Use Case Driven Extension of ProjectIT-RSL to Support Behavioral Concerns

David de Almeida Ferreira, Alberto Rodrigues da Silva
INESC-ID / Instituto Superior Técnico
Rua Alves Redol 9, 1000-029 Lisboa, Portugal
david.ferreira@inesc-id.pt, alberto.silva@acm.org

***Abstract*—Requirements Engineering is a system engineering discipline of paramount importance. Its primary deliverable is the requirements specification, a document that entails the detailed description of business-specific needs to which the target software system must comply to. Despite the advances brought by modeling techniques, the specification of software systems still consists mostly in manually writing down requirements in natural language. Resorting to natural language to convey requirements has some advantages, such as its expressiveness and the stakeholders proficiency at using it for communication purposes, when compared with semi-formal or formal languages. However, ad-hoc natural language has intrinsic characteristics that make it error-prone, some of which may even hinder the effort of complying with good requirement criteria, regardless the usage of an automated requirements tools. To address this problem, within the scope of the ProjectIT initiative, a requirements specification language named ProjectIT-RSL was developed, based on common linguistic patterns. Despite covering several structural aspects of the software system under specification (e.g., actors, components, entities, properties, and relationships), ProjectIT-RSL still lacks the support for capturing behavioral concerns.**

This paper presents a Use Case Driven extension for ProjectIT-RSL , with the purpose of supporting the specification of functional requirements according to a controlled natural language approach. The proposed solution combines patterns from Use Case textual description best practices with Pseudocode specifications.

***Keywords*-Requirements Engineering; Specification Language; Controlled Natural Language; Use Cases; Pseudocode.**

I. Introduction

The size and complexity of modern large-scale software systems demand for high-levels of abstraction, conceptual reasoning, and validation upfront. Namely, a clear definition of the real business-specific needs is required. Therefore, regardless of the achievements in both industry and academia, the development of software systems is still a rather challenging process. From all system engineering disciplines, it is argued that Requirements Engineering (RE) is one of the most important regarding this issue, since it supports the work of several other disciplines throughout the entire software product life-cycle [1]: it provides useful scope and status information to Project Management upstream and provides a stable basis for development downstream, namely for design and testing.

Typically, the development of a software system begins with Requirements Engineering (RE) [2]. RE deals with the early activities related with the process of discovering the purpose, scope, stakeholders, boundaries, and actors of the system being specified [3]. These early activities can be classified as the *requirements development process*, whereas the activities of evolving accepted requirements (e.g., dealing with change requests, impact analysis, tracing, and status-tracking) can be regarded as the *requirements management process* [4]. In short, RE is concerned with real-world goals for software systems functionality, and also how it can be precisely specified and maintained throughout the software development process [5].

However, often the importance of RE is underestimated, resulting in a large amount of rework: any attempt to start technical work beforehand, without a deep understanding of the target software system's purpose, will certainly jeopardize the project outcome [6].

The main deliverable of RE is an artifact (usually called *requirements document*) that contains the detailed textual description of *what the target system should do*, or *constraints on its behavior* [7]. However, this form of specification (based on natural language) is ambiguous and, in many cases, unverifiable because of the lack of a standard machine-executable representation [7]. The main problem appears during the translation of natural language requirements into a formal or, at least, semi-formal computer model [8]. Despite these complementary approaches, natural language textual specifications are still the most suitable, fast, and preferred manner (by non-technical stakeholders) to contribute and validate requirements specifications [7].

Taking these facts into consideration, within the scope of the ProjectIT initiative [9], [10], a requirements specification language named ProjectIT-RSL [11] was developed. This initiative seeks to address some of the aforementioned problems, since it advocates that the software development process should be focused on higher-level activities (such as Project Management, Requirements Engineering, and Architectural Design), instead of repetitive and error-prone manual activities. Following this mindset, ProjectIT-RSL was designed to increase the rigor of requirements specifications to enable tool support, namely to provide user feedback

This work is supported by FCT (INESC-ID multi-annual funding and PhD Scholarship SFRH/BD/36972/2007) through the PIDDAC Program.

during the specification process and the automatic generation of initial design drafts upon validated requirements.

Despite the concrete results achieved, there are still several open issues, namely the lack of support for the definition of software system behavior. To cope with real-world problems, the controlled natural language approach provided by ProjectIT-RSL needs to cover not only static aspects of the target system (e.g., *domain model* concepts) but also dynamic aspects related with functional requirements.

The remainder of this paper is organized as follows. Section II presents a review of related work regarding the specification of software system behavior. Section III presents the Use Case Driven extension of ProjectIT-RSL and introduce an illustrative example. Finally, Section IV concludes this paper with final thoughts and discussion.

II. Related Work

Historically, the requirements specification process has consisted in creating a natural language description of *what the target system should do* or *constraints about its behavior* [4], [7]. As opposed to constructed, artificial, or any other kind of engineered languages (with specific purposes in mind), the term *natural language* refers to any language commonly used by humans in their daily communication. Its usage to support RE activities has already been studied for a long time [8]. However, this form of specification is both ambiguous and, in many cases, unverifiable because of the lack of a standard machine-executable representation [7].

To mitigate the undesirable effects of natural language, one can impose some restrictions to reduce its complexity while maintaining its naturalness. *Controlled Natural Languages* (CNL) are subsets of natural languages whose grammars and vocabularies have been engineered in order to reduce (or even eliminate) both ambiguity (to improve computational processing) and complexity (to improve readability). CNLs are typically designed for knowledge engineering, not addressing behavioral aspects. A thorough analysis on the state-of-the-art of CNLs is provided in [12]. Attempto Controlled English (ACE) [13] is a good representative of this category of textual languages, because it is a mature general-purpose CNL with a solid toolset and a wide literature body of knowledge[1]. Furthermore, ACE possesses some interesting (and useful) characteristics, namely every ACE sentence is unambiguous, even if people may perceive the sentence as ambiguous in full English [14].

To address functional aspects, namely to specify complex behavior (implementation-independent algorithms and other abstract computations), one can resort to *Pseudocode* [15]. The high-level algorithmic descriptions achieved with Pseudocode could be integrated with the previous CNL approaches to address their weaknesses in terms of behavioral expressivity. However, Pseudocode lacks a widely accepted and well-defined standard (only some guidelines are defined [16]). We considered that *PlusCal* [17], [18] is a suitable representative for this approach. Despite having formal roots, PlusCal was designed having in mind some of the Pseudocode's guidelines (e.g., simplicity), which in turn reflects on *PlusCal*'s design principles and syntax. PlusCal is an algorithm language that can be used to replace Pseudocode, for both sequential and concurrent algorithms [17]. Specifications in PlusCal can be automatically translated into another formal language enabling the verification of formal properties through model-checking.

Additionally, one can resort to semi-formal *graphical* approaches. This category of languages is well represented by *UML*, the general-purpose language endorsed by OMG and considered by many as the *de facto* standard for modeling and documenting object-oriented software. Usually, modelers use Activity and Sequence UML diagrams to specify software system behavior. However, the information conveyed by UML models is usually incomplete, informal, imprecise, and sometimes even inconsistent. These flaws are caused by the diagrams' limitations. These diagrams cannot fully convey the details of a thorough specification.

Also, there are hybrid approaches such as Use Cases: UML Use Case diagrams can be complemented with their textual descriptions counterpart. Use Cases provide a natural approach to organize and describe the overall system functionality through descriptions, in abstract terms, of how actors interact with the system to accomplish their goals [19]. Although they are not suitable to describe functional requirements of all kinds of systems, they are adequate to specify general information systems [19]. Also, since they describe the system functionality from an actor's perspective, they can be easier to read and understand by non-technical stakeholders than more formal or complex notations (formal methods or UML diagrams enhanced with OCL). However, although it might seem a trivial task, the relatively small number of concepts, allied with the lack of a rigid set of structuring rules (or compositional rules, when dealing with Use Case diagrams), makes people often struggle to write (or model) useful and adequate Use Cases.

Finally, there are some combined approaches. Natural Language Processing (NLP) approaches to automatically derive conceptual models from requirements texts are not new [8], and neither is the usage of formal methods to support consistency checking through mapping techniques [20]. Debnath et al. [21] propose a solution to integrate requirements written in natural language into the OMG's Model-Driven Architecture. They defined manual derivation strategies to obtain conceptual models and formal specifications from textual requirements. Their approach follows an OCL-based transformation process to define Computer Independent Model (CIM) models from natural language oriented models. Mala et al. [22], [23] present research regarding Use Cases combined with NLP techniques. They

[1] http://attempto.ifi.uzh.ch/site/pubs/

present an approach for the automatic construction of object--oriented design models in UML from the natural language requirements specification. Unlike others, their approach combines domain modeling, Use Case descriptions (written in a restricted form of natural language), and non-functional taxonomy to derive non-functional requirements prioritization through trade-off analysis. Konrad et al. [24] also attempts to ease the integration of formal specification languages through the process of specifying properties of formal system models in terms of natural language and formally analyzing these properties using existing formal analysis tools. Still regarding natural language approaches, the work of Meziane et al. [8] follows a different approach to mitigate miscommunication problems by generating natural language specifications from UML class diagrams.

Despite this wide range of techniques and approaches, many problems concerning the pragmatic usage of these solutions are still unresolved [25].

III. ProjectIT-RSL Extension

ProjectIT-RSL is a controlled natural language aligned with a metamodel derived from common linguistic patterns [11]. This language was designed to convey information in such manner that a tool can extract requirements models from its textual specifications. ProjectIT-RSL metamodel already provides concepts (e.g., Operation, Activity, and Action) to convey information about software system behavior. However, until now, the concrete realization of the dynamic part of the metamodel in terms of language constructs has not been addressed [10]. Thus, ProjectIT-RSL lacks the means to convey behavioral concerns about the software system under consideration. The specification of complex interaction patterns between actors and the target system is crucial for gaining a thorough understanding of the software system's functionality, namely to perform a more extensive analysis (e.g., consistency checking and model-checking) on requirements specifications written in this controlled natural language. This kind of feedback is crucial to increase stakeholder engagement in RE activities related with verification and validation.

According to the object-oriented paradigm, so far the conceptualizations that ProjectIT-RSL conveys are typically referred to as a *domain model*. This kind of conceptualization describes the various entities, their properties and relations (an example is provided in [26]). The domain model provides a structural view of the domain of interest, which can be complemented by other views focusing dynamic aspects, such as the one presented in this paper.

Considering the original mindset and design principles of ProjectIT-RSL [11], namely the grammar and writing style, we follow the seminal results presented in [27]. Hence, our proposal extends the previous version of ProjectIT-RSL by combining Use Case textual descriptions with Pseudocode.

A. Reusing and Structuring with Use Cases

Besides the metamodel previously mentioned, ProjectIT--RSL specifications also follow another metamodel (depicted in Figure 1) with different concerns. The former is related with the specifications' content, whereas the latter addresses the best practices regarding requirements document's structure. Our proposal in this paper focuses on providing an extension for the *Functional Requirements* section.

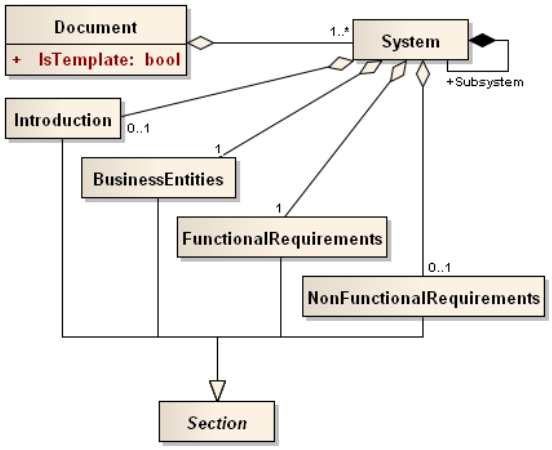

Figure 1. ProjectIT-RSL structural metamodel.

Given the match between our goals in terms of representation of functional requirements and the purpose of Use Cases specifications, we consider that, to avoid "reinventing the wheel", Use Cases are the best underpinning for developing the ProjectIT-RSL extension. Use Cases employ a small set of straightforward concepts that are easily understood by non-technical stakeholders. We also benefit from a well--defined body of knowledge [28]–[30], namely how these specifications can be structured and reused in an easy to understand and industry-standard manner. Finally, bearing in mind the current ProjectIT-RSL support for domain modeling, by adopting Use Cases for conveying functional requirements, we become aligned with the early stages of the well-known ICONIX process [30]. The major pitfall of using Use Cases as a requirements specification technique is the lack of well-defined rules. Nevertheless, this problem can be addressed by providing a Use Case context-free grammar that is aligned with the metamodel presented in Figure 2.

This extension enhances the *Functional Requirements* section by providing a subsection named *Use Cases*, which contains all Use Case specifications of the (sub)system being considered. An illustrative example is presented in Listing 1. This Use Case describes a simple and common interaction with a web application, in which a registered user tries to recover a password. The textual description already provides syntax highlighting according to a context-free grammar aligned with the metamodel presented in Figure 2.

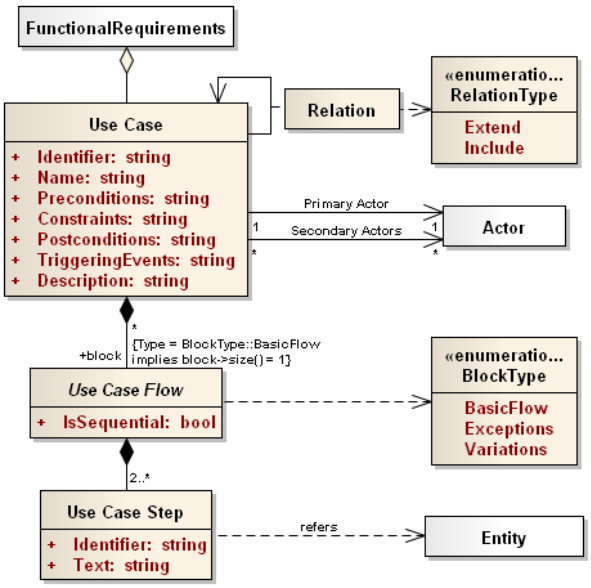

Figure 2. Use Case Driven metamodel.

B. Complex Behavior Specification with Pseudocode

To cope with the description of complex software system behavior, we advocate the usage of Pseudocode-like descriptions. According to our proposal, these Pseudocode--like [15], [16], [31] snippets act as adornments for Use Case steps. They only should be used when a fine-grain level of detail is required, namely to customize the intended action meaning or to entail the detailed specification of a complex behavior (not provided by built-in actions). The verbs used while describing detailed Use Case steps have linguistic properties (such as thematic relations[2]) that allow us to consider them as extension points to which functions in Pseudocode can be hooked up (if the function presents the same number of arguments and constraints as the verb theme). Gottesdiener [19] provides a comprehensive list of suggested verbs for both *informative* Use Cases that provide information to actors (e.g., *find*, *list*, *notify*, *select*, *view*) and *performative* Use Cases that allow actors to handle complex tasks (e.g., *approve*, *authorize*, *choose*, *send*, *submit*, *validate*). By resorting to this hook up mechanism we gain an additional level of flexibility regarding the possibility of providing custom-made functional specifications for the specific needs of the project at hand.

Pseudocode often appears in the literature as a simplified programming language to describe algorithms. Regarding the Pseudocode syntax, we adopt PlusCal's due to its benefits: we do not "reinvent the wheel" by designing a concrete syntax for Pseudocode, and we also automatically ensure compatibility with PlusCal's formal verification tools. PlusCal's documentation [17] already provides a context-free grammar that we reuse within the ProjectIT-RSL extension for specifying Pseudocode-like snippets.

```
1. System "MyWebApp"
  1.1. Section Business Entities (...) End Section
  1.2. Functional Requirements
    1.2.1. Section Actors Declaration (...) End Section
    1.2.2. Section Use Cases
      BeginUseCase
      Identifier: "UC22"
      Name: "Recover Password"
      PreConditions: Previous{"UC21"/"Authenticate User"} ;
                     State{"The User is not authenticated."}
      PostConditions: State("The system updates the User password.")
      PrimaryActor: "Anonymous User"
      SecondaryActors: "Mail Server"
      TriggeringEvent: "The user does not know the password."
      Description: "Verify if the e-mail is valid and if the e-mail is associated
                    with a registered user. Generate a new password and send the
                    new password to the user's email."
      BasicFlow: InTheFollowingOrder
          1. "The system asks the Anonymous User for the registration e-mail."
          2. "The Anonymous User inserts the registration e-mail."
          3. "The system verifies the registration e-mail."
          4. "The system generates a new password."
          5. "The system sends an e-mail with new password to the Anonymous User
              through the Mail Server."
          6. "The system displays a PasswordRecovered success message to the
              Anonymous User."
      Exceptions:
          3.   "The Anonymous User provides an invalid e-mail."
          3.1a "The system displays an InvalidEmail error message to the
                Anonymous User."
          3.1b "The system terminates the use case."
          3.   "The Anonymous User provides an inexistent e-mail."
          3.2a "The system displays an InexistentEmail error message to the
                Anonymous User."
          3.2b "The system terminates the use case."
          5.   "The Mail Server is unavailable."
          5.1a "The system displays a ServiceUnavailable error message to the
                Anonymous User."
          5.1b "The system terminates the use case."
      EndUseCase
    End Section
  End Section
End System
```

Listing 1. Use Case example: "Recover Password".

Following the example of Listing 1, we can illustrate the usage of Pseudocode to specify the behavior related with the verb *"send"* used within step 5 (*"The system sends an e-mail with new password to the Anonymous User through the Mail Server."*). According to VerbNet, the sentence's main verb (*"send"*) has three roles (arg0 the "sender", arg1 the "sent", and arg2 the "sent-to"). While mapping these roles we notice that step 5 contains additional information provided by a prepositional sentence ("through the Mail Server"). This piece of information will not be considered while hooking up the verb arguments with the built-in actions, unless we introduce a Pseudocode extension. We can specify a custom-made Pseudocode snippet (illustrated in Listing 2) to support this additional verb role (arg3 the "by-means").

```
--algorithm Send

variables maxAttempts = 3, secondsBetweenAttempts = 60

procedure SentToBy( variables sender, message, destination, channel )
  variables count = 0
  while ( count < maxAttempts )
    if ( call SendToChannel( message, destination, channel ) )
      return 'true'
    else
      count := count + 1
      call SleepSeconds( secondsBetweenAttempts )
    end if
  end while
  return 'false'
end procedure

end algorithm
```

Listing 2. "Send-To-By" Pseudocode specification example.

[2]Thematic relation is a linguistic term used to express a syntax–semantic correlation, namely the roles that a noun phrase plays with respect to the action or state described by a sentence's main verb.

C. Parsing of ProjectIT-RSL Extension

This extension of ProjectIT-RSL resorts to different parsing techniques, being the pipeline divided in two different stages. First, we make use of a structural template, derived from Use Case textual description best practices [28]–[30]. This derived template is processed with a context-free grammar similar to those used by programming languages. Next, after the structural information has been parsed, the process follows to the second stage of the pipeline. At this stage we make use of NLP techniques, namely shallow parsing[3] of the pieces of information previously captured according to its context within the structural template. The extracted information, namely from detailed Use Case steps, will populate models aligned with the ProjectIT-RSL metamodel. Pseudocode is used to specify complex behavior not covered by built-in actions associated with the verbs used while describing detailed Use Case steps.

We resort to shallow parsing techniques to address complexity of natural language, which hinders the possibility of building a general-purpose representations of meaning from ad-hoc natural language text [32]. To ease the burden of full parsing, we only look for very specific kinds of information in detailed Use Case steps, namely to populate models according to the *"Actor performs Operation on Entity"* core structure emphasized by the ProjectIT-RSL metamodel. Our shallow parsing approach can be considered as Chunking [32, Ch. 7, p. 261], more specifically a regular expression cascaded chunker. The possibility of adding new parsing rules that express linguistic patterns, namely to extract specific and contextualized pieces of information from detailed Use Case steps, makes the language extensible.

IV. Conclusion

Requirements Engineering is a mature discipline, and throughout the last decades several approaches to improve its processes and artifacts have been proposed. Yet, there are still open issues to address, namely problems regarding communication and requirements quality.

The ProjectIT initiative advocates that a greater attention should be given to the rigor and quality of requirements specifications. Namely, it emphasizes the need for tool support, because most RE techniques are manually applied, thus making them error-prone and less cost-effective when dealing with modern software systems, due to their size and complexity. Since requirements provide the foundation for most of subsequent technical work, these activities would certainly benefit from consistency checking of the requirements specifications. Also, unless suitable tool support is provided, to validate and generate an early design draft from requirements specifications, it will be difficult to seamlessly integrate RE with the Model-Driven Engineering paradigm.

[3] As opposed to full parsing, where one or more complete parsing tree are derived, shallow parsing techniques try to identify and extract information from relations between words.

Considering these facts, this paper presents an extension for the ProjectIT-RSL language, to address its current lack of support for the specification of behavioral aspects. To convey information about the software system behavior with ProjectIT-RSL, namely to specify functional requirements, we propose an integration of well-known techniques, more specifically textual Use Cases with Pseudocode specifications. The former is an industry-standard technique that provides a template for structuring functional requirements, thus reducing the complexity of natural language parsing with well-defined contexts. The latter provides a technology-agnostic approach to express complex behavior (the computations associated with the customized-actions, i.e., the verbs used within Use Case steps). This extension complements the current ProjectIT-RSL support for domain modeling and, although it is process-independent, it can be well supported by processes such as ICONIX or RUP.

As future work, in the short-term we plan to validate the ProjectIT-RSL language with real-world case studies. First, we are going to develop the required tool support by extending the current ProjectIT toolset, namely the parsing algorithms and text editor. In the long run, we plan to use the requirements models entailed within ProjectIT-RSL to infer or verify properties of the target system (e.g., model-checking or theorem proving) in order to ensure the overall quality of its specification.

References

[1] C. Hood, S. Wiedemann, S. Fichtinger, and U. Pautz, *Requirements Management: The Interface Between Requirements Development and All Other Systems Engineering Processes*, 1st ed. Springer, December 2007, ISBN: 978-3540476894.

[2] D. Firesmith, "Modern Requirements Specification," *Journal of Object Technology*, vol. 2, no. 1, pp. 53–64, March 2003.

[3] B. Nuseibeh and S. Easterbrook, "Requirements Engineering: a Roadmap," in *Proc. of the Conference on The Future of Software Engineering (ICSE'00)*. New York, NY, USA: ACM, 2000, pp. 35–46, ISBN: 978-1-58113-253-3.

[4] K. Wiegers, *Software Requirements*, 2nd ed. Microsoft Press, March 2003, ISBN: 978-0735618794.

[5] P. Zave, "Classification of research efforts in requirements engineering," *ACM Computing Surveys*, vol. 29, no. 4, pp. 315–321, December 1997, ISSN: 0360-0300.

[6] R. R. Young, *The Requirements Engineering Handbook*. Artech Print on Demand, November 2003, ISBN: 978-1580532662.

[7] H. Foster, A. Krolnik, and D. Lacey, *Assertion-based Design*. Springer, 2004, ch. 8 - Specifying Correct Behavior.

[8] F. Meziane, N. Athanasakis, and S. Ananiadou, "Generating Natural Language specifications from UML class diagrams," *Requirements Engineering*, vol. 13, no. 1, pp. 1–18, January 2008, DOI: 10.1007/s00766-007-0054-0.

[9] A. Silva, "O Programa de Investigação ProjectIT (whitepaper)," October 2004.

[10] A. Silva, J. Saraiva, D. Ferreira, R. Silva, and C. Videira, "Integration of RE and MDE Paradigms: The ProjectIT Approach and Tools," *IET Software Journal*, vol. 1, no. 6, pp. 217–314, December 2007.

[11] C. Videira, D. Ferreira, and A. Silva, "A Linguistic Patterns Approach for Requirements Specification," in *Pro. of the 32nd EUROMICRO Conf. on Soft. Eng. and Advanced Applications*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 302–309, ISBN: 0-7695-2594-6.

[12] T. Kuhn, "Controlled English for Knowledge Representation," Ph.D. dissertation, Faculty of Economics, Business Administration and Information Technology of the University of Zurich, 2010.

[13] N. E. Fuchs, U. Schwertel, and R. Schwitter, "Attempto Controlled English – Not Just Another Logic Specification Language," in *Logic-Based Program Synthesis and Transformation*, ser. Lecture Notes in Computer Science, P. Flener, Ed., no. 1559, Eighth International Workshop LOPSTR'98. Manchester, UK: Springer, June 1999.

[14] N. E. Fuchs, K. Kaljurand, and T. Kuhn, "Attempto Controlled English for Knowledge Representation," in *Reasoning Web, Fourth International Summer School 2008*, ser. Lecture Notes in Computer Science, no. 5224. Springer, 2008, pp. 104–124.

[15] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, June 2004, vol. 5565, ch. The Pseudocode Programming Process, ISBN: 978-0735619678.

[16] J. Dalbey, "Pseudocode Standard," Retrieved Friday 18th February, 2011 from http://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html.

[17] L. Lamport, "The pluscal algorithm language," in *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, ser. ICTAC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 36–60, Retrieved Wednesday 15th December, 2010 from http://research.microsoft.com/en-us/um/people/lamport/pubs/pluscal.pdf. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03466-4_2

[18] Microsoft Corporation, "The PlusCal Algorithm Language," January 2009, Retrieved Wednesday 15th December, 2010 from http://research.microsoft.com/en-us/um/people/lamport/tla/pluscal.html.

[19] E. Gottesdiener, *The Software Requirements Memory Jogger: A Desktop Guide to Help Software and Business Teams Develop and Manage Requirements*, 1st Spiral-Bound ed. Goal/QPC, October 2009, ISBN-13: 978-1576811146.

[20] W. E. McUmber and B. H. C.Cheng, "A General Framework for Formalizing UML with Formal Languages," in *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 433–442, ISBN: 0-7695-1050-7.

[21] N. Debnath, M. C. Leonardi, M. V. Mauco, G. Montejano, and D. Riesco, "Improving Model Driven Architecture with Requirements Models," *Information Technology: New Generations, Third International Conference on*, vol. 0, pp. 21–26, April 2008, ISBN: 978-0-7695-3099-4.

[22] G. S. A. Mala and G. V. Uma, *PRICAI 2006: Trends in Artificial Intelligence*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, July 2006, vol. 4099, ch. Automatic Construction of Object Oriented Design Models [UML Diagrams] from Natural Language Requirements Specification, pp. 1155–1159, ISBN: 978-3-540-36667-6.

[23] ——, *Elicitation of Non-functional Requirement Preference for Actors of Usecase from Domain Model*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, December 2006, vol. 4303, pp. 238–243, ISBN: 978-3-540-68955-3.

[24] S. Konrad and B. H. C. Cheng, *Satellite Events at the MoDELS 2005 Conference*, ser. Lecture Notes in Computer Science. Springer-Verlag, January 2006, vol. 3844, ch. Automated Analysis of Natural Language Properties for UML Models, pp. 48–57.

[25] B. Cheng and J. Atlee, "Research Directions in Requirements Engineering," *ICSE FOSE*, 2007.

[26] D. Ferreira and A. Silva, "A Requirements Specification Case Study with ProjectIT-Studio/Requirements," in *Proceedings of the 2008 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2008, pp. 656–657.

[27] ——, "Survey on System Behavior Specification for Extending ProjectIT-RSL," in *Proc. of the 7th Int. Conf. on the Quality of Information and Communications Technology (QUATIC 2010)*. Los Alamitos, CA, USA: IEEE Computer Society, September 2010, pp. 210–215, ISBN: 978-0-7695-4241-6.

[28] A. Cockburn, *Writing Effective Use Cases*, 1st ed. Addison Wesley, October 2000, iSBN-13: 978-0201702255.

[29] S. Adolph, P. Bramble, A. Cockburn, and A. Pols, *Patterns for Effective Use Cases*, 1st ed. Addison Wesley, August 2002, iSBN-13: 978-0201721843.

[30] D. Rosenberg and M. Stephens, *Use Case Driven Object Modeling with UML: Theory and Practice*, 1st ed. Apress, January 2007, iSBN-13: 978-1590597743.

[31] C. Borysowich, "Guidelines for Pseudocode in Documenting Specifications," Retrieved Friday 18th February, 2011 from http://it.toolbox.com/blogs/enterprise-solutions/guidelines-for-pseudocode-in-documenting-specifications-16011.

[32] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*, 1st ed. O'Reilly Media, June 2009, ISBN-13: 978-0596516499.