A Methodology for Scenario Development

Giuseppe Della Penna, Benedetto Intrigila, Anna Rita Laurenzi, Sergio Orefice
Dipartimento di Informatica, Università di L’Aquila, Italy
{dellapenna, intrigila, laurenzi, orefice}@di.univaq.it

Abstract

The pervasive use of scenarios in the development of computer systems and software has motivated the need of formalisms for the description and manipulation of scenarios. In this paper we propose a scenario–based methodology to support requirements engineering. This methodology enables to exploit the emerging XML technologies in order to offer powerful ways to create, maintain, distribute and use scenarios.

1. Introduction

Usually, there are many sources of requirements, such as customer information, engineering needs, safety constraints, legislation and product safety. The elicited requirements have to be translated in a more implementation–oriented format, becoming the software requirements of the system being implemented.

In this paper we propose a scenario–based methodology, namely SMDP (Scenario Model Development Process), to formalize, manipulate and visualize software requirements. The SMDP methodology consists of five main phases: scenario definition (the inception phase, where software requirements are organized in the form of scenarios), scenario refinement (the elaboration of more detailed scenarios), scenario composition (the composition of different scenarios), scenario transformation (the derivation of other forms of specification from scenarios) and scenario validation (the consistency and completeness checking on the scenarios).

2. Scenario Model Development Process

The SMDP methodology is an iterative and incremental process which consists of the following phases: scenario definition, refinement, composition, transformation and validation.

We assume that user requirements have been previously documented or explicitly elicited from the stakeholders. In this way, the SDML user works on an existing user requirements document in order to formalize the software requirements by applying the methodology.

In the following we give an overview of the single phases, describing the tasks accomplished in each of them and their SDML counterpart. We begin with the first three phases that are grouped in a resulting macro–phase called scenario construction.

2.1. Scenario Construction

Scenario Definition In the scenario definition phase the software requirements are formally organized in the form of scenarios, conceived as concrete sequences of interactions between the user and a system.
The main tasks accomplished during this phase are the identification of the actors, the items and the main scenarios. In the actors and items identification, the actors and items of the scenario, respectively, are extracted from the informal specification of the problem. Actors are all the active agents (human or otherwise) that interact with the system, whereas items are the objects of the system application domain used by the actors to interact with the system. In the main scenarios identification, the set of goals representing the system functionalities are identified. For each goal a main scenario is constructed, which contains a trigger, a set of preconditions, a flow of interactions ending with success and a set of postconditions satisfying that goal.

Using the SDML formalism, all the actors and items are defined in object–oriented structures that formally describe their properties (attributes) and the actions that can be applied on them (methods). All the actors and items that belong to a particular domain are grouped in separate documents and assigned to a namespace. The scenarios can then import one or more namespaces and refer to the actors or items they define. This allows to give a precise semantics to each object referred in the scenario and to share the domain knowledge between different scenarios. The SDML language syntax allows to define actor and item namespaces through the <actorList> and <itemList> elements, respectively.

Figure 1 shows the overall scenario knowledge. The scenario goal is described textually and the interactions are modelled by a main flow containing the scenario knowledge. Such interactions describe the communication between the user and the system through a formal structure in terms of a sequence of actor–action–item, and include the description of the scenario trigger, preconditions and postconditions. Some interactions could be guarded by a particular activating condition.

An interaction is represented using the SDML <interaction> element containing the <actor>, <action> and <item> elements. The scenario goal, trigger and the set of preconditions and postconditions have also a description in the SDML syntax. The goal is represented by the <goal> element, whereas the trigger and the preconditions are described through a set of interactions contained in the <trigger> and <preconditions> elements, respectively. Finally, the success postconditions are also specified as a set of interactions contained in the <success> element at the end of the scenario <mainFlow>.

To support the definition phase, we have developed a graphical editing environment which assists the SDML user in the construction of scenarios. This editor has been implemented in Java using the Xerces and Xalan packages for the manipulation and transformation of XML, and then results to be highly portable among different platforms.

![Figure 2. Main interface of the SDML editor.](image)

Figure 2 shows the main interface of such editor. The editor has a left pane that shows the tree–like structure of the scenarios and of the namespaces contained in the current SDML project. Each element in the tree can be clicked to edit the corresponding data. The right pane of the editor window contains a view of the selected element, which can be of different kinds. Typical views consist of a set of wizard–like dialogs that can be used to fill the various SDML structures, or of a text editor where the generated SDML code can be viewed, modified and checked for validity. In Figure 2 the editor is configured in order to enter the scenario information. After filling this form, the user can begin to write the scenario interactions through the “body” section.

We have also developed a visualization system which has been implemented through XSLT stylesheets generating a set of HTML documents animated with Dynamic HTML. As an example, Figure 3 shows how the basic structure of a scenario is visualized. The visualization is composed of three parts, showing general information on the scenario (title, author, preconditions, etc), all the namespaces used in the scenario and the scenario interaction flow, respectively. The interaction flow is decomposed as a sequence of steps, each one numbered and associated with a textual explanation, and it always terminates with one the keywords SUCCESS, FAILURE or GOTO.
Scenario Refinement In the scenario refinement phase, scenarios are restructured to make them easy to understand and more reusable. Unlike other works [9] where refinement operations do not increase the contents of scenarios, refinement is used in our approach both to add more details to scenarios (through the notion of variant flow) and to abstract common functionalities in order to reuse them in other scenarios (through the notions of inclusions and extensions).

The main tasks accomplished during this phase are the identification of redundant flows, extension flows, failing variant flows and alternative flows. In the identification of redundant flows, sub-flows of interactions common to different scenarios are individuated in order to create new scenarios. These new scenarios will be then recalled in the starting scenarios through appropriate include rules. The aim of this task is to remove possible redundancy in the interaction flow through the modularization provided by the inclusion feature. In the identification of extension flows, some new flows of interactions are added to the scenarios through appropriate extend rules. This information provides further details to the scenarios without changing their post-conditions. As far as the identification of failing variant flows is concerned, the failure of a scenario means, in general, that its interaction flow does not satisfy the goal. This happens when some condition leads to a ramification of the flow that does not terminate with success. The aim of this task is to identify such kind of conditions and produce the corresponding variant flows. Finally, in the identification of alternative variant flows those interaction flows that may be split in groups of alternative equivalent variant flows are individuated. These flows allow to have more execution paths that satisfy the scenario goal.

Figure 4 shows the overall organization of the SDML scenario refinement elements. A subsequence of interactions in the scenario main flow can be refined by splitting it into different sequences of interactions (failing or alternative variants) that are subject to an activating condition, or by introducing an optional sequence of interactions in a particular point of the main flow as a secondary scenario (extension). Moreover, a scenario can be modularized through groups of sub–flows of interactions that are included in the main flow as new scenarios.

The SDML language syntax includes the element <variantFlow> to contain failing or alternative sequences of interactions, and the elements <extend> and <include> for referring to extensions and inclusions, respectively.

Scenario Composition The role of the scenario composition phase is to compose different scenarios in order to show the dependencies and interactions among the corresponding subsystems. In the SMDP methodology the composition is accomplished directly at the scenario level without introducing other formalisms like it often happens in other approaches.

The main task accomplished in this phase is the identification of relations among scenarios. In general, different scenarios can be related through notions like “sequential”, “parallel”, “mutually exclusive”, “repeated”. The aim of this task is to identify all these relationship existing among the involved scenarios and to build a structured composite scenario where they explicitly appear. Thanks to this, it is possible to model features of the system that could not be captured when using a single scenario to describe it.

Figure 5 shows the overall organization of the SDML scenario composition elements. The SDML language syntax provides the notion of simple and composite sce-
Scenario Composition

Scenario Composition

- Sequential Composition
  - <sequence>
  - <parallel>
  - Mutual Exclusion
  - Repetition

Scenario Composition

Scenario Composition

- Sequential Composition
  - <sequence>
  - <parallel>
  - Mutual Exclusion
  - Repetition

Scenario Composition

Scenario Composition

- Sequential Composition
  - <sequence>
  - <parallel>
  - Mutual Exclusion
  - Repetition

Figure 5. SDML scenario composition.

Figure 6. Visualization of the composition in a scenario.

Figure 6 shows how a composite scenario is visualized as a diagram in our visualization system. In the diagram, each component scenario is represented by a box labelled by its identifier. Moreover, whenever a scenario includes other scenarios, these are depicted within the former scenario box. Successive steps of the scenario sequential flow are separated by arrows. Scenarios that must be executed in parallel are visualized on the same row. Depending on the type of parallel composition used, the label parallel or choice appears over the row. Repeated scenarios are instead visualized inside a dashed box having the label repeat.

As said at the beginning of the section, we use the name scenario construction to refer to the first three phases of the SMDP methodology described above. The output of the construction phase is the scenario model, i.e. a set of scenarios which covers the overall user requirements. The scenario model is built through an incremental process where the requirement coverage gradually increases over iterations among the scenario definition, refinement and composition phases. Each incremental step may generate a new main scenario, refine a scenario adding variants flows, inclusions, extensions, or integrate several scenarios into a composite one.

2.2. Scenario Transformation

In the scenario transformation phase, scenarios are translated in different forms of specification which are used in other phases of the software development process, such as validation and testing, and to produce documentation, too.

In general, this task requires a substantial manual effort and needs the introduction of intermediate formalisms. This is avoided in our approach where the formalization of the scenarios allows to directly translate them into target formats.

As an example, in the following let us describe how our methodology addresses the case of the automatic generation of testing artifacts such as test cases. A test case for the specified system contains a sequence of actions that must be performed during the testing session, followed by the expected system response. Moreover, a test case includes a list of preconditions that must be satisfied before its execution. If the system reacts as expected to the sequence of interactions, then the test is successful.

We have developed a test case generation algorithm that takes a SDML document as input and produces in output a set of test cases by applying three main steps. First of all, the external references (i.e. extensions and inclusions) are resolved in order to build a self-contained scenario. Then, the variant tree is visited and each possible control flow is written separately. These flows are associated to a set of true/false values assigned to the conditional steps they contain, which represent the instance of the preconditions for the test case generated from each flow. Finally, the interactions in each flow are analyzed to generate the corresponding test table. Each test case is then completed by adding the preconditions and other information that are directly copied from the SDML document header. The final
test case is formatted using a standard industrial template and it is expressed in HTML fashion to be easily viewed through a common web browser.

### 2.3. Scenario Validation

The primary goal of validation is to confirm the elicited requirements and to detect inconsistency, ambiguity and redundancy. The validation should guarantee the correctness and completeness of the requirement specification respect to the user intentions. In general, the validation is said to be static if it does not require the execution of the specified software artifact on sample input data, dynamic otherwise.

In the SMDP methodology, dynamic checks are accomplished on the target formats generated by the transformation phase, whereas the static verification of some correctness and completeness properties can be directly accomplished on the scenario description.

We have also developed a prototype application written in XSLT based on a meaningful set of quality measures for scenarios, like for example the max depth of the variant tree and the max depth of the failing variant tree. In fact, variants should not be nested too deeply in order to maintain the scenario complexity under an acceptable level.

### 3. Related Work

In the last years, much research has been done in the scenario-based requirements engineering and a number of approaches has been developed. In the following we give a brief synthesis of some techniques for representing and using scenarios that have been developed so far. A wider survey can be found in [6], where our approach is also more precisely illustrated in comparison with the scenario literature.

In the Potts’ et al. approach, [1, 8], scenarios are in textual form following some tabular notations. The requirements engineering process is supported by a hypertext tool in which scenarios and requirements are annotated with requirements discussions, rationales and change requests. Therefore, while inspecting a requirement or a scenario fragment, the user can retrieve, through hypertext links, the open questions, responses and arguments that have been posed on this element and the change requests referring to it as well.

In [4], a concrete style for single scenario representation and a new concept for systematically structuring scenarios and relationships in a set of scenarios are presented. The scenario structure is obtained by combining natural language text with the formal structure of statecharts. Moreover, interaction flow can contain alternatives and iterations.

Sutcliffe et al., [10], define a meta-schema for modeling use cases and scenario based knowledge. The methodology commences by acquisition and modeling of a use case. The use case is then compared with a library of abstract models that represent different application classes, where each model is associated with a set of generic requirements for it class. The authors also provide the CREWS–SAVRE tool to support this methodology.

Leite et al., [7], describe a common scenario construction process and cope with further issues regarding scenario management, in particular the scenario organization. In this approach, scenarios are described in a structured way, using a simple conceptual model together with a form-oriented language.

Hong et al., [5], propose HOONet, a hierarchical object-oriented Petri net, as a method to specify the scenarios and also suggest a technique to integrate scenarios, including different abstraction levels, as well as redundancy, incompleteness and inconsistency.
Only few works in the scenario–based requirements engineering literature exploit the XML technology. Among these, Ralytè, [9], presents an implementation using SGML–HTML to store scenario based approaches in multimedia hypertext documents and illustrates the retrieval of components meeting the requirements of the user by the means of SGMLQL queries.

Duran et al. [3] use XML to represent software requirements and XSLT to support the requirements verification in order to guarantee some quality properties. Nevertheless, scenario representation continues to be semi–formal. For example, the scenario steps are not structured, and the variants miss an activation condition. However, this work is mainly oriented to the implementation and does not aim to the reuse of software specifications.

Not only are scenarios a very relevant research topic, but also they are increasingly adopted in industrial applications. As an example, a survey on the use of scenarios in the software development practice can be found in [11], where fifteen software projects developed using different scenario based approaches are analyzed and compared among them.

4. Concluding remarks

In this paper we presented SMDP, a formal methodology that describes the software specification at various detail levels and allows to reuse the domain knowledge. The SMDP methodology is supported by the underlying SDML formalism, which has been enhanced with a visual editing environment, to create and refine the scenarios, and with a graphical representation system that supports all the SMDP phases and allows to dynamically navigate through the scenario model.

The SMDP methodology has been experimented on a large variety of case studies that have been formalized through the SDML language. For example, some case studies concerning banking or library transactions can be found at the URL http://dellapenna.univaq.it/sdml/examples.asp. These experiments have also shown how the SDML editor makes easier the application of the formalism, balancing the difficulties in the learning and the use of SDML due to the high level of formalization adopted.

It is worth noting that each SMDP phase has an underlying XML schema, where a set of appropriate elements has been defined in order to guarantee the traceability among the various phases. Moreover, a specific <preTraceability> SDML element supports the traceability of software requirements towards user requirements. More details on these elements can be found in [6].

As further research, we plan to enhance the SMDP validation and transformation phases. In fact, we are currently studying more powerful verification rules to validate scenario models and we are investigating how to derive different formalisms, such as statecharts and SRDs (Software Requirements Document), from SDML specifications.

References