A Framework for Defining and Checking Constraints in Requirement Gathering and Definition

Rodolfo Gómez and Pablo Fillottrani
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur, Bahía Blanca, Argentina
[rgomez, prf]@cs.uns.edu.ar

1 Introduction

Requirements capture user needs about a system [KG99]. They stand for those functional and nonfunctional attributes the system must posses to be considered correct. Requirements also dictate how the system should respond to user interaction. Therefore, they play a very important role in software development. They state what the system should do for the user to be satisfied, and therefore they lead all development stages in the lifecycle. They are used to communicate user needs to all people involved in the development process, the system architecture is built upon them and the testing stage uses them as satisfaction criteria.

Requirement gathering is the activity of bringing requirements together, *requirement definition* is the activity of documenting requirements and organizing them into something understandable and meaningful, i.e. a *requirement specification*. Unfortunatly these tasks are quite difficult. Users usually cannot precise what they want from the system. Usually there are more than one user of the system and many conflictive views appear about what the system should do. Frequently, even if the users agree in system functionality, they are not able to see the "big picture" and the specification is incomplete. It is also possible for requirements to be stated by someone who's not the real user of the system, for example company managers. This may results in requirements that express interactions that do not add real value to users.

Another problem with requirements is that they evolve while users are learning about their needs. Hence, requirements suffer a lot of changes until they can be precisely defined. Also, requirement gathering itself can be a difficult task if users do not have enough time to be properly interviewed, or they are reluctant to cooperate in this process. Finally, requirement specification itself may be cumbersome as it often has no more structure than a simple list. Because this list tries to capture every requirement, including those which add no value to the user real needs, its volume become unmanageable by subsequent development stages (e.g. design and testing).

All difficulties just mentioned result in a poor quality requirement specification, with redundant, inconsistent and even incomplete requirements. Recently, UML's use cases and use diagrams has been adopted as a succesfull methodology in requirement gathering and definition [KG99]. They have helped to solve the problems we have previously mentioned and are still

useful to drive the entire software development process [IJR99b].

Despite the many advantages use cases offer in requirement definition, we feel this process can be improved by letting requirement redundancy, inconsistence and incompleteness to be automatically controlled as much as possible. The proposed framework should benefit from use-cases structure for discovering patterns expressing redundancy, inconsistence and incompleteness. Also, the framework will be a tool for analysts as it will allow the evolution of requirement definition to be automatically controlled and verified.

2 Use cases in Requirement Gathering and Definition

Use cases [IJR99a] are text descriptions of the interactions between some outside actors and the computer system. An actor represents a coherent set of roles that users of use cases play when interacting with these use cases. Typically, an actor represents a role that a human, a hardware device, or even another system plays with a system. Use case diagrams are graphical depictions of the relationships between actors and use cases and between two use cases.

Use cases play an important role in requirement gathering and definition. The interactions that use cases illustrate form the basis of most of the requirements that must be documented. They are effective communication vehicles between analysts and users. Generally, functional requirements can be put into terms of interaction between the actors and the application. Non-functional requirements, such as performance, extensibility and maintainability, can often be stated in terms of use case stereotypes [IJR99a]. Use cases also provides requirement traceability through the lifecycle because they are a building block for system design, units of work, construction iterations, test cases and delivery stages. Finally, use cases discourage premature design because these faults become quite evident when appearing into the use case structure.

Briefly, a use-case driven approach to requirement gathering proceeds as follows [KG99]. Use cases go through a series of refinement iterations until they are complete. Those that model the most valuable functions for the user are identified first. These cases are usually incomplete as they only express brief descriptions of particular interactions. As they are refined, they are added basic course of events, alternative course of events, scenarios, preconditions, postconditions, etc. Last steps of refinement have to do with use case diagrams as a whole, where opportunities for merging use cases with similar functionality, reusing and generalization are looked for.

3 Improving requirement gathering and definition

Requirement gathering still relies on the analyst abilities for discovering inconsistences, redundancy and incompleteness in requirements. Of course, all of these problems have become much more tractable by use cases, but we think there are some ways in which the process can be improved. The goal of our research is to define a framework based on use cases which allow the requirement specification evolution to be automatically controlled and verified.

Use cases become more detailed through refinement iterations. The next iteration completes or corrects the previous one. However, it would be useful for the analyst to define controls to be

carried on while the use case evolves.

Advantages appear when the analyst wishes a given use case diagram semantics to be an invariant with respect to subsequent iterations. These invariants include, for example, actors e.g., in the form of *"this actor cannot perform operation A before operation B has be completed"*, use case relationships e.g., in the form of *"actors can never interact with use case A before they have finished with use case B"*, politics (or business rules as called in [KG99]) e.g., in the form of *"use case diagrams should never contain more than 5 actors and 10 use cases, otherwise they must be decomposed"* and use case structure e.g., in the form of *"all preconditions stated for a use case must be provided as postconditions of other previously-performed use cases"*. Other benefits become clear when analysts postpone some details for future iterations, but they know enough about these details to be documented in the current iteration. For example, the analyst could adorn the use case diagram with a comment expressing that a new actor should be added in the next iteration.

We are currently developing a framework that help analysts to automatically define and maintain the sort of controls they could be interested on during the requirement gathering and definition activities. Roughly, we can compare this framework with a CASE tool in the sense that it will allow the visualization, edition and maintenance of requirement gathering deliverables such as use cases and use case diagrams. Beside that, the focus of our research is concerned with other, paramount features. We think this framework should provide a language which has to be expressive enough to define useful controls while at the same time be capable of automatic verification. For instance, a possible research line may consider executable logics as definition languages and model checkers as verification procedures (see e.g. [BBC+99] and [Hol97]).

As an overview of the framework possibilities, consider the following. Firstly, the framework could provide controls over invariants, alerting the analyst if they are not met in new iterations. For example, a previously defined actor could interact with a different use case by adopting a semantics that is inconsistent with that it was originally defined for. Secondly, the framework could remind the analyst about postponed details, such as refining a new actor in the next use case iteration. Finally, it could identify opportunities for reducing redundancy in requirements by finding flows of events which are common to various use cases. Therefore, it would be possible for the analysts to merge the cases or derive a new use case with common functionality.

We are warned about the existent of related work in the area of formal specifications. However, these works seem not to be at the level of abstraction which is present in requirement gathering. For example, RSL, the formal specification language of RAISE (Rigorous Approach to Industrial Software Engineering) [GRO92], [GRO96] is useful for modeling design specifications, i.e. a contract between the system architect and the implementors or for modeling component specifications, i.e. a contract between the user of the program module and the module developer. The major difference between the requirement gathering process our research is concerned about and the process adressed by RSL has to do with inputs. Requirements adressed by use cases come directly from users, so they are much more general, abstract and unestructured than those adressed by RSL, where refinement activities such as analysis and design have transformed these requirements in documents which are more structured, specific and complete. Indeed, we expect our framework to be capable of managing incomplete specifications.

Other work that is worthy to be mentioned is OCL (Object Constraint Language) [WK99]. This language is used to define constraints over both elements of UML models and elements

of the UML metamodel. We still have to investigate if this language can be directly used or otherwise be extended as a mean to specify those controls we are concerned about in requirement gathering. For example, it is possible that preconditions and postconditions for use cases could be specified in OCL to a certain extent, but it is likely OCL does not provide the means for managing common flows of events among use cases.

Our research, then, is currently focused on discovering what kind of controls are useful for the analyst when designing use cases. Then we have find to how these controls are related to use case diagrams. Once these steps are finished, it will be possible to define the expressiveness required for defining and maintaining the controls.

References

[BBC+99] N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, B. Sipma, and T. Uribe. Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. *Formal Methods in System Design*, 1999.

[GRO92] GROUP, T. R. M. *The RAISE Specification Language*. Prentice-Hall, 1992.

[GRO96] GROUP, T. R. M. *The RAISE Development Method*. Prentice-Hall, 1996.

[Hol97] Gerard Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[IJR99a] Grady Booch Ivar Jacobson and James Rumbaugh. *The Unified Modeling Language User Guide*. ACM Press, Addison-Wesley, 1999.

[IJR99b] Grady Booch Ivar Jacobson and James Rumbaugh. *The Unified Software Development Process*. ACM Press, Addison-Wesley, 1999.

[KG99] Daryl Kulak and Eamonn Guiney. *Use Cases, Requirements in Context*. ACM Press, Addison-Wesley, 1999.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.