Using Correctness, Consistency, and Completeness Patterns for Automated Scenarios Verification

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

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

***Abstract*—Scenario-based approaches are often used for Software Requirements Specification (SRS). Since scenarios are usually written in natural language, they may be ambiguous and, sometimes, inaccurate, which impair requirements quality. One of the major factors for this problem is because interactions among scenarios are seldom represented explicitly. As such, the importance of correctness, consistency and completeness, in the context of scenario-based representations should be rethought. In this paper, we employ the NFR approach to organize the non-functional requirements related to correctness, consistency and completeness as a catalog of non-functional requirements (NFR). We represent the initial catalog as non-functional requirements patterns. These initial patterns can be effectively used for automated scenario-based SRS verification. The identified patterns can be operationalized by evaluating properties related to these NFRs. We demonstrate the applicability of this catalog by instantiating it to the evaluation of an SRS based on a scenario language.**

***Index Terms*—requirements, analysis, verification, scenario, correctness, consistency, completeness, pattern, NFR, Petri-Net.**

I. Introduction

Many problems and high-risk issues that arise during the software development (SD) process are related with deficiencies in the requirements engineering (RE) process. Assessing that Software Requirements Specification (SRS) satisfices (relative satisfaction) the necessary quality is crucial to the success of any software development project, since the SRS is the anchor for software development. However, assessing the quality of a SRS is not a simple process, mainly, because a system must often support multiple users with different viewpoints and needs, which may be contradictory.

Nowadays, Scenario-based representations are frequently used in RE for requirements specification or modeling. In this context, requirements are represented as a collection of scenarios, which are described by specific flows of events. The use of scenarios helps understanding a specific situation in an application, prioritizing their behavior [7]. Some of the most prominent languages to write scenarios are restricted-form of use case or scenario descriptions [2][7][19][20]; however, they lack precise semantics to support the analysis of structural and behavioral properties of the application.

Scenarios are a key concept for writing SRS [13], because: (1) they describe requirements such that users and developers can easily understand, (2) they may support the early detection and resolution of ambiguity or quality problems. Due to these and others characteristics, a scenario-based SRS has the potential to influence positively the SD process, requirements engineers need to pay special attention to its quality, in special with respect to *correctness*, *consistency* and *completeness* since they will anchor further development.

As stated before, scenarios are usually written in natural language, in order to improve the interaction between all the actors involved in RE process. However, natural language is by definition *ambiguous* leading to *incorrectness*, *inconsistent* and *incompleteness*. *Inconsistent* requirements occur when two or more users have conflicting requirements, or the captured requirements are internally inconsistent when one or more requirements override others. *Incomplete* requirements occur because the world is complex; as such, users or clients are not able to fully understand the impact of present decisions. *Incorrect* requirements may occur when the acquired requirements do not accurately reflect the facts, or erroneous predicts about future states.

Numerous techniques have been developed to deal with these quality problems in SRS and each one with its own context of applicability. However, most SRS still do not meet these qualities. According to Glinz [13], it is not only a problem of applying the right methods and processes for SRS until they yield the desired qualities; the qualities themselves are part of the problem.

In this paper, we employ the NFR approach [3] to: (1) model the relationships between correctness, consistency and completeness qualities, (2) organize the related properties to these qualities as a catalog of NFR patterns, and (3) show what kind of operationalizations can be made to support of these NFR patterns. These initial NFR patterns can be effectively used for automated scenario-based SRS verification.

We demonstrate the applicability of this catalog by instantiating it to the evaluation of SRS based on a scenario language evolved from previous works [7].

Details of our work are presented through the background, the NFR catalog of Correctness, the strategy to evaluate Correctness, towards an example application and conclusions.

II. BACKGROUND

A. Software Requirements Specification Verification

The terms Verification and Validation are commonly used in software engineering to mean two different types of analysis. According to Boehm [1], the usual definitions are:

- Verification: to establish the truth of the correspondence between a software product and its specification, i.e. are we building the product right?
- Validation: to establish the fitness or worth of a software product for its operational mission, i.e. are we building the right product?

In other words, validation is concerned with checking that the software meets user's actual needs, while verification is concerned with whether the software is well-engineered, error-free, and so on. Verification helps to determine whether the software is of quality, but it doesn't ensure that it is useful [16].

Validation is concerned with checking that the software meets user's actual needs, while verification is concerned with whether the software is well-engineered, error-free, and so on. Verification helps to determine whether the software is of high quality, but it does not ensure that the software is useful [16].

Verification is a relatively objective process. It includes the activities associated with producing high quality software: inspection, analysis, simulation or checklist heuristics for evaluating that specifications are expressed precisely enough.

In contrast, validation is a subjective process. It should confirm that the *Universe of Discourse* situations, occurrences, have been reported in accordance with the real world needs of the users. Requirements validation includes activities such as structured interviews or meetings with users.

In this paper, we consider that a SRS represents the expectations, needs or desires of users. So, verifying a SRS should: (1) check if the set of requirements are consistent 2) check if the degree of incompleteness is acceptable. If the SRS does not meet these criteria, then the SRS should be re-constructed.

A SRS can be verified by the following approaches: (1) *inspection*, to examine carefully and critically, especially for flaws; (2) *analysis,* a series of logical deductions based on logic oriented representations; (3) *simulation,* execution of a model, usually with a computer program; (4) *checklist*, an examination of the SRS by pre-defined rules or patterns.

Each of the above-mentioned approaches for SRS verification has its own advantages and drawbacks. *Formal verification* through model checking technique can be used for analysis of structural and behavioral properties; however, this approach is not suitable for models with large number of states (the complexity of the generated reachability graph is exponential).

B. Scenario

For practical reasons, and in order to allow for an easy communication between users and developers, requirements are written using natural language-based textual templates, such as in [2][7][13]. Textual scenario-based approaches 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 [5].

Unfortunately, textual scenarios exhibit some shortcomings: (1) scenarios informally specified are usually hard to analyze, because natural language is by definition ambiguous; (2) modularity is poorly supported, because the interactions among scenarios are rarely represented explicitly; and (3) currently**,** there are no systematic approaches to represent concurrency opportunities since initial requirements descriptions (scenarios are rarely truly independent, they interact [5]).

Many researches have shown the importance to systematize the informal aspects of scenarios in order to benefit from automated scenarios verification [4] [6] [5][10][11] [13][19].

So, the development of lightweight approaches to support the automated verification of SRSs is a challenging topic.

The term *scenario* is used with different meanings in different contexts. We therefore state a definition from [18].

Definition 1: A ***scenario*** is a collection of partially ordered event occurrences, each guarded by a set of conditions (pre-condition, 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.

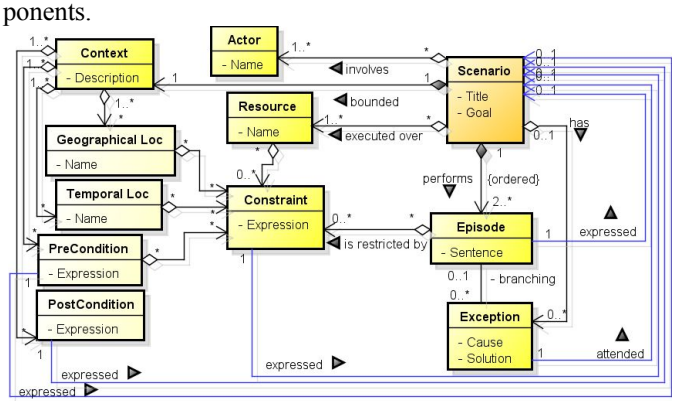

Fig. 1. Scenario model, from [18]

According to conceptual model of Figure 1, the ***scenario language*** is composed of the Scenario, Context, Resource, Actor, Episode, Exception and Constraint entities.

In this language, a *scenario S* starts in an initial state (*context*) with all necessary *Resources*, 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* [18].

Fig. 2 presents a scenario example. It shows the situation "***Submit Order***" using the scenario language used in this work. After, a Customer has submitted an order; the Online Broker System broadcasts it to the suppliers (SupplierA, SupplierB and SupplierC). "SupplierA Bid", "SupplierB Bid" and "SupplierC Bid" scenarios are executed concurrently; they submit a bid for

the order. The remaining situations in the "Online Broker System" example are shown in [18]. Reference [18] explains what each ***scenario*** field means and how it should be filled.

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. **REPEAT** 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

Fig. 2. Description of "Submit Order" scenario in the Broker System [18].

C. NFR Framework

The NFR Framework [3] is a Goal-Oriented RE approach for capturing NFRs in the domain of interest, and defining their interdependencies and operationalizations. We chose the NFR Framework because it allows to: (1) model the NFRs and their decomposition, (2) design alternatives for different NFRs, (3) deal with conflicts, tradeoffs, and priorities, and (4) evaluate the decisions impact centered on NFRs. These NFRs are modeled using a Softgoals Interdependency Graph (SIG). The SIG graphically represent NFRs as softgoal nodes (clouds); their refinements using AND/OR decompositions links; their positive/negative inter-dependencies as some+ (help), some- (hurt), some++ (make), some-- (break) contribution links; their operationalizations as leaf nodes; and claims as annotations in natural language. Generally, softgoals are named using the convention Type [Topic1, Topic2...] where Type is the softgoal and Topic is the field of application of Type; Topic is optional.

Fig. 3 illustrates a SIG that models Correctness, by decomposing in Consistency and Completeness (HELP links).

D. NFR Pattern

The NFR pattern approach [14] focuses on the reuse of NFR knowledge [3]. NFR patterns may be specialized to create more specific patterns, composed to build larger patterns, or instantiated to create occurrence patterns using existing patterns as templates. NFR Patterns can be seen as an evolution of the NFR catalogs. Supakkul *et. al* [14] defined four types of patterns for capturing and reusing knowledge of NFRs: (1) *Objective Patterns* are used to capture the definition of NFRs in terms of specific softgoals to be achieved; (2) *Problem Pattern* captures knowledge of problems or obstacles to achieving goals; (3) *Alternatives Pattern* is used to capture different means, solutions, and requirements mappings; (4) *Selection Pattern* allows to choose the best alternative considering their side-effects. In [15], Serrano and Leite presented another type of pattern: the *Questions Pattern,* which helps the refinement of softgoals towards operationalizations via *Alternative Patterns*.

III. Modeling Correctness as a NFR Catalog

Our proposal is one of the first to represent SRS *correctness*, *consistency* and *completeness* as a catalog based on NFR framework and NFR patterns. The NFR catalog was modeled taking into account the decomposition into softgoals to be achieved by SRS, which is described using natural language-based scenarios. Moreover, our contribution also exposes the links and impacts between the softgoals related to the main NFR (Correctness). We introduce a novel perception of *correctness* and its complex relationships with *consistency* and *completeness*, describing it as a quality that should be satisficed by contributions of related qualities or properties.

Based on the literature, we have developed a SIG for SRS *Correctness* (Figure 3), which will be the base for cataloged information and will be detailed by a series of NFR-Patterns. In order to elaborate the NFR catalog to achieve Correctness, we defined a set of three steps, which are detailed below.

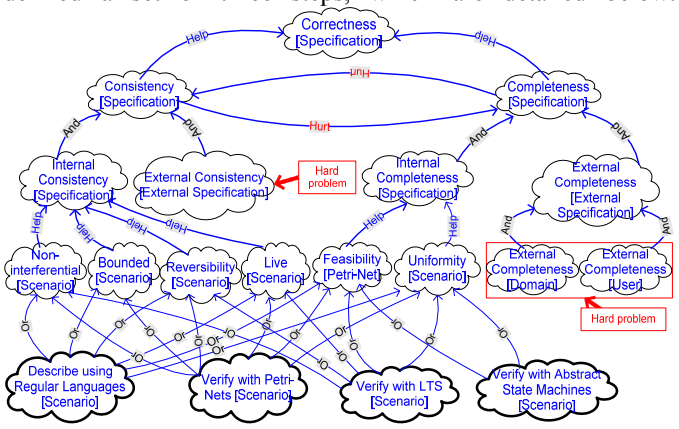

Fig. 3. SIG of SRS correctness, consistency and completeness.

A. Defining the Main NFRs

In the NFR Approach, correctness, consistency and completeness are non-functional requirements (NFR) that need to be satisficed (relative satisfaction) by the specifications.

In our work, it is assumed that: (1) *correctness* is the most important quality [13], and (2) there is an important causal relationship between *consistency*, *completeness* and *correctness* of SRS [12]. Zowghi [12] 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 the SRS can reduce the completeness, thereby again diminishing correctness.

Definition 2: 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 3: 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 4: ***Correctness*** or *Adequacy* means the quality of being able to meet a need satisfactorily [13]. Thereby, having a *consistent* and *complete* set of scenarios contributes for requirements specification *correctness*.

B. Modeling the SIG

In order to evaluate *correctness*, *consistency* and *completeness*, we apply the NFR qualitative reasoning approach [3], the goal here is to achieve good *correctness* in scenario-based SRS. Fig. 3 illustrates the SIG that models the SRS *Correctness* and how SRS *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 real user's needs. Interdependency between SRS Consistency and Completeness negatively impacts (hurt) on both [12]. *Consistency* and *Completeness* are decomposed – using AND links – in *Internal* and *External* softgoals, following the lead of [1] and [12]. Evaluating External Consistency and External Completeness is a hard problem because it depends on external specifications, external domain models and user's needs satisfaction [1][12]. *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 [4], OR (2) Verifying Scenarios with Petri-Nets [5][6][7][10][11], OR (3) Verifying Scenarios with Abstract State machines [20]; OR (4) Verifying Scenarios with Labeled Transition Systems – LTS [19]. *Non-interferential*, *Bounded*, *Reversibility* and *Live* can be operationalized by: (1) writing scenarios using Regular Languages [4], OR (2) Verifyng Scenarios with Petri-Nets [5][6] [10][11]; OR (3) OR Verifying Scenarios with LTS [19].

C. Patternizing

The knowledge modeled in Figure 3 is captured as NFR patterns using the approach proposed by Supakkul in [14].

1) Objective Pattern: Using *NFRIdentification* and *NFRDecomposition* it is possible to describe the *Correctness* SIG as an *Objective Pattern*.

Fig. 4 shows the *Correctness* SIG represented as an *Objective pattern*. Six refinement rules are necessary to express the Correctness SIG. The first refinement rule, R1, relates the Correctness NFR to the software requirements specification resource. R2 defines the contributions between the two main *Correctness*-related NFRs and the Correctness NFR. R3 and R4 define the contributions from the four NFRs to the main *Correctness*-related NFRs (Consistency and Completeness). R5 and R6 define the contributions from six NFRs (Non-interferential, Bounded, Reversibility and Live, Feasibility, Uniformity) to the four NFRs (Internal/External Consistency and Completeness) related to the main *Correctness*-related NFRs (Consistency and Completeness). For example, R2 states that the Consistency and Completeness NFR help the Correctness NFR. Moreover, R3 states that Consistency NFR is decomposed – AND – in Internal and External Consistency.

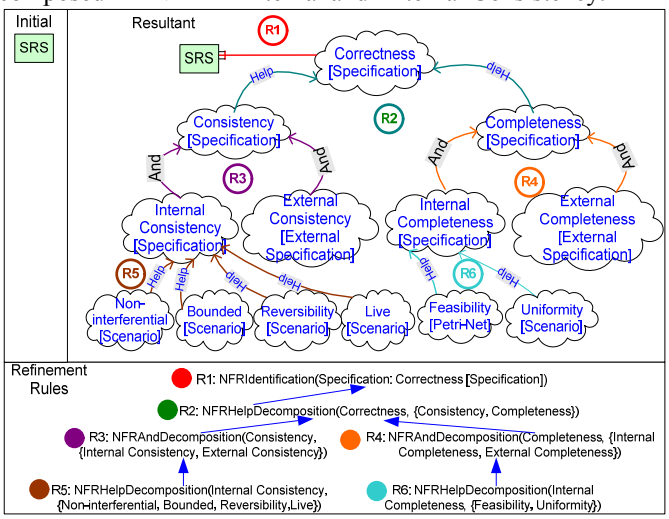

Fig. 4. Correctness objective pattern.

2) Problem Patterns: In Figure 5 and Figure 6 as examples, the presence of *Conflicting Events [scenario]* and *inherent Ambiguity of Natural Language [Scenario]* are soft-problems that can **HURT** the *Live* and *Feasibility* softgoals, respectively. *Conflicting Events [scenario]* is caused by the use of *Textual Narratives [Scenario]* to write scenarios, which is composed of *Lacking of Constructs to represent interactions among events* AND *Lacking of Constructs to represent interactions among scenarios threats*, where the former threat was made possible by *Lacking of Syntax and Semantic Rules. Ambiguity of Natural Language [Scenario]* is caused by the use of *Textual Narratives [Scenario]* to write scenarios, which is composed of *Lacking of Guidelines* OR *Lacking of Scenario Model* threats, where the former threat was made possible by *Lacking of Syntax and Semantic Rules.*

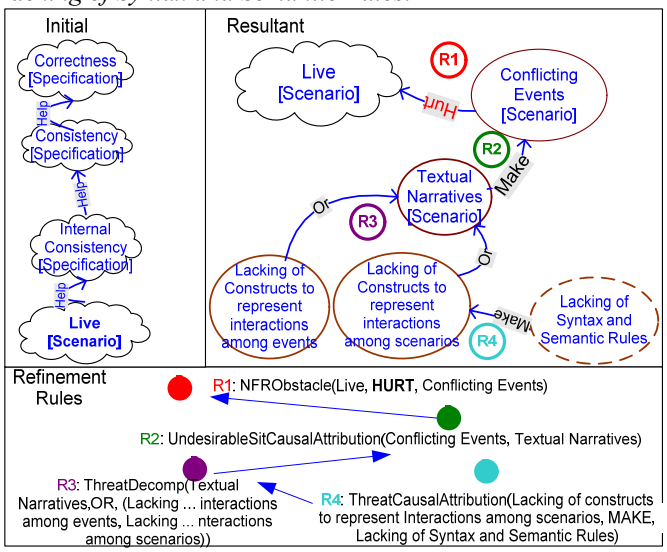

Fig. 5. Conflict can HURT the Live NFR. (problem pattern)

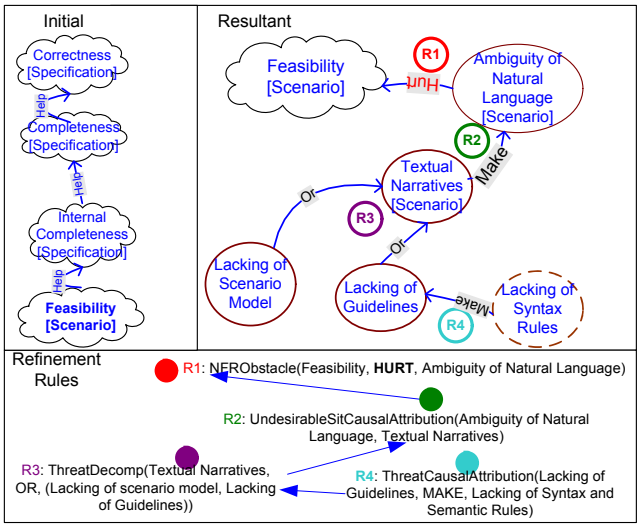

Fig. 6. Ambiguity can HURT the Feasibility NFR. (problem pattern)

3) Question Patterns: Fig. 7 shows the groups and questions (Goal-Question-Operationalization - GQO method [15]) that should help the requirements engineer to identify good practices that operationalize the *Live* NFR represented as a *Question pattern*. The *GroupsIdentification* refinement rule, R1, defines three question groups for the *Live* NFR. Each *QuestionsIdentification* refinement rule, R2 to R4, lists the good practices questions pertinent to each group. For example, R1 states that *"Simulation techniques"* and *"Non-Determinism analysis"* are two question groups of the *Live* NFR. R2 to R4 list two or three questions that help indentifying good practices, or operationalizations, that answer to the *"Simulation techniques"*, *"Non-Determinism analysis"* and *"Deadlock analysis"* question groups.

Fig. 8 shows the groups and questions that should help the requirements engineer to identify good practices that operationalize the *Feasibility* NFR represented as a *Question pattern*. R2 and R3 list two or three questions that help indentifying good practices, or operationalizations, that answer to the "Use of Model" and "Dependency analysis" question groups.

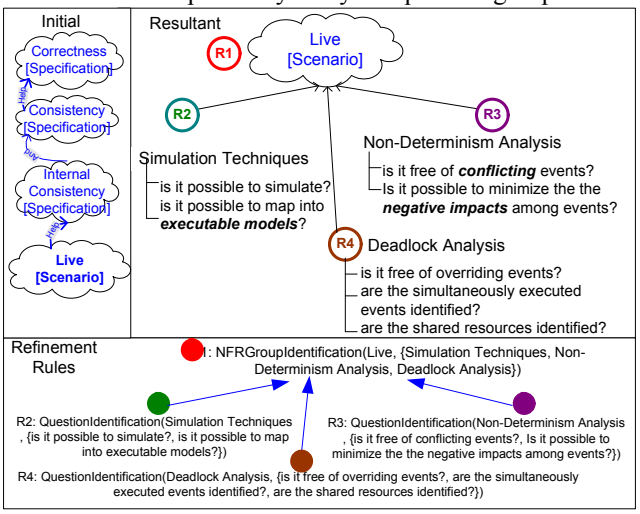

Fig. 7. Live question pattern.

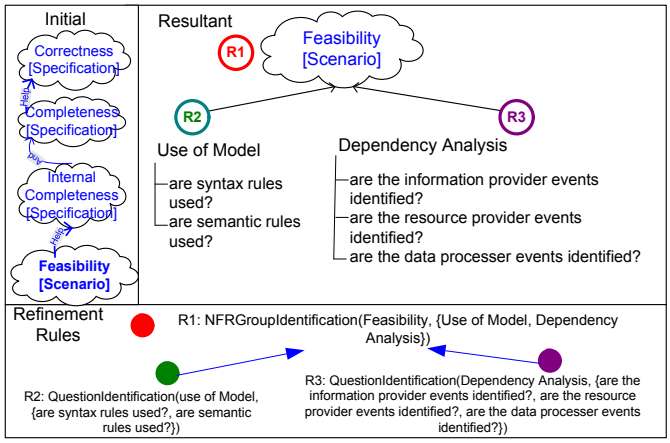

Fig. 8. Feasibility question pattern.

"Non-determinism and deadlock analysis" are related to the identification and resolution of negative interactions (conflict, negative impact, cancel or override) among requirements or scenarios. "Dependency analysis" is related to the identification of positive interactions (require, inform, configure or flow) among requirements or scenarios. These interactions and their types are defined and detailed in [17].

According to [17], there is an interaction when two or more requirements have some effect on each other. These interactions can be caused by different viewpoints of stakeholders, change or re-use of requirements, component-based development, among others.

4) Alternative Patterns: Using the alternative solutions pattern in Figure 9 as an example, *Describe Using Regular Languages*, *Verify with Petri-Nets* and *Verify with LTS* are alternative solutions that **BREAK** *Lacking of Syntax and Semantic Rules* soft-problem. The solutions are captured by *R1:RiskTransfer*, *R2:RiskTransfer* and *R3:RiskTransfer* rules. All alternatives have MAKE contributions towards *Non-Intereferential*, *Feasibility* and *Uniformity* softgoals.

The *Verify with Petri-Nets* alternative has two MAKE contributions towards *Availability* (of Petri-Net analysis tools, such as PIPE2 [8]) and *Portability* (related to an interchangeable format between tools - Petri Net Markup Language - PNML) softgoals, while the *Verify with LTS* alternative has a HURT contribution towards *Portability* (there is not an interchangeable format) and a HELP contribution towards *Availability*. The side-effects are recorded by *R4:SideEffect* to *R9:SideEffect* rules for the *Verify with Petri-Nets* solution, and by *R10:SideEffect* and *R12:SideEffect* rules for the *Verify with LTS* solution.

Fig. 9 shows the alternative solution pattern that breaks the *Lacking of Syntax and Semantic Rules* soft-problem. This solution pattern contains possible operationalizations that answer to questions related to "Simulation", "Deadlock analysis" and "Non-determinism analysis" in *Live* question pattern from Figure 7.

We prioritized the operationalizations of the *Live* NFR because its operationalizations positively contribute into the *Non-interferential*, *Feasibility* and *Uniformity* NFRs.

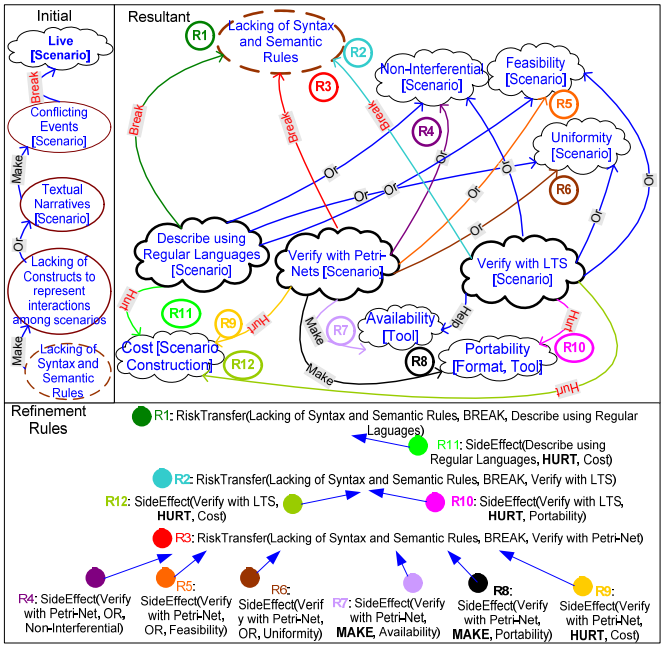

Fig. 9. Lacking of syntax and semantic rules alternative solution pattern.

5) Selection Patterns: It was used the *Weight-based Quantitative* Selection Pattern [14] for alternatives selection.

Using Figure 10 as an example, three alternatives, *Describe Using Regular Languages*, *Verify with Petri-Nets* and *Verify with LTS* are alternatives for mitigating *Lacking of Syntax and Semantic Rules* vulnerability, where the former has three positive side-effects towards *Non-Intereferential*, *Feasibility* and *Uniformity* softgoals.

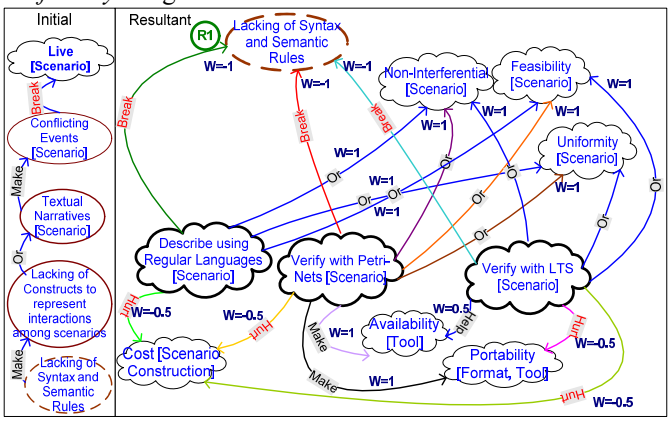

Fig. 10. Weight-based selection of alternative solution pattern.

Let us suppose that the stakeholders agree with the following weight assignments: 1.0, 0.5, -1.0, -0.5 to each of Make (++), Help (+), Break (–), and Hurt (-) contributions respectively, as shown in Figure 10.

For selecting among these three alternatives, we use the multi-step bottom-up process defined in [14]. In this example, we have one-level, then, *Describe Using Regular Languages*, *Verify with Petri-Nets* and *Verify with LTS* are evaluated and selected using as parameter their fitness values.

Next, *Verify with Petri-Nets* is selected for its higher fitness value of 3.5, over *Verify with LTS*, which has a fitness value of 1.5. The higher fitness is obtained from:

fitness(Petri-Nets)
= weight(Break(Petri-Nets,Lacking Syntax Rules) +
weight(Make(Petri-Nets,Non-Interferential) +
weight(Make(Petri-Nets,Feasibility) +
weight(Make(Petri-Nets,Uniformity) +
weight(Hurt(Petri-Nets,Cost) +
weight(Make(Petri-Nets,Portability) +
weight(Make (Petri-Nets,Availability) +
= -1 + 1 + 1 + 1 + (-0.5) + 1 + 1
= 3.5

IV. Requirements Specification Verification Using Correctness, Consistency and Completeness Patterns

A. Related Work

In order to perform an automated verification of structural and behavioral properties in scenario-based SRS, some research focused on (1) developing formal semantics for scenario representations based on regular languages [4]; or (2) developing techniques to translate them from informal to formal representations, like Petri-Nets or LTS [19]. The verification outcome can be used to improve the SRS, since the identified problems can be traced to the scenarios.

Among the approaches to formalize informal specifications of scenarios using Petri-Net notations include [5][10][11]. In these approaches, scenarios are translated into Petri-Nets, which are used as the mechanism to enable rigorous verification. Petri-Net based approaches exhibit the following shortcomings: (1) scenarios are described in relation to formal definition of pre and post-conditions; (2) lacking of systematic procedures on how to represent the scenarios; (3) the procedures to transform scenarios into Petri-Nets are not automated and intermediate models must be created; and (4) scenario notations do not make explicit interactions among scenarios.

B. Petri-Nets

A Petri-Net is composed of nodes that denote places (Place) or transitions (Transition). Nodes are linked together by arcs (Arc). *Transitions* are active components that model the activities that can change the state of the system. *Places* are passive components and placeholders for tokens, that model communication medium, 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. *Arcs*: Input arcs start from places and ends at transitions, while output arcs start at a transition and end at a place.

C. Our Method

So, in order to detect some problems in SRS, we will evaluate some static and dynamic properties in Scenario-based SRS and Petri-Nets. More details of this method are presented in [18]. The SRS verification comprised the following steps:

1) Transforming Scenarios into Petri-Net Models: Once scenarios are constructed (conform metamodel of Figure 1), it

is possible to automatically generate Petri-Net formal specifications. In the context of Petri-Nets, a *Scenario* ***S***:

- Starts at an *idle state* with all necessary resources, pre-conditions or constraints;
- Performs a collection of partially *ordered event occurrences* (episodes or exceptions), each guarded by a set of *conditions* (pre-condition, post-condition) and restricted by a set of *constraints*;
- Returns to the *idle state* and *releases* the *resources*, pre-conditions or constraints after completion.

Details of the procedure to transform Scenarios into Petri-Nets are shown in [18].

2) Reusing NFR Patterns: The related work in using the potential of Petri-Nets for scenario formalization indicates that Petri-Nets are an effective mechanism for scenario-based SRS verification. The motivation behind using Petri-Nets as executable models can be attributed to two reasons: (1) the reachability analysis can reveal the incorrect behavior of scenarios (Bounded, Safe, Deadlock free); (2) the availability of Petri-Net tools, such as PIPE2 [8].

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 [9] for verification of: (1) static properties like *Correct Token Passing* and *Fully Connected* (*Feasibility*); and (2) dynamic properties like *Determinism*, *Bounded*, *Reversibility* and *Deadlock free* (related to *Non-interferential*, *Bounded*, *Reversibility* and *Live*). Fig. 11 shows that pattern **P1** is specialized sub-pattern from pattern **P2** where **P1** captures a definition of *Verify with Petri-Nets* that is more specific than the general definition defined in **P2** (Figure 11).

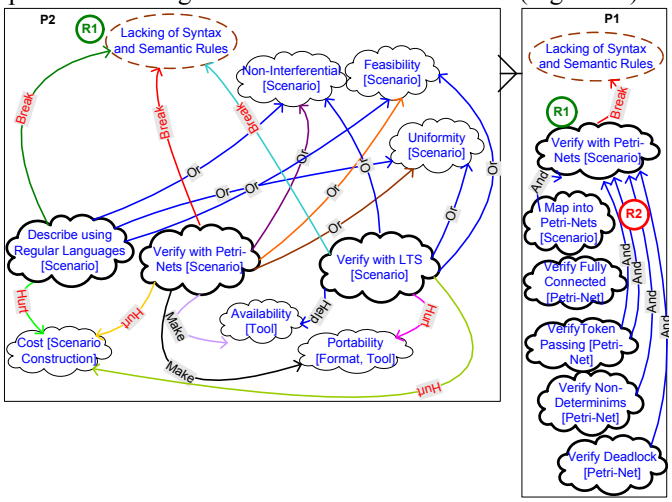

Fig. 11. Verify with Petri-Nets pattern specialization.

3) Verifying Scenarios with Petri-Net Models: 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 element in scenario and its relationships must be described as established in the scenario model (more details of style checks in [7]).

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

- A place in the PN model should be sent from a particular transition and be received by a destination transition. If they are missing, the tokens in the PN model *cannot pass correctly* [6].
- The transitions in the PN model 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* [6][11].

To evaluate ***non-interferential***, we find wrong information (dynamic) 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 [5][6].

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 [11].
- *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* [5][6].

D. Sample Application

For illustration, the Petri-Net (Figure 12) of the *Submit Order* scenario was obtained from its scenario (Figure 2). For *Submit Order* scenario, 15 *events* are identified (13 in the episodes and 2 in the exceptions): 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).

Fig. 12 shows the verification of the Petri-Net using the PIPE2 tool [8]. A deadlock occurs when an exception hinds the achievement the scenario goal.

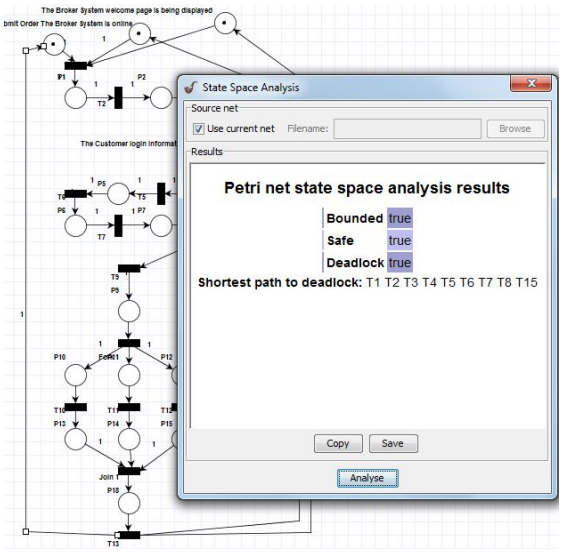

Fig. 12. Petri-Net of "Submit Order" in the Broker System.

V. Conclusion

There is a lack of systematic approaches to model and organize the related properties to *Correctness*, *Consistency* and *Completeness* of scenario-based SRS. We have used a catalog of NFR patterns to help the organization of such properties. The given scenarios may interact or compete with each other by *communication channels* or *shared resources* that would be prone to erroneous situations such as deadlocks and unbounded. Our approach provides benefits due to the following reasons: (1) it identifies properties to be evaluated due to interacting scenarios; (2) it shows what kind of operationalizations can be made to support the evaluation of properties due to interacting scenarios described using the scenario language proposed in [18]; and (3) it is possible to reuse the patterns and specialize them for more specific scenario languages.

The process of SRS verification is a complicated activity; no single solution is effective to resolve the challenges of leading with correctness, consistency and completeness. We have presented the potential of *Uniformity, Non-interferential*, *Live* and *Feasibility* patterns with operationalizations using Petri-Net models for automated SRS verification. Our sample application involved simple sentences and simpler scenario.

Limitation: The mapping from scenarios into executable models works well if a requirements engineer can properly write scenarios using the syntax and semantic rules described as restricted-form of natural language (RNL). It is our assumption that the use of RNL is accepted by the most stakeholders in RE process, and it is amenable to automated processing.

We *further* aim to work towards complex scenarios and investigate the scalability of our approach.

Acknowledgment

Leite acknowledges the support from CNPq.

References

[1] B. W. Boehm, "Guidelines for verifying and validating software requirements and design specifications," Proc. European Conf. Applied Information Technology (IFIP ',79), pp. 711-719, Sept. 1979.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

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

[10] S. Somé, Petri Nets based formalization of textual Use Cases. Tech. Report in SITE, TR2007-11, Uni. of Ottawa, 2007.

[11] 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.

[12] 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.

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

[14] S. Supakkul, T. Hill, L. Chung, T. T. Tun and J.C.S.P. Leite, An NFR Pattern approach to dealing with NFRs, 18th IEEE Intl. Requirements Engineering Conference, 2010.

[15] M. Serrano and J. C. S. P. Leite, Capturing transparency-related requirements patterns through argumentation, 1st International Workshop on Requirements Patterns (RePa), 2011.

[16] S. Easterbrook, 2010. available at http://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation/

[17] E. Sarmiento, M. R. S. Borges, and M. L. M. Campos, Applying an event-based approach for detecting requirements interaction, In ICEIS, 2009.

[18] E. Sarmiento, E. Almentero and J. C. S. P. Leite, Analysis of scenarios with Petri-Net models, In Brazilian Symposium on Software Engineering - SBES, 2015.

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

[20] 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.