Analysis of Scenarios with Petri-Net Models

Edgar Sarmiento, Julio Cesar Sampaio do Prado Leite
Informatics Department
PUC - Rio, Brazil
{ecalisaya, julio}@inf.puc-rio.br

Eduardo Almentero
Mathematics Department
Federal Rural University of Rio de Janeiro, Brazil
almentero@ufrrj.br

***Abstract*—Requirements analysis plays a key role in the software development process. Scenario-based representation is often used for software requirements specification (SRS). Scenarios written in natural language may be ambiguous, and, sometimes, inaccurate. This problem is partially due to the fact that interactions among scenarios are rarely represented explicitly. As scenarios are used as input to subsequent activities of the software development process (SD), it is very important to enable their analysis; especially to detect flaws or problems due to wrong or missing information. This work proposes a Petri-Net based approach as an effective way to analyze the acquired scenarios. To enable the automated analysis, scenarios are translated into equivalent Place/Transition Petri-Nets. Scenarios and their resulting Petri-Nets can be automatically analyzed to evaluate some properties related to correctness, consistency and completeness. The identified problems can be traced back to the scenarios, allowing their revision. We also discuss how correctness, consistency and completeness of scenario-based SRSs can be decomposed in related properties using the NFR approach. Proposal's workability is demonstrated through two running examples.**

Keywords-component; requirements analysis, requirements verification, scenario, Petri-Net

I. INTRODUCTION

Scenario-based representations are frequently used in Requirements Engineering. In this context, SRS are represented as a collection of scenarios and described by specific flows of events and their guard conditions. The use of scenarios helps understanding a specific situation in an application, prioritizing their behavior [12]. Some of the most prominent languages to write scenarios are restricted-form of use case or scenario descriptions [5][12], UML dynamic behavior models or Message Sequence Charts[1].

For practical reasons, and in order to allow for an easy communication between clients and developers, requirements are written using natural language-based scenarios. Natural language-based scenarios offer several practical advantages: (1) scenarios are easy to describe and understand; (2) they are scalable; the behavior of a large system can be represented as a collection of independently and incrementally developed scenarios; and (3) it is relatively easy to provide requirements traceability throughout the design and implementation [10].

Unfortunately, natural language-based scenarios exhibit some shortcomings: (1) informally specified scenarios are usually hard to analyze, because natural language is by definition ambiguous; (2) modularity is poorly supported, because the relationships among scenarios are rarely represented explicitly; and (3) there are no systematic approaches to represent concurrency since initial requirements. From the concurrency perspective, scenarios are rarely truly independent in practice; they interact or compete with each other by *communication channels* or *shared resources*, what can lead to erroneous situations such as *deadlocks*.

Scenario's specifications are normally informal or semi-formal, and, they cannot be used for further analysis (including graphical notation based models) because they lack of formal semantics to support it. In order to perform an automated analysis of scenarios, it is necessary to translate them from informal or semi-formal to formal representations, like Petri-Nets. The analysis outcome can be used to improve the scenario descriptions, since the identified problems can be traced to the scenarios. Petri-Nets [14] 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 main contribution of this work is an automated analysis approach to evaluate some static and behavioral properties related to *correctness*, *consistency* and *completeness* in scenario-based SRS, as early as possible in software development. Other contributions are: (1) the definition of a scenario language (conforming to a model) based on a restricted-form of natural language (RNL); (2) the development of a systematic procedure (mapping rules) that transforms scenarios and their interactions stated in a RNL to Petri-Nets (conforming to a Petri-Net metamodel); and (3) the modeling of *correctness*, *consistency* and *completeness*, and their decomposition in related properties, using the NFR approach [6].

This approach allows benefiting from both the usability of textual scenarios and the precision of Petri-Nets. It also allows an easier integration to available Petri-Net tools like [13].

This work is organized as follows. Section II presents the scenario language used as input to the approach. Section III details the transformation from scenario into Petri-Net. Section IV presents the scenario analysis strategy. Section V compares our work with related work. Finally, Section VI presents the conclusions and some suggestions for future work.

II. SCENARIO

The term *scenario* is used with different meanings in different contexts. We therefore state a definition from [12] and [8]. In [8], a scenario is an ordered set of interactions between partners, usually between a system and a set of actors external to the system. In [12], a scenario is a partial description of the application behavior

978-1-4673-9272-3/15 $31.00 © 2015 IEEE
DOI 10.1109/SBES.2015.13

that occurs at a given moment in a specific geographical context - a *situation*. A scenario may comprise a concrete sequence of interaction steps (*instance scenario*) or a set of possible interaction steps (*type scenario*).

Definition 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. An ***event*** is an actor operation or the interaction: (1) between the user and the system through its interface, (2) between the environment and the system, or (3) between system's components. A ***condition*** is an actor/system/resource state or the availability of some resource. An ***actor*** can be a user, the system or system's components.

The natural language based-scenario used in this work is an adaptation of [12]. Unlike [12], our focus is the analysis of scenarios. For this purpose, a new scenario language is defined by adding the *pre-condition* and *post-condition* attributes, and the *repetition structure control* to the syntax proposed in [12].

Fig. 1 shows a model for the Restricted-form of Natural Language Scenario description used in this work. It defines an abstract conceptual model for the proposed scenario language, using a class diagram.

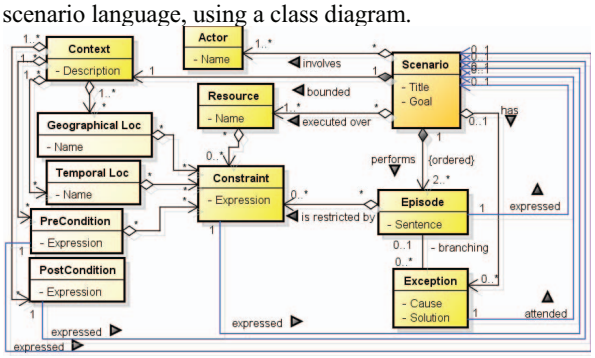

Figure 1. Scenario model.

According to our conceptual model, the ***scenario language*** is composed of the Scenario, Context, Resource, Actor, Episode, Exception and Constraint entities.

In this language, 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*.

The ***Title*** of a ***scenario*** identifies the scenario using a declarative sentence, and it must be unique.

The ***Goal*** describes the purpose of the scenario using a declarative sentence, and it gives a general idea about the scenario main purpose and how it is achieved.

The ***Context*** describes the scenario initial state using declarative sentences, and it must be described through at least one of the following sub-components: pre-condition, post-condition, geographical or temporal location.

Resources are an enumeration of passive entities used by the actors in the episodes to achieve scenario's goal. Resources must appear in at least one of the episodes.

Actors are an enumeration of active entities directly involved with the *situation*. Actors must appear in at least one of the episodes.

***Episodes*:** they are a set of actions that give an operational description of behavior. They represent the main flow, which is a sequence of steps where everything works as expected.

A basic ***episode e*** is a 5-tuple (Id, Sentence, Constraint, Pre-condition, Post-condition). Every *episode* is identified through an identifier *Id*. An episode performs an action - *Sentence* (imperative) that can modify resources and be executed by actors. Optionally, the following attributes can be added: (1) *constraints* that restrict the quality with witch the episode is performed; (2) *pre-conditions* that we expect are already satisfied before the episode is performed; (3) *post-conditions* that we expect will be achieved after the episode occurs.

Pre-condition and *post-condition* are described as declarative sentences involving relevant actor/resource/system states (e.g. the availability of a resource). They are different of context's pre-condition and context's post-condition: (1) context's pre-conditions are the state of the system before the scenario is started; (2) context's post-conditions are the state of the system when the episodes are carried out. These attributes are important in the modeling of concurrent systems [10].

The *episodes* 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.

A sequence of episodes implies a precedence order, but a non-sequential order can be bounded by the symbol "#". This is used to describe concurrent or parallel episodes (#<Episode Series>#).

Exceptions are situations that prevent the proper course of the scenario. Its treatment should be described using a sentence. It represents the alternative/exceptional flows.

An ***exception ex*** is a 4-tuple (Id, Cause, Solution, Post-condition). Every *exception* is identified through an identifier *Id*. An exception: (1) is caused by invalid input data or critical resource/system states - *Cause*; (2) is treated by an imperative sentence – *Solution*; and optionally (3) may generate effects on the resource/system states, or simply produce a message – *Post-conditions*. An exception is always thrown from an episode of the main execution flow.

Constraint: Scope or non-functional aspect described as declarative sentence that restricts the quality with witch the: (1) *goal* is achieved, (2) *resources* are needed and (3) *episodes* are performed. It is an attribute of resources, episodes or context's sub-components.

Constraint, geographical location, temporal location, pre-condition and post-condition can be expressed by one or more single sentences linked by the logical connectors "*AND*" or "*OR*".

A. Restricted-form of Natural Language

In order to reduce ambiguity in natural language requirements descriptions, we have defined a concrete syntax based on a restricted-form of natural language (RNL) for writing scenarios in accordance to their conceptual model (Fig. 1). This syntax defines the interactions or situations involving users, the system, the environment or system's components.

Table I shows the linguistic patterns for writing scenario elements based on a RNL using partial Extended-BNF. The scenario model should be seen as a syntax and structural guidelines to: (1) obtain a homologous description style, (2) demonstrate the aspects that scenarios can cover and (3) facilitate the automated analysis [12].

According to the syntax described in Table I, a *Scenario* must be described by the attributes: *Title*, *Goal*, *Context*, *Resource*, *Actor*, *Episodes* and *Exception*.

In Table I, + 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.

TABLE I. LINGUISTIC PATTERNS

TYPE	DESCRIPTION
<Scenario>	<Title> + <Goal> + <Context> + $\{<Resource>\}_1^N$ + $\{<Actor>\}_1^N$ + <Episodes> + {<Exception>}
<Title>	([Actor | Resource] + Verb + Predicate) | Phrase
<Goal>	[Actor | Resource] + Verb + Predicate
<Context>	**{**<Geographical Location>**}** + **{**<Temporal Location>**}** + **{**<Pre-condition>**}** + **{**<Post-condition>**}**
<Geographical Location>	Phrase + {<Constraint>}
<Temporal Location>	Phrase + {<Constraint>}
<Pre-condition>	(<Condition> | <Title>) + {<Constraint>}
<Post-condition>	<Condition> | <Title>
<Resource>	Name + {<Constraint>}
<Actor>	Name + {<Constraint>}
<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> **CR**
<Conditional Episode>	<Id> **IF** $\{<Condition>\}_1^N$ **THEN** <Episode Sentence> **CR**
<Optional Episode>	**[** <Id> <Episode Sentence> **]** **CR**
<Loop Episode>	<Id> **DO** <Episode Sentence> **WHILE** $\{<Condition>\}_1^N$ **CR**
<Episode Sentence>	(([Actor | Resource] + Verb + Predicate) | <Title>) + {<Pre-Condition>} + {<Post-Condition>} + {<Constraint>}
<Exception>	<Id> **IF** $\{<Cause>\}_1^N$ **THEN** <Solution>
<Cause>	<Condition>
<Solution>	(([Actor | Resource] + Verb + Predicate) | <Title>) + {<Post-Condition>}
<Condition>	([Actor | Resource] + Verb + Predicate) | Phrase
<Constraint>	([Actor | Resource] + [*Must*][Not] Verb + Predicate) | <Title> | Phrase

B. Scenario Relationships-based Modularity

The Scenario language is designed with modularity in mind, mainly using a mereology operator for decomposition; *integration scenario* is an important method for modularity. Besides, other *relationships* (pre-condition, sub-scenario, exception and so on) also help the modularization. These features provide *modularity* through the inter-connectivity among related scenarios. For instance, an episode or the solution of an exception can be detailed in other scenarios. Modularity is considered a mechanism to deal with the scenario explosion problem [10][12].

When facing large systems, the number of scenarios could be unmanageable and the requirements engineers become sunk in details, losing the global vision of the system. Or simply, the requirements engineers are most likely interested in a subset of scenarios. In order to face this problem, Leite et al. [12] proposes the construction of *integration scenarios* based on the existing scenarios. An ***integration scenario*** gives an overview of the relationship among several scenarios of the system, since each integration scenario episode corresponds to a scenario.

Therefore, the scenario language makes explicit the sequential relationships among scenarios. In scenarios descriptions, UPPERCASE words or phrases are references to other scenarios. Scenarios can be connected to other scenarios through the following relationships: (1) ***sub-scenario,*** an episode of a scenario can be described as a scenario; this allows the decomposition of complex scenarios, facilitating both its writing and understanding; (2) ***pre-condition,*** it is defined within the context of a scenario; a scenario that is pre-condition to another must be executed first; (3) ***post-condition,*** it is defined within the context of a scenario; a scenario that is post-condition to another must be executed last; (4) ***exception,*** a scenario can be used to detail the treatment of an exception – solution; the scenario that treats the exception is only executed when exception's cause is triggered in the main scenario; and (5) ***constraint,*** it 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 can also ***interact concurrently*** (non-sequential) by explicit and non-explicit relationships. These scenarios can be considered as a set of concurrently executing threads. Explicit concurrency is described using the concurrency construct (#<episodes series>#), when some of the concurrent episodes are detailed in other *scenarios* (like in Figure 2). Non-explicit concurrency may be described as *pre-condition* or *post-condition*; when a pre-condition or post-condition described in a scenario appears like *episode's pre-condition* or *episode's post-condition* of another scenario, then, these scenarios might interact concurrently by *message passing* using communication *channels* or *shared resources*.

C. Running Example

In a *scenario*, the explicit concurrency between some episodes may be described using the concurrency construct (#<episodes series>#), such as illustrated in the scenario of Figure 2 (episodes 10, 11 and 12).

Fig. 2 describes the interactions between the ***Online Broker System*** and its partner services, ***SupplierA***, ***SupplierB*** and ***SupplierC*** using the scenario language. The "Submit Order" scenario interacts with the "SupplierA", "SupplierB" and "SupplierC" scenarios (Fig. 3). After, a Customer has submitted an order; the Online Broker System broadcasts it to the suppliers (executed concurrently), and they submit a bid for the order. The remaining scenarios in the "Online Broker System" example are shown in [17].

In Figure 2, the episodes *10* (SUPPLIERA BID FOR ORDER), *11* (SUPPLIERB BID FOR ORDER), *12* (SUPPLIERC BID FOR ORDER) and *13* (PROCESS BIDS) are detailed in other scenarios. Fig. 2 shows the

sequential interaction among scenarios by sub-scenario relationship (PROCESS BIDS), and non-sequential interaction by explicit concurrency (SUPPLIERS).

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 Customer 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. **Post-condition**: An Order has been broadcasted
10. **#** SUPPLIERA BID FOR ORDER
11. SUPPLIERB BID FOR ORDER
12. SUPPLIERC BID FOR ORDER **#**
13. PROCESS BIDS.

EXCEPTIONS
4.1 **IF** the Customer login information is not accurate **THEN** the Broker System displays an alert message
8.1 **IF** the order is empty **THEN** the Broker System displays an error message

Figure 2. Submit Order scenario in the Online Broker System.

TITLE: ***SupplierA bid for order***
GOAL: SupplierA has bidded
CONTEXT: Create a Bid for an Order
PRE-CONDITION: An Order has been broadcasted
POST-CONDITION: SupplierA has bidded
ACTOR: SupplierA, Broker System
RESOURCES: Order, Bid
EPISODES
1. SupplierA receives the Order and examines it
2. SupplierA submits a Bid for the Order
3. The Broker System sends the Bid to the Customer

EXCEPTIONS
1.1 **IF** SupplierA can not satisfy the Order **THEN** SupplierA passes on the Order

Figure 3. SupplierA bid for order scenario.

Scenarios may interact with each other, in some cases by non-explicit concurrency. Episode's *pre-condition* and episode's *post-condition* attributes may be used to specify the *message passing* between concurrent scenarios using communication *channels* or *shared resources*. A *pre-condition* or *post-condition* may specify the state or availability of some *communication channel* or *shared resource*.

In concurrent scenarios, the identification of episode's *pre-conditions* and episode's *post-conditions* proceeds in two steps: (1) identify relevant events (episodes) after which the shared resources (or communication channel) is first needed; and (2) identify relevant events (episodes) after which the shared resources (or communication channel) is no longer needed.

Fig. 4 depicts the scenarios which describe a simple solution for the "Readers-Writers Problem. The Reader-Writer Problem pertains to any situation where a *shared memory* (data structure, database, or file system) is read and modified by concurrent processes. While a *data structure* is being written or modified, it is often necessary to bar other processes from reading, in order to prevent a reader from interrupting a modification in progress and reading inconsistent or invalid data [7]. We selected an example of the Readers-Writers problem in order to verify problems due to concurrent interactions among scenarios by *shared resources*.

In Figure 4, the *Data Set is* a shared resource. Reader and Writer tasks are enabled only when the *Data Set is available*. Episodes 1 of Reader and Writer (*Get access to read, Get access to write*), and Episode *2* of Writer (*Write Data Set*) specify as pre and post-conditions the availability of *Data Set*.

Title: ***Reader Task***
Goal: Allow multiple readers access the Data Set
Context:
Resources: Data Set
Actors: Reader
Episodes:
1. Get access to read. ***Pre-condition***: Data Set is available.
2. ***Read Data Set***.

Title: ***Writer Task***
Goal: Only one writer can access the Data Set
Context:
Resources: Data Set
Actors: Writer
Episodes:
1. Get access to write. ***Pre-condition***: Data Set is available.
2. Write Data Set. **Post-condition**: Data Set is available.

Figure 4. Scenarios for Readers-Writers problem.

III. PETRI-NET MODEL GENERATION

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

A. 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 [14].

A Petri-Net (Fig. 5) is composed of: (1) ***transitions*** or active components; they 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; (2) ***places*** or passive components and placeholders for tokens; they 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; (3) ***tokens*** model physical or information object, collection of objects, indicator of state or indicator of condition; (4) ***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.

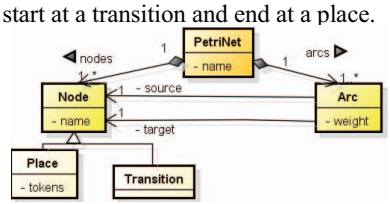

Figure 5. Petri-Net metamodel.

Definition 2. A ***place-transition*** Petri-Net is a 5-tuple ***PN*** = (P, T, F, W, M_0) where P = {p_1, p_2, ..., p_n} is a set of places, T = {t_1, t_2, ..., t_m} is a set of transitions, F $\subseteq$ (P×T) $\cup$ (T×P) is a set of arcs, W : F $\rightarrow$ {1, 2, ...} is a weight function, M_0 : P $\rightarrow$ {0, 1, 2, ...} is the initial marking and P $\cap$ T = $\varnothing$ and P $\cup$ T $\neq \varnothing$.

Definition 3. For a PN = (P, T, F, W, M_0), a ***marking*** is a function M : P $\rightarrow$ {0, 1, 2, ...}, where M(p) is the number of tokens in p. M_0 represents PN with an initial marking.

B. Transforming Scenarios into Petri-Nets

We assume that a ***scenario*** **S**: (1) *starts* at an *idle state* with all necessary resources, pre-conditions or constraints; (2) *performs* a collection of partially ordered *event*

occurrences (episodes or exceptions), each guarded by a set of *conditions* (pre-conditions, post-conditions, or causes) and restricted by a set of *constraints*; and (3) *returns* to the *idle state* and releases the resources, pre-conditions (if it is not returned by some previous event) or constraints after completion (adapted from [4]).

A Petri-Net is generated from a scenario 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 occurrence. *Input places* are created to denote the locations of its *pre-conditions* 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 is then created to denote the initial state, in which tokens are added into input places that represent *pre-conditions* or *constraints*. Execution of the scenario begins at this initial marking which means the initial state, including the availability of all resources. It ends at the same marking that means the release of these resources, pre-conditions or constraints.

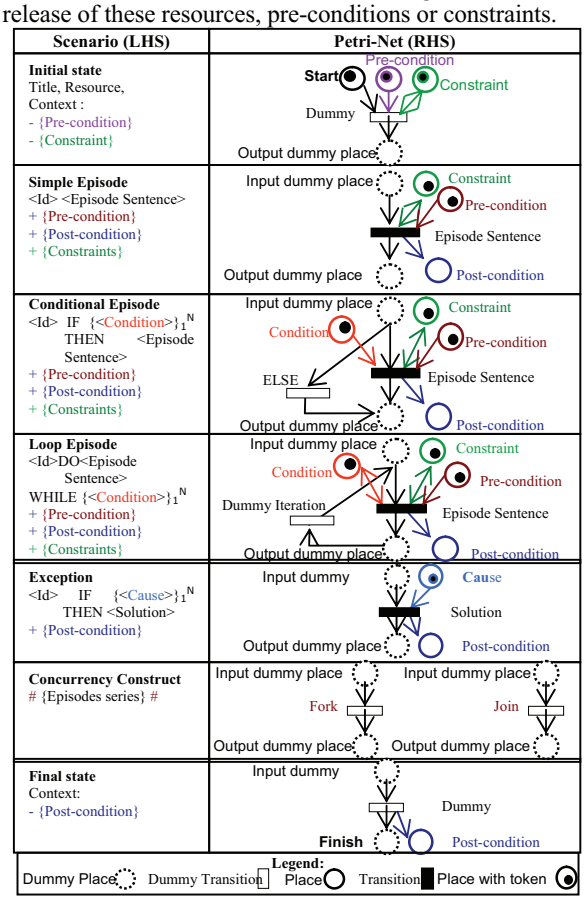

Figure 6. Mapping scenario constructs into Petri-Net elements.

As the ***first step*** of the algorithm for transforming a scenario into an equivalent Petri-Net model (***Algorithm 1***), we define mapping rules to translate scenario elements into Petri-Net elements (*transition*, *place* and *arc*).

The different mapping rules to generate 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.

Fig. 6 depicts the visual transformations (LHS→RHS) from Scenario elements into Petri-Net elements. Transformation performs the tasks (mapping rules) defined in ***Algorithm 1***. (A first version of these tasks was detailed in [15]).

As the ***second step*** of the ***Algorithm 1***, the sub Petri-Nets generated from scenario's elements are composed into a whole Petri-Net by Fusion Place or Modified Fusion Place operations.

Definition 4 (Fusion Place): A sub Petri-Net can be fused with other sub Petri-Net by fusing the *output dummy place* of the first sub Petri-Net into the *input dummy place* of the last sub Petri-Net.

Definition 5 (Modified Fusion Place): Any sub Petri-Net can be fused with other sub Petri-Net by fusing at least a *common place* among them.

Algorithm 1: Transforming Scenario into Petri-Net

Algorithm 1: Transforming Scenario into Petri-Net
Input: Scenario ***S*** = (Title, Goal, Context, Resource, Actor, Episodes, Exception);
Output: Petri-Net ***PN*** = (P, T, F, W, M_0);
Begin:
1. Generate a sub Petri-Net for ***scenario triggering - initial state***, containing:
→A *dummy transition*; an "*output dummy place*";
→An *input place* with one initial token (p.token = 1) for the title and resources:
→**For** every *constraint* or *pre-condition* in *Context*, generate an *input place* with one initial token (p.token = 1);
2. **For** every ***episode*** generate a sub Petri-Net containing:
→A main *transition* for episode's sentence and a *dummy transition* for conditional/optional/loop episodes;
→An "*input dummy place*" and an "*output dummy place*" of the main *transition*;
→**IF** sentence starts with "#" **Then** generate a *fork transition* with an *input* and an *output dummy place;*
→**IF** sentence ends with "#" **Then** generate a *join transition* with an *input* and an *output dummy place;*
→**For** every *condition*, *option, pre-condition* or *constraint* in {Conditions ∪ Options ∪ Constraints ∪ Pre-conditions}, generate an *input place* with one initial token;
→**For** every *post-condition* in *Post-conditions*, generate an *out place*;
3. For every ***exception*** generate a sub Petri-Net containing:
→A *transition* for *exception's solution*;
→**For** every *cause*, generate "*input place*" with an initial token;
→**For** every *post-condition* in *Post-conditions*, generate an *output place*;
4. **Link** sub Petri-Nets of *exceptions* to sub Petri-Nets of *corresponding episodes*;
→Apply *Fusion Place* to sub Petri Nets from episode and exception (Definition 4);
5. **Link** sub Petri-Nets between a *fork and a join transitions* as concurrent sub Petri-Nets;
→Apply *Fusion Place* to sub Petri-Nets from ***fork*** and episode (Definition 4);
→Apply *Fusion Place* to sub Petri-Nets from episode and ***join*** (Definition 4);
6. Generate a sub Petri-Net for the ***scenario completion***, containing:
→A *dummy transition* with an "*input dummy place*" and an "*output dummy place*" (*Finish*);
→**For** every *post-condition* in Context's *Post-conditions*, generate an *output place*;
7. **Compose** the sub Petri-Nets into a *complete* ***Petri-Net***:
→**For** every *sub Petri-Net*
→Apply *Fusion Place* operation (Definition 4);
→Apply *Modified Fusion Place* operation (Definition 5);
8. **FOR** every *input place* of the *first transition* (initial state):
→**IF** *input place* has not *input arcs* **Then Link** *last transition* to the *input place*;
9. Return **Petri-Net**;
End

C. *Integrating Petri-Nets*

We generate a Petri-Net for every scenario in order to integrate these partial Petri-Nets into a consistent whole ***integrated Petri-Net***. The ***integrated Petri-Net*** obtained reflects exactly the functionality of the scenarios.

In our scenario language, scenarios are related to other scenarios by explicit sequential relationships (pre-condition, post-condition, constraint, sub-scenario or exception). When an *integration scenario* (or not) is used as a ***root scenario*** and translated into a ***root Petri-Net***, the 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 algorithm for *integrating Petri-Nets* ***(Algorithm 2)***, each sequentially related scenario is translated into a Petri-Net. After it, each one of these Petri-Nets must be replaced into the corresponding place or transition of the *root Petri-Net*. Our first step is the *substitution of places or transitions*.

Definition 6 (Substitution Transition): Any *transition* (not dummy) of a Petri-Net can be replaced by any other Petri-Net. Then the *input dummy place* of the *transition* is fused with the first *input dummy place (Title)* of the *replacing Petri-Net* and the *output dummy place* of the *transition* is fused with the last *output dummy place (Finish)* of the *replacing Petri-Net.*

Definition 7 (Substitution Input Place): Any *input place* (not dummy) of a Petri-Net can be replaced by any other Petri-Net. Then the last *output dummy place (Finish)* of the *replacing Petri-Net* is fused with the *input place*.

Definition 8 (Substitution Output Place): Any *output place* (not dummy) of a Petri-Net can be replaced by any other Petri-Net. Then the first *input dummy place (Title)* of the *replacing Petri-Net* is fused with the *output place*.

Scenarios are also related to other scenarios by non-explicit concurrent interactions described as *pre-condition* or *post-condition*. If a ***root scenario*** is mapped into a ***root Petri-Net***, these conditions are mapped into *input* or *output places*.

As the ***second step*** of the algorithm for *integrating Petri-Nets* ***(Algorithm 2)***, each concurrently 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 need to be uniquely represented from the system point of view [4]. Our second step is basically the *fusion of these common places*.

Definition 9 (Concurrent Fusion Place): Any Petri-Net can be fused with other Petri-Net by fusing at least a *common place* (from *pre-condition* or *post-condition*) among them.

Algorithm 2: Integrating Petri-Nets

Algorithm 2: Integrating Petri-Nets
Input: Scenario ***S*** = (Title, Goal, Context, Resource, Actor, Episodes, Exception);
Root Petri-Net ***PN*** = (P, T, F, W, M_0); -- Derived from Scenario **S**
Output: Integrated Petri-Net ***IPN*** = (P, T, F, W, M_0)
Begin:
1. Identify *sequential relationships* of the ***Scenario*** by *Pre-condition. Post-condition, Constraint, Sub-scenario or Exception*;
2. Identify *concurrent interactions* of the ***Scenario*** by *Pre-condition* or *Post-condition*;
3. Integrate Petri-Nets to obtain a whole *Integrated Petri-Net* from the selected ***Scenario***:
→**For** every *sequential scenario*,
→Transform *scenario* into a *Petri-Net* (***Algorithm 1***);
→**IF** *current* Petri-Net represents a *Sub-scenario* or *Exception* **Then**:
→Apply *Substitution Transition* to the corresponding "*transition*" of the ***Root Petri-Net*** (Definition 6);
→**IF** *current* Petri-Net represents a *Pre-Condition* or *Constraint* **Then**:
→Apply *Substitution Input Place* to the corresponding "*place*" of the ***Root Petri-Net*** (Definition 7);
→**IF** *current* Petri-Net represents a *Post-Condition* **Then**:
→Apply *Substitution Output Place* to the corresponding "*place*" of the ***Root Petri-Net*** (Definition 8);
→**For** every *concurrent scenario*
→Transform *scenario* into a *Petri-Net* (***Algorithm 1***);
→Apply *Concurrent Fusion Place* to *current Petri-Net* and ***Root Petri-Net*** (Def. 9);
4. Return ***Root Petri-Net***;
End

D. *Petri-Net Example*

For illustration, Figure 7 depicts the integrated Petri-Net obtained from the "*Submit Order*" scenario by *Algorithm 2*, it was created through the interconnection of sub Petri-Nets derived from its scenario specification and substituting the non-explicit concurrently related scenarios (*SupplierA*, *SupplierB* and *SupplierC* scenarios). In order to manage the state explosion problem of Petri-Nets, the sequentially related scenarios (PROCESS BIDS) are not included (See Section IV.C).

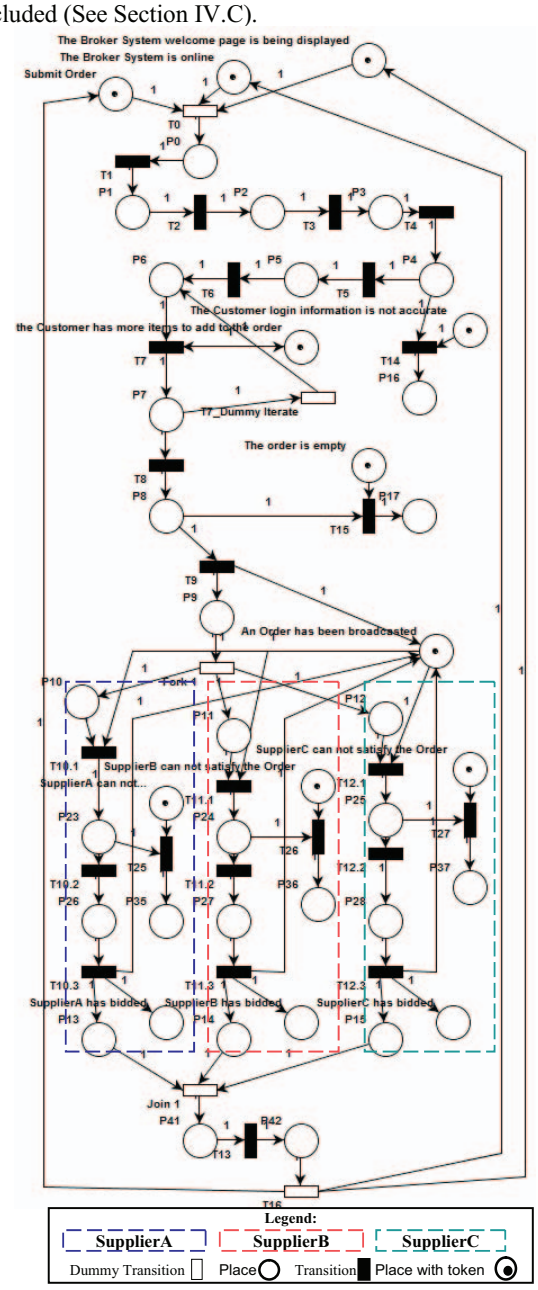

Figure 7. Integrated Petri-Net of "Submit Order".

For "Submit Order" scenario, 15 event occurrences are identified (13 in the main flow – episodes and 2 in the exceptional flows): T1 (The Customer loads the login page), T2 (The Broker System asks for the Customer login information), T3 (The Customer enters her login information), T4 (The Broker System checks the provided login information), T5 (The Broker System displays an order page), T6 (The Customer creates a new Order), T7 (The Customer adds an item to the Order), T8 (The Customer submits the Order), T9 (The Broker System broadcast the Order to the Suppliers), T10 (SUPPLIERA

BID FOR ORDER), T11 (SUPPLIERB BID FOR ORDER), T12 (SUPPLIERC BID FOR ORDER), T13(PROCESS BIDS), T14 (The Broker System displays an alert message) and T15 (The Broker System displays an error message). We construct a Petri-Net by creating transitions T1, T2,… , T13, T14 and T15 to denote these events and appending to each transition 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 transitions (Fork1 and Join1) are created for synchronization of concurrent transitions T10, T11 and T12; and (2) two transitions are created to denote the scenario triggering (t0) and the scenario completion (t16). Revisiting the "Submit Order" scenario, episodes 10, 11 and 12 are detailed in other scenarios (sub-scenario) like "SupplierA bid for order", "SupplierB bid for order" and "SupplierC bid for order". It means that Petri-Nets should be generated for referenced scenarios (SupplierA bid for order, SupplierB bid for order and SupplierC bid for order) and replaced into the root Petri-Net of "Submit Order".

Fig. 8 depicts the integrated Petri-Net obtained from the "Reader" and "Writer" scenarios (Fig. 4), 4 event occurrences are identified (2 in the main flow of "Reader" and 2 in the main flow of "Writer"): R_1 (Get access to read), R_2 (Read Data Set), W_1 (Get access to write), W_2 (Write Data Set). We construct a Petri-Net by creating transitions R_1, R_2, W_1 and W_2 to denote these events and appending to each transition input and output places to denote: (1) internal dummy input and output places, or (2) pre-conditions and post-conditions. Additionally: (1) two transitions (R_0 and R_3) are created for "Reader" to denote the scenario triggering and the scenario completion; (2) two transitions (W_0 and W_3) are created for "Writer" to denote the scenario triggering and the scenario completion. These scenarios (Fig. 4) interact concurrently by non-explicit relationships using *shared resources* (Data Set) for communication.

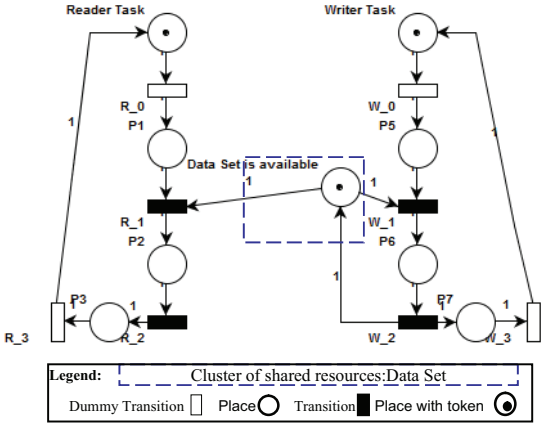

Figure 8. Integrated Petri-Net of "Readers-Writers" problem.

E. *Preservation of Properties*

Our approach preserves the consistency between models because the equivalent Petri-Net preserves the event sequences and conditions described with a scenario. According to ***Algorithm 1,*** every scenario element and their attributes are mapped into Petri-Net elements, and a path is created from the first place (Initial state) to every Petri-Net node.

Moreover, the properties of the Petri-Nets derived from related scenarios are preserved when they are synthesized into a whole Petri-Net, because the synthesis procedure (***Algorithm 2***) does not introduce new non-deterministic situations (Non-determinism is the main source of synchronization problems).

We prove that the integrated Petri-Net obtained according to ***Algorithm 2*** preserves the original properties of the synthesized Petri-Nets; because: (1) the *substitution* (Definition 6, 7 and 8) of places or transitions by sequentially related Petri-Nets do not create any new arcs between these Petri-Nets, i.e. the substitution of places or transitions is done by fusing with the first "*input dummy place (Title)*" or the last "*output dummy place*" (Finish) of the *replacing Petri-Net*; and (2) the *fusion* (Definition 9) of concurrently related Petri-Nets do not create any new arcs between these Petri-Nets, i.e. the fusion of places is done by fusing common places. Fig. 9 illustrates the application of *substitution input place* and *concurrent fusion place* operations to obtain two integrated Petri-Nets: (a) First example, a *constraint* of a scenario is detailed in other scenario (sequential relationship); and (b) Second example, two scenarios interact concurrently because the post-condition of the first one has the same label that the pre-condition of the second one.

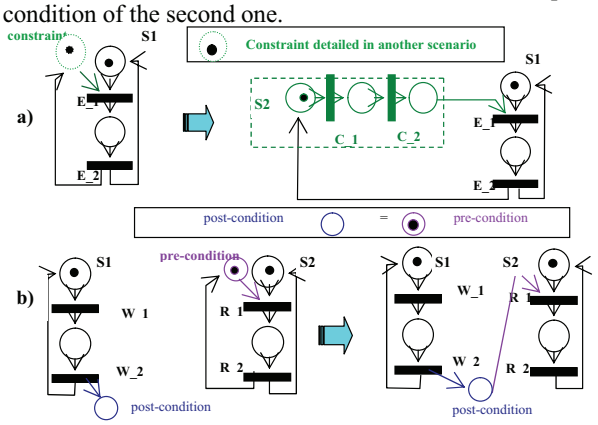

Figure 9. Substitution input place (a) and concurrent fusion place (b).

IV. ANALYSIS OF SCENARIOS

In our proposed analysis approach: ***(1)*** *user's requirements* are described by requirements engineers in an abstract way, textual scenarios are constructed to describe the different situations in the system; additionally, *Integration Scenarios* can be constructed based on the existing scenarios; ***(2)*** by an automatic transformation from scenarios (and their interactions) to Petri-Nets, we obtain an executable model; ***(3)*** from these models (Scenarios and Petri-Nets), we can check some properties related to correctness, consistency and completeness, using the *reachability* analysis of structural and behavioral properties of Petri-Nets; and ***(4)*** finally, the tool sends a feedback to the requirements engineer, if some problem exist. The tool also shows the source of problems – errors in scenarios or their interactions. With the feedback provided by the tool, the requirements engineer can improve the requirements description and then the process starts again. If no problem is found, these

scenarios and models can be used in next activities of the software development process.

A. Modeling Correctness Using the NFR Approach

In our work, it is assumed that: (1) Correctness is the most important quality, and (2) there is an important causal relationship between Consistency, Completeness and Correctness of SRS [19]. Zowghi [19] argues as increasing the completeness of a SRS can decrease its consistency and hence affect the correctness of the final product. Conversely, improving the consistency of a SRS can reduce the completeness, thereby again diminishing correctness. Fig. 10 illustrates this reasoning.

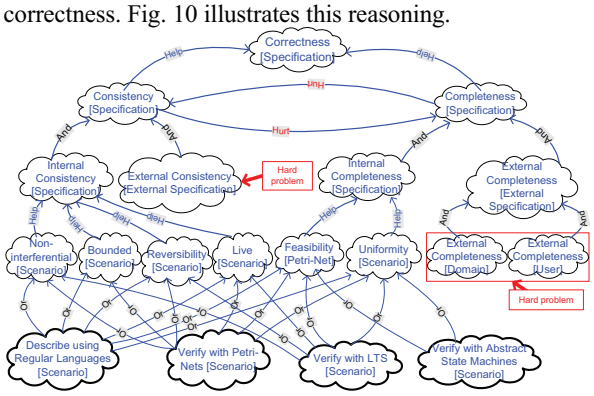

Figure 10. Correctness, consistency and completeness SIG [16].

Definition 10: A specification is ***complete*** to the extent that all of its parts are present and each part is fully developed [1]. We understand that a fully developed SRS is *uniform* and *feasible*, i.e. (1) each operation or condition is constructed using syntax and semantic rules; and (2) it is possible to perform each operation described by them and each internal/external condition is not violated.

Definition 11: A specification is ***consistent*** to the extent that its provisions do not conflict with each other or with governing specifications and objectives [1]. Therefore, some properties that influence the consistency are *non-interferential*, *bounded*, *reversibility* and *live, i.e.* (1) every operation that negatively affect on others should be identified, (2) resources have a finite capacity, (3) the behavior should reach its initial state again, and (4) every operation is enabled in execution.

Definition 12: ***Correctness*** or *Adequacy* means the quality of being able to meet a need satisfactorily [8]. Thereby, having a *consistent* and *complete* set of scenarios contributes for requirements specification *correctness*.

Fig. 10 illustrates the Softgoals Interdependency Graph (SIG) that models the SRS *Correctness* using the NFR qualitative reasoning approach [6], and shows how *Consistency* and *Completeness* positively impacts (HELP) to *Correctness*. We assume that a SRS is *correct*, if it is perceived as *complete* and *consistent* with respect to user's real needs. Interdependency between Consistency and Completeness negatively impacts (HURT) on both [19]. *Consistency* and *Completeness* are decomposed in *Internal* and *External* softgoals (AND), following the lead of [1] and [19]. *Internal Consistency* is decomposed – using HELP links – in *Non-interferential*, *Bounded*, *Reversibility* and *Live* softgoals; and *Internal Completeness* is decomposed – using HELP links – in *Uniformity* and *Feasibility* softgoals. *Uniformity* and *Feasibility* can be operationalized by: (1) writing Scenarios using Regular Languages [9], OR (2) Verifying Scenarios with Petri-Nets [10][11][17][18], OR (3) Verifying Scenarios with Abstract State machines [21]; OR (4) Verifying Scenarios with Labeled Transition Systems – LTS [20]. *Non-interferential*, *Bounded*, *Reversibility* and *Live* can be operationalized by: (1) writing scenarios using Regular Languages [9], OR (2) Verifyng Scenarios with Petri-Nets [10][11][17][18]; OR (3) OR Verifying Scenarios with LTS [20]. In [16], it is discussed the details of the modeling and reasoning to select *Petri-Nets* for scenario analysis.

B. Analyzing Scenarios with Petri-Nets

The process of SRS verification involves checking some static and dynamic properties in *Scenario-based SRS* and equivalent *Petri-Nets*. In order to detect some problems due to non-determinism and synchronization issues, we have made use of Place-Transition Petri-Nets [14] for verification of: (1) static properties like *Correct Token Passing* and *Fully Connected* (related to *Feasibility*); and (2) dynamic properties like *Determinism*, *Bounded*, *Reversibility* and *Deadlock free* (related to *Non-interferential*, *Bounded*, *Reversibility* and *Live*).

Criteria or heuristics to evaluate some properties related to *Completeness and Consistency*, and consequently *Correctness* in SRS by analyzing its *Scenario Specifications* and *Petri-Nets (PN)* are summarized as below.

To evaluate ***uniformity***, we detect missing information by following a checklist with heuristics in Scenario-based SRS:

- The syntax of each component in scenario and its relationships must be described as established in the scenario model (details of style checks in [12]).

To evaluate ***feasibility***, we detect missing information (static properties) by traversing the equivalent Petri-Net (PN):

- Every place in the PN must have at least an input arc (that is not an initial place: pre-condition) and an output arc (that is not a final place: post-condition). If they are missing, the tokens in the PN *cannot pass correctly* [11].
- The transitions in the PN should interact with each others to exchange information (tokens). If there are places or transitions that do not interact with others, it will cause *isolated sub Petri-Nets* [11][18].

To evaluate ***non-interferential***, we find wrong information (dynamic properties) by analyzing the reachability graph of the PN:

- A *non-deterministic* behavior occurs when a set of transitions are simultaneously enabled due to presence of tokens in their input places. If the reachability graph reveals non-deterministic execution paths, a warning is reported to indicate wrong information [10][11].

To evaluate ***bounded***, ***reversibility*** and ***live***, we find wrong information by analyzing the *reachability graph* of the PN:

- An *overflow* exists in a PN when the number of tokens in some place exceeds a finite number k for any marking reachable from initial marking M_0. If

PN is not *bounded*, overflow exists in some place [18].

- *Reversibility* of a PN guarantees that the described behavior reaches its initial state again. If the PN is not reversible, the automatic error recovery is not possible.
- The PN must be free of dead transitions. If the reachability analysis reveals a set of transitions that are never enabled (unreachable code in programs), the PN is not *deadlock free* [10][11].

C. Managing the State Explosion

State explosion issue is a serious problem when applying *Petri-Net analysis* to large systems. A contribution of this paper is a MULTI-STEP BOTTOM-UP analysis approach to manage this problem. The reachability analysis of an *Integrated Petri-Net* can be performed in a compositional way. It is possible because the process to obtain an *Integrated Petri-Net* from a *root scenario* and its interactions does not introduce new arcs when a Petri-Net corresponding to a related scenario is fused or substituted into a place or transition of the *Integrated Petri-Net*.

Using this approach, the *Integrated Petri-Net* is divided into a set of Petri-Nets that preserves the properties and concurrency characteristics of the *Integrated Petri-Net*. This approach: (1) takes the *Petri-Nets* corresponding to *sequentially related scenarios* and adds them to a set; (2) analyzes each one of these Petri-Nets; (3) analyzes the *Integrated Petri-Net*; and (4) returns the feedback, if some problem results from the Petri-Nets analysis. The Petri-Nets corresponding to *concurrently related scenarios* are preserved into the *Integrated Petri-Net* because it might interact by *message passing* among them.

In Figure 7, the transition ***T13*** must be replaced by the Petri-Net corresponding to *PROCESS BIDS* scenario because it is sequentially related to the root scenario (Submit Order) by sub-scenario relationship. However, it is not replaced, because it does not interact concurrently with some related scenario or the root scenario, and it can be independently analyzed.

This approach: (1) reduces the state explosion problem; (2) increases the feasibility of Petri-Nets and (3) may enable verification of properties which may fail on Integrated Petri-Net due to a state explosion problem.

D. Sample Application

Two application cases were used to illustrate the proposed approach; an e-business and a concurrency problem. These applications exhibit some concurrent processes.

1) Online Broker System: In the Online Broker System, Suppliers' scenarios are executed concurrently, as shown in Figure 2 and the integrated Petri-Net (PN) of Figure 7. The Petri-Net corresponding to PROCESS BIDS transition (T13) is first analyzed.

Assessing Uniformity, ***Feasibility*** and ***Non-interferential***: (1) the root scenario and its related scenarios are uniform, because all scenario events has particular name and involves the participation of actors and the use of resources; (2) the resulting Petri-Nets (Fig. 7) are feasible, because all places have input and output arcs (except final places), and they do not present isolated sub Petri-Nets; (3) the presence of a token in "An Order has been broadcasted" place indicates that the transitions "T10.1", "T11.1" and "T12.1" (corresponding to suppliers scenarios) are enabled simultaneously (non-determinism) in the PN (Fig. 7). It is a warning, not an error.

Assessing Bounded, Safe, Live and Reversible: The reachability analysis of the PN using the PIPE2 tool [13] reveals that PN is: (1) *bounded* and not safe; (2) not *live*, because the firing sequence ⟨T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, Fork 1, T10.1, T10.2, T10.3, T11.1, T11.2, T11.3, T12.1, T12.2, T12.3, Join 1, T13, T16, T0, T1, T2, T3, T4, T5, T6, T7, T8, T15⟩ is a shortest path to Deadlock; and (3) not *reversible*, because it is not *bounded*, not *safe* and not *live.* There is a deadlock when the *Order of the Customer is empty*.

2) Readers-Writer Problem: In order to detect problems due to concurrency, we used a faulty version of Reader-Writer scenarios. In Reader scenario, the *post-condition* of the episode 1 is not specified, as shown in Figure 4 and the integrated Petri-Net (PN) of Figure 8.

Assessing Uniformity, ***Feasibility*** and ***Non-interferential***: (1) these scenarios are uniform, because all scenario events has particular name and involves the use of resources; (2) the resulting Petri-Net (Fig. 8) is feasible because all places have input and output arcs, and it does not present isolated sub Petri-Nets; (3) the presence of a token in "*Data Set is available*" place indicates that the transitions "*R_1*" and "*W_1*" are enabled simultaneously (non-determinism) in the PN (Fig. 8). It is a warning, not an error.

Assessing Bounded, Safe, Live and Reversible: The reachability analysis of the PN using the PIPE2 tool [13] reveals that PN is: (1) *bounded* and safe; (2) not *live*, because the firing sequence ⟨R_0,R_1,R_2,R_3,R_0, W_0⟩ disables the transitions ⟨W_1,W_2,W_3⟩ of the *Writer* (Deadlock); and (3) not *reversible*, because it is *bounded*, *safe* and not *live.* There is a deadlock because the Reader does not return the shared resource "*Data Set"*.

V. Related Work

Many researches have shown the importance to formalize the informal aspects of scenarios in order to benefit from automated scenarios analysis. Some research focused on developing the formal semantics for scenario representations, like [9] and [20]; others are focusing on developing techniques to translate scenarios into executable models. Among the approaches to translate scenarios into Petri-Net notations, we include [10][17][18].

In [10], it is proposed a systematic procedure to convert use case descriptions into Constraint-based Modular Petri-Nets (CMPNs), allowing the analysis of use cases. To facilitate the transformation, use cases are described in relation to formal definition of *pre and post-conditions*, and represented like Action-Condition tables. Use cases are considered as a collection of interacting and concurrently executing units of system functionalities. Petri-Net analysis techniques can be used to evaluate *completeness* and *consistency* properties in CMPNs. It is the unique approach that manages the state explosion problem by dividing the CMPN into a set of slices. However, intermediate models are created and alternative/exception flows of use cases are not considered.

In [17], it is proposed semantics for use cases based on Petri-Nets where the property of *consistency* can be evaluated. This approach deals with sequential relationships among use cases (include and extend). However, the language to describe use cases does not deal with communication between concurrent use cases.

In [18], it is proposed an approach to formalize use cases with Petri-Nets. A semi-formal language is proposed for use case syntax. This syntax is based on message sender and receiver objects, and the events in use cases can be sequential, conditional, iteration or concurrent (parallelism). Petri-Nets are derived extracting objects and messages between objects. Based on the obtained Petri-Net model, criteria to detect *incompleteness, inconsistency* and *incorrectness* properties are described. However, it is necessary to create intermediate "event frames" for the extraction of the objects and messages from each one of the sentence events.

The related Petri-Net based approaches exhibit the following shortcomings: (1) scenarios are described in relation to formal definition of pre and post-conditions; (2) there is a lack of systematic procedures on how to represent the scenarios; (3) the procedures to translate scenarios into Petri-Nets are not automated and intermediate models are created; (4) interactions among scenarios are rarely represented - *modularity*; and (5) the state explosion problem is not managed.

In contrast, our approach: (1) uses a semi-structured natural language to write scenarios; (2) defines an abstract model and concrete syntax for scenarios; (3) implements automated transformation rules; (4) provides powerful characteristics to deal with modularity, (5) identifies concurrency problems; and (6) manages the state explosion in Petri-Net analysis.

VI. CONCLUSION

In this paper, we introduced an approach for the automated analysis of scenarios through Petri-Nets. On the basis of this approach, it is possible to: (1) show how the properties of uniformity, feasibility, non-interference, bounded and live contribute to correctness, consistency and completeness; (2) show problems related to correctness, consistency and completeness in Petri-Nets derived from scenarios and their interactions, as early as possible in SD; (3) provide modularity by first developing local processes using scenarios, then composing them to one component; and (4) support traceability, indicating the problems in Petri-Nets and showing the source of the problems in scenarios (or their interactions).

Our approach provides benefits due to the following reasons: (1) it manages the *state explosion* issue in Petri-Nets; (2) it preserves the consistency between scenarios and their equivalent Petri-Nets; (3) it is possible to build a tool implementing our approach, like C&L tool [3]; (4) it starts with the software development process; and (5) it shows that our approach is applicable to scenarios that involve concurrency.

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 paper, 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.

Future research: (1) investigating other properties related to the main qualities considered; (2) developing tools to support the compositional analysis of Petri-Nets.

REFERENCES

[1] M. Andersson, and J. Bergstrand, Formalizing Use Cases with Message Sequence Charts, Master's thesis, Lund Inst. of Technology, 1995.

[2] B. W. Boehm, "Guidelines for verifying and validating software requirements and design specifications," Proc. European Conf. Applied Information Technology, pp. 711-719, Sept. 1979.

[3] C&L, Scenarios & Lexicons, 2014. Available at: http://pes.inf.puc-rio.br/cel.

[4] 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.* vol. 79, num. 6 , pp. 772-790, 2006.

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

[6] L. Chung and N. Subramanian, ''Software architecture adaptability: an NFR approach,'' Proceedings of the 4th International Workshop on Principles of Software Evolution, pp. 52--61, 2001.

[7] A. B. Downey, The Little Book of Semaphores. Green Tea Press, 2005. Available at http://greenteapress.com/semaphores.

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

[9] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen, Formal Approach to Scenario Analysis, in IEEE Software, pp. 33-41, 1994.

[10] W. Lee, S. Cha, and Y. Kwon, Integration and analysis of use cases using Modular Petri Nets in requirements engineering, in IEEE Trans. on Software Engineering, vol. 24, num. 12, pp. 1115-1130, 1998.

[11] J. Lee, J. I. Pan, and J. Y. Kuo, Verifying scenarios with time petri-nets, Inf. Softw. Technol., vol. 43, num. 13, pp. 769–781, 2001.

[12] J. C. S. P. Leite, G. Hadad, J. Doorn and G. Kaplan, A scenario construction process, Requirements Engineering Journal, Springer-Verlag London Limited, vol. 5, num. 1, pp. 38-61, 2000.

[13] PIPE2, Platform Independent Petri net Editor 2, 2014. Available at http://pipe2.sourceforge.net

[14] W. Reisig, Petri Nets: An Introduction, Springer-Verlag, Berlin, Heidelberg, 1985.

[15] E. Sarmiento, E. Almentero, J. C. S. P. Leite, and G. Sotomayor, Mapping textual scenarios to analyzable Petri-Net models, In International Conference of Enterprise Information Systems, May 2015, to be published.

[16] E. Sarmiento, E. Almentero, and J.C.S.P. Leite, Using Correctness, Consistency, and Completeness patterns for automated scenarios analysis, In 5th IEEE Workshop of Requirements Engineering Patterns - RePa, 2015.

[17] S. Somé, Petri Nets based formalization of textual use cases. Tech. Report in SITE, TR2007-11, Uni. of Ottawa, 2007.

[18] J. Zhao, and Z. Duan, Verification of use case with petri nets in requirement analysis, In ICCSA, Part II, vol. 5593, O. Gervasi, D. Taniar, B. Murgante, A. Laganà, Y. Mun and M.L. Gavrilova, Eds. LNCS, Springer, Heidelberg, 2009, pp. 29-42.

[19] D. Zowghi and V. Gervasi, "On the interplay between Consistency, Completeness, and Correctness in requirements evolution", Information and Software Technology, vol. 45, pp. 993-1009, November, 2003.

[20] D. Sinnig, P. Chalin, and F. Khendek, LTS semantics for use case models, In *ACM symposium on Applied Computing*, pp. 365-370, 2009.

[21] M. Barnett, W. Grieskamp, W. Schulte, N. Tillmann, and M. Veanes, Validating use-cases with the asmL test tool, In 3rd International Conference on Quality Software, 2003.