Survey on System Behavior Specification for Extending ProjectIT-RSL

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*—The ever increasing demand for more complex and larger software systems stresses the importance of having well-engineered requirements. Requirements Engineering (RE) can no longer be an isolated upfront phase: RE is critical to ensure the success and quality of the delivered system. This paper presents a study of approaches, techniques, and languages for specifying system behavior. From this survey we synthesize a greatest common subset of constructs, which can be adapted for extending the current ProjectIT-RSL language. The purpose is to use the specifications written in this controlled natural language to enable automatic consistency checking to increase stakeholders engagement in RE tasks related with verification and validation.**

I. Introduction

Despite the research and industrial efforts over the last decades, the engineering of modern software systems is still a rather complex process. The dimension and sophistication demanded for developing these large-scale systems require a high-level degree of abstraction and conceptual design upfront [1], before the actual implementation takes place. Otherwise, the rework and cost required to fix concealed errors at later stages can compromise the project success [2].

Typically, the development of a software system begins with Requirements Engineering (RE) activities [3]. This system engineering discipline deals with the early activities related with the process of discovering the purpose, scope, stakeholders, boundaries, and actors of the system being specified [4]. Since, ultimately, the measure of success of a software system is based on the degree to which it meets the purpose and expectations for which it was intended by business stakeholders, RE plays a key role within the whole software development life-cycle. 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 system development [5].

It is argued that requirements specification is one of the most problematic activities of the whole software development process [2]. Historically, it consists of creating a natural language description of what the target system should do, or constraints on its behavior [6]. However, this form of specification is ambiguous and, in many cases, unverifiable because of the lack of a standard machine-executable representation [6]. Hence, aside from communication-related misinterpretation issues, the main problem appears during the process of transforming natural language requirements into a formal or, at least, semi-formal computer model [2]. The former is supported by formal methods (or languages) such as the B-Method [7]; whereas the latter are addressed by graphical notations, such as the Unified Modeling Language (UML). Despite the existence of these visual approaches, textual specifications are still the most suitable, fast, and preferred manner (by non-technical stakeholders) to begin the requirements specifications of the envisioned software system [6].

The ProjectIT initiative [8] advocates that the software development process should be focused on higher-level activities, such as those related with Project Management, Requirements Engineering, and Architectural Design. On the other hand, the production effort, namely programming and testing activities, should be minimized and performed automatically. The main research topics of ProjectIT are related with the RE, Quality Engineering (QE), and Model-Driven Engineering (MDE) domains. The endeavor of our research under this initiative resulted into the ProjectIT approach [9], which is supported by a set of modeling languages. One of these languages is ProjectIT-RSL [10], which consists of a controlled natural language, based on linguistic patterns, for deriving requirements models from textual specifications. However, despite the concrete results achieved, there are still several open issues. The current RE component of ProjectIT initiative still has some limitations. For instance, neither does ProjectIT-RSL supports aspects related to the definition of software behavior, nor does it have a complementary graphical notation. To succeed, the controlled natural language approach provided by ProjectIT-RSL needs to cover not only static aspects of the target system (e.g., domain model aspects) but also dynamic aspects related with functional and non-functional requirements. These facets are crucial to specify complex interaction patterns of actors with the target system. Hence, ProjectIT-RSL needs to be extended to capture dynamic aspects that characterize the software system behavior (i.e., the way in which it works or functions in response to a particular situation or stimulus).

Despite some skeptical opinions [11], we advocate that Natural Language Processing (NLP) techniques can significantly contribute to leverage the current tool support for RE activities [2]. Our research aims to automatically extract concepts and relations of the target system from textual specifications, for early detection of inconsistencies within these generated models. Also, we seek to use these models to infer or verify properties of the target system, by using inference or formal methods techniques and tools, such as model-checking or theorem proving. The goal of our research is to provide tool support that fosters proper business stakeholders engagement and participation throughout the development process, thus addressing the social and collaborative nature of RE [12].

The main contributions of this paper are twofold: (1) a broad survey on methods, techniques, and languages for specifying requirements related to software behavior; and (2) the identification of a greatest common set of constructs, which a specification language oriented towards non-technical stakeholders should support, in order to capture the target system behavior. Currently, neither the NLP approach to automatically derive conceptual models from requirements texts is new [2], nor is the usage of formal methods to support consistency checking, through mapping techniques [13]. However, the contributions of this paper are novel in the sense that we go beyond the traditional research focused on statical/structural aspects of the target system, by also considering behavior aspects.

The remainder of this paper is organized as follows. Section II presents a review of related work regarding the specification of the behavior of software systems. In Section III we discuss the results of the behavior-related survey. Section **??** presents some considerations about how these results can be applied to support behavior-related extensions for ProjectIT-RSL. Finally, Section V provides some concluding remarks, highlighting its main contributions and presenting some thoughts for future work.

II. Related Work

Although the object-oriented (OO) paradigm brought several advantages regarding the proficiency of software development process [13] by introducing more cognitive-amenable concepts, we think that this survey still ought to encompass older approaches that may have been forgotten in the meantime, specially those related with behavior specification. The dominance of mainstream modeling languages (e.g., UML) may lead to the well-known "law of the instrument", in which one observes a over-reliance on a familiar tool. Thus this survey was conducted in a "back to roots" fashion, seeking to identify benefits of alternative methods, techniques, and languages. This approach follows the recommendation that "researchers need to think beyond current RE and SE knowledge" [14], even at the risk of possible failure.

A. Tabular Representations

One of the oldest (but still in use [15]) approaches to represent software behavior derives from early computer science theory, namely from transition tables. Although tabular representations may seem to be too low-level notation to deal with, they are indeed useful, even for the specification of complex systems. They have proven to be easier to understand and review than source code. Moreover, their simplicity and the high-level degree of abstraction in which they are used, allows them to be coupled with Domain-Specific Languages [16]. However, although it is argued that they scale well for several conditions with many possibilities, their main weakness may arise from the lack of contextual information and scalability issues in terms of legibility and overwhelming amounts of information in a single table (a side-effect that a divide-and-conquer approach can lessen).

Some of the most reckoned tabular-based techniques are *Control Tables* and *Decision Tables* [17]. The former are simple tabular representations that control the program flow by associating *conditions* with *actions* to be performed. In contrast, *Decision Tables* are more elaborated structures. Unlike those found in traditional programming languages, *Decision Tables* can associate many independent *conditions* with several *actions* in an elegant fashion. *Decision Tables* are often represented as a tabular structure with four quadrants. Following the Cartesian coordinate system notation, we have: (1) the 2nd quadrant contains the list of conditions; (2) the 1st quadrant presents the rules that arise from the composition of the possible values of conditions; (3) the 3rd quadrant lists all the available actions; and finally (4) the 4th quadrant indicates for each rule which are the actions that become enabled if the rule (composition of condition values) holds. However, depending on the specific needs of the domain of application, the condition values can be: (1) boolean, following the construct IF-THEN-ELSE; (2) numbered, following the construct SWITCH-CASE; and (3) fuzzy logic or stochastic. Furthermore, action entries can simply represent whether an action or, in more advanced scenarios, if a sequence of actions is to be performed.

Since these tabular representations are a precise and yet compact way to model complicated control flow, they can be executed by a proper interpreter, but at a higher level of abstraction. By associating table actions to custom-built or customized services, these services can be composed and executed by the interpreter, which effectively acts as a virtual machine that executes the program embedded within the control table. Therefore, these tabular representations can drive the behavior of a software system in an executable manner.

B. Pseudocode

Pseudocode [18] is a mix of natural language with structured programming constructs to specify or document system behavior. It is widely used to describe computer science algorithms and there is even an approach named Pseudocode Programming Process [19]. Pseudocode allows the designer to be concerned with the software behavior, without being distracted by details of the syntax arising from the formal language used. The designer is encouraged to use vocabulary of the problem domain, not in terms of implementation details (of the solution domain). Pseudocode can be though as a narrative for someone who knows the requirements (i.e., the problem domain) and is trying to tackle a possible solution by iteratively refining a draft. However, by making use of constructs with formal semantics (i.e., programming language or mathematical constructs), pseudocode can entail elaborated software specifications. After a few iterations, when the lowest level is reached, the pseudocode can be straightforwardly translated into the target implementation language, when combined with a data dictionary. To achieve this translation, pseudocode needs to be complete. Regarding pseudocode completeness, there are studies [19] that emphasize the benefits and the preference for pseudocode as a tool to detect insufficiently detailed designs and for the ease of documentation and ease of modification it provides. Despite its benefits, pseudocode has a drawback. Its specifications are more suitable for stakeholders with technical background. In short, pseudocode doesn't have a well-defined notation (there is no "universal standard"), since it is intended to be read by people, not to be directly interpreted by a computer [18].

Borysowich [20] presents a handful of useful guidelines for writing pseudocode: (1) for structuring the pseudocode, one should use indentation to emphasize scope nesting, place each statement on a separate line, and group together logically-related sequences of statements into modules; (2) the specification process should be iterative: in each refinement, statements should be replaced by more lines of pseudocode or by a direct call to another procedure; (3) keywords should be taken from the formal language being used (i.e., a fixed vocabulary of structured programming constructs); (4) one should adopt a consistent writing style, namely using capitalization only for keywords, procedure and module names, and also a consistent convention for naming data elements; and finally (5) there should exist a data dictionary with the definition of all data element names, and the binding to these data elements should be made through the descriptive name assigned to them.

The core structured programming constructs recommended for pseudocode specifications are [18]: (1) sequence, which is used to represent a linear execution where one statement is performed sequentially after another; (2) conditional branching, in which a choice is made between alternative courses of execution (e.g., the IF-THEN-ELSE construct); and (3) loop, in which a block of statements is repeated until a conditional holds (e.g., the WHILE construct). It has been proven that these three basic constructs for control flow are sufficient to properly implement any kind of algorithm [18]. However, there are also other constructs that may be useful to include [18].

C. Structured Natural Language

A way of addressing part of the ambiguity intrinsically introduced by the usage of natural language, during the specification process of the target system behavior, is to force some structuring rules into the textual specifications. This approach supports narrative-like natural language specifications as a series of blocks that uses capitalization and indentation rules to represent the scope of bindings and hierarchical structure associated with control-flow logic. The introduction of these constrains enforces a certain degree of rigidity, similar to a computer program, but staying at an higher-level degree of generality regarding detailed aspects of the solution, namely design or implementation issues.

The typical constructs of structured natural language, can be classified according to three main topics [21]: (1) sequence; (2) conditional; and (3) repetition. *Sequence* constructs are used to express the top-down ordered execution of the actions entailed within the semantics of each statement. *Conditional* constructs support the cases where a sequence of one or more statements is executed only if a certain logical condition is true (e.g., the typical IF-THEN-ELSE control structures). These constructs encompass scenarios of mutually exclusive conditions (e.g., the usual SWITCH-CASE control structure). *Repetition* constructs provide a way of expressing recurring behavior, where a set of actions is executed multiple times. Within these constructs one can distinguish between: (1) those that have a guard condition at the beginning (e.g., WHILE control structure), thus allowing zero or more executions, and (2) those that place the repetition condition at the end of the set of statements (e.g., DO-WHILE or REPEAT-UNTIL control structure), hence allowing at least one execution. Finally, although not mentioned explicitly in [21], the author mentions *scoping* constructs for grouping blocks of statements together, with the possibility of providing a meaningful name that describes their purpose. These constructs allow the definition of logically bound steps of behavior and binding environments for names. Borysowich [21] also presents guidelines for writing structured natural language. These guidelines allow one to significantly narrow down to a few couple of possible interpretations (preferably just one). We synthesized these guidelines into the following groups:

a) Structure: One should use capitalization style to emphasize scope definition (e.g., through scope labeling constructs or with core constructs of the language that entail scopes *per se*) and use indentation to stress the logical hierarchy of scopes (i.e., nesting). All constructs that entail scope should provide a beginning and ending keyword to wrap the inner block of statements, thus making the scope explicitly identifiable. Also, each statement should be clearly separated from others, hence one should write one statement per line. Finally, there should be an escape character (e.g, an asterisk) to support side-notes or other kind of remarks such as comments.

b) Control-Flow: Since these constructs can be thought of as a metalanguage (i.e., the language to structure natural language), they should also follow the capitalization style. The workflow that entails the behavior addressed by the specification must be written in terms of sequential, conditional, and repetition constructs previously mentioned.

c) Semantics: To reduce ambiguity, each statement should follow a grammatical form that expresses direct commands or requests (i.e., the imperative mood). Also, during name resolution (i.e., binding of names to their meaning) one should adopt a different writing style, like underline or quotation marks, for expressing that the definition or meaning of a word or noun phrase exists and is available elsewhere in a data source (e.g., a glossary or a detailed dictionary). Moreover, by using keyword capitalization one can significantly reduce ambiguity regarding core language constructs. Finally, one can use parentheses to clarify precedence within conditional statements.

Borysowich also distinguish *Structured Natural Language* from *Tight English* [22], and draw some considerations on how to reduce ambiguity while at representing logic in natural language documents [23].

D. Use Cases

Use Cases are a popular requirements modeling technique. The behavior of the system under development is often specified through *Use Cases*, which are basically structured stories or scenarios describing what should happen when the system is used in a stimulus-response fashion (i.e., according to a "black box" model). By using a relatively simple handful of concepts, *Use Cases* provide the means to describe the behavioral aspects of a wide range of systems. *Use Cases* allow one to attain a high-level overview of the interactions of actors (i.e., user roles or other systems) with the system being specified, according to a functional perspective. Although it might seem a trivial task, the relative 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* for the specific problem at hand, specially for relatively complex systems.

To avoid such situations, Adolph et al. [24] describe proven solutions (almost three-dozen patterns) to the specific problems of writing *Use Cases* on real projects, hence providing guidelines for writing and organizing them effectively, and, most importantly, to help judge their quality. It is argued wether the process should start with textual descriptions or with Use Case diagrams. Since both variants have theirs strengths and weakness, the problem can be reduced to a matter of proficiency on the approach and suitability to the problem being addressed.

Misuse Case is an interesting business process modeling technique, which has derived from Use Cases [25]. In contrast with the original concept of Use Case, a Misuse Case describes something that should not occur, i.e., the steps of interactions that may affect the expected behavior of the system [25]. This technique is extremely useful when tightly coupled with Use Cases because it often leads to the discovery of new requirements, which again may be expressed in Use Cases. Besides the benefits of triggering the chain reaction of iteratively refining system behavior, it is argued that this modeling tool has the additional strengths: (1) although Use Cases aim to model functional requirements, Misuse Cases provide the means to address non-functional requirements without requiring a cognitive disruption in terms of concepts used to model system requirements; (2) the refutation-based approach used avoids taking premature design decisions based on oversimplified assumptions about the problem domain; and (3) it inherits the benefits of mitigating the communication gap and fostering consensus-building among stakeholders at different levels of technical competence. On the other hand, regarding its key weaknesses, one can point out: (1) simplicity, which may imply the use of other complementary modeling tools; and (2) lack of well-defined methods guidelines.

Concerning modeling concepts, Misuse Cases introduce two counterparts to those available in *Use Cases*: (1) *Misuse Case*, which consists in the sequence of steps that may lead the system to exhibit unexpected behavior; and (2) *Misuser*, the actor that intentionally or inadvertently initiate the *Misuse Case*. Since Misuse Cases are a specialization of Use Cases, they also use the same relation types. However, they introduce two new relation types: (1) *Mitigates*, which is used to express that a Use Case lessens the likelihood of a Misuse Case to occur (i.e., a countermeasure); and (2) *Threatens*, which means that a Misuse Case can hinder the achievement of a Use Case purpose. Sindre and Opdahl [25] synthesize these new concepts, along with those pertaining to Use Cases, into a common metamodel.

III. Discussion

There is a wide set of research proposals and work thoroughly validated in the literature regarding the specification of software behavior. Moreover, the use of natural language to support RE activities has already been studied for a long time in the RE community [2]. Nevertheless, many problems concerning the use of developed techniques in practice are still unresolved [14]. In our opinion, the state of the art of RE methods, techniques, and languages still lack a consolidated approach to lessen the effort to overcome the conceptual gap between informal text-based requirements (in the problem domain), usually specified by non-technical stakeholders, and a formal representation that enables some level of automation regarding verification and generation of artifacts.

Most of the research of formal methods and consistency model checking is still focused on low-level implementation details (e.g., for embedded systems [13], [26]), whose mindset and working methods are relatively close to the underlying mathematical theory used to formalize the specifications. Moreover, these research proposals make use of graphical notations (e.g., often a subset of UML diagrams) as their primary specification language.

Although we do not disagree with the use of graphical notations to aid the comprehension of requirements, we think that these diagrams could be automatically generated from textual specifications or, if they are used as a specification language, it should be provided a way of converting them back into a textual representation. Regarding RE we consider graphical notations as an auxiliary notation. For instance, the Object-Process Methodology, and its supporting tool OPCAT [27], deal with system structural and behavioral aspects by adopting a single unified model that can be represented through a visual formalism, which in turn can be automatically translated into a subset of natural language. To achieve this goal one could use an approach similar to the work of Meziane et al. [2], which generates natural language specifications from UML class diagrams.

To address the problem presented above we decided to conduct the broad scope analysis presented in this paper. All the approaches that we considered do not provide formal semantics, or do not even follow any particular standard. Nevertheless, they are widely used in practice, which justifies taking them into account within this survey.

Decision Tables capture the requirements through a list of conditions and a list of corresponding actions. Then for each combination of criteria we have a selection of actions. Therefore, the main concepts of *Decision Tables* are: conditions and actions. When compared with *Decision Tables*, *Pseudocode* and *Structured Natural Language* benefit from not only supporting branching logic as *Decision Tables*, but also from enabling step-wise statements for fine-grained definition of behavioral aspects in a more amenable manner.

There are no standard constructs for *Structured Natural Language* and the constructs for *Pseudocode* depend on the target implementation language. However, from the recommendations one can identify the following desirable constructs: (1) a BEGIN construct or a capitalized name for initiating a labeled scope, and the corresponding EXIT construct; (2) IF-THEN-ELSE and SWITCH-CASE for conditional constructs; and (3) WHILE, DO-WHILE, REPEAT-UNTIL, and FOR to address constructs for specifying recurring behavior. The main difference between *Structured Natural Language* and *Pseudocode* is that the former seems like a spoken language, whereas the latter resembles a programming language (encompassing constructs with formal semantics). The translation from *Structured Natural Language* to an executable notation would be more difficult because it specifies behavior at an higher-level of abstraction and doesn't require to be complete, thus omitting several effective execution-related details.

Still regarding the comparison of Pseudocode with other approaches presented in this paper, one can expect the following benefits [19]: (1) it lessens the burden of the review process, by discarding low-level syntax and focusing on problem domain (requirements), which is important for the verification process, and even for the subsequent validation, if non-technical users are able to understand the pseudocode specification, as intended; (2) if follows an iterative top-down refinement approach from high-level specifications to detailed design (similar to some formal methods [7]); (3) it is change-friendly, since it is easier to change pseudocode statements than larger amounts of detailed specifications; (4) it is economical, because it identifies errors sooner, hence avoiding their propagation; (5) it records the design rationale, since pseudocode statements become comments afterwards; and (6) it is suitable for communication and documentation, as long as inline comments are maintained.

When compared with other approaches, *Use Case*s are suitable for specifying large complex systems, when a technique to discover requirements at a high-level of abstraction is needed. But, sometimes the number of *Use Case*s required to model such systems can become inefficient and inappropriate for requirements analysis and as a specification method. Nevertheless, Mala et al. [28], [29] present a research regarding *Use Cases* combined with NLP techniques, from which we can gain some insight on the subject. Although their work has a different goal, they present an approach for automatic construction of object-oriented design models in UML from the natural language requirement specification. Unlike others, their approach combines domain modeling, Use Cases descriptions (written in a restricted form of natural language), and non-functional taxonomy to derive non-functional requirements prioritization through trade-off analysis.

IV. EXTENDEDING PROJECTIT-RSL

With the information gathered from this survey, the approach for defining the ProjectIT-RSL behavioral extensions (earlier sketched in [30]) is going to consist in the interweaving of *Use Cases* with *Pseudocode*. We opted by these two approaches because they are complementary: the best practices of *Use Cases* descriptions [24] will provide an overview and high-level structuring, whereas the *Pseudocode*-like specifications will provide enough expressive power to describe the *Use Cases* behavior in detail. However, instead of using constructs from a programming language, the *Pseudocode*-like specifications will rely on high-level constructs built-in into a *Pseudocode Virtual Machine*. The theoretical underpinning of this virtual machine can be supplied by a workflow language that supports most of the workflow patterns [31], which provide a framework to formally assess the expressiveness of these kind of languages. Furthermore, besides being a well-established and known technique, *Use Cases* descriptions can be easily converted into their graphical notation, for communication purposes. Finally, Misuse Cases [25] exemplify how Use Cases can be extended to model non-functional requirements.

Besides some new keywords and content-related scoping rules, these behavioral extensions will follow the previous design principles of ProjectIT-RSL [10]: to define linguistic patterns for capturing the most of the stakeholders' representational information from their textual specifications, without imposing a particular writing style. Moreover, these extensions will require the already defined structural concepts, as suggested in Figure 1. To achieve this goal, we are going to

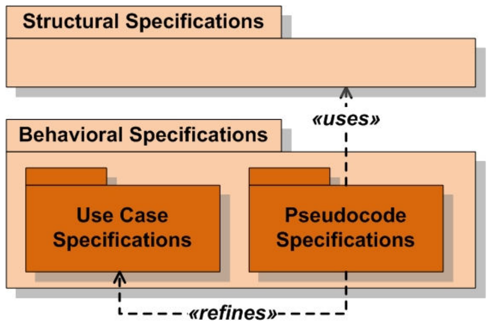
Structural Specifications
«uses»
Behavioral Specifications
Use Case Specifications
Pseudocode Specifications
«refines»

Figure 1. Overview of ProjectIT-RSL behavior-oriented extension.

research the variations of *Pseudocode* specifications through the study of a large number of examples from the literature. Afterwards, we are going to develop rules for removing potential ambiguities in this subset of natural language. This is justified by the fact that requirements specifications primarily contain narrative natural language textual statements that document all constraints on the behavior of the envisioned system. The ultimate purpose of ProjectIT-RSL is to empower and encourage non-technical stakeholders to specify, by themselves, parts of the software behavior in an iterative and reusable manner. This does not imply the absence of the supervision of a requirements engineer for more detailed concerns, namely to enrich or correct the specifications in order for them to be amenable for procedural automation.

V. Conclusions

In this paper we present a survey of approaches to capture and specify the behavior of a software system. The work presented is novel in the sense that most of the approaches and tools that use NLP techniques (to analyze natural language software requirements documents and produce initial object-oriented conceptual models) aim at structural aspects of the system being specified, such as domain models with UML class and object diagrams [32]. Also, we discuss the survey results and, based upon them, we introduce a greatest common set of constructs that a specification language ought to have, in order to cope with the needs for behavioral specification of software systems. With the information gathered, we lay ground for the definition of extensions for ProjectIT-RSL.

Since we are still at a preliminary stage of this research, numerous directions for future work are possible. However, the main short-term goal of this research is to enable ProjectIT-RSL to surpass the current lack of support for behavior-related specifications [30]. In the medium term, we intend to provide tool support for these behavior extensions. After that, we plan to evaluate the language main characteristics [33], and finally to evaluate the tool support through blind trials against a corpus of requirements.

References

[1] S. W. Lee and R. A. Gandhi, "Ontology-based Active Requirements Engineering Framework," *Asia-Pacific Software Engineering Conference*, pp. 481–490, April 2005, ISSN: 1530-1362.

[2] 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, ISSN: 0947-3602.

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

[4] 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: 1-58113-253-0.

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

[7] J. Abrial, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, October 1996, iSBN-13: 978-0521496193.

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

[9] 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.

[10] 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.

[11] K. Ryan, "The Role of Natural Language in Requirements Engineering," in *Proc. of the IEEE International Symposium on RE*. IEEE Computer Society Press, January 1993, pp. 240–242, ISBN: 0-8186-3120-1.

[12] D. Ferreira and A. Silva, "Wiki-based tool for requirements engineering according to the projectit approach," *Software Engineering Advances, International Conference on*, vol. 0, pp. 359–364, 2009.

[13] 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.

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

[15] G. Brooke, "From Requirements to Tables to Code and Tests," Retrieved Friday 9th April, 2010 from http://commons.oreilly.com/wiki/index.php/From_Requirements_to_Tables_to_Code_and_Tests.

[16] M. Fowler, "Domain-Specific Language," Retrieved Friday 9th April, 2010 from http://www.martinfowler.com/bliki/DomainSpecificLanguage.html.

[17] H. Schumacher and K. C. Sevcik, "The Synthetic Approach to Decision Table Conversion," *Communications of the ACM*, vol. 19, no. 6, pp. 343–351, 1976.

[18] J. Dalbey, "Pseudocode Standard," Retrieved Friday 9th April, 2010 from http://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html.

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

[20] C. Borysowich, "Guidelines for Pseudocode in Documenting Specifications," Retrieved Friday 9th April, 2010 from http://it.toolbox.com/blogs/enterprise-solutions/guidelines-for-pseudocode-in-documenting-specifications-16011.

[21] ——, "Guidelines for Structured English in Documenting Specifications," Retrieved Friday 9th April, 2010 from http://it.toolbox.com/blogs/enterprise-solutions/guidelines-for-structured-english-in-documenting-specifications-15987.

[22] ——, "Guidelines for Tight English in Documenting Specifications," Retrieved Friday 9th April, 2010 from http://it.toolbox.com/blogs/enterprise-solutions/guidelines-for-tight-english-in-documenting-specifications-15997.

[23] ——, "Things to Consider When Expressing Logic," Retrieved Friday 9th April, 2010 from http://it.toolbox.com/blogs/enterprise-solutions/things-to-consider-when-expressing-logic-15970.

[24] S. Adolph, A. Cockburn, and P. Bramble, *Patterns for Effective Use Cases*, 1st ed. Boston, MA, USA: Addison-Wesley, August 2002, ISBN-13: 978-0201721843.

[25] G. Sindre and A. L. Opdahl, "Eliciting security requirements with misuse cases," *Requirements Engineering*, vol. 10, no. 1, pp. 34–44, January 2005, ISSN: 0947-3602.

[26] S. Konrad, L. A. Campbell, B. H. C. Cheng, and M. Deng, "A Requirements Patterns-Driven Approach to Specify Systems and Check Properties," in *SPIN*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 18–33.

[27] OPCAT, "What is OPM?" Retrieved Friday 9th April, 2010 from http://www.opcat.com/products_opm.htm.

[28] 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.

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

[30] D. Ferreira and A. Silva, "A Controlled Natural Language Approach for Integrating Requirements and Model-Driven Engineering," in *Proc. of the International Conference on Software Engineering Advances*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 518–523, ISBN: 978-0-7695-3777-1.

[31] N. Russell, A. ter Hofstede, W. van der Aalst, and N. Mulyar, "Workflow Control-Flow Patterns: A Revised View," BPM Center, BPMcenter.org, Tech. Rep. BPM-06-22, 2006.

[32] H. Harmain and R. Gaizauskas, "CM-Builder: A Natural Language-Based CASE Tool for Object-Oriented Analysis," *Automated Software Engineering*, vol. 10, no. 2, pp. 157–181, April 2003, ISSN: 0928-8910.

[33] A. Caplinskas and J. Gasperovic, *Information Systems Development*. Springer, 2005, ch. Functionality of Information Systems Specification Language: Concept, Evaluation Methodology, and Evaluation Problems, ISBN: 978-0-387-25026-7.