_j, then, S_i and S_j might interact concurrently.
- **Synchronization**, it compares *pre-conditions* against *post-conditions*. When a pre-condition described in a scenario S_i appears like post-condition in another scenario S_j, and a pre-condition described in S_j appears like post-condition in S_i, then, S_i and S_j might interact concurrently.

3.4 Running Example

The system under consideration is an *Online Broker System* [23]. The *Broker System* interacts with its partner services, *Local Supplier* and *International Supplier*. The *goal* of the system is to allow customers to find the best supplier for a given order. A customer fills up an online order form and after submission, the system broadcasts it to the local and international suppliers. Each supplier after examining the order may decide to decline or submit a bid. A local supplier needs to add taxes to the order total, while an international supplier needs to ensure an order does not

include items restricted for export. Submitted bids are sent back to the broker to be shown to the customer, *who eventually asks the system to proceed with a bid.* The full scenarios of the "Online Broker System" example are shown in [20].

If we select the "Submit Order" scenario (Table 2) as the ***main scenario***, the *episodes* 10 (LOCAL SUPPLIER BID FOR ORDER), 11 (INTERNATIONAL SUPPLIER BID FOR ORDER), 12 (PROCESS BIDS), and *exception* 1.1 (REGISTER CUSTOMER) are detailed in another scenarios. Thus, from the ***main scenario***, it is possible to identify the sequentially (PROCESS BIDS, REGISTER CUSTOMER) and explicit non-sequentially related scenarios (LOCAL SUPPLIER AND INTERNATIONAL SUPPLIER).

The "Submit Order" scenario (Table 2) includes a brief description about the submit order process, the *pre-conditions* to start the scenario and five *execution flows*: the main and four exception flows. The main execution flow would pass through all its episodes until *episode* 12, after which it successfully terminates. The first *exception* execution, which describes the situation when the "The customer is not registered" starts from *episode* 1, just after the *episode* 1, the "REGISTER CUSTOMER" scenario is enabled. The second exception describes the situation when the "The customer does not inform her login after 60 seconds" starts from *episode* 1, just after the *episode* 2 is performed, "The Broker System displays a login timeout page". The third *exception* describes the situation when the "The Customer login information is not accurate" (invalid input), starts from *episode* 1, just after the *episode* 4 is performed. In this case, given the user action, the "The Broker System displays an alert message". The fourth *exception* execution describes the situation when the "The order is empty" (invalid input), starts from *episode* 1, just after the *episode* 8 is performed. In this case, given the user action, the "The Broker System displays an error message".

PROCESS BIDS, LOCAL SUPPLIER BID FOR ORDER and INTERNATIONAL SUPPLIER BID FOR ORDER are presented in Table 3 and detailed in [20]. PROCESS BIDS references sequentially to HANDLE PAYMENT scenario.

4 Petri-Net Model Generation

Once scenarios are constructed, it is possible to automatically generate Petri-Net formal models.

In order to improve the efficacy of scenario transformation method, it is necessary to remove the irrelevant information and formatting symbols, such as URLs, HTML tags, parenthesized comments and bullets. Details of the pre-processing step are presented in [22] and [20].

4.1 Petri-Net

Petri-Net is a graphical and mathematical modeling and analysis language for describing and studying systems that are characterized as concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic [17].

A Petri-Net (Figure 3) is composed of nodes that denote *places* or *transitions*.

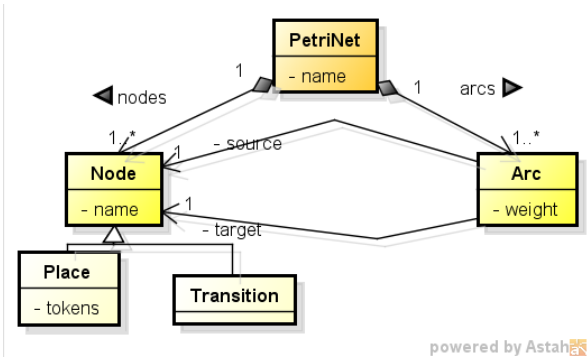

Fig. 3. Petri-Net metamodel [20]

Nodes are linked together by *arcs*. ***Transitions*** or active components model the activities that can occur, thus changing the state of the system; transitions are only allowed to fire if they are enabled, which means that all the pre-conditions (input places) for the activity have been fulfilled. ***Places*** or passive components (placeholders for tokens) model communication channel, resource, buffer, geographical location or a possible state (*condition*); the current state of the system being modeled is called *marking*, which is given by the number of tokens in each place. ***Tokens*** model physical or information object, collection of objects, indicator of state or indicator of condition. ***Arcs*** are of two types; input arcs start from places and end at transitions, and output arcs start at a transition and end at a place.

When a ***transition fires***, it removes tokens from its input places and adds at all of its output places. The number of tokens removed/added depends on the *weight* of each arc.

Definition 4.1 A ***place-transition*** Petri-Net is a five-tuple $PN = (P, T, F, W, M_0)$ where $P = \{p_1, p_2, \cdots, p_n\}$ is a finite set of places, $T = \{t_1, t_2, \cdots, t_m\}$ is a set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, $W : F \to \{1, 2, \cdots\}$ is a weight function, $M_0 : P \to \{0, 1, 2, \cdots\}$ is the initial marking and $P \cap T = \varnothing$ and $P \cup T \neq \varnothing$.

Definition 4.2 For a $PN = (P, T, F, W, M_0)$, a ***marking*** is a function $M : P \to \{0, 1, 2, \cdots\}$, where $M(p)$ is the number of tokens in p. M_0 represents PN with an initial marking.

Definition 4.3 A transition t is ***enabled*** at a marking M if $M(p) \geq w(p, t)$ for any $p \in {}^\circ t$, where ${}^\circ t$ is the set of input places of t. On firing t, M is changed to M' such that $\forall p \in P : M'(p) = M(p) - W(p, t) + W(t, p)$. $M[t > M'$ denotes firing t at marking M.

Definition 4.4 For a PN, a sequence of transitions $\sigma =< t_1, t_2, \cdots, t_n >$ is called a ***firing sequence*** if and only if $M_0[t_1 >, [t_2 >, \cdots, [t_n > M_n$. In notation, $M_0[PN, \sigma > M_n$ or $M_0[\sigma > M_n$.

Definition 4.5 For a $PN = (P, T, F, W, M_0)$, a marking M is said to be ***reachable*** if and only if there exists a firing sequence σ such that $M_0[\sigma > M$. In notation,

$M_0[PN, * > M$ or $M_0[* > M$, represents the set of all reachable markings of PN.

Definition 4.6 The ***reachability tree***, also called marking graph, of a Petri-Net $PN = (P, T, F, W, M_0)$ is a directed graph $G = (N, A)$, where nodes N corresponds to reachable markings ($N = \{M_0[* > M\}$) and arcs A correspond to feasible transitions ($A = \{T\}$). Thus, *Markings* are states reached from the *initial marking* (initial state) by firing transitions (which effect the change from one marking to another by firing).

4.2 Transforming Scenarios into Petri-Nets

A Petri-Net PN is derived from a scenario S as follows: We identify the *event occurrences* (*episodes* and *exceptions*) and their *pre-conditions* (or *causes*), *constraints* and *post-conditions*. For each *event*, a *transition* is created for denoting the location of *event occurrence*. *Input places* are created to denote the locations of its *pre-conditions*, *causes* and *constraints* (They restrict but do not impede - TRUE). *Output places* are created to denote the location of its *post-conditions*. Event labels, condition labels and constraint labels are assigned to these transitions and places accordingly. The initial marking M_0 of the PN is then created to denote the initial state, in which tokens are added into input places that represent *pre-conditions*, *causes* or *constraints*. Execution of the scenario begins at this initial marking which semantically means the system initial state, including the availability of all *resources*, *pre-condition*, *causes* or *constraints*. It ends at the same marking that semantically means the release of these *resources*, *pre-conditions*, *causes* or *constraints* [20].

The ***first step*** of the *transformation method* defines mapping rules to translate scenario elements (Title, Goal, Context, Resource, Actor, Episodes, Exception) into Petri-Net elements (*transition*, *place* and *arc*). Figure 4 depicts the visual transformations from Scenario into Petri-Net elements.

For each scenario element, a sub Petri-Net which contains places, transitions and arcs is derived. The different mapping rules to derive a sub Petri-Net from a scenario element are described using a structure composed of left and right hand sides (LHS and RHS). LHS is the conditional part of the rule (scenario element), and RHS is basically the expected result of the rule (sub Petri-Net). *Input* and *Output dummy places* are created for bridging to previous and next sub Petri-Nets.

As the ***second step*** of the *transformation method*, the sub Petri-Nets generated from scenario's elements are composed into a whole Petri-Net by Fusion Place operations. Formal definition of the *transformation method*, mapping rules and fusion place operations are detailed in [20].

For illustration, we applied the *transformation method* to obtain the Petri-Net of the "Submit Order" scenario (Table 2). It was derived through the mapping the scenario components of the main execution flow - episodes and exceptions. Figure 5 depicts the Petri-Net for the "Submit Order" scenario.

For "*Submit Order*" scenario (Table 2), 16 event occurrences are identified (12 in the main flow - episodes and 4 in the exceptional flows): T_1 (The Customer

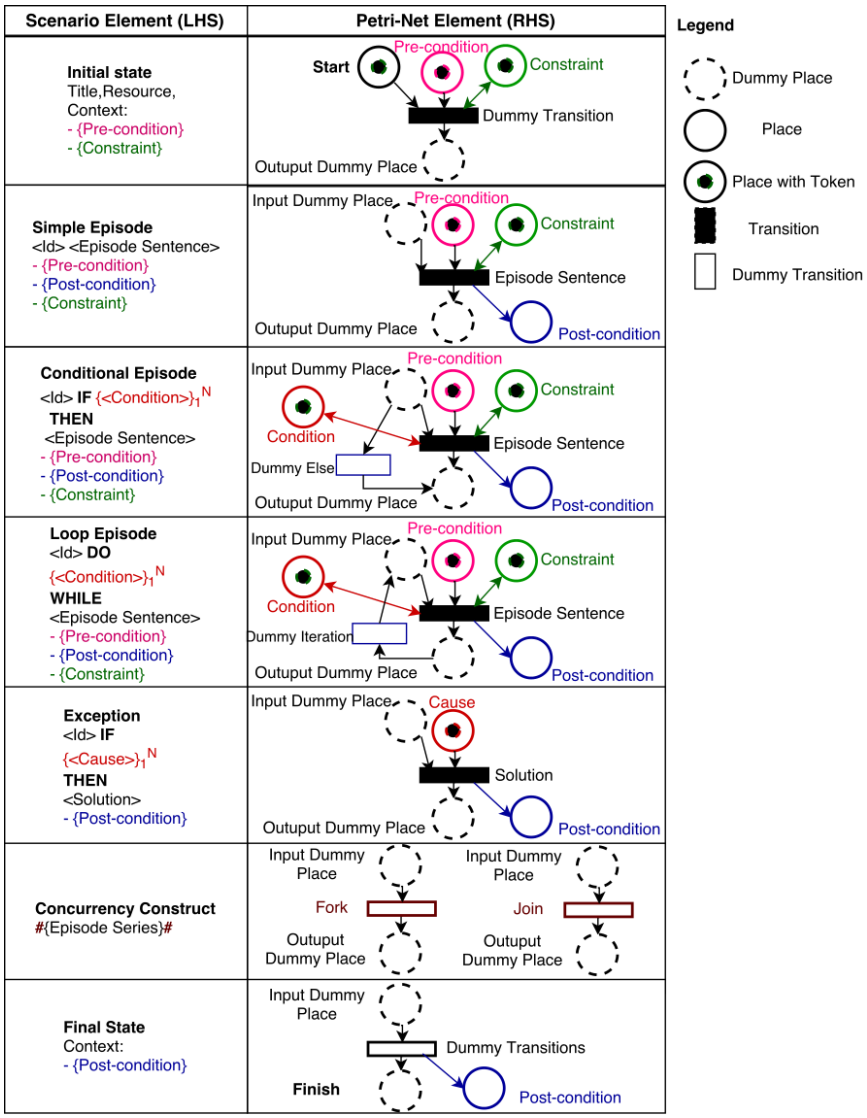

Fig. 4. Mapping scenario elements into Petri-Net elements [20].

loads the login page), T_2 (The Broker System asks for the Customer login information), T_3 (The Customer enters her login information), T_4 (The Broker System checks the provided login information), T_5 (The Broker System displays an order page), T_6 (The Customer creates a new Order), T_7 (The Customer adds an item to the Order), T_8 (The Customer submits the Order), T_9 (The Broker System broadcast the Order to the Suppliers), T_{10} (LOCAL SUPPLIER BID FOR ORDER), T_{11} (INTERNATIONAL SUPPLIER BID FOR ORDER), T_{12} (PROCESS BIDS), $T_{1.1}$ (REGISTER CUSTOMER), $T_{2.1}$ (The Broker System displays a login timeout page), $T_{4.1}$ (The Broker System displays an alert message) and $T_{8.1}$ (The Broker System displays an error message). We construct a Petri-Net by creating transitions $T_1, T_2, \cdots, T_{11}, T_{12}$ and T_{13} to denote these events and appending to each transition

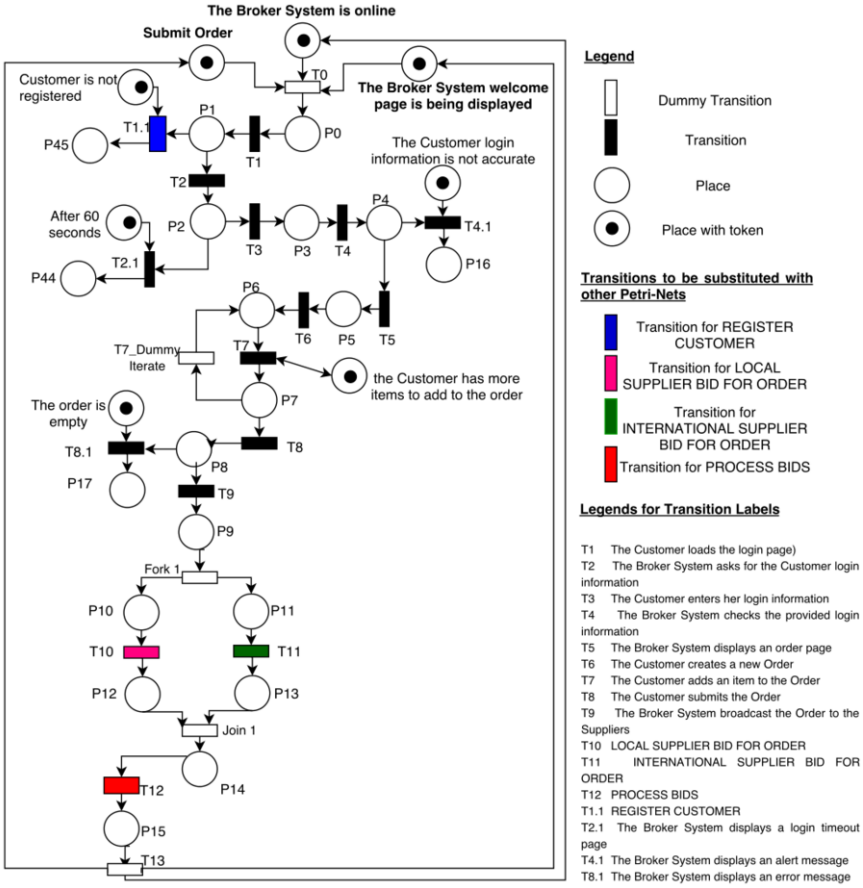

Fig. 5. Mapping Submit Order scenario into Petri-Net model [20].

input and output places to denote: (1) internal dummy input and output places, or (2) input conditions (exception's cause or episode's condition) and post-conditions. Additionally: (1) two dummy transitions ($Fork_1$ and $Join_1$) are created for synchronization of concurrent transitions T_{10} and T_{11}; and (2) two transitions are created to denote the scenario triggering (T_0) and the scenario completion (T_{13}).

4.3 Integrating Petri-Nets into an Integrated Model

For every scenario and its related scenarios, we generate partial Petri-Nets in order to integrate these partial Petri-Nets into a consistent whole Integrated Petri-Net. The Integrated Petri-Net reflects exactly the original properties of the synthesized Petri-Nets (Demonstrated in [20]).

In scenario language, scenarios are related to other scenarios by explicit sequential relationships (Section 3.3). When a *scenario* is chosen to be a ***main scenario***, and translated into a ***main Petri-Net***, its sequentially related scenarios are mapped into *input places* (pre-conditions or constraints), *output places* (post-conditions) or *transitions* (episodes' sentence or exceptions' solution).

As the ***first step*** of the method for *integrating Petri-Nets*, each sequentially

related scenario is translated into a Petri-Net. Then, each one of these Petri-Nets must be replaced into the corresponding place or transition of the *main Petri-Net*. The first step is the *substitution of places or transitions*.

If a ***main scenario*** is mapped into a ***main Petri-Net***, the interaction with non-explicit and non-sequentially related scenarios is described by common *pre-conditions* or *post-conditions*, these common conditions are mapped into *input places* or *output places*.

As the ***second step*** of the method for *integrating Petri-Nets*, each non-sequentially related scenario is translated into a Petri-Net. Among the Petri-Nets, there are *common places* (with the same labels) that denote the same *pre-condition* or *post-condition*, and they need to be uniquely represented from the system point of view [3]. The second step is basically the *fusion of common places*.

The method for *integrating Petri-Nets* produces an integrated Petri-Net from a given set of related Petri-Nets. A formal definition of the method for *integrating Petri-Nets*, *substitution of places* or *transitions* and *fusion places operations* are detailed in [20].

Revisiting the "Submit Order" scenario (Table 2), the *exception* 1.1 and *episodes* 10, 11 and 12 are detailed in another scenarios (*exception* and *sub-scenario*) like REGISTER CUSTOMER, LOCAL SUPPLIER BID FOR ORDER, INTERNATIONAL SUPPLIER BID FOR ORDER and PROCESS BID. It means that Petri-Nets should be generated for referenced scenarios (Register Customer-$T_{1.1}$, Local Supplier bid for order-T_{10}, International Supplier bid for order-T_{11} and Process Bids-T_{12}) and replaced into the ***main Petri-Net*** of "Submit Order".

Figure 5 shows the transitions where the referenced scenarios must be substituted in "Submit Order" Petri-Net. and Figure 6 shows the Integrated Petri-Net of "Submit Order" scenario. To manage the state explosion problem of Petri-Nets, the sequentially related scenarios (REGISTER CUSTOMER and PROCESS BIDS) are not included (See [22]).

5 Test Scenario Generation

The Scenario-Petri-Net mapping strategy presented in Section 4 enables the translation of the behavior defined in the scenario, which is formally represented here as a Petri-Net model. While establishing a formal model is essential to ensure consistency, this task is usually considered as a barrier to the practical application of MBT.

In order to improve the accuracy of *Test Scenario Generation Strategy*, it is necessary to detect and fix defects in the scenarios descriptions and the resulting Petri-Nets that can hurt *Unambiguity*, *Completeness*, *Consistency* and *Correctness* properties. If defects are found, the analysis feedback is used to improve the scenario descriptions, since the identified problems can be traced to the scenarios. If no defects are found, the resulting *Petri-Nets* are used to derive test scenarios. Details of this analysis strategy are presented in [22].

Concerning the proposed approach, it is considered that requirements engineers

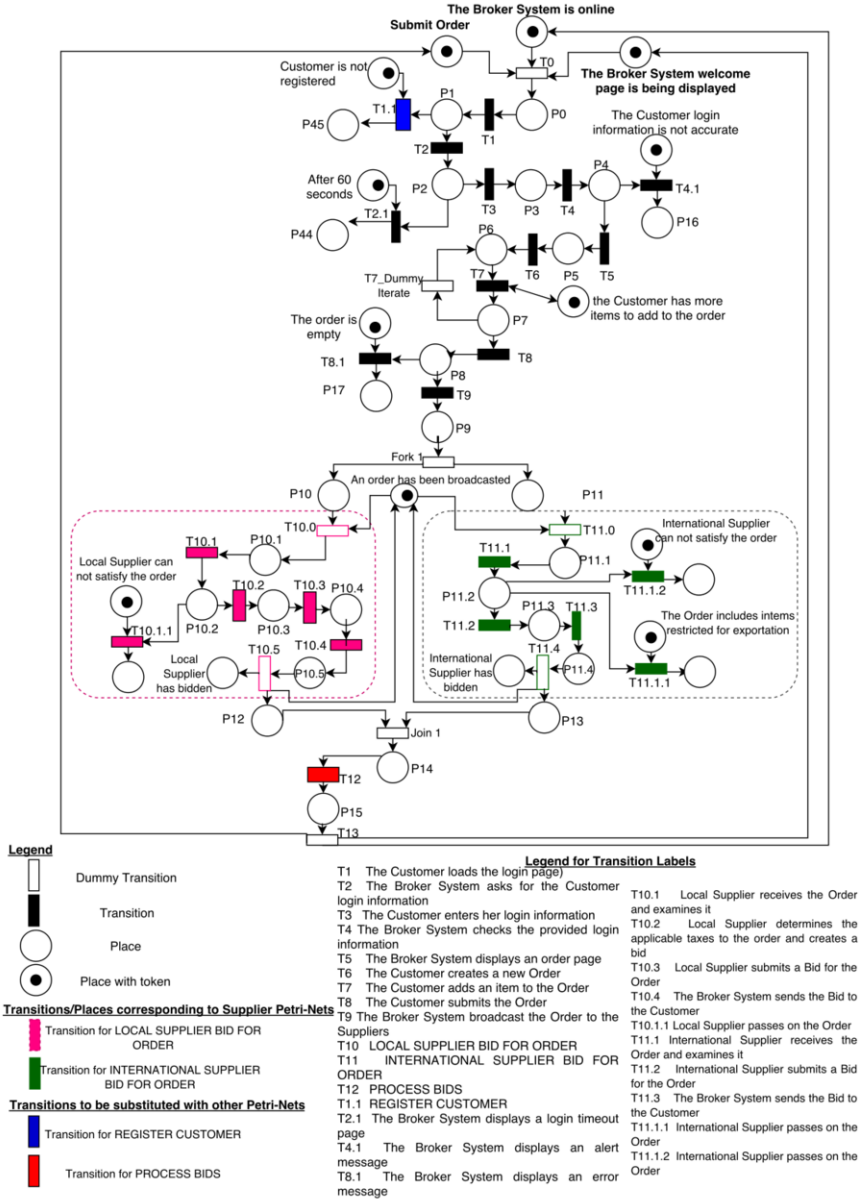

Fig. 6. Integrated Petri-Net of "Submit Order" in the Broker System [20].

are not usually familiar to work with formal notations. In our approach to generate test scenarios from scenario specifications, the intermediate formal model is described in terms of Petri-Nets. The translation to Petri-Net notation is hidden and entirely accomplished without the manipulation of requirement engineers.

The ***test scenario generation process*** is divided into four main steps:

(i) Generate Petri-Net models from scenario specifications (Section 4).

(ii) Generate the Reachability Tree for Petri-Net models using an available Petri-Net tool like PIPE2 [16].

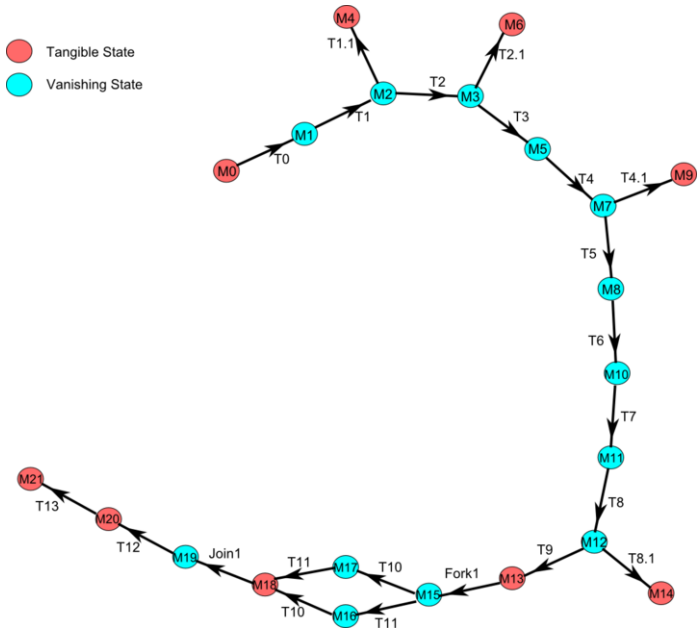

Fig. 7. Reachability tree of "Submit Order" Petri-Net.

(iii) Generate test sequences from Reachability Tree of a Petri-Net. A test sequences is a theoretical path between the initial state M_0 and a reachable final state.

(iv) Generate test scenarios from the generated test sequences while taking guard conditions (input and output) into consideration, satisfying the test coverage and adequacy criteria.

5.1 Generating Reachability Tree from Petri-Net

With Petri-Nets derived from scenarios, the reachability analysis strategy can be applied next. The reachability analysis strategy generates a *Reachability Tree* $G = (N, A)$ from a Petri-Net $PN = (P, T, F, W, M_0)$, which contains reachable markings as nodes and transitions as arcs (which effect the change from one marking to another by firing). From initial marking M_0, we can get an overview about possible markings (states).

As an example, Figure 7 shows the reachability tree for the Petri-Net of "Submit Order" scenario (Figure 5). This reachability tree was automatically generated using the PIPE2 tool [16]. In fact, in the "Submit Order" example, all markings are reachable from initial marking M_0. Table 5. shows an excerpt of the reachable markings. From Table 5, it can be seen that there are 21 states with 21 transitions.

In Fig. 7, M_0 is the initial marking (or initial state), and M_4, M_6, M_9, M_{14} and M_{21} are reachable final makings. A marking represents the current state of the Petri-Net, i.e., the number of tokens in each one of the places of the Petri-Net.

Table 5
Reachable Marking for "Submit Order" Petri-Net.

Marking	Arc from (By Firing)	Marking Corresponds to Places {after 60 seconds, Customer is not registered, P0, P1, P10, P11, P12, P13, P14, P15, P16, P17, P2, P3, P4, P44, P45, P5, P6, P7, P8, P9, Submit Order, The Broker System is online, The Broker System welcome page is being displayed, the Customer has more items to add to the order, The Customer login information is not accurate, The order is empty }	Arc to (By Firing)
M_0		1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1	$M_1(T_0)$
M_1	$M_0(T_0)$	1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1	$M_2(T_1)$
M_2	$M_1(T_1)$	1,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1	$M_3(T_2), M_4(T_{1.1})$
M_3	$M_2(T_2)$	1,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1	$M_5(T_3), M_6(T_{2.1})$
M_4	$M_2(T_{1.1})$	1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,1,1	
M_5	$M_3(T_3)$	1,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,1	$M_7(T_4)$
M_6	$M_3(T_{2.1})$	0,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,1	
M_7	$M_5(T_4)$	1,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,1	$M_8(T_5), M_9(T_{4.1})$
M_8	$M_7(T_5)$	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,1,1	$M_{10}(T_6)$
M_9	$M_7(T_{4.1})$	1,1,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,1	
M_{10}	$M_8(T_6)$	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,1,1	$M_{11}(T_7)$
M_{11}	$M_{10}(T_7)$	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,1,1	$M_{12}(T_8)$
M_{12}	$M_{11}(T_8)$	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1	$M_{13}(T_9), M_{14}(T_{8.1})$
M_{13}	$M_{12}(T_9)$	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1	$M_{14}(T_{8.1}), M_{15}(Fork_1)$
M_{14}	$M_{12}(T_{8.1})$	1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0	
M_{15}	$M_{13}(Fork_1)$	1,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1	$M_{16}(T_{11}), M_{17}(T_{10})$
M_{16}	$M_{15}(T_{11})$	1,1,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1	$M_{18}(T_{10})$
M_{17}	$M_{15}(T_{10})$	1,1,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1	$M_{18}(T_{11})$
M_{18}	$M_{16}(T_{10}), M_{17}(T_{11})$	1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1	$M_{19}(Join_1)$
M_{19}	$M_{18}(Join_1)$	1,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1	$M_{20}(T_{12})$
M_{20}	$M_{19}(T_{12})$	1,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1	$M_{21}(T_{13})$
M_{21}	$M_{20}(T_{13})$	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1	

5.2 Generating Test Sequences

If $G = (N, A)$ is the *Reachability Tree* obtained from a Petri-Net $PN = (P, T, F, W, M_0)$, the different execution sequences between the initial state M_0 and the final states represent the finite set of ***Test Sequences***. These ***test sequences*** can be automatically identified by traversing the reachability tree G by breadth first traversal algorithm–*BFS*. During traversal, we look for paths (sequence of arcs–transitions) between the Initial state M_0 (It triggers the initial transition) and the *Vanishing states* or *Tangible states* (this states are reachable by firing a final transition). A *final transition* represents the last episode or exception of the main execution flow or exceptional flow of the corresponding scenario.

Thus, a *test sequence* consists of a sequence of transitions between the initial transition t_0 and a final transition; a transition represents a scenario episode sentence or scenario exception solution.

Definition 5.1 A sequence $TS =< t_0, t_1, t_2, \cdots, t_m >$ is a ***test sequence*** in a Petri-Net PN, where t_0 is the initial transition and t_m is a final transition, $t_i \in T$ of PN, $i = 0, 1, 2, \cdots, m$.

For instance, in the reachability tree (Figure 7) for the Petri-Net of "Submit Order" (Figure 5), the set of test scenarios for exercising the submit order scenario successfully is as below:

- $TS_5 : [T_0] \rightarrow [T_1] \rightarrow [T_2] \rightarrow [T_3] \rightarrow [T_4] \rightarrow [T_5] \rightarrow [T_6] \rightarrow [T_7] \rightarrow [T_8] \rightarrow [T_9] \rightarrow [Fork_1] \rightarrow [T_{10}] \rightarrow [T_{11}] \rightarrow [Join_1] \rightarrow [T_{12}] \rightarrow [T_{13}]$

Table 6
Algorithm for generating test scenarios from a Petri-Net.

INPUT: Scenario S
OUTPUT: Test Scenarios Set TCS
BEGIN
1. Obtain a Petri-Net PN from a Scenario S
2. Generate the Reachability Tree G from PN
3. Generate Test Sequence Set TS: Traverse G from initial state M_0 to final states by applying breadth first search (BFS)
4. For each test sequence ts in TS
  a. Initialize a test scenario tsc with empty string;
  b. For each transition $t_i \in ts$, where $i = 0, \cdots, n$,
    - find the transition t_i in Petri-Net PN;
    - get input places of t_i in PN;
    - update guard condition s_i of t_i, where s_i = {input places of t_i};
    - append $< s_i t_i >$ to test scenario tsc;
    - IF t_i is the last transition THEN;
      update expected results K of tsc, where K = {output places of t};
      append $< K >$ to test scenario tsc;
  c. Add the test scenario tsc to the set of test scenarios TSC
5. Return TSC;
END

- $TS_6 : [T_0] \rightarrow [T_1] \rightarrow [T_2] \rightarrow [T_3] \rightarrow [T_4] \rightarrow [T_5] \rightarrow [T_6] \rightarrow [T_7] \rightarrow [T_8] \rightarrow [T_9] \rightarrow [Fork_1] \rightarrow [T_{11}] \rightarrow [T_{10}] \rightarrow [Join_1] \rightarrow [T_{12}] \rightarrow [T_{13}]$

5.3 Generating Test Scenarios from Petri-Net

To generate ***test scenarios*** that satisfy the test coverage and adequacy criteria, we first enumerate all possible ***test sequences*** from the initial transition t_0 to final transitions in the reachability tree G. Then, each test sequence is revisited to generate ***test scenarios***. During revisit, we look for guard conditions of each one of the transitions.

The set of guard conditions of a transition t consist of the input places of t in the corresponding Petri-Net PN. This set of input places enables the transition t for firing. Table 6 shows the algorithm to obtain test scenarios from a Scenario.

Thus, a *test scenario* consists of a sequence of transitions and guard conditions linked by arcs, which represent scenario episode sentences (with input conditions or constraints) or scenario exception solutions (with cause).

Definition 5.2 A *sequence* $TSC =< s_0 t_0, s_1 t_1, \cdots, s_m t_m \ k >$ is a ***test scenario*** in a Petri-Net PN, where t_0 is the initial transition and t_m is a final transition, s_0 is the set of initial guard conditions for all test scenarios, s_i is a set of guard conditions of t_i, k is the set of expected results of the current test scenario, $k \subseteq P$ of PN, $s_i \subseteq P$ of PN, $t_i \in T$ of PN, $i = 1, 2, \cdots, m$.

5.4 Test Criteria

The test scenario generation process satisfies the sequential and concurrent programs test coverage and adequacy criteria [10]. These criteria are described below:

- **Path Coverage Criterion**, each path in a model is executed at least once in testing.

Table 7
Test scenarios for "Submit Order" Petri-Net.

CREATING REACHABILITY TREE G OF PETRI NET "Submit Order"
$S0$ is the initial guard condition for the transition $T0$ and all test scenarios.
$S0$ = {Submit Order, The Broker System is online, The Broker System welcome page is being displayed}
The set of expected results is empty.
********** EXCEPTIONAL EXECUTION TEST SCENARIOS **********
********** ALL PATHS BFS from $M0$ to $M4$ in G
⟶ TSC1: $S0 \rightarrow [T0] \rightarrow S1 \rightarrow [T1] \rightarrow S2 \rightarrow [T1.1] \rightarrow S4$
$S2$ is the guard condition for firing the exceptional transition $T1.1$
$S2$ = {Customer is not registered}
********** ALL PATHS BFS from $M0$ to $M6$ in G
⟶ TSC2: $S0 \rightarrow [T0] \rightarrow S1 \rightarrow [T1] \rightarrow S2 \rightarrow [T2] \rightarrow S3 \rightarrow [T2.1] \rightarrow S6$
$S3$ is the guard condition for firing the exceptional transition $T2.1$
$S3$ = {after 60 seconds}
********** ALL PATHS BFS from $M0$ to $M9$ in G
⟶ TSC3: $S0 \rightarrow [T0] \rightarrow S1 \rightarrow [T1] \rightarrow S2 \rightarrow [T2] \rightarrow S3 \rightarrow [T3] \rightarrow S5 \rightarrow [T4] \rightarrow S7 \rightarrow [T4.1] \rightarrow S9$
$S7$ is the guard condition for firing the exceptional transition $T4.1$
$S7$ = {Customer login information is not accurate}
********** ALL PATHS BFS from $M0$ to $M14$ in G
⟶ TSC4: $S0 \rightarrow [T0] \rightarrow S1 \rightarrow [T1] \rightarrow S2 \rightarrow [T2] \rightarrow S3 \rightarrow [T3] \rightarrow S5 \rightarrow [T4] \rightarrow S7 \rightarrow [T5] \rightarrow S8 \rightarrow [T6] \rightarrow S10 \rightarrow [T7] \rightarrow S11 \rightarrow [T8] \rightarrow S12 \rightarrow [T8.1] \rightarrow S14$
$S12$ is the guard condition for firing the exceptional transition $T8.1$
$S12$ = {The order is empty}
********** NORMAL EXECUTION TEST SCENARIOS**********
********** ALL PATHS BFS from $M0$ to $M21$ in G **********
⟶ TSC5: $S0 \rightarrow [T0] \rightarrow S1 \rightarrow [T1] \rightarrow S2 \rightarrow [T2] \rightarrow S3 \rightarrow [T3] \rightarrow S5 \rightarrow [T4] \rightarrow S7 \rightarrow [T5] \rightarrow S8 \rightarrow [T6] \rightarrow S10 \rightarrow [T7] \rightarrow S11 \rightarrow [T8] \rightarrow S12 \rightarrow [T9] \rightarrow S13 \rightarrow [Fork1] \rightarrow S15 \rightarrow [T10] \rightarrow S17 \rightarrow [T11] \rightarrow S18 \rightarrow [Join1] \rightarrow S19 \rightarrow [T12] \rightarrow S20 \rightarrow [T13] \rightarrow S21$
⟶ TSC6: $S0 \rightarrow [T0] \rightarrow S1 \rightarrow [T1] \rightarrow S2 \rightarrow [T2] \rightarrow S3 \rightarrow [T3] \rightarrow S5 \rightarrow [T4] \rightarrow S7 \rightarrow [T5] \rightarrow S8 \rightarrow [T6] \rightarrow S10 \rightarrow [T7] \rightarrow S11 \rightarrow [T8] \rightarrow S12 \rightarrow [T9] \rightarrow S13 \rightarrow [Fork1] \rightarrow S15 \rightarrow [T11] \rightarrow S16 \rightarrow [T10] \rightarrow S18 \rightarrow [Join1] \rightarrow S19 \rightarrow [T12] \rightarrow S20 \rightarrow [T13] \rightarrow S21$

- **Interaction Coverage Criterion**, all interactions of a concurrent program are executed at least once in testing.

5.5 Test Scenarios Example

In Figure 7, the ***test scenarios*** are generated traversing the reachability tree by *BFS* from the initial state M_0 to the final states M_4, M_6, M_9, M_{14} and M_{21}. The test scenarios can be automatically generated with tool support.

Table 7 shows the test scenarios generated for "Submit Order" Petri-Net (Figure 5). We generate 2 test scenarios for the main execution flow and 4 for the exceptional flows.

In Table 7, the initial guard condition $S0$ = {Submit Order, The Broker System is online, The Broker System welcome page is being displayed} represents the set of initial input conditions for all test scenarios.

The union of the guard conditions $S0 \cup S2$ = {Submit Order, The Broker System is online, The Broker System welcome page is being displayed, Customer is not registered} represents the set of input conditions for the test scenario of the first exceptional execution flow (path from $M0$ to $M4$).

$S0 \cup S3$ = {Submit Order, The Broker System is online, The Broker System welcome page is being displayed, after 60 seconds} represents the set of input conditions for the test scenario of the second exceptional flow (path from $M0$ to $M6$).

$S0 \cup S7$ = {Submit Order, The Broker System is online, The Broker System welcome page is being displayed, Customer login information is not accurate} is the set of input conditions for the test scenario of the third exceptional path ($M0$ to $M9$).

$S0 \cup S12$ = {Submit Order, The Broker System is online, The Broker System welcome page is being displayed, The order is empty} represents the set of input conditions for the test scenario of the fourth exceptional flow (path from $M0$ to $M14$).

And, $S0$ = {Submit Order, The Broker System is online, The Broker System welcome page is being displayed} represents the set of input conditions for the test scenarios of the main execution flow (paths from $M0$ to $M21$).

All these test scenarios do not have explicit expected results, because the scenario specification used as input (Submit Order) does not describe post-conditions in its context, episodes or exceptions.

6 Conclusion

6.1 Related Work

Many researches have shown the importance to formalize the informal aspects of scenarios in order to benefit from automated scenarios analysis and testing. Some research focused on developing the formal semantics for scenario representations [3], [2]; others are focusing on developing techniques to translate scenarios into formal models.

UML Sequence Diagrams, Activity Diagrams or Message Sequence Charts [1] are frequently used as formalisms for scenarios. However, these models are either informal or semi-formal and, they are hard to be used in automated test generation without the help of additional models.

Several works have been done for test scenario generation from use case, activity, state and sequence diagrams; however, in most of reviewed works, it is necessary to refine the input models into intermediate models (not automated) and create additional domain models to make explicit test inputs or conditions of them [7], [11], [15], [12] and [25].

Use case descriptions are widely used to specify requirements, however test cases generated from these models are usually described at high level of abstraction, and commonly it is necessary to refine them because external inputs (conditions required to execute test scenarios) are not explicit in the initial descriptions of requirements artifacts used as input. Denger and Medina [6] discuss some approaches for generating test cases from use cases described in natural language; these approaches are incomplete and not automated.

Some approaches have shown how testing tasks can be automated. These approaches generate test cases from activity [9] and sequence diagrams [15] derived from use cases descriptions. The test variables referenced in the use cases are described as operational variables or glossary terms [9].

In [15] is proposed an approach to test generation from use case extended with contracts and sequence diagrams; it creates sequence diagrams to represent use case scenarios. Object Constraint language (OCL) is used to write contracts. And, the input artifacts are enriched before automate testing tasks.

In a previous work [21], we generate test cases (with test variables and conditions) from natural language scenario descriptions. In this work, activity diagrams are generated as intermediate models, and using the C&L prototype tool [4].

To our knowledge, there is no systematic approach based on activity diagrams, which generates all test scenarios from scenarios or use cases that contain complicated non-sequential or concurrent steps. [9], [12], [15] and [21] generate partial test scenarios using activity diagrams (or extensions) with fork-join structures.

In our work, we represent scenarios using a restricted-form of the natural language [22]. These scenarios are translated into Petri-Nets, which are used as the mechanism to enable rigorous requirements analysis and test generation. Other approaches to formalize scenarios based on restricted-form of natural language and Petri-Net notations include [13] and [23].

In [18], it is proposed an MBT approach to generate test cases using Scenarios and Petri-Nets; however, this approach depends on semi-formal specification of scenarios.

The related Petri-Net based approaches exhibit the following shortcomings: (i) Scenarios are described in relation to formal notations; (ii) There is a lack of systematic procedures on how to represent scenarios; (iii) The procedures to transform scenarios into Petri-Nets are not automated and depend on intermediate models; and (iv) Scenario notations do not provide adequate constructs to support *modularity*.

On the other hand, our approach: (i) Uses a semi-structured natural language to write scenarios; (ii) Defines an abstract and concrete syntax for scenarios; (iii) Implements automated transformation rules; (iv) Provides powerful characteristics to deal with modularity; (v) can generate test scenarios for concurrency; and (vi) No additional models are required for test scenario generation. However, in order to generate test data from the test scenarios, additional domain models are needed. Our approach generates the conditions for test data.

6.2 Conclusion

Natural language based requirements specification, like the scenario technique explored in this work, helps developers to identify the test scenarios to exercise the different execution flows of the system. As such, it improves the quality of the product from the initial stages of software production, contributing to the reduction of failures and the reduction of maintenance costs after the final product was delivered.

The scenario language used in this work is generic enough to permit the specification of any application. This language holds the main characteristics of other scenario definitions, such as use case descriptions [5]. However, the scenario gram-

mar (Table 4) seems to offer more support for further transformation into executable models.

Our approach provides benefits due to the following reasons: (a) it preserves preserves the original properties of scenarios when they are translated and synthesized into Petri-Nets, such as demonstrated in [22]; (b) it is capable to generate test scenario more comprehensively and consistently (using available Petri-Net tools [16]) than the existing approaches; (c) it derives test scenarios from scenario specifications based on semi-structured natural language, existing approaches are based on semi-formal models; (d) it generates test scenarios for applications with concurrency characteristics; (e) it starts with the software development process; and (f) it improves the accuracy of the test generation process by checking the quality of the inputs (scenarios and derived Petri-Nets) [22].

Limitation. The transformation procedure from scenarios into Petri-Nets works well if a requirements engineer can properly write scenarios using the syntax and semantic rules described in this work, i.e. following the linguistic patterns and putting the correct markers (IF THEN, Constraint, and so on) on sentences, it is our assumption that the use of RNL scenarios is well accepted by the most stakeholders in RE process, and it is amenable to automated processing.

The scalability of Petri-Nets and the state explosion of the generated reachability tree can be considered limitations, however these limitations are overcame in previous works [22] and [20].

Future Work. The C&L prototype tool has been used and evolved by the PUC–Rio requirements engineering group. Methods for model transformation are being improved. Their results are positive and therefore its evolution continues.

In the future, we plan (a) to provide more details of test scenario generation for concurrent applications; and (b) to deal with the testing of exceptions and non-functional requirements. In this work we have drafted some criteria for mapping exceptions and non-functional requirements described (constraints) in scenarios to behavioral models and testing.

References

[1] M. Andersson and J. Bergstrand, *Formalizing use cases with message sequence charts*, m.sc. thesis, Lund Institute of Technology, 1995.

[2] G. Carvalho, D. Falcão, F. Barros, A. Sampaio, A. Mota, L. Motta, and M. Blackburn, *Test case generation from natural language requirements based on SCR specifications*, in 28th Annual ACM Symposium on Applied Computing, 2013, pp. 18–22.

[3] K. S. Cheung, T. Y. Cheung, and K. O. Chow, *A petri-net-based synthesis methodology for use-case-driven system design*, J. syst. Softw., 79 (2006), pp. 772–790.

[4] C&L, *Scenarios & lexicons*, 2014. http://pes.inf.puc-rio.br/cel.

[5] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2001.

[6] C. Denger and M. Medina, *Test case derived from requirement specifications.* Report, 2003.

[7] M. J. Escalona, J. J. Gutierrez, M. Mejías, G. Aragón, I. Ramos, J. Torres, and F. J. Domínguez, *An overview on test generation from functional requirements*, Journal of System and Software, 84 (2011), pp. 1379–1393.

[8] M. Glinz, *Improving the quality of requirements with scenarios*, in Second World Congress for Software Quality (2WCSQ), Yokohama, 2000, pp. 55–60.

[9] J. J. Gutiérrez, N. Clémentine, M. J. E. M. Mejías, and I. M. Ramos, *Visualization of use cases through automatically generated activity diagrams, in MODELS*, LNCS Springer, 5301 (2008), pp. 86–96.

[10] T. Katayama, E. Itoh, and Z. Furukawa, *Test-case generation for concurrent programs with the testing criteria using interaction sequences*, 1999.

[11] M. Khandai, A. Acharya, and D. Mohapatra, *A novel approach of test case generation for concurrent systems using UML sequence diagram*, ICECT, 6 (2011), pp. 157–161.

[12] H. Kim, S. Kang, J. Baik, and I. Ko, *Test cases generation from uml activity diagrams*, in 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007, pp. 556–561.

[13] W. Lee, S. Cha, and Y. Kwon, *Integration and analysis of use cases using modular petri nets in requirements engineering*, IEEE Trans. on Software Engineering, 24 (1998), pp. 1115–1130.

[14] J. C. S. P. Leite, G. Hadad, J. Doorn, and G. Kaplan, *A scenario construction process*, Requirements Engineering Journal, 5 (2000), pp. 38–61.

[15] C. Nebut, F. Fleurey, Y. L. Traon, and J. M. Jezequel, *Automatic test generation: A use case driven approach*, IEEE Transactions of Software Engineering, 32 (2006).

[16] PIPE2, *Platform independent petri net editor 2*, 2014. http://pipe2.sourceforge.net.

[17] W. Reising, *Petri Nets: An Introduction*, Berlin, Heidelberg, 1985.

[18] H. Reza and S. D. Kerlin, *A model-based testing using scenarios and constraints-based modular petri nets*, in Information Technology: New Generations (ITNG), 2011 Eighth International Conference on, 2011, pp. 568–573.

[19] A. W. Roscoe, *The Theory and Practice of Concurrency*, Prentice Hall, 1998.

[20] E. Sarmiento, *Analysis of Natural Language Scenarios*, PhD thesis, Departament of Informatics, PUC-Rio, Brazil, 2016.

[21] E. Sarmiento, J. C. S. P. Leite, and E. Almentero, *Generating model based test cases from natural language requirements descriptions*, in 1st International Workshop on Requirements Engineering and Testing (RET'14), 2014, pp. 32–38.

[22] ———, *Analysis of scenarios with petri-net models*, in 29th Brazilian Symposium on Software Engineering (SBES), 2015.

[23] S. Sóme, *Petri nets based formalization on textual use cases*, 2007. Report in SITE, TR2007-11.

[24] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*, Morgan Kaufman publishing, 2007.

[25] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, *Automatic generation of system test cases from use case specications*, in Proceedings of the 2015 International Symposium on Software Testing and Analysis, ser. ISSTA 2015, 2015, pp. 385–396.