Automating Scenario-Driven Structured Requirements Engineering

Hong Zhu
School of Computing and Mathematical Sciences, Oxford Brookes University, Oxford OX3 0BP, UK
Email: hzhu@brookes.ac.uk

Lingzi Jin
Email: lingzi@cherwell.com

Abstract

Scenario analysis is a vehicle of separating concerns in the elicitation of users' requirements. It is also a means of requirements validation and verification. In the practical application of the method, a number of key activities must be supported by automated tools, which include: (a) the analysis of consistency among scenarios, (b) the analysis of the consistency and completeness of a set of scenarios with respect to requirements models, (c) the synthesis of requirements models from a set of scenarios, and (d) the generation of scenarios from requirements models for requirements validation. This paper presents an automatic toolkit that supports these activities. It adapts the idea of scenario analysis originally proposed in OO analysis to the structured analysis method.

1. Introduction

Requirements analysis and specification are concerned with eliciting, clarifying, documenting and validating users' requirements of computation systems [1]. Many studies have shown that errors made at this stage are very costly, even impossible, to rectify. Neglected or only partially completed requirements analysis tends to lead to problems later in development. It is perceived as an area of growing importance. In the analysis and specification of users' requirements, software engineers are often confronted with difficulties due to the complexity of the problem, the communication barriers between people of diverse backgrounds, the inconsistency and incompleteness of information and frequent changes of users' requirements. A number of proposals have been advanced in the literature to overcome these difficulties. These include the deployment of various vehicles for separation of concerns, the design and use of various kinds of representations and notations for the description of users' requirements, and the development of software tools to support requirements engineering process. In this paper, we report our research on an automated toolkit RASS for requirements engineering.

1.1 The POTOR process model

We regard requirements engineering as a process that starts with informal statements of users' requirements and targets at producing a requirements specification in a formal notation. In this course, software engineers not only need to formalise the information, but more importantly to elicit the necessary information from the users and domain experts. To bridge the vast gap between the two ends, we proposed the POTOR (Progressive and Orderly Transition of Representation) process model of requirement engineering [2, 3, 4]. It divides the process into a number of iterating phases. Each phase aims at increasing the formality of representation by a small step with the help of users' interactions and supported by automated tools. The process includes the following typical phases.

- **Initial statement**: an initial statement of user's requirements is produced, which is usually in a natural language.
- **Requirements elicitation**: users' requirements are elicited and described, say, in the form of scenarios and use cases. In this paper, we use the word 'scenario' as a synonym of 'use case'.
- **Model construction**: models of users' requirements are constructed, analysed and validated, where a model should be in the form of diagrams and other formally defined notations, but may contain terminology informally defined, even undefined.
- **Knowledge formalisation**: the terminology used in the models are formally defined and specified in the first order logic or other mathematical notations in this phase.
- **Formal specification**: formal requirements specifications are produced according to the models of users' requirements and formally defined system specific knowledge and domain knowledge.

As the formality of representation increases, more and more formal checking of consistency and completeness can be performed, which may result in backtracking to earlier phases and iteration.
1.2 Requirements Definition Language RDL-2

The requirements definition language provided notational links between different stages in requirements engineering. The RDL-2 language and its earlier version NDRDL [2] are based on the classic methods of structured analysis, see e.g. [5]. The core part of RDL-2 consists of three diagrams commonly used in structured analysis, which include an entity-relationship diagram (ERD), a data flow diagram (DFD) and a state transition diagram (STD). These diagrams are the focus of model construction stage. To link from model construction to knowledge formalisation, we extended the dictionary to include formal definitions of glossaries. To link requirements elicitation to model construction, we extended the diagram notation to support the description of scenarios, see section 2.2. A set of scenarios can be included in a requirements definition as appendix.

1.3 The RASS system

In addition to the facilities that are provided by existing CASE tools, we are more concerned with the following activities in requirement engineering:

- the automatic transformation of information from one representation into another, e.g., from scenarios into requirements models, from requirements models into formal specifications;
- the automatic analysis of the properties of the information and its representations, such as the consistency and completeness of information at each phase;
- the validation and verification of the consistency between different views and across different representations.

The RASS system consists of a set of editors, a set of information management subsystems, a set of automatic consistence and completeness checkers, and a set of automatic synthesisers and generators. These tools are invoked through menu items in a tool bar in graphic user interface. They can be grouped into a number of subsystems to provide support to various phases and activities in requirements analysis and specification. This paper only discusses the part related to scenario analysis. Readers are referred to [6] for more details of the system and [2–4] for its earlier version NDRASS.

2. Scenario Analysis

Scenarios and use cases are vehicles for separation of concerns, e.g. [7, 8]. They are concerned with situations that might reasonably occur in the operations of a system. In recent years, the role of scenarios in requirements elicitation, modelling and analysis has been widely recognised [9], but mainly studied in the context of object-oriented methods [10]. By identifying a set of scenarios, i.e. a set of situations of common characteristics, and working on each scenario separately to elicit information, a complicated problem can be decomposed systematically and naturally. Information from different sources can be elicited independently. A requirements definition can be synthesised from the descriptions of the scenarios. Secondly, by analysing the consistency between requirements of different scenarios, conflicts between different types of users, different use purposes and different operation conditions can be discovered. Moreover, scenario analysis also provides a basis for requirements validation. By analysing the consistence and completeness of requirements definition with respect to a set of well-defined scenarios, the requirements definition can be validated and verified. However, for large software systems, there might be a large number of scenarios. Dealing with a large number of scenarios demands tool support.

2.1 The process of scenario analysis

A process model of scenario analysis is depicted in Figure 1. It consists of the following interactive and iterating activities.

- Identification of scenarios
- Elicitation of information and description of scenarios
- Analysis of the consistence and completeness of a set of scenarios
- Synthesis of requirements models from a set of scenarios
- Validation of requirements models using scenarios

![Figure 1. A process model of scenario-driven RE](image)
2.2 Description of scenarios in RDL-2

An appropriate representation of scenarios should satisfy at least two basic requirements. Firstly, it should be simple and easy for software engineers and users to understand. Secondly, it should facilitate automatic analysis of scenarios as well as the construction of requirements models. Although various representation forms have been proposed in the literature, unfortunately, few satisfy both of the requirements. For example, in the HCI area, scenarios often appear in the form of activity lists, which are lists of sentences in natural languages [11]. In [12], a tabular notation is used. In [13], the use of structured natural language is proposed. These informal and tabular representations are simple and easy to use, but do not facilitate formal analysis. In [14], use cases were first described in a tabular form, called partial use case table, and organised into a tree to form a complete use case. Use cases were further linked together by temporal relations. Then, such descriptions were transformed into Petri nets manually to facilitate formal analysis. Although such formal representations facilitate formal analysis, they are difficult for software engineers and users to use.

Another very important requirement for scenario representation is that it should facilitate the integration of scenario-driven requirements analysis into other well established requirements engineering methodologies such as multiple views and structured analysis. The notion of scenario has been incorporated in the study of multiple views either explicitly or implicitly. In [14], a use case is described in the form of trees of tables as domain expert-centred viewpoint and equivalently in Petri net as analyst-centred viewpoint, while in [15], a view also has the characteristic of specific user agents, use purpose and operation condition.

In RDL-2 [16], a scenario description contains three views. These views are extensions of ERD, DFD, and STD. In the design of RDL-2, we recognised some special requirements for scenario description that are not satisfied by the existing models. Among the most important facilities that existing requirements models lack are the facilities to describe information that is vague or unknown in a scenario and the facilities to describe the relationships between scenarios.

Due to the restriction of a scenario on a particular use purpose of the system and the limitation of the users involved in the scenario, the description of a scenario may need to describe vague information about the system's functional, behavioural and other requirements. Such information could be unknown for its detail to the people involved in the elicitation of the requirements, or even not directly relevant to the scenario, hence should be excluded. However, that information should be elicited and described in other parts of the system through other scenarios. For instance, consider a banking system and the scenario that a customer deposits into an account by presenting a cheque. From a customer's point of view, such a scenario consists of the process of filling a pay-in slip, presenting the cheque and pay-in slip to the cashier, and then somehow the correct amount of money will be added to his account within a certain period of time. However, how the money comes to his account is unknown to the customer. In fact, the customer is probably not interested in the detailed process after the paying in activity as long as the correct amount of money is added into the account within an acceptable period of time. Without a facility to describe such situations, one either has to describe the detailed process not directly related to the scenario, or results in a collection of unrelated pieces of information that may not make sense and could hardly be considered as a partial model. The description of a scenario should therefore not just simply delete irrelevant information from a requirements model. In RDL-2, we introduced the notion of virtual data flow into the data flow description of scenarios and the notion of virtual state transition into the state transition description of scenarios as such a facility. Virtual data flows and virtual state transition are depicted as dashed arrows in diagrams. Figure 2 gives the data flow and state transition descriptions of the scenario of paying-in-by-cheque.

In an extended data flow diagram, a dashed arrow means that the need of a data flow path in the whole system is recognised but its details are not clear or not relevant in this scenario. Therefore, it represents a path in the data flow model of the whole system. Similarly, a dashed arrow in an extended state transition diagram means that the need of a state transition or a sequence of state transitions in the whole system is recognised and represents a path in the state transition model of the whole system. The introduction of virtual data flow and virtual state transition are extensions of ERD, DFD, and STD. In the design of RDL-2, we introduced the notion of virtual data flow into the data flow description of scenarios and the notion of virtual state transition into the state transition description of scenarios as such a facility.
state transition, thus, also provides a facility of describing the relationships between scenarios. For example, a scenario describing how a cheque is processed would relate to the scenario of paying-in-by-cheque through the virtual state transition and virtual data flow in Figure 2. Such relationships cannot be described in other existing scenario representations, although UML’s use case diagram can specify the 'include' and 'extend' relationships between use cases.

2.3 The notions of consistency and completeness

There are two notions of consistency related to scenarios. One is the consistency of a scenario with respect to a requirements model. The other is the consistency among a set of scenarios. These notions of consistency have their own uses in requirements analysis. Both of them can be automatically analysed by RASS tools.

The consistency between a scenario and a requirements model is a property that the information contained in the scenario does not conflict with the requirements model. Since a scenario is essentially a special case of the requirements model, consistency means the information contained in a scenario is a subset of the information contained in the model. This property can be formally defined and automatically analysed by RASS tools.

This notion of consistency enables us to automatically validate a requirements model against a set of well-defined scenarios. If a requirements model is consistent with respect to a set of scenario descriptions, then the requirements model is valid with respect to the scenarios. Otherwise, errors are found in the requirements models or the scenarios. Such validation is proved effective in our case studies of the RASS tool kit. It is particularly useful when the user changes the requirements by modifying a description of a scenario. Automatic analysis of consistency between the modified scenario and the existing requirements models enables software engineers to identify the required modifications on the models. Hence, the impact of the modification can be identified.

A set of scenarios is consistent if the scenarios do not contain conflict information among themselves. This notion of consistency can be formally defined by using the notion of consistency between a scenario and a requirements model, because a set of scenarios is consistent if and only if there is a requirements model that all scenarios in the set are consistent with respect to the model. However, this property can be automatically analysed without constructing the requirements model first. Checking the consistency among a set of scenarios is particular useful during requirements elicitation stage when users express their requirements through scenarios. Conflict requirements can be identified and resolved at very early stage without any effort of constructing a requirements model.

When a requirements model is validated against a set of well-defined scenarios, the question arises of how adequate such a validation is. This leads to the notion of completeness of a set of scenarios with respect to a requirements model. It means that all the information contained in the model is covered by at least one scenario. Therefore, checking the consistency against the set of scenarios covers all the information contained in the model. In this sense, the completeness of a set of scenarios with respect to a requirements model means the adequacy of validation. On the other hand, checking such completeness also means checking if a requirements model contains information not contained in the scenarios. Assuming that a set of scenarios is complete in the sense that it already contains all the information of the required system, incompleteness then means that the model must contain unnecessary or incorrect information. Therefore, checking completeness is also an important part of the validation of requirements models. This notion of completeness can be formally defined and it is automatically checked by RASS.

Given a set of descriptions of scenarios, to synthesise a requirements model we require that the result should include all the information contained in the scenarios, but contain no information that cannot be inferred from the scenarios. The result model should also be consistent with respect to all of the scenarios. It was formally proved that from any consistent set of entity-relationship diagrams, we can always synthesise an entity-relationship model. However, due to the existence of virtual flows, consistency is not sufficient for synthesising a data flow model, nor a state transition model, from a set of scenario descriptions. This is caused by the possibility of the incompleteness of the information contained in the scenario descriptions. For example, a data flow scenario description contains a virtual flow, but there no scenario in the whole set contains further information about the data flow. Obviously, the information is incomplete. This leads to the second notion of completeness, i.e. the self-completeness of a set of scenarios. This notion of completeness was also formally defined. It was proved that a data flow model \( M \) can be uniquely synthesised from a set of scenarios that is consistent and complete with respect to \( M \), if and only if the set of scenarios is consistent and self-complete. A similar theorem holds for state transition diagrams.

3. Requirements validation

Existing requirements validation methods fall into two classes, static testing and prototyping. Static testing is concerned with the review of requirements documents. Typical static testing methods include checklists guided inspection [17] and structured walk-through in formal review [18]. They have been widely used in practice and claimed to be effective to detect errors. Prototyping involves developing and demonstrating prototype software to show if the software satisfies user's requirements. It is effective for validating user interface and certain specific
features of the system, such as timing and scheduling aspects in real-time applications [19].

In comparison with the methods for testing other software artefacts such as executable code [20], the methods for testing requirements are less systematic, less cost effective and less easy to apply. A common problem is that requirements documents are difficult to grasp, especially without tool support [21]. During the process of testing, the testers have to translate, in their mind, the static description of the requirements so as to visualise the dynamic behaviour of the described system. Such translations often have to co-ordinate information scattered over the requirements documents, especially when the system is described by multiple views [18]. This process is further complicated due to the difficulty of casting two-dimensional diagrammatic descriptions into one-dimensional, temporal sequences of events. Not only is this process time consuming and expensive, but also its effectiveness depends on the experience and training of the human tester. There is no well-established method to specify, measure or control the strictness of such tests.

The potential application of scenario analysis in requirements validation has been recognised by a number of researchers [16, 22]. The basic idea is to check requirements in various scenarios or use cases that may occur in the operation of the software. As discussed in section 2, RASS provides automatic checks for the consistency between a scenario description and requirements models. However, such consistency checks have some weakness. For example, to apply the method, there must be a set of scenarios that thoroughly covers the requirements models. The problem is how to chose scenarios and where the descriptions of scenarios come from. Obviously, use of the scenarios obtained during requirements acquisition is not an ideal solution especially when the requirements models are automatically generated from the scenarios, because validation should be independent of the production of requirements models.

As a complementary to the above approach, we proposed and implemented a second approach to software requirements validation using scenarios [23, 24]. The basic idea was to select a set of test scenarios automatically and systematically according to the requirements models, and then, to generate readable descriptions of system's behaviour in each scenario for validation review. The following discusses two key issues related to this approach of requirements validation, i.e. how to select test scenarios, and how to generate detailed descriptions of system's behaviour for a given scenario.

The method is inspired by task analysis techniques developed in HCI [11]. The key idea of the proposed approach is to test the correctness of a requirements definition by analysing the behaviour of the specified system on a set of task scenarios. As all scenarios, a task scenario represents a set of situations that may occur in the use of the software system. However, as a test case, such task scenarios should be simple and can be clearly described so that system's behaviour can be effectively analysed and validated.

The result of such behaviour analysis is an activity list, which is a notion borrowed from HCI task analysis techniques but extended to describe system's behaviour. An activity list is a linear sequence of the events that happen inside the system in temporal order. In the design of the requirements validation tools, we identified the following aspects of software behaviour as the most important to be analysed during requirements validation:

- how information is exchanged between components of the software and between such components and the agents in the system’s environment;
- what sequence of computations are performed by each component;
- how the software reacts to stimuli from its environment;
- how the internal states of the software are updated and effect the behaviour of the system.

Consequently, there are four types of actions used in the description of system's behaviour: (a) receiving or sending information from one agent to another; (b) obtaining information from internal state of the software; (c) updating the internal state of the software; (d) performing computation by a software component.

An activity list is presented in the form of a sequence of simple sentences in structured natural language. For example, the following sentence describes an activity.

Process P sends data \( \mathbf{x} \) of type real to the terminator display where \( \mathbf{x} > 0 \).

One of the most important advantages of activity lists is that a sentence in activity list combines relevant information originally scattered over the requirements definition documents in various diagrams and dictionaries. Information irrelevant to the scenario is filtered out. Therefore, when validating the behaviour of a specified system by reviewing the activity lists, the tester need not to search for all the information scattered over various diagrams and dictionaries, and to translate two-dimensional diagrams into temporal sequences of events. It is this characteristic of activity lists that differs from existing representations of scenarios such as diagrams e.g. [10, 8] and formal notations. It thus offers a more testable description of system's behaviour so that formal review can be performed more effectively and efficiently [23].

As all other testing methods, the effectiveness of testing depends on the selection of test scenarios. We developed three types of selection criteria. Data flow testing methods select test scenario according to the data flow model of requirements definition. State transition testing methods are based on the state transition model. Entity testing methods are concerned with the entity relationship model.
These adequacy criteria fall into a hierarchical structure of subsume relations, see [24] for details.

The current version of the RASS has implemented a test scenario generation tool that automatically generates various sets of test scenarios according to the state transition testing criteria. It also implemented a test adequacy measurement tool that measures test adequacy according to a subset of data flow test criteria. The description of system's behaviour in each test scenario in the form of activity list can be display on screen for the tester to validate the requirements.

A number of empirical studies of the tool shown that the numbers of task scenarios and the lengths of activity lists are acceptable for the practical use of the testing method. Details of the empirical studies can be found in [24].

4. Conclusion

Our experiments with automated toolkit for requirements engineering indicates that a significant degree of automation can be achieved at requirements stage. Automatic tools can provide software engineers power to cope with difficulties in requirements elicitation, analysis and specification. We are planning further case studies of the toolkit in industry environment.

Acknowledgement

The authors are grateful to those who participated in the implementation and case studies of the tool, especially Q. Huo, C. Zhang, G. Bai. The authors would also like to thank Prof. J. Xu of Nanjing University, Dr. Bob Champion of Oxford Brookes University, Prof. Dan Diaper and Prof. Martin Shepperd of Bournemouth University for their valuable comments on an earlier version of the paper and discussions on many related issues.

References


