FROM REQUIREMENTS TO READY TO RUN SOFTWARE: A BRIEF THOUGHT ON HOW TO MECHANIZE THE SOFTWARE DEVELOPMENT PROCESS

Alexandre R.S. Correia[1] and Juliano M. Iyoda[2] and Carla T.L.L. Silva[2]

[1]Campus Petrolina, The Federal Institute of Technology at Sertao Pernambucano
Postal code 56314-520, Petrolina, Pernambuco, Brazil
[2]Informatics Center, The Federal University at Pernambuco State
Postal code 50740-540, Recife, Pernambuco, Brazil

ABSTRACT

This paper presents a vision on how the software development process could be a fully unified mechanized process by getting benefits from the advances of Natural Language Processing and Program Synthesis fields. The process begins from requirements written in natural language that is translated to sentences in logical form. A program synthesizer gets those sentences in logical form (the translator's outcome) and generates a source code. Finally, a compiler produces ready-to-run software. To find out how the building blocks of our proposed approach works, we conducted an exploratory research on the literature in the fields of Requirements Engineering, Natural Language Processing, and Program Synthesis. Currently, this approach is difficult to accomplish in a fully automatic way due to the ambiguities inherent in natural language, the reasoning context of the software, and the program synthesizer limitations in generating a source code from logic.

KEYWORDS

Software Engineering, Requirements Engineering, Natural Language Processing, Program Synthesis, Software development process

1. INTRODUCTION

The software development process usually comprises a set of the following tasks: Firstly, a group of stakeholders writes requirements using a common text editor to envision the requirements of the system. Secondly, those requirements are passed to Requirement Analysts who build a set of formal models (like UML diagrams, textual requirements, use cases, etc). Thirdly, a group of Software Architects works on those models and on the requirements documents and refines them to accomplish a "well-ended design model" or a "final model." We call these tasks the "translation of software requirements into design specifications." Fourthly, after producing a "final model," a group of Programmers implements a set of code (source code, test code, documentation code, etc). We call this last specific task the "transformation of design specifications into source code". Fifthly, a group of Testers compiles and tests the software in order to verify if the software is working in conformance to the design. Finally, a group of Deliverers presents the software ready-to-run to those stakeholders to check its validity (to check if the software is working in conformance of what was required). Figure 1 shows this process in detail.

Particularly, the tasks of translating software requirements into design specifications and of transforming design specifications into source code have always been done manually. These manual tasks take more time, money and are more error-prone than the tasks that are fully or semi-automatic. Therefore it would be a good contribution to the software development process if these tasks were amenable to mechanization.

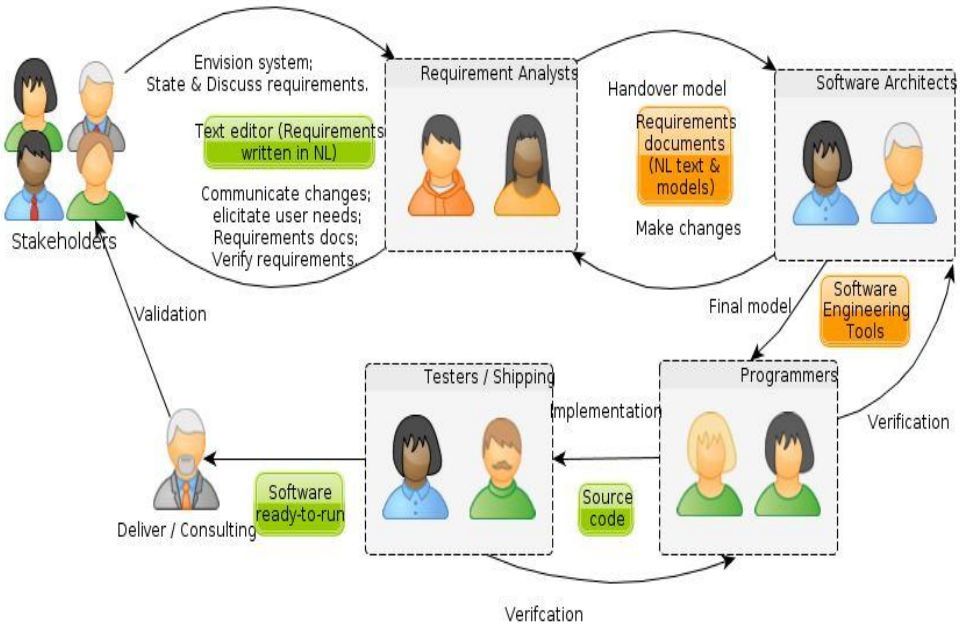

Figure 1: The software development process (adapted from Tichy & Koerner. 2010 [1]).

There are some approaches in the literature to mechanize the translation of software requirements into design specifications. One approach processes requirements written in natural language to generate UML diagrams, such as class diagrams, state diagrams, and sequence diagrams [1,16,17,18,19] by identifying and removing ambiguities inherent to natural language through ontology, grammars, or dictionaries [2, 3, 4].

Also, in order to transform design specifications like UML diagrams into source code, there are some tools developed in the context of Model Driven Development [20,21,22].

Moreover, it is possible to extract test cases from requirements as well [5, 6, 23]. Finally, by using controlled natural language (such as a subset of English with a formal grammar) it is possible to mechanized the translation to another formal language [24, 25].

Although the set of techniques available provide useful information about how to mechanize the current software development process, there is still space left for improvements.

This paper presents how these currently available technologies could be integrated resulting in a fully unified mechanized software development process. We carried out an exploratory research on Requirements Engineering [7,8], Natural Language Processing [9,10], and Program Synthesis [11,12] in order to put them all together as a way of producing executable code from requirements written in natural language with no manual human intervention between the intermediates tasks.

A proposed way to integrate Natural Language Processing and Program Synthesis is described as follows: firstly, a stakeholder writes a set of requirements using natural language in a common text editor. Secondly, those requirements are given as input to a translator able to map natural language text into sentences in logical form. Thirdly, those sentences are given to a synthesizer that is capable of generating source code. And finally, the task of translating source code to ready-to-run software is mechanically done by an off-the-shelf compiler (see Figure 2). The approach proposed is a vision for a faster, cheaper, more reliable, and more maintainable software production process.

The rest of this paper is organized as follows: Section 2 describes natural language processing into the context of Requirement Engineering. Section 3 describes Program Synthesis, Synthesizers, and Translators. Section 4 describes a way on how these technologies (Natural Language Processing and Program Synthesis) can be integrated in a unified process and summarizes a set of difficulties and contributions toward this technological integration. Section 5 concludes the paper.

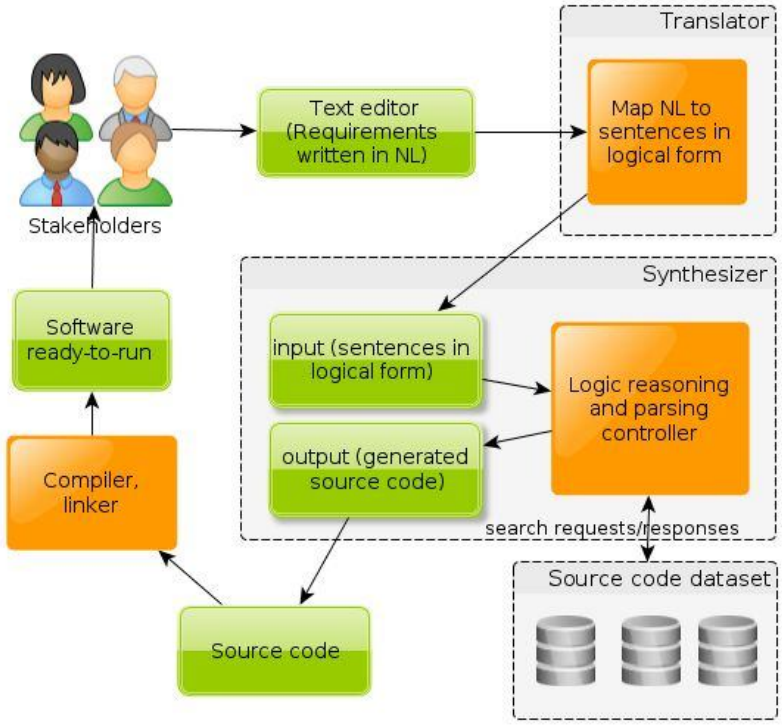

Figure 2: The proposed mechanized process.

2. NATURAL LANGUAGE PROCESSING INTO REQUIREMENTS ENGINEERING CONTEXT

This section defines what is Natural Language Processing into the context of Requirement Engineering. It also describes a usual process of transforming the human intent into software.

2.1. Natural Language Processing

Natural Language Processing (NLP) aim to make computers to “understand” the meaning of the language that humans naturally use [9]. NLP deals with the design and the build of software that

analyzes, understands, and generates languages that humans use naturally, so that eventually one will be able to address to your computer in the same way one addresses another person.

This goal is not trivial to reach. "Understanding" a language means, among other things, knowing what concepts a word or phrase stands for and knowing how to link those concepts together in a meaningful way. Natural language, the communication system that is easy for humans to learn and to use, is hard for a computer to master.

2.2. From human intent to ready-to-run software: an approach

Software development involves the process of transforming the human intent into software. Very often, future users of the system need to elicit this intent. The result is a requirements document which is written in natural language to make it understandable to the stakeholders. A set of models and artifacts are then manually extracted from the requirements. The requirements are refined into a software design which is then programmed as source code. Finally, source code is translated automatically into an executable code. In parallel, test cases are also built. Often, all of these steps are manual, except the final one that translates the source code to the executable code [1].

2.3. Requirements Engineering: a definition

A software system depends on how well it fits the needs of its users, customers, other stakeholders, and its environment. Software requirements comprise these needs, and requirements engineering is the process by which the requirements are determined. Requirements Engineering involves the understanding of the needs of the users, the customers, and the stakeholders. Moreover, understanding the contexts in which the software will be developed and will be used is also crucial. Modeling, analyzing, negotiating, documenting and validating the stakeholders' requirements and managing requirements evolution are also major challenges of Requirements Engineering. In general, the field of Requirements Engineering is decomposed into five types of tasks: elicitation, modeling, requirements analysis, validation and verification, and requirements management [7].

3. SYNTHESIZERS AND TRANSLATORS

This section describes Program Synthesis, Synthesizers, and Translators as a way to map natural language sentences into their equivalent logical representations.

3.1. Program synthesis: a definition

Program synthesis is the task of generating an executable program from the user intent expressed in the form of logical constraints. Unlike compilers, which take as input programs written in a well known programming language and perform syntax translations, synthesizers are able to accept a mixed form of constraints (such as natural language, input-output examples, partial or inefficient programs), and perform a search of the source code.

Program synthesis applications have an impact on end-users, non-skilled and skilled programmers. The synthesis technology can be deployed in several fields such as discovery of new algorithms, end-user's automation of repetitive programming, automated problem solving in teaching, and general assistance to a programmer, in addition to broader domains like transportation, health-care, etc [13].

3.2. Program Synthesis: a Synthesizer

A synthesizer is characterized by three processes: the user intent expressed in terms of logical representations (a set of constraints); a set of program templates over which a synthesizer searches, and the search technique it employs. The user intent can be expressed in the form of logical relations between inputs and outputs, input-output examples, demonstrations, natural language, and inefficient or similarly related programs [13].

3.3. Mapping natural language sentences into logical representations: a Translator

Some algorithms can map natural language sentences to their representations in logical form without taking into account the context of those sentences. In other words, those algorithms assess each sentence in isolation.

Sentence 1: to list bus departures from "Petrolina city" on Monday night.

```
Logical Form 1: λx.bus(x) ∧ to(x, petrolina city) ∧
  day(x, mon) ∧ during(x, night)
```

Sentence 2: to present bus departures after 7pm.

```
Logical Form 2: λx.bus(x) ∧ to(x, petrolina city) ∧
  day(x, mon) ∧ depart(x) > 1900
```

Other algorithms are able to map natural language sentences whose underlying meanings depend on the context they appear. For instance, Sentence 2 have to be determined based on the context established by previous sentences, such as Sentence 1 [14]. Mapping natural language sentences into context-dependent logical representations is a potential scenario for mechanize requirements.
A thirdly alternative is to design a controlled natural language: a language that is a subset of English and that resembles English as much as possible but that has a formal syntax. Controlled natural languages are amenable to translation without ambiguity to another formal language (like logic, Communicating Sequential Processes, Java, etc.) [15].

4. PUTTING ALL TOGETHER

This section describes briefly a way on how these technologies can be integrated in a unified process and summarize a set of difficulties and contributions toward this technological integration.

4.1. An approach to join Natural Language Processing and Program Synthesis

The proposed approach aims to reduce (or even to replace) the duties of Testers, Programmers, Software Architects, and Requirement Analysts. This is only possible once translators are able to map natural language into logical representations and synthesizers that take as input the user intent expressed in terms of logical representations and generate program source code become available.

Any stakeholder could start the software development process by writing a set of requirements using (controlled) natural language in a common text editor. The integration of natural language processing and program synthesis in order to get a ready-to-run software could be mechanized as follows: the requirements are given as input to a translator that maps natural language text to sentences in logical form. The sentences in formal logic are given as input to a synthesizer that generates source code. And finally, a compiler translates the source code into software read-to-run.

4.2. Main difficulties

Requirements Engineering is difficult. It describes what the system should be and how the environment affected by its introduction. In other words, Requirements Engineering is about defining precisely the problem that the software is to solve. The task of Requirements Engineering starts from a set of ill-defined ideas of what the proposed system should to be, which should evolve towards a single, detailed, technical, and unambiguous specification of such system. It is probably iterative and involves many players: ideally, we should put together those various backgrounds and expertise. Once requirements are defined, the remaining Software Engineering phases deal with refinement of the proposed software up to the point of getting an executable code [7].

Mapping context-dependent natural language is difficult. Each sentence has to be translated to a number of many possible contexts. An approach to accomplish that task is by using a probabilistic Combinatorial Categorical Grammar to solve an underspecified meaning representation by creating a sequence of modifications that depend on the previous basis; and applying a weighted linear model that is used to make a range of parsing and context-resolution decisions [14].

Program Synthesis is difficult. First of all, creating an interface that is able to take as input any sort of user intent without ambiguity to synthesize source code is a challenge in any real world's domain. Secondly, program synthesizers usually need to receive, besides the formal logic expressions, other kinds of input like code templates or input-output data samples. Code templates describe primitives structures (such as conditional or unconditional branches, simple or nested loops, program termination, exceptions, and so on.) in terms of their types, quantities, and their relationships. These templates guide a synthesizer on how the code must be built (or improved). Thirdly, the synthesizer needs to strike a good balance between expressiveness and efficiency of the search space. The better the expressiveness, the larger the search space. And finally, there is not a unified search technique able to deal with any domain. Therefore the performance of such search depends on the domain and the input constraints [13].

4.3. Important contributions

Important contributions in the field of Natural Language Processing have reached significant progress on both the task of mapping natural language into logic and of addressing the problem of context-independence:

(I) *Inductive Logic Programming* (ILP) is defined as the intersection of inductive learning and logic programming. Therefore, ILP employs techniques from both machine learning and logic programming. Machine learning uses tools and techniques to induce hypotheses from observations or examples and to synthesize new knowledge from experience. Logic programming provides the representational mechanism for hypotheses and observations formalism, such as its semantic orientation, and various well-established techniques [26];

(ii) *Statistical Machine Translation* (SMT) treats the translation of natural language as a machine learning problem. By examining many samples of human-produced translation, the statistical machine translation algorithms automatically learn how to translate [27].

Also, several authors have worked on the problem of context-dependent mappings from natural language to formal languages:

(I) *A fully statistical approach to natural language interfaces,* which is fully supervised and produces a final meaning representation in System Query Language (SQL) [28];

(ii) *Learning context-dependent mappings from sentences to logical form* is carried out from examples annotated with lambda-calculus expressions that represent only the final, context-dependent logical forms [14]; and
(iii) *Wide-coverage semantic parsing* implemented as a system that generates semantic representations with relevant background knowledge from texts and perform first-order inferences on the result. The key feature of this approach is its wide coverage achieving more than 95% [29].

The field of Program Synthesis have achieved significant progress on the task of synthesis of programs from user intent:

(i) *Deductive (constructive) synthesis* is the approach of successively refining a given specification using proof steps, each of which corresponds to a programming construct. By having the human programmer to guide the proof refinement, the synthesizer is able to extract the insight behind the program from the proof. In that approach the synthesizer is seen as a compiler from a high-level (possibly non-algorithmic) language to an executable (algorithmic) language, guided by the human [30];
(ii) *Schema-guided synthesis* takes a template of the desired computations and generates a program using a deductive approach. Some heuristic techniques for automating schema-guided synthesis have been proposed, but they yield a very limited schematic of programs, and therefore those techniques are limited in their applicability. Schema-guided synthesis specialized in arithmetic domain has been proposed using a constraint-based solving methodology [31];
(iii) *Inductive synthesis* is the approach of generalizing from instances in order to generate a program that satisfy all instances. The instances could be positive ones that define valid behavior or counterexamples that eliminate invalid behavior [30];
(iv) *Proof-theoretic synthesis* is a midway between deductive and schema-guided synthesis. The user provides a set of information through a synthesis scaffold (such as an input-output functional specification, a description of the atomic operations in the programming language, and resource constraints). This technique synthesizes a program, if there exists one, that meets the input-output specification and uses only the given resources. The synthesis algorithm works by creating a program with unknown statements, unknown guards, unknown inductive invariants (proof certificate for safety), and unknown ranking functions (proof certificate for termination). It then generates constraints that relate the unknowns, which we show that can be solved using of-the-shelf Satisfiability Modulo Theories (SMT) solvers. The feasibility of that approach by synthesizing programs was verified in three different domains (arithmetic, sorting, and dynamic programming) by synthesizing programs for complicated arithmetic algorithms including Strassen's matrix multiplication, Bresenham's line drawing, several sorting algorithms, and several dynamic programming algorithms [31].

The advances described above may, in future, provide us a way to enable the stakeholders to make software with little help from software developers. This is going to yield a faster development and a cheaper, less error-prone, and more mechanized software production process. However, the integration of these technologies and the evolution of them into an industrial-scale technology is currently a grand challenge.

5. CONCLUSIONS

We presented a vision on how the software development process could become a fully unified mechanized process by combining the benefits from Natural Language Processing in the context of Requirements Engineering (Section 2) and Program Synthesis (Section 3).

In our vision, humans use natural language to describe a problem for building a software solution and gives such description to a computer. With the push of a button, a software ready to execute is produced. Unfortunately, despite the great progress seen on the natural language processing and the program synthesis fields in the last decade, this vision is currently a non trivial task to accomplish.

We do not know exactly if (and when) science will ever reach the proposed vision in its entirety, but many of the current obstacles have been addressed by important contributions from both fields, which makes our vision closer to reality.

ACKNOWLEDGMENTS

The authors would like to thank all those who encouraged, advised and supported this work, extend our deepest appreciation.

REFERENCES

[1] Walter F. Tichy and Sven J. Koerner (2010) Text to software: developing tools to close the gaps in software engineering. In Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER '10). ACM, New York, NY, USA, 379-384. http://doi.acm.org/10.1145/1882362.1882439.

[2] Nuno Ramos Carvalho. (2013) An ontology toolkit for problem domain concept location in program comprehension. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). IEEE Press, Piscataway, NJ, USA, 1415-1418.

[3] Umber, Ashfa; Bajwa, ImranSarwar; Asif Naeem, M. (2011) NL-Based Automated Software Requirements Elicitation and Specification. Advances in Computing and Communications. Communications in Computer and Information Science Volume 191, 2011, pp 30-39. Springer Berlin Heidelberg. http://dx.doi.org/10.1007/978-3-642-22714-1_4.

[4] Seresht, Shadi Moradi, and Olga Ormandjieva. (2008) Automated assistance for use cases elicitation from user requirements text. Proceedings of the 11th Workshop on Requirements Engineering (WER 2008). Vol. 16.

[5] Suresh Thummalapenta, Saurabh Sinha, Nimit Singhania, and Satish Chandra. (2012). Automating test automation. In Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012). IEEE Press, Piscataway, NJ, USA, 881-891.

[6] Pablo Pedemonte, Jalal Mahmud, and Tessa Lau. (2012). Towards automatic functional test execution. In Proceedings of the 2012 ACM international conference on Intelligent User Interfaces (IUI '12). ACM, New York, NY, USA, 227-236. http://doi.acm.org/10.1145/2166966.2167005.

[7] Cheng, Betty HC, and Joanne M. Atlee. (2007) "Research directions in requirements engineering." 2007 Future of Software Engineering. IEEE Computer Society.

[8] Bashar Nuseibeh and Steve Easterbrook. (2000) Requirements engineering: a roadmap. In Proceedings of the Conference on The Future of Software Engineering (ICSE '00). ACM, New York, NY, USA, 35-46. http://doi.acm.org/10.1145/336512.336523

[9] Indurkhya, Nitin, and Fred J. Damerau, eds. (2010) Handbook of natural language processing. Vol. 2. Chapman and Hall/CRC.

[10] Harald Wahl, Werner Winiwarter, and Gerald Quirchmayr. (2010) Natural language processing technologies for developing a language learning environment. In Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services (iiWAS '10). ACM, New York, NY, USA, 381-388. http://doi.acm.org/10.1145/1967486.1967546.

[11] Srivastava, Saurabh, Sumit Gulwani, and Jeffrey S. Foster. (2010) "From program verification to program synthesis." ACM Sigplan Notices. Vol. 45. No. 1. ACM.

[12] Zohar Manna and Richard J. Waldinger. (1971). Toward automatic program synthesis. Commun. ACM 14, 3 (March 1971), 151-165. http://doi.acm.org/10.1145/362566.362568.

[13] Gulwani, Sumit. (2010) "Dimensions in program synthesis." Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming. ACM.

[14] Zettlemoyer, Luke S., and Michael Collins. (2009) Learning context-dependent mappings from sentences to logical form. Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2-Volume 2. Association for Computational Linguistics.
[15] Gustavo Carvalho, Diogo Falcão, Flávia Barros, Augusto Sampaio, Alexandre Mota, Leonardo Motta, and Mark Blackburn. Test Case Generation from Natural Language Requirements based on SCR Specifications. Proceedings of the 28th Annual ACM, pages 1217-1222.
[16] D. K. Deeptimahanti and R. Sanyal. (2009) An innovative approach for generating static UML models from natural language requirements. In Advances in Software Engineering, volume 30 of Communications in Computer and Information Science, pages 147–163. Springer Berlin Heidelberg.
[17] Ilieva, M. G., and Olga Ormandjieva. (2005) "Automatic transition of natural language software requirements specification into formal presentation." Natural Language Processing and Information Systems. Springer Berlin Heidelberg. 392-397.
[18] Gnesi, S., Lami, G., Trentanni, G. (2005) An automatic tool for the analysis of natural language requirements. CSSE Journal 20(1), 53–62.
[19] Franců, Jan, and Petr Hnětynka. (2011) "Automated generation of implementation from textual system requirements." Software Engineering Techniques. Springer Berlin Heidelberg. 34-47.
[20] Soley, Richard. (2000) "Model driven architecture." OMG white paper 308.
[21] SPARX SYSTEMS. (2007). MDA Overview. [Online]. Available at: http://www.omg.org/mda/mda_files/MDA_Tool-Sparx-Systmes.pdf >. Accessed: 10/Jul/2913.
[22] Usman, Muhammad, and Aamer Nadeem. (2009) "Automatic generation of Java code from UML diagrams using UJECTOR." International Journal of Software Engineering and Its Applications 3.2: 21-37.
[23] de Figueiredo, André LL, Wilkerson L. Andrade, and Patrícia DL Machado. (2006) "Generating interaction test cases for mobile phone systems from use case specifications." ACM SIGSOFT Software Engineering Notes 31.6: 1-10.
[24] Schwitter, Rolf, and Norbert E. Fuchs. (1996) "Attempto-from specifications in controlled natural language towards executable specifications." arXiv preprint cmp-lg/9603004.
[25] Hinchey, Michael G., James L. Rash, and Christopher A. Rouff. (2005) "Requirements to design to code: Towards a fully formal approach to automatic code generation." NASA tech. monograph TM-2005-212774, NASA Goddard Space Flight Center.
[26] Stephen Muggleton, Luc de Raedt, Inductive Logic Programming: Theory and methods, The Journal of Logic Programming, Volumes 19–20, Supplement 1, May–July 1994, Pages 629-679, ISSN 0743-1066, http://dx.doi.org/10.1016/0743-1066(94)90035-3.
[27] Adam Lopez. 2008. Statistical machine translation. ACM Comput. Surv. 40, 3, Article 8 (August 2008), 49 pages. DOI=10.1145/1380584.1380586 http://doi.acm.org/10.1145/1380584.1380586]).
[28] Scott Miller, David Stallard, Robert J. Bobrow, and Richard L. Schwartz. 1996. A fully statistical approach to natural language interfaces. In Proc. of the Association for Computational Linguistics.
[29] Johan Bos, Stephen Clark, Mark Steedman, James R. Curran, and Julia Hockenmaier. 2004. Wide-coverage semantic representations from a CCG parser. In Proceedings of the 20th international conference on Computational Linguistics (COLING '04). Association for Computational Linguistics, Stroudsburg, PA, USA, , Article 1240 . DOI=10.3115/1220355.1220535 http://dx.doi.org/10.3115/1220355.1220535.
[30] Basin, D., Deville, Y., Flener, P., Hamfelt, A., & Nilsson, J. F. (2004). Synthesis of programs in computational logic. In Program Development in Computational Logic (pp. 30-65). Springer Berlin Heidelberg.
[31] Srivastava, S. (2010). Satisfiability-based program reasoning and program synthesis. Chapter 4. 76-105p. Chapter 8. 142-149p. PhD dissertation.

Authors

Alexandre R.S. Correia received his B. Sc. in Civil Engineering from the Federal University of Pernambuco – Brazil (1995) and his M. Sc. in Computer Science from the University of Coimbra – Portugal (2010). He is a PhD student in Computer Science at the Federal University of Pernambuco – Brazil. He is an assistant professor at the Federal Institute of Education, Science and Technology at Sertao Pernambucano – Brazil. He is interested in Software Engineering and formal methods.

Juliano M. Iyoda received his B. Sc. degree in Computer Science from the Federal University of Pernambuco – Brazil (1997), M. Sc. in Computer Science from the Federal University of Pernambuco – Brazil (2000), and PhD. in Computer Science from the University of Cambridge – United Kingdom (2007). He is an assistant professor at the Federal University of Pernambuco – Brazil. He is interested in Formal Methods applied to hardware and software design, automated reasoning and Testing.

Carla T.L.L. Silva received her B. Sc. degree in Computer Science from the Federal University of Campina Grande – Brazil (2001), M. Sc. in Computer Science from the Federal University of Pernambuco – Brazil (2003), and PhD. in Computer Science from Federal University of Pernambuco – Brazil (2007). She is an assistant professor at the Federal University of Pernambuco – Brazil. She is interested in Software Engineering and Requirements Engineering applied to model driven software design.