A Survey of Naturalistic Programming Technologies

OSCAR PULIDO-PRIETO and ULISES JUÁREZ-MARTÍNEZ, Instituto Tecnológico de Orizaba

Mainly focused on solving abstraction problems, programming paradigms limit language expressiveness, thus leaving unexplored natural language descriptions that are implicitly expressive. Several authors have developed tools for programming with a natural language subset limited to specific domains to deal with the ambiguity occurring with artificial intelligence technique use. This article presents a review of tools and languages with naturalistic features and highlights the problems that authors have resolved and those they have not addressed, going on to discuss the fact that a "naturalistic" language based on a well-defined model is not reported.

CCS Concepts: • **Software and its engineering** → **Very high level languages**; **Syntax**; **Semantics**; • **Theory of computation** → *Grammars and context-free languages*;

Additional Key Words and Phrases: Naturalistic programming, controlled natural english, expressiveness, automatic source code generation

ACM Reference format:
Oscar Pulido-Prieto and Ulises Juárez-Martínez. 2017. A Survey of Naturalistic Programming Technologies. *ACM Comput. Surv.* 50, 5, Article 70 (September 2017), 35 pages.
https://doi.org/10.1145/3109481

1 INTRODUCTION

For many years, researchers have sought to use natural languages (NL) in the development of software; however, they have had to face the fact that computers cannot deal with the ambiguity that traditional programming languages resolve with formal specifications. While the human cognitive process helps us to resolve ambiguity, computers lack this ability, thus comprising the main gap between natural languages and those used in software development. The Cognitive process helps humans to understand and distinguish not only verbs from nouns but also meaning in a particular context (Liblit et al. 2006). As a solution, several authors propose various ways of obtaining a natural-like programming language that is also restricted enough to be formalized to be processable by computers.

In recent years, the gap between humans and computers has been reduced through the emergence of various paradigms, languages, and tools that facilitate software development. Despite these technological advances, limitations remain that affect software maintenance and evolution. The first problem comes when, to read programming, programmers must learn to understand

This research was supported by the Consejo Nacional de Ciencia y Tecnología (CONACyT).
Authors' addresses: O. Pulido-Prieto and U. Juárez-Martínez, Department of Posgraduate and Research, Instituto Tecnológico de Orizaba, Av. Oriente 9 No. 852, Col. Centro, Zip 94320, Orizaba, Veracruz, México; emails: {opulidop, ujuarez}@ito-depi.edu.mx.

https://doi.org/10.1145/3109481

code written by others, where, self-documented code aside, the best scenario is code accompanied by detailed documentation about how it works and clear description of every line. Lacking the required discipline or working under tight delivery schedules, many programmers see this as a waste of time and prefer to document code solely when they think it necessary, meaning that this ideal scenario often does not always occur. Furthermore, ideas must generally be adapted to a specific programming language and even a specific paradigm. The second problem is the lack of expressiveness that would enable the description of abstractions from a particular domain, which domain-specific languages (DSL) deal with by enabling specific tasks to be translated into general-purpose languages (GPL) and domain abstractions to be interpreted easily. The third problem stems from the inability to correctly and flexibly interpret abstractions, which is addressed by new paradigms that treat correctness and flexibility separately. The main challenge to improving GPL expressiveness is keeping them generic enough to encompass all domains, while also maintaining clearly defined abstractions.

Although DSLs are expressive, they are focused on a particular domain; on the other hand, GPLs cover all domains with relative efficiency but lack the expressiveness of DSLs. Based on the idea that natural languages refer not only to abstractions but also the language structure itself, naturalistic programming can be used with existing paradigms to describe instructions in a more expressive way. In Pane (1996), authors present a report describing various problems that novice programmers report having to deal with when learning to program, and, having found no formal and organized research in this area, performed an extensive review, with the main problem identified being the match between the system and the real world. Authors found that a program designer must choose a correct anaphor consistent between the real world and the domain problem, while users generally experience problems attempting low-level compositions due to psychological problems such as code synthesis or apparently unrelated primitives. Moreover, natural language users generally avoid describing situations that are expected to be inferred, and prefer a "go to," such as notation for bifurcations, preferring to repeat an instruction N times rather than use iterations. This article reviews proposed and commercial languages that cover several naturalistic approaches. Section 2 describes fundamental concepts, while Section 3 analyses reported technologies with naturalistic elements, and Section 4 performs an analysis of naturalistic technologies. Section 5 presents the open issues that need to be researched to improve naturalistic programming, and, finally, Section 6 presents the conclusions of this study.

2 BACKGROUND

As the first programs were written with binary numbers, their expressiveness was null. To solve this problem, an intermediate language that gives meaning to those instructions was found, thus giving birth to assembler languages, which enabled programmers to create programs with which they could manipulate variables and record that which they then translated to executable code. The expressiveness level at the time was insufficient for programmers to break down its ideas to convert them into instructions. The so-called *third-generation programming languages* were created to solve this problem through the use of verbs, which are natural language concepts that represent the concrete task that a program performs. Later, object-oriented programming (OOP) languages used nouns as instance identifiers, *objects*, which enable programmers to define real-world abstractions that perform specific tasks defined by verbs. Thus, programmers were then able to work with simple sentences (Booch et al. 2007).

Naturalistic programming is related to linguistic elements, such as deixis and complex phrases, which are taken to provide a level of expressiveness that is closer to natural language. The following section describes these concepts.

2.1 Naturalistic Programming

In Lopes et al. (2003), authors define naturalistic programming as the use of natural languages elements to design more expressive programming languages from a human perspective. In the same article, they address the advantages and disadvantages of using programming languages and natural languages for software development, proposing, as a solution, a median point—a formal programming language that possesses elements of natural languages, such as anaphoric references. The *Oxford English Dictionary* (Press 2016) defines "Naturalistic" as

> Copying the way things are in the natural world.

From this definition, a naturalistic thing can be inferred as being artificial and designed to be similar to its natural counterpart. Programming languages are not designed to be similar to natural languages, instead designed to be representations of formal models that enable a computer to process instructions.

According to Clark et al. (2010), another detail to be considered is that a programming language is based on a formalized natural representation that could be classified in two categories: formalistic or naturalistic.[1]

A language is naturalistic when it represents more simple natural language, while it is formalistic when it represents a formal and deterministic implementation of natural language. Formalistic languages reduce ambiguity to a minimum, while naturalistic languages maintain ambiguity and resolve it in a number of different ways. The aim of formalistic languages is to be correctly understood by users, while naturalistic languages aim to be interpreted by computers dealing with ambiguity through heuristic translations to eventually give support to a full natural language.

Finally, programmatic semantics are procedural relations inferred from linguistic structures (Liu and Lieberman 2005b). Nouns are associated with data structures, verbs with functions, and adjectives with properties (Mihalcea et al. 2006). The principles of programmatic semantics are as follows: *syntactic features* that provide differentiation between nouns and verbs and enable the semantic understanding of an expression; *procedural features* that consist in how to express rules for conditionals, iteration, or recursion and are divided into subjunctive and conditionals, possibility, and the concept of "how," which works as a base to create conditional expressions; *relational features and set theory*, represented as grammatical constructions that enable the description of a set of steps and loops; and, *representational equivalence*, which enables entities to be represented through narrative.

2.2 Deixis

The *Oxford English Dictionary* defines "deixis" as

> The function or use of deictic words or expressions whose meaning depends on where, when, or by whom they are used.

The term *Deictic* indicates that a sentence or phrase depends on the context in which it is used. For example, in the sentence *the list is complete*, a list is described, but its meaning depends on the context in which the sentence is used, thus the list could be a guest list, a data structure, or a shopping list. Deixis is present in virtually every natural language of the world, because it enables the expression of ideas and reduces word number to a minimum. Such sentences will contain ambiguity that humans resolve through the use of their cognitive process, unconsciously, using reasoning to complement partial ideas (Liblit et al. 2006). Deixis is related to *ellipsis*,[2] because we

[1] In Clark et al. (2010), authors use "formalist" and "naturalist" instead of "formalistic" and "naturalistic."
[2] Ellipsis is defined as the omission of words without losing the context of the sentence (Lobeck 2007).

omit redundant words to produce sentences such as *I will go to the classroom and you too*, from *I will go to the classroom and you will go to the classroom too*, thus shortening sentences without losing semantic meaning.

Jaaa is a programming language that uses anaphora in a concrete and particular manner. Jaaa was born as a Java dialect that uses an anaphor limited to one parameter per method (Lohmeier 2011). The author defines direct anaphora as a reference to a previously described element, where, for example, *it* or *those* would address elements such as *the hard disk*, if it had previously been mentioned. On the other hand, he defines an indirect anaphora as a non-explicit textual relationship found in the user's mental model, with an expression such as *a computer part* implied by *the hard disk*.

Through deixis, we replace the *signifier*[3] with a generic meaning that varies according to a particular context, thus ambiguous but shorter sentences are used even when the context depends on the direct reference and its associated referent.

Another feature of indirect references is the ability to describe things not only in space, but also in time, which has been dealt with by logic programming as an extension where several languages provide temporal features to enable the verification and specification of programs in terms of time (Gaintzarain and Lucio 2009). The following temporal unary operators are described: *always* indicates a condition that is forever true; *next* indicates a condition that is true in the incoming state; *eventually* indicates a condition that must be true some time in a future state; and *all* indicates a condition that is forever true (like *always*). The binary operators *until* and *release* are also described.

Moreover, modal extensions that include possibility and necessity are reported and are used in mathematics and artificial intelligence.[4]

2.2.1 Endophora. An endophora is an indirect reference defined in the same text and is divided into two classifications: *anaphora* and *cataphora*.

- —Anaphora. Indirect reference whose referent has been previously defined in the same text: "Take the key, use it in the lock."
- —Cataphora. Indirect reference whose referent is subsequently defined in the same text: "After taking it, use the key in the lock."

2.2.2 Exophora. Exophora is something defined in another text, a particular type of which, called a *homophora*, is a reference that depends on a particular context. One example is the *Integer* type, which is conceptually defined as a data type used to manipulate integer numbers, but with particular features that depend on a specific programming language or the machine in which a program is executed.

2.3 Expressiveness

The *Oxford English Dictionary* defines expressiveness as

> Showing or able to show your thoughts and feelings.

Expressiveness, therefore, is a natural language feature that allows us to describe ideas as precisely as we want. In contrast with natural language, GPLs use formal specifications that, as

[3]In semiotics, a signifier describes a sign component, which is a conjunction of three elements: *referent*, which is a real or abstract thing; *signifier*, which is the referent in the symbolic representation relation; and, *signified*, which is the mental idea to which the referent corresponds (Ogden et al. 1923). However, Ferdinand de Saussure says that, as there are signs without referents, he only considers signifier and signified (De Saussure 1916).

[4]In Orgun and Ma (1994), an analysis of temporal logic, modal logic and multi-modal logic is presented.

formalized languages, lack the expressiveness of natural language. While DSLs enable a higher level of expressiveness, these are limited to particular domains. DSLs are not programming limited, but they do help non-programmers to develop domain-specific tasks without having to deal with GLPs and use particular-domain elements. In van Deursen et al. (2000), authors report 75 DSL-related publications about programming, software engineering, multimedia, and telecommunication.

The most notable barrier to human communication is the fact that there is no single language in the world, which has led to the creation of various controlled English languages since 1930. These languages, covering diverse areas such as manuals for construction tools or commercial aviation, have the common feature of being controlled versions of the English language that release users from ambiguous elements and regionalisms. As these languages are limited to particular areas, they employ the slang used in their domains to avoid misunderstanding.[5]

In O'Brien (2003), eight controlled English languages were reviewed to extract a common set of rules, from which the authors identified just one as a human-oriented controlled language (HOCL) with the rest identified as computer-oriented controlled languages (COCL). As a result, the following classification for three types of rules is presented:

—Lexical. Focused on choosing words and how meaning is affected.
—Grammatical. Focused on affecting syntax.
—Textual. Focused on affecting design or word information (structural) or affecting text purpose and user response to it (pragmatic).

As a result, all eight of these languages share only 1 rule from the 61 obtained, while 7 common rules are shared by at least 4 of them.

In Miller (1981), the author describes how natural language specifications are translated into programming expressions in his study.

Miller concludes the following: natural languages do not possess the explicit declarations of programming languages; natural languages have implicit references, while programming language references are explicit; indexing in natural languages occurs using words such as "previous" and "next," while this is performed in programming languages using numbers and variables; programming languages have explicitly defined format specifications while natural languages are contextual; and, there is a high volume of defined data in programming languages, while there is no distinction between them in natural languages.

In reference to control structures, Miller concludes the following: the extent of control structures is a major factor in programming languages but rare in natural languages; conditional blocks are a major factor in programming languages, while natural languages only function with "if-then" conditions, thus avoiding the "else" part; programming languages possess mechanisms that detect exceptions, mechanisms that are not present in natural languages; natural languages have linear block structures, while programming languages feature recursion, co-routines, and non-linear structures, among others; natural languages have structure calls as a major control mechanism, which are specified by context, while, in programming languages, these are frequent and completely specified; and, in natural languages, the argument passing feature is implicit, while it is explicit in programming languages.

Miller concludes the following about the general features of languages: programming languages have very limited support for lexicons, while natural languages have many synonyms that can be restricted; programming languages have imperative and conditional sentences, while natural languages also have declarative sentences; and, programming languages have a rigid syntax, while

[5]In Kuhn (2014), the author reviews 100 natural controlled English languages from several domains.

natural languages have an extremely variable and complex syntax that depends on the cognitive process.

2.4 Phrases

Phrase complexity is another element highlighted, because phrases integrated into a naturalistic language provide the ability to create more complex instructions with a high level of expressiveness from a programmer perspective, unlike formal programming languages where phrases are decomposed to leave simple nouns and verbs. There are two basic types of phrases: noun phrases and verb phrases (Bolshakov and Gelbukh 2004).

A *Noun phrase* is a word set that represents the sentence subject and direct and indirect verb objects and that enables a property-based description for an abstraction. Thus, instructions can be defined in terms of properties. Noun phrases can also be used to inquire about hierarchy, so phrases like *a house with a garden* can be a *House* type abstraction with a *Garden* property—a *HouseWithGarden* or even a mixture of both where a *house with a garden and pool* can be described as a *HouseWithGarden* type with a *Pool* property.

2.5 Anaphoric Relations

In natural languages, abstractions are not only classified according to their type, but these also using their properties, so phrases like *the house* and *the house with garden* not only identify two different abstractions but also the same abstraction where the second example has a more detailed description. Furthermore, naturalistic languages require explicit typing, because structural anaphors are type-based for efficient expressiveness, thus indirect references like *the house described in this paragraph* are valid. A higher expressiveness level is achieved using full support for noun phrases, implying modifiers, determinants, and extensions that enable not only a hierarchy limited classification but also one based on instance properties (Knöll et al. 2011).

An anaphora allows efficient support for indirect references but only considering the latter ones. The optimal manner of achieving full support is to consider support for full deixis, where indirect references are considered irrespective as to whether these are pre- or post-definition references or whether they are defined in the same or in another text.

Phrases are another element used by natural languages, as they enable an instruction to be described in more detail. Verbs and nouns are considered by OOP, while simple sentences are considered by aspect-oriented programming (AOP). The drawback of AOP can be found in the limited support that a phrase can provide, as a phrase with stronger support could enable more detailed descriptions in a naturalistic language. For example, the action described in the sentence "the neighbor's dog barks when an explosion happens" is based on the capabilities and limitations of language:

- —Functional. A function *bark* is called from another function, receiving a string with the owner's name and the occurrence of an explosion (as a Boolean value). The caller function invokes *bark* if both the owner and the occurrence of the explosion are true.
- —Objects. A *Dog* class with a *bark* method is defined. This method must be validated through the occurrence of an explosion and whether or not the dog owner is a neighbor, to validate whether or not a dog barks.
- —Aspects. The *Dog* definition is kept, but the explosion and dog owner are validated in an aspect with a pointcut that works with explosion, calling *bark* after the explosion occurs, so a *Dog* instance must be kept in the aspect to use it without relating *bark* with *explosion* directly.

—Naturalistic. Unlike other paradigms, natural language expressiveness is sought, in that the main description "dog barks" and crosscutting concerns (validate the dog ownership and the occurrence of the explosion) can be described in the same instruction, which is also separated from verb definitions.

2.6 Context

Context is a set of proven circumstances that occur around an event. In software development, there is a close relationship between the problem domain and the problem to be solved. While many tools are focused on a particular context, leaving others to one side, GPL must cover all contexts so that they have abstractions and syntax that cover all domains. Because of this, a programmer is able to increase code when another context is required; however, the previously written code will be affected or modified by the user and this results in a lost context. To deal with this problem, language designers must provide a clear abstraction definition to avoid loss of context, or use solely the elements required by the context. While a solution consists of making layer definitions that can be activated at any time for every context (Hirschfeld et al. 2008), this has the disadvantage that these layers must be explicitly activated, for which an event mechanism is required (Kamina et al. 2010).

On the other hand, DSL cannot respond to changes in the system context, because these represent complex semantics with simple solutions. Complex semantics result in languages that lack the flexibility required to respond to the changes made when domain context is changed. In Laird and Barrett (2009), authors propose an adaptable method to avoid these problems that occur with DSLs, where an adaptable part is used to decouple implementations from intentions to obtain dynamically adaptable systems after these are deployed using a component-based solution involving a dynamic reconfiguration of interpreter components. This DSL dynamic interpretation model offers a high level of expressiveness for developing specialized applications.

2.7 Ambiguity

Ambiguity is a natural property of language that describes a word or sentence that has many interpretations, which vary according to a particular context. While we use our reasoning to solve ambiguity, computers cannot do this. Thus, several mechanisms have been developed that enable a computer to resolve ambiguity when analyzing sentences from natural language, as an ambiguous description can result in unexpected behaviors. There are many mechanisms proposed to formalize a natural language description and thus make it understandable by a computer without any ambiguity.

Anaphora resolution is a technique that has been researched for some years and comprises the process of how to identify the anaphora referent through a conceptual link (Qiu et al. 2004). In Arnold and Lieberman (2010a), authors propose the use of dialogs for the treatment of ambiguous ideas, where, while the user refines an idea, they do not propose a model, instead proposing a mechanism to generate code in an existing programming language from a natural language description. Arnold and Lieberman (2010b) describe the *Zones* tool that uses code documentation to find reusable code through imprecise sentences. *Zones* has a back-end named *ProcedureSpace* that allows it to find code-sentence pairs associated with context-dependent fragments. Other tools limit ambiguity, using data dictionaries such as *Pegasus* and *Macho*, or establishing keywords to maintain formal and strict control.

Begel and Graham (2006b) propose an extended generalized left-to-right (XGLR) algorithm to deal with the ambiguity present in the embedded languages used in voice programming. This analysis consists in two grammars, one with unique tokens and another with multiple tokens. This algorithm uses several generalized left-to-right (GLR) instances that analyze several LR instances

to process the ambiguity of these instances separately. The *extended* part comes from the ability to deal with homophone words.

3 NATURALISTIC TECHNOLOGIES

This section presents a discussion about the different technologies focused on the development of software using a formalized English subset. The term "technologies" is used because it includes not only languages, but also frameworks and integrated development environments (IDE). Among these technologies are software-generation tools such as Pegasus, Natural Language Computer, Macho, Metafor, to name but a few, programming languages such as Natural Java, or documentation-focused languages such as Attempto Controlled English. The focus on solving particular problems is a feature that, notably, limits implementation to particular situations in a similar way to a DSL or, to be more precise, fourth-generation languages (4GL), which are enterprise-oriented and based on an expressive syntax (Heering and Mernik 2002; Mernik and Žumer 2001).

3.1 FLOW-MATIC

FLOW-MATIC (also known as B-0) is a programming language developed for the UNIVAC I in 1955, with languages like Common Business Oriented Language (COBOL) inspired by its English-like syntax (Wexelblat 1981). FLOW-MATIC divides instructions into two sections: English-like instructions and a particular data definition syntax.

As it was intended to close the gap between programming and natural languages, developers had, up to that point, used mathematical notation to write code. When designing FLOW-MATIC, Grace Hopper proposed the use of English for executable instructions and printed forms for the definition of data to work on them separately.

3.2 COBOL

COBOL is a business-focused structured programming language, the principal features of which are English-like imperative and procedural language, with an object extension added during the 1990s and standardized in 2002 (Lämmel 1998). In addition to these features, several modifications are necessary when a company needs to adapt COBOL to particular specifications or when an English-like expressiveness is required.

Many COBOL-reserved words are synonyms or enable defining expressions with different quantities. Another COBOL feature is that some redundant elements can be omitted, for example, *a IS GREATER THAN b* can be simplified to *a GREATER THAN b* or even as $a > b$. A *DISPLAY* word before a message is used when a value is presented on screen. A previously defined variable can be assigned using the syntax *MOVE ORIGIN-VAR TO DESTINYVAR* where the action is indicated by *MOVE* and, as is shown, a relation between variables compared to an action is established by the *TO* preposition. Whereas variables are described by nouns, actions are described by verbs. COBOL uses classes to create instances as a mechanism to deal with object-oriented programming that is tasked with creating interfaces between object-oriented languages using COBOL as a back-end.

Despite its benefits and widespread business use, COBOL syntax is imperative, procedural, and object-oriented when necessary. While the incorporation of an aspect-oriented paradigm into the language could have marked the greatest step towards the establishment of a naturalistic language, it was only reported as a conceptual proposal (Lämmel and De Schutter 2005). Therefore, while COBOL lacks the required features of naturalistic language, such as indirect references and temporality management, it does have a significant expressiveness level and sufficient formality for system development regardless of domain.

3.3 Heidorn's Dialog Language

In Heidorn (1973), the author studies an experimental natural language system that "converses" with the user about a problem to solve it. In this system, the user defines problems using natural language statements; then, if the problem cannot be solved, the system requests more statements from the user. When it has sufficient statements, the system takes a moment to solve the model to be executed, after which, the user can either request, by means of text in English, analysis as to whether the system understood the entry statements and whether they were correctly processed, or a representation in GPSS.[6] This system is based around simulating an English-like dialogue with the user to produce a DSL implementation.

3.4 Structured Query Language (SQL)

Structured Query Language (SQL) is a DSL whose focus is database creation and handling (Kim 1982). Originally, SQL was intended to be named *SEQUEL*, but its name was changed due to copyright issues (Chamberlin and Boyce 1974). SQL was intended as a declarative language with an expressiveness level that allows a clear and simple way to define database elements. These elements are: entity, which is reflected as a table; attribute, which is represented as a table field; relation, which is an association between two or more tables; and registry (or tuple), which is the data kept in the database. A sentence defines a particular action, which can be the creation of a table (with keys and relations between tables), an insert, a delete, a modification, or a data query. As SQL has an expressiveness level that is higher than traditional programming languages, it has an easy and clearly defined declarative syntax, due to the fact that a pure SQL lacks the control structures that other GPLs have. While several programmers have developed SQL extensions that provide supporting procedures, these are generally restricted to a particular environment and are not interchangeable. A simple SQL example would be

```
SELECT id, name, age, wage
FROM workers_db
WHERE position = 'manager'
```

As can be seen, this example is very expressive and easy to understand: we will obtain a query with the id, name, age, and wage of all managers.

It is noteworthy that in I. Androutsopoulos and Thanisch (1995), authors reviewed how natural language interfaces could enable the generation of SQL queries from a controlled natural language input that they named *natural language interfaces to database* (NLIDB), identifying advantages such as the use of a natural language instead of an artificial one that uses question-like queries and supports some level of elliptical sentences. These authors describe such disadvantages as: a very limited subset of natural language; the fact that the system cannot understand elements such as conjunctions or false positives and lacks human conceptual and linguistic abilities; and, that the user could assume that he is dealing with an intelligent system. In Nihalani et al. (2011), authors also present a historical review of several NLIDBs.

3.5 Natural Language Computer (NLC)

In Biermann and Ballard (1980), authors present Natural Language Computer (NLC), a programming language focused on performing operations over arrays with a limited English subset. While NLC solves expressiveness problems using natural language to program, there was no technology at that time that provided semantic construction capabilities, thus NLC users cannot generate

[6]Gordon's Programmable Simulation System (GPSS) is a general purpose programming language used in discrete time simulations (Schriber 1990).

system-acceptable expressions, because natural languages have a broad range of rules and word sets.

The authors propose two rules for the use of natural languages as programming languages:

—Only in-screen data structures are referenced when an instruction is written with simple sentences.
—Imperative verbs for instructions.

The authors describe English as being as precise as a programmer wants it to be. NLC decomposes sentences and compares tokens with either data dictionary or user-defined entities, and then performs an orthographic correction and checks for abbreviations and ordinals. NLC processes nouns, checking phrases and words that alter meaning to create an element set that refers to screen-displayed structures. NLC analyzes imperative verbs without processing them, updating them on screen. If a verb cannot be processed, then it is passed to a semantic module where some predefined verbs are processed with user-defined verbs.

A pair feature set consists in a dictionary that establishes whether a word is a verb, a pronoun, an adjective, or a noun. NLC also has spellcheck support to suggest alternative options.

The user describes an executed action as an imperative verb, a noun phrase (a noun with its determinant) and a *verbicle* (defined by authors as an imperative-verb-associated preposition), for example: *multiply X by Y*, where *by* is the verbicle. A sentence *add X to Y* is processed by the parser in the following order:

```
imperative, noun phrase, verbicle, noun phrase, end (period).
```

For conjunction, NLC uses a MIX A expression, as shown below:

```
A, separator (comma), A, separator (comma), (optional) "and", A.
```

NLC was tested by 23 students after they had received training. From 1,581 instructions, 81% were processed immediately, while, from the remaining 19%, half were rejected by system limitations and the other half by human error. In conclusion, 300 words were nearly sufficient to solve problems, with only eight words detected as missing. Incomplete definition words were found, because implicit elements were involved in more natural forms. Correctly processed instructions were identified at between 70% and 90%. Error messages were incomplete. Finally, a refinement is necessary for processor quantification.

3.6 Layered Domain Class (LDC)

In Ballard and Lusth (1983), authors present Layered Domain Class (LDC), which is a medium-sized language focused on enterprise data queries and learning new English domains. A domain is divided into layers if it has an NLC matrix comprising a structural relation where entities conform a lower level (for example, a calendar has months, days and weeks as low level entities). LDC has two components: Prep (performed by the knowledge acquisition component) and User-Phase (performed by the User-Phase processor). Prep establishes domain entities, relations, and associated nouns that the user must provide, and data structures such as involved entities. It also establishes whether these are or are not composed, whether or not an entity is plural, or whether it has ordinals. Later, Prep requests nouns to verify entities and verbs and deal with them. During User-Phase, LDC receives words and compares them with others stored in the dictionary that was created during Prep execution, and then creates a syntactic structure called *bubble*, which is the formal representation for user queries.

LDC supports several modifiers: ordinals (*the second floor*); superlatives (*the largest office*); adjectives (*the large room*); noun modifiers (*conference rooms*); subtypes (*offices*); argument-taking nouns

(*classmates of Jim*); prepositional phrases (*students in CPS215*); comparative phrases (*students better than Jim*); trivial verb phrases (*instructors who teach AI*); implied-parameter verb phrases (*students who failed AI*); operational verb phrases (*students who outscored Jim*); argument-taking adjectives (*offices adjacent to X-238*); and negations (*the non-graduated students*). LDC also has support for solely anaphorical syntax modifiers, such as: anaphoric comparatives (*better students*); anaphoric argument-taking adjectives (*adjacent offices*); anaphoric implied-parameter verbs (*failing students*); and anaphoric argument-taking nouns (*the best classmate*) (Ballard 1984).

3.7 Transportable English-Language Interface (TELI)

In Ballard and Stumberger (1986), the authors present Transportable English-Language Interface (TELI), which is a natural language analyzer that enables the modification of known domain-specific words in a system. TELI first uses normal form database tables to analyze and learn about problem domain concepts to perform natural language queries. The authors say that implementations fail either, because the user does not provide clear grammatical rules and reserved words or does not receive the grammatical constructions required to deal with a domain problem. TELI depends on natural language, because its syntax is based on LISP. It provides several features such as passive verbs, prepositional and adjective-reduced relatives, quantifiers, and numbers. Also, TELI provides semantic processing where definitions are provided by the user, meaning that modifiers are inferred as predicated-managed extensions and that semantic analysis is compositional. Semantics are LISP-like definitions that are stored and associated with an index that is a defined word. The drawback of TELI is that it has no support for arithmetic operations.

3.8 HyperTalk

HyperTalk is an Apple HyperCards-embedded programming language used for data manipulation and hypermedia with an English-like grammar (Wheeler 2004) that is based on Macintosh Toolbox English-like grammar and based on an API. HyperTalk has a Pascal-like logic structure that is organized through events, and was designed with non-programmer users in mind to close the gap between them and programming. HyperTalk describes HyperCard element transitions known as *cards*, which are containers for objects such as buttons or text fields. HyperTalk allows for indirect references such as *it*, *the third*, *last*, and *before*, to name but a few.

In Eisenstadt (1993), the author describes eight problems for HyperTalk, a language that is easy to learn and use but hard to debug, because HyperCards have a complex structure and lack tools to analyze and debug code. The user himself must deduct the location of the error to debug code. The debugging process requires many secondary objectives and the interpreter is inaccessible during execution, while the user can neither use support tools nor observe a coherent execution view. The user must have a deep understanding to ensure the correct interpretation of the interrelated dataflow, and even requires a deeper understanding to ascertain the object's inner state, which is not always equal to its apparent state.

3.9 AppleScript

As AppleScript is derived from HyperTalk, it is a scripting language developed by Apple and used for Inter-Application Communication through *AppleEvents* (Apple Inc. 2016). It works in a similar way to a UNIX console. AppleScript is designed to be used with a predefined object library that is released by Apple. AppleScript follows the HyperTalk idea of creating an easy to understand natural-language-based programming language. A list from which an element can be selected is shown in the example below:

```
set chosenListItem to choose from list {"A", "B", "3"}
        with title "List Title"
        with prompt "Prompt Text"
        default items "B"
        OK button name "Looks Good!"
        cancel button name "Nope, try again"
        multiple selections allowed false
        with empty selection allowed
```

The *if* word defines conditionals:

```
if x is greater than 3 then
     -- commands
else
     -- other commands
end if
```

The word *repeat* is used to define loops and is used in the example below to describe a loop that repeats itself ten times:

```
repeat 10 times
    -- commands to be repeated
end repeat
```

3.10 Attempto Controlled English (ACE)

In Fuchs and Schwitter (1996) and Fuchs et al. (1999), the authors present Attempto Controlled English (ACE), an English formalized version that can be translated into first-order logic and looks like natural language. ACE lacks ambiguity and is understandable by English speaking humans and computers alike. ACE consists of various elements. Attempto Parsing English (APE) is a Prolog-based interpreter that incorporates ACE construction and interpretation rules, with text unambiguously translated into Discourse Representation Structures (DSR). DSRs are first-logic variants easy to translate into any formal language. Attempto Reasoner (RACE) is a Prolog-programmed tool that enables automatic reasoning to be used to verify the influence of one text over another. Also, the authors describe Semantic Web ACE extensions, such as the translation between this and languages including OWL, SWRL, RuleML, Protune and R2ML, while ACE can be used as an MIT Handbook query language. AceWiki is a merge between ACE and a Semantic Web tool focused on wikis (Fuchs et al. 2008).

Reserved words (taken from functions and some anaphora-like words) and content words constitute the ACE vocabulary, with grammar definition restricting sentence form and meaning. Text is an anaphorically-related declarative sentence sequence that can be simple or composed, where composed sentences use *and*, *or*, and *if-then* expressions to connect simple expressions. Query sentences use a question format with the *?* character and words such as *what*, *which*, or *how*, while those finishing with *!* phrases indicate executable commands. To deal with ambiguity, ambiguous phrases are decomposed into two or more phrases to replace words without losing the meaning. For anaphoric references, ACE searches for anaphors and replaces them with the last and more specific noun that is consistent with gender and number. Test results indicate that software artifacts documented with ACE are clear and defined with a controlled English (Kuhn and Bergel 2014), with ACE performing better than Prolog and SOUL based on a questionnaire applied to 20 software engineering experienced graduates and undergraduates, considering execution time and user preferences.

3.11 MOOSE Crossing

In Bruckman (1997) and Bruckman (1998), the authors present MOOSE Crossing, an educational text-based MUD[7] Object-Oriented game that has a DSL named MOOSE, which has an English-like syntax and is focused on children aged 9 to 19 years old. Children use MOOSE to describe actions, creatures or places using *procedures* that are denoted as verbs, strings, numbers or references received by verbs. MOOSE is an easy-to-learn language where simplicity is preferred over formality and in which non-alphanumeric characters are avoided as much as possible.

3.12 NaturalJava

In Price et al. (2000), authors present NaturalJava, which is a user interface based on three subsystems: *Sundance*, a natural language processor; *Programming Instruction Synthesis Module* (PRISM), a structure interpreter; and *TreeFace*, a Java abstract syntax tree manager. While NaturalJava generates Java code from natural language expressions, it does not support arrays and nested classes.

NaturalJava limits inference to reduce ambiguity, and uses the Sundance natural language partial parser, which, focusing on information extraction technology, solves fragile expressions. Unlike full parsers that generate an abstract syntax tree, NaturalJava generates sentence fragments as a plain syntactic representation using partial parsers.

Information extraction is a natural language processing that extracts predefined types from a natural language text, focusing on important details and leaving aside irrelevant information. Information extraction systems can include several domains, with, in this case, natural language definitions providing Java domain constructions that Sundance processes to generate 27 case frames. A case frame is a representation of a program construction used to describe concepts.

PRISM provides a user interface for NaturalJava with which sentences can be added, and using which addition, deletion, modification and AST information queries can be performed via commands. Case frames are divided into two tasks by PRISM: a user determines desired actions; and, obtains necessary information for test cases to fulfil a petition. TreeFace generates an AST, while an instance (TreeFace object) maintains the edition context registry with information to be performed and determined using PRISM's stored concept. Also, TreeFace provides new interfaces, classes, fields, methods, local variables, loops, conditionals, and assignations as well as enabling the creation of return values.

3.13 HANDS Tool

Pane et al. (2002) present the Human-centred Advances for the Novice Development of Software (HANDS), which is a tool focused on GPL and usability for children from fifth grade level or higher. These authors discard other features such as efficiency, correctness, and similarity with other languages to improve usability.

The authors conducted a study to analyze how children and adults describe structures before learning programming (Pane et al. 2001), observing that non-programmers use event-based and rule-based descriptions where nouns are combined in sets and avoid data structures. Mathematical concepts are described in natural language terms, while "things" knowing their state implicitly and solely becoming explicit when this changes.

Usability is described as something to bear in mind when designing new languages, Usability must be increased or decreased according to the user who will use the language.

HANDS works with its programming environment, consisting in a compiler, a debugger, and other elements, and also has self-complementary visual and textual languages. As HANDS is

[7] A Multi-User Dungeon (MUD) is usually a text-based role video game (Bartle 2003).

user-focused, it avoids mathematical concepts, with the article *the* able to be used in any part of the code, and also accepts sentences such as *end if*. HANDS uses graphical card representations for memory variables that are managed through an agent called *Handy*. An example of HANDS is presented below:

```
set nectar of flower to 100
set flower's nectar to flower's nectar + 25
add 25 to flower's nectar
```

Where the *nectar* is a property of the card *flower*, with the latter instructions representing the same event, while the second instruction is more natural from a user perspective.

Also, HANDS uses a query-like mechanism to work with iterators:

```
all flowers evaluates to orchid, rose, tulip
all bees evaluates to bumble
all snakes evaluates to the empty list
all (flower and (nectar < 100)) evaluates to orchid
```

Where the word *all* is used as an indirect reference for all cards associated with *flower*.

3.14 Action-Based Application to Natural Language Interface

In Chong and Pucella (2004), authors present a framework that enables the creation of interfaces in natural language for action-based applications. They describe an application interface as a set of links that facilitates the interaction between the user and the system itself. The user interface deals with user inputs and presents the system responses to those inputs.

To work properly, a correct separation between interface and the application is required, so many interface types can be used in a transparent way or, at least, an adapter can be used to enable the framework to work correctly.

The framework translates natural language into a high-order logic using categorial grammars that return a semantic representation for the input. As a result, the framework generates a λ-calculus simple expression that is executed through user-interface procedure calls.

3.15 Metafor

In Liu and Lieberman (2005a), authors present Metafor, a tool that enables the abstraction of the necessary elements to help new programmers increase their abstraction ability. Experienced programmers use for brainstorming in the early stages of software development. Text is written by the user as a story that Metafor analyzes through programming semantics, generating Python code, which is not directly executed, to help consolidate the user's ideas through a programmatic perspective. This technique is presented as an alternative that enables the user to program interactively, instead of using programming books as learning tools. Metafor cannot analyze all English grammar and allows source code to be written in such a way as to generate an approximate natural language interpretation.

Metafor can be used to resolve object-oriented programming problems, consisting in the fact that the objects are inefficient when describing protocols or abstractions involving patterns, which are then solved using pattern languages. In addition, OOP lacks support for non-functional requirements that are solved using AOP. The differences between AOP and pattern languages are that the latter encapsulate abstractions that cover many objects, while AOP encapsulates abstractions scattered through objects. Although abstraction is the process of taking some elements and ignoring others, it is undesirable in context-sensible applications (Lieberman 2006).

Metafor analyzes pronouns like *it* and *he* via pre-processing, using ambiguity advantageously unlike other tools, because, with traditional languages, programmers take essential decisions from the first steps of development onwards and restructure all code if said decisions are found to be counterproductive. With figurative representations, which are an efficient mechanism for corrections to be undertaken, object design is simplified. While this task is generally performed using formal languages, it is a challenge that authors propose to solve with a more neutral object representation in a tuple form with format (full name, arguments, body). A type analyzer dynamically examines body and arguments. Metafor uses normalization to convert an adjective into a noun to transform expressions like *the drink is sweet* into *the drink has sweetness*.

3.16 Spoken Java

In Begel and Graham (2005), authors present Spoken Java, a tool for Java voice programming. Spoken Java enables coding in three ways—by spelling words, using natural language, or paraphrasing text. Spelling is a slow and tedious task, while paraphrasing text entails abstraction problems that hinder text comprehension for inexperienced programmers. The authors describe pronunciation as a drawback, because programming languages use clearly defined writing not focused on spelling, leading to unpronounceable or hard to pronounce structures. Furthermore, users generally talk about the code they want to write instead of talking about the code itself. Spoken Java is a Java superset with extra rules and an increased semantic and syntactic ambiguity.

Ten programmers with the following characteristics recited Java code that was used in Spoken Java tests (Begel and Graham 2006a): Five knew Java and five did not; five were native English speakers and five were not; and five were students from the United States and five were not. The recited code was translated into text. There are elements with a complex translation into code, for example, *array[i]* expression was translated as *array sub i* for some programmers, as *array of i* by others, and as *i from array* for yet others. Homophones were another drawback, because sounds like *for* could be understood as a loop, a *4* number or a *fore* variable. The use of capital letters is also a problem, because a class definition recited explicitly as *that class with capital c* by some authors, requiring the analyzer to deduce whether the word means *c* or *see*. Yet another problem was whitespaces, because the system cannot recognize the difference between the *dropStack Process* and the *drop stack process*. Authors mention that partial word pronunciation was a problem for non-native English speakers, because they either enunciated the word or phrase completely or omitted sections as with *printLn*. English speakers omitted punctuation when they spoke with authors detecting hindered interpretation in sentences like *ex.printStackTrace()*, because some users paraphrased as *ex printStackTrace* and others *ex dot printStackTrace*. Users described constructions in several ways and used expressions like *no arguments* as a synonym for *()*. Finally, prosody, a voice variation used to express ambiguous ideas in their correct form, was employed by users to recite expressions such as *array sub i plus plus* interpreted as *array[i]++* or *array[i++]*. The former example defines an *array sub i <pause> plus plus*, while the second defines an *array sub <pause> i plus plus*. Some non-English speaker users had difficulty with this, because they are not familiar with English prosody.

3.17 Computer-Processable Language (CPL and CPL-Lite)

In Clark et al. (2005) and Clark et al. (2010), authors describe Computer-Processable Language (CPL), which is a controlled natural language developed using a naturalistic approach, and which has a formalistic variant called CPL-Lite.

CPL uses heuristics for making ambiguity decisions to produce human-expected implementations. CPL accepts three sentence types—facts, questions, and rules. These include auxiliaries and

grammatical particles[8] with a verb, and in which a noun can be modified with other nouns, prepositions, and adjectives.

Authors also developed CPL-Lite, a mechanism to define queries in a comprehensible and controllable way. Unlike CPL, CPL-Lite does not uses heuristics, compensating by using a more restricted interpreter to avoid ambiguity and thus only using words that can be translated into an ontology-defined concept.

Both CPL and CPL-Lite have the same expressiveness level, but CPL requires fewer words, because it has several mechanisms that could give an expression the correct meaning. In the following example taken from Clark et al. (2010), CPL is used in the AURA tool context, a knowledge-based system for physics, chemistry and biology:

```
A man drives a car along a road for 1 hour.
The speed of the car is 30 km/h.
```

Next, the same example is used with CPL-Lite:

```
A person drives a vehicle.
The path of the driving is a road.
The duration of the driving is 1 hour.
The speed of the driving is 30 km/h.
```

CPL infers the concept and necessary elements, whereas, in CPL-Lite, the user describes them explicitly. Despite this, both approaches have apparent incompatibility, with CPL-Lite embedded into CPL's deterministic core, meaning that CPL includes CPL-Lite's 113 sentence pattern, and both languages use the same interpreter.

3.18 Natural Language to Python Prototype

In Vadas and Curran (2005), authors present a prototype that receives natural language inputs and returns Python code. They indicate that that programming tasks and code analysis are generally complicated if we do not possess a natural language explanation, and that programming languages possess uncertain details from a natural language perspective. In the following example, they describe a desirable user input:

```
read in a number
add 2 to the number
print out the number
```

which is converted into a Python representation as shown below:

```
number = int(sys.stdin.readline().strip())
number += 2
print number
```

As shown, while Python conversion is transparent, ambiguity remains when a number is read, in this case adding two, because the prototype did not know whether an addition or a string concatenation was required. Another variant could be

[8] According to the *Oxford Dictionary*, a grammar particle is any one of the class of words, such as *in, up, off, over*, used with verbs to make phrasal verbs. A particle is a word without a clear meaning and cannot be inflected.

```
read in 2 numbers
add them together
print out the result
```

whose Python equivalent is

```
number1 = int(sys.stdin.readline().strip())
number2 = int(sys.stdin.readline().strip())
result = number1 + number2
print result
```

As shown in the above example, the prototype could generate two variables, because the expression *read in 2 numbers* and *add them together* implies an indirect reference variable creation and invocation, unlike the first example where *number* is described explicitly in the three instructions.

In the first example, ambiguity is solved through assumptions: if a variable is called *number*, it is more likely to be used as a numeric value than as a string. On the other hand, the second example is resolved when the prototype analyzes the expression *2 numbers*, the scope of *them* and the correct association with *result*. The latter example is still without a proper solution.

The authors performed a study of how tasks are described by programmers, with the corpus used consisting in 370 sentences from 12 programmers. From this corpus, the authors found that programmers generally described tasks in programming terms instead of using a simpler natural language solution. From the examples, they found that some programmers described procedural constructions, even adding an *end loop* sentence that was partially corrected with Combinatory Categorial Grammar, with the corpus then analyzed by a parser and the errors corrected manually. The authors also describe how corrections were required for nearly all sentences.

For future research, they describe how they would use a better prototype capable of correct recognition for a greater corpus range, and that they would refine the parser and the semantic engine and use a quiz system to obtain additional information.

3.19 Pegasus

In Knöll and Mezini (2006), authors present a tool that not only eases software development through natural language sentences, but also provides multi-language support to facilitate teamwork among people from different countries who speak a variety native languages without affecting the overall system. In its actual stage, Pegasus generates simple programs in English and German languages, while Mefteh et al. (2012) describe an Arabian language extension named Ara_Pegasus.

Authors describe four problems relating to how the gap between the developer's expectations and programming techniques cannot be closed:

- —Mental problem. An idea must be adapted to a programming language, decomposing or grouping its elements.
- —Programming language problem. An algorithm must be translated into several languages that use new technologies and concepts, meaning that old languages still have a strong presence.
- —Natural language problem. In the modern era, while people from all around the world work in teams, they generally have a different native language and their own particular regionalisms.
- —Technical problem. In system design, the developer's contribution and time is invested by programmers thinking about how to best adapt ideas to produce an efficient implementation.

Pegasus has a limited human thought model where the concept of the idea is described as a human perception of the external world. As an idea is a personal perception that varies between two humans, a perception is an ideas set. Atomic ideas are those that cannot be decomposed into simpler ideas; therefore, a table idea is an idea set that could be atomics, such as color or a horizontal panel, or composed ideas, such as texture, as defined by the element from which the table is made (wood, for example), or a particular smell. The authors propose *notation idea* to formalize ideas. An example of this concept, taken from Knöll and Mezini (2006), is presented below:

```
wood: [smell of wood, warmth, solidness, brown color]
table: [horizontal panel, vertical bars, wood, solidness]
```

This example defines two complex ideas, with simpler ideas specified between square brackets. Furthermore, the wood idea can be integrated into the table example, thus:

```
table: [horizontal panel, vertical bars, [smell of wood, warmth, solidness,
brown color], solidness]
```

A composed idea is defined as the composition of representations. In this case, for example, the ideas table and brown can become a composed idea summarized as follows:

```
(table, brown)
```

or extended as

```
([horizontal panel, vertical bars, [smell of wood, warmth, solidness,
brown color], solidness], brown)
```

As long as a complex idea is the association of atomic or complex ideas, composed ideas associate several complex ideas through a particular moment.

Pegasus is a tool that enables natural language text, conducting analysis to return executable code as a result. Pegasus uses three basic features to function: it reads natural language; it generates source code; and, it expresses natural language. The authors describe such features as comprising a *brain* that presents concepts as the notation of ideas.

The Pegasus brain has three parts:

- —Mind: Focused on understanding the meaning of a sentence.
- —Short-term memory: A queue comprising eight cells of the most recently used elements, with the first element deleted when the stack is full and a new element is placed in the queue.
- —Long-term memory: A data dictionary where ideas, their grammatical variations and semantic knowledge are defined.

Pegasus analyzes the grammatical structure of a sentence to obtain its ideas and relationships, and then identifies the context of the sentence. An idea appears as an entity, action or property. Pegasus analyses an idea by comparing the meaning of the abstracted sentence idea with ideas stored in an entity dictionary. Then, Pegasus decomposes the idea into sub-ideas to resolve its meaning. Once an idea is solved, Pegasus generates Java code, although it is not limited to this language. To generate text in another natural language, Pegasus compares the input with the ideas dictionary and generates an output.

3.20 Little and Miller's Prototypes

In Little and Miller (2006), the authors describe a mechanism for the translation of simple instructions into executable code related to a particular domain. The authors present two prototypes—a Web-based browser and another based in Microsoft Word.

Similar to programming-by-demonstration (PbD), the translation is performed without the user knowing how the script functions. This approach enables the implementation of description and filling mechanisms applied to forms for practically any website. The user is also able to record his actions in a command set that works on the website, even for form filling tasks that can be shared to other people.

Expressive functions such as *left margin two inches* are described, where the system translates it into Visual Basic code *InchesToPoints* due the word "inches" being shared in both descriptions. As the system infers that this is the most appropriate function in the sentence context, the result *ActiveDocument.PageSetup.LeftMargin = InchesToPoints(2)* is returned.

The system can infer string sequences such as *UIST 2006 into search textbox*, translated into a search for *UIST 2006* that is executed with an implicit "enter" in the textbox with name *search*.

With the system tolerating nested functions, expressions like *pick the 4GB RAM option* is successfully translated into *pick(findOption("4GB RAM"))*. This nesting is limited to continuous strings, so expressions such as *pick option 4GB RAM* can be correctly inferred, but not expressions such as *option pick 4GB RAM*. This is because *findOption* receives a string, while pick does not, thus generating *findOption()* in the latter example.

The prototype uses various formats for values such as *2*, which can be replaced by *2nd, two or second*, among others. While quotes are optional for strings, their use reduces ambiguity. The system deals with arguments using several approaches, one of which being explicit naming. For example, while, for the function definition *click(integer x, integer y)*, the sentence *click 300 y 200 x* is valid and the arguments are ordered, *200 300 click* is also valid.

3.21 MOOIDE

In Lieberman and Ahmad (2010), authors present MOOIDE, which is a tool for creating MOOs using natural language. MOOIDE is formed by a parser that analyzes English text and provides information from a knowledge base. With MOOIDE, users add new elements in order that other users can interact with them. MOOIDE uses anaphora resolution and *Commonsense knowledge* through *Open Mind Common Sense* to guarantee that objects and characters are text coherent. An example is presented below:

```
There is a chicken in the kitchen.
There is a microwave oven.
You can only cook food in an oven.
When you cook food in the oven, if the food
is hot, say "The food is already hot."
Otherwise make it hot.
```

MOOIDE learns about situations, so, if we write this:

```
Cook chicken in microwave oven
```

the system associates the sentence with the previously defined rules. In this case, an error will be returned due to the fact that there is no relation defined between *microwave oven* and *chicken*. To resolve this, we can write:

```
People eat chicken
```

Then, the system can associate this with the previous sentence and then print *the chicken is now hot*.

As MOOIDE is restricted to real world properties, something cannot be defined based on magic, and Python code is returned as executable code.

3.22 Macho

In Cozzie et al. (2011), authors present Macho, which is a tool that facilitates the generation of source code from simple natural language instructions. Macho works by analyzing text and transforming it into executable queries to obtain code fragments from a database filled with source code examples that are integrated to return a functional program.

Due to its features, Macho only allows very simple UNIX comand-line utilities without options, and one or two arguments. Based on the generation of automatic code from user-defined requirements, Macho focuses on requirement analysis and Java code generation. To reduce ambiguity, Macho uses combined code fragments.

Macho analyzes, processes and transforms simple natural language instructions into queries to select source code fragments to construct a program. To do this, it has four subsystems:

- —Natural language parser. Analyzes verbs, nouns, prepositions, abbreviations and conjugations that can contain relevant information for code implementation.
- —Database. A library storing more than 1,200 code fragments, from which fragment candidates are selected.
- —Stitcher. Fragment codes are combined, from which candidate programs are returned.
- —Automatic debugger.

Macho combines code fragments by analyzing two expressions via a variable if the output type matches the other input type, or by using the Stitcher subsystem to search for code to find a fragment for a type conversion. Macho provides limited control flow, where a natural language parser generates *if* sentences and synthesizer infers, generating loops if suggested by the system. The debugger analyzes and classifies code in five cases:

- —Exception (code is inserted into an "if").
- —A correct output superset (print is inserted into an "if").
- —Garbage (subsequent program is tested).
- —Correct output subset (some further prints are tested).
- —Correct output (process end).

Macho generates source code from expressive natural language sentences. Due to sentence ambiguity, the programmer must provide Macho with code fragments. Macho assists programmers without replacing them, because it does not resolve ambiguity. As reported in Cozzie and King (2012), the authors performed a study on Macho, in which they found that, from 69 tests, Macho generates 55 correctly in a time that varies from 2 to 20min.

In Cozzie and King (2011), the authors also describe early work on an update named Macho II, where the computer "learns" to generate compositions from these patterns to generate Java code. Macho II will work using natural language processing techniques and a probabilistic model for Java code. With this in mind, the authors say that it is possible for a model to learn to predict based on operations to generate code in almost any programming language.

3.23 Aiaioo NLC

In Carlos (2011), the author presents Aiaioo NLC, a natural language programming system that uses class sequential rules. This system is limited to dealing with real numbers operations, in a process consisting in the decomposition of instructions into sentences and their classification into predefined categories. Based on this classification, the system extracts entities for use in code generation, while for natural language code recognition, it facilitates the creation of new rules

and the recognition of new instructions that the system classifies into the existing categories and which can even be translated into another language to be used via the system's learning ability.

The author describes "proceduralization" as a construction process guided by steps, blocks, conditionals and loops. Imperative words are verbs that indicate function calls or change commands. The system uses questions to evaluate a variable or expression. The author requests entry tests via the internet, to express natural language instructions, and then classifies the responses according to notation rules described as follows: conditional primitives express conditions or actions; loop primitives evaluate primitives repeatedly; operations with side effects change the state of the variable; operations without side effects do not change the state of the variable; and relational operations compare values. The author defines class sequential rules as a token-ordered sequence where a sentence is correct only if its evaluated rule is correct.

3.24 Similar to C Spoken Language

In Gordon and Luger (2012), the authors present an imperative and C-like programming language that is easy to recite through simple English voice processors. This language has a typical grammar but omits punctuation and unpronounceable symbols, with the interface freeing the user from the typing task and carrying out compiling traditionally. The language was tested as an Eclipse plug-in using CMU Sphinx as a voice recognition engine and ANTLR for grammar definition. The authors describe how most voice recognition engines lack traditional programming language support and present two approaches to solve this problem. The first features programming models with built-in voice processing engines while, in the second, the tool generates code from English constructs.

3.25 Automated Test Automation

In Thummalapenta et al. (2012), authors present a technique to automatize automatic tests in software systems performed by testers who are, generally, not programmers. They perform the tests based on test cases without losing the informality, flexibility and ambiguity related to natural languages, while, at the same time, maintaining the accuracy of the scripting languages. Testers generally describe test cases in natural language that are captured in requirements and use cases to be translated into scripts, which then validate system quality. Testers perform this task manually.

The authors propose a mechanism to extract relations between actions and user interface elements without using an advanced natural language. Instead, the authors propose employing a step-by-step description where the tester describes his interaction with the user interface. The main problem is that, as a stylized description could contain ambiguity, heuristics and application-specific rules are required.

This approach deals with ambiguity by exploring alternatives to obtain a result or, when the system reaches an unresolvable point, performing a rollback to the last decision, after which the system continues with another alternative. Thus, the system analyzes a step-divided test that is ordered has identified objectives, and which, if an ambiguity is found, then generates a bifurcation. The last two steps are the identification of the segments that refer to interface elements, and their execution in the browser with consequent decisions to explore different flows.

3.26 Programming by Example-based and Natural Language Hybrid

In Manshadi et al. (2013), authors present a framework that uses natural language descriptions over the course of one or more examples. With this in mind, authors propose that Programming by Example (PbE) is reinforced by using natural language descriptions to obtain an expected output.

To achieve this goal, this framework uses ambiguous natural language descriptions that are complemented with examples that resolve the ambiguity, generating input-output pairs for every

possible solution, and then selecting the best option based on how well-adapted it is to the natural language description.

To resolve a problem from PbE, the framework decomposes a problem into simpler problems that are individually solved with defined restrictions for each. While the system uses Space algebra to deal with this decomposition, this returns near-infinite regular expression compositions; therefore, these expressions are intersected for all pairs. It also uses a probabilistic PbE (PPbE) method to reduce the number of possible solutions.

For the natural language component, the authors used a dependency parser for each description to code not only the syntactic but also the semantic structure.

3.27 SmartSynth

In Le et al. (2013), authors describe SmartSynth, a smartphone script-generation tool. SmartSynth produces scripts in SmartScript, a DSL created for this purpose. As SmartSynth processes natural language descriptions, it functions as a program synthesizer that serves as a translator between the user and the smartphone.

In SmartSynth, the user enters a natural language description that translates with an algorithm and obtains the API-related elements from the input. Thereafter, the system interacts with the user to resolve ambiguous or unknown elements. Then, SmartSynth relates descriptions and returns an executable script.

Eleven students tested SmartSynth, produced 750 different descriptions for 50 tasks, of which 640 met with the specifications and were used to validate SmartSynth. Based on this corpus, SmartSynth identified 64.3% of components using the base algorithm only, to a precision level of 77.1% combining all features, and 90.3% with user interaction. The drawback was that a rule-based resolution provided an 86% precision level. Taking this into account, an overall level of 58.7% was achieved using NLP-related techniques only, while, using program synthesis, this improved to 90%.

3.28 Natural Language Command Interpreter (NLCI)

In Landhäußer et al. (2016), the authors present Natural Language Command Interpreter (NLCI), which, based on an ontology that models an API that translates natural language inputs into API intentions, is a tool focused on generating executable code from English descriptions. The authors used two APIs to test NLCI, openHAB and Alice, with NLCI constructing API calls with an accuracy of between 67% and 78% for 50 written scripts. The ontology serves as a bridge between natural language and the code that implements its meaning.

The NLCI's focus is to resolve non-sequential problems occurring when humans express ideas non-sequentially, meaning that it is not necessary to express a program description in execution order. This problem was observed when children were instructed to use Alice; Alice is a 3D tool designed to teach children to program to create animations.[9] A limitation of NLCI is that it is unable to handle temporal elements, of which authors describe three English expression types: tense, temporal prepositions, and temporal adverbs (Landhäußer et al. 2014).

In a first NLCI test described as "AliceNLP," the tool divides text into ordered steps, and analyses and organizes it according to found temporal expressions. NLCI has an 86% success rate from 24 tests with the Alice implementation, with the time line correctly established in 17 tests. Also, NLCI is not limited to working with Alice, because its approach is based on an ontology for a library description, thus, when the ontology is changed, another library can be used (Landhäußer and Hug 2015).

[9]While an Alice ML language was developed as a Standard ML dialect by Saarland University (University 2014), in this article, the term "Alice language" is used only for the tool developed by Carnegie Mellon University.

Later, the authors performed a test for both openHAB-based and Alice-based software. In openHAB, they wrote 5 scripts with 15 commands based on the ontology generated from the website demonstration configuration. To feed the Alice-based ontology, they programmed ten animations and two static scenarios that were described by both programmers and non-programmers using natural language, after which this information was fed into NLCI, thus returning a sequence of API calls. The authors created a corpus of 86 scripts, with only 50 usable for animations, whereas 30 were executed for static scenarios and six were discarded. The overall level of precision was 78%, with 67% recorded for correct API calls. Because openHAB and Alice cannot provide support to deal with pre-conditions and post-conditions, NLCI could not resolve them, while the authors also expressed their interest in improving a voice translation system.

3.29 NLyze

In Gulwani and Marron (2014), authors present NLyze, a natural language-based interface for spreadsheets. This interface is based on a DSL that uses algebraic calculations and several mechanisms for table management focused on inexpert users. It was implemented as a plug-in for Microsoft Excel.

The authors designed a DSL that was sufficiently expressive to solving tasks while being sufficiently restricted to maintain efficient translation. With this in mind, the DSL enables mapping, filtering, and reducing operations. An algorithm called Translate receives a natural language instruction and a spreadsheet, then returning a ranked set of program candidates generated in the DSL that are possible interpretations of the instruction over the spreadsheet. This algorithm contains two other algorithms, *Synth* and *Rule*.

Synth focuses on type-based compositions, generating them by applying a recursive decomposition to obtain smaller fragments. *Rule* focuses on a common pattern rule set to return word-based instructions from the sentence. *Rule* is limited to syntax and fails when it receives sentences with words outside the expected pattern, which *Synth* then complements to increase the algorithm effectiveness of the translation.

While SmartSynth does not solve ambiguity, it returns both candidate programs based on user input and a natural language description for all candidate programs in order that they are easily understood by the user.

3.30 NL to DSL Framework

In Desai et al. (2016), authors present a framework that takes natural language as an input and returns several DSL code blocks that the tool ranks according to how near they are to the expected result, based on user input. The authors indicate that in 80% of the tests, the correct code was shown at the first attempt, while correct code was shown in the first three attempts in 90% of the tests, which were undertaken on a corpus of 1272 natural language inputs.

This Framework is DSL-independent and its specification and text-code pairs are required to train the tool. This framework uses a synthesis algorithm that generates candidate programs from text, based on the relation between terminals and text words. A data dictionary establishes this relation during the training phase, and also filters typos, solves homonyms and, finally, assigns a score for every program according to how near it is to the user-expected input. Also, its noteworthy that this framework uses techniques presented in SmartSynth and NLyze, so it can be seen as an refinement or extension of both tools, because it is not limited to a single DSL.

The authors implemented it with three domains: text editors with Microsoft Excel with a DSL to manipulate spreadsheet; automata for intelligent tutoring problems in which a DSL was also designed; and, an ATIS system for air traffic, in which a SQL-like DSL was designed.

3.31 Fourth-Generation Languages

The term fourth-generation languages is generally associated with enterprise data-managing languages optimized for database access, report generation and graphical interfaces (van Deursen et al. 2000). Due to their enterprise focus, language developers create 4GLs with a higher expressiveness level and more restricted abstractions; however, these langauge do not have a "4GL standard design," so every 4GL has its own grammar and properties. This section reviews several 4GLs that have natural language features.

3.31.1 Advanced Business Application Programming Language. Advanced Business Application Programming Language (ABAP/4) is a report-focused language that uses procedures to develop complex enterprise applications. ABAP derives from SAP R/2 and is embedded into SAP R/3 (Kemper et al. 1999). ABAP/4 enables database access through Native SQL and OpenSQL, and can manage events. It has support for object-oriented programming through the use of ABAP objects, features intern tables similar to C *"structs"*, and enables the definition of COBOL-like instructions (SAP-AG 2014). ABAP provide mechanism to describe verbose instructions like *ADD 2 TO result*, which is the naturalistic equivalent for *result += 2*; it is noteworthy that the ABAP instruction is equivalent to an English sentence with an elliptic subject.

3.31.2 Informix-4GL. Informix-4GL is a report-focused language with a limited imperative operations capability, designed to resemble natural language. Producing bytecode as well as C code as output, Informix-4GL is a very popular language due its clones, which have extended functionality, such as Graphical Interfaces or Java code translators, one of which being Aubit-4GL (Aubit-4gl team 2001). Users generate reports and connect them to many database managers. Informix-4GL also has an embedded SQL query language support (IBM-Corp 2015). Informix-4GL has a powerful support for graphical interfaces and the programmer can describe tasks with a English-like syntax like *MOVE WINDOW*, *NEXT FIELD "fieldname,"* or *NEXT FORM NEXT OPTION "optname,"* to name a few (O'Gorman 2010).

3.31.3 Visual FoxPro. Visual FoxPro (VFP) is a data-oriented programming language that was developed by Microsoft with FoxPro capabilities but that can also be used to develop user interfaces simply. As Microsoft designed VFP with data-dependent Windows application development in mind, VFP has database support integrated into its grammar, and displays information easily due its simple syntax (Hennig et al. 2005). Due to VFP is focused to build user interfaces, it is has an imperative grammar that allows working with graphical environments and databases with a formal, but expressive grammar where a sentence *DEFINE POPUP table MARGIN RELATIVE SHADOW COLOR SCHEME 4*, describes a popup table named "tables," which is "margin relative" with a "shadow color scheme" assigned to 4 (Pinter 2004).

3.31.4 Nomad Software. Nomad Software is a relational database-oriented language that combines both interactive and batch processing. It enables database definition, information handling and report generation and also has a database-oriented language for manipulating reports. Unlike other languages, Nomad is end-user focused, because the users are generally big corporations using languages in batch production loops, Web applications, and Web or desktop applications for report generation. It is an intuitive database-focused language with an interactive development environment in which an instruction can be immediately written and executed. Also, Nomad generates relational and normalized databases that are compatible between several database systems, such as VASM, IMS, IDMS, DB2, Oracle, and SQL Server (McCraken 1980).

3.31.5 SAS. SAS is a PL/I syntax-based language developed by the SAS Institute, whose focus is the statistical data analysis of databases or tables. SAS displays these data in graphic or document

form, analyzing and then modifying tables in several ways to present results as a reports, and also analyzes SQL code. A remarkable detail is that SAS has no restriction in terms of reserved words, which can be used as identifiers. The unrestricted use of reserved words results in an ambiguous language that requires previously defined instructions that depend on context. SAS has had a built-in R[10] language since its 9.2 version, which enables the use of its features (Li 2013).

3.31.6 OpenEdge ABL. As OpenEdge ABL is a language that depends on data and has an English-like syntax, end-users can develop applications without knowing a programming language. Based on graphical interfaces and data queries, it has an expressiveness syntax that enables the construction of data queries, inserts, modifications, and the deployment of simple tasks (Sadd 2006).

3.31.7 LiveCode. LiveCode is a HyperTalk-based language that was originally named Revolution, which, in 2010, its authors changed to LiveCode. LiveCode is an easy-to-learn language that runs on/above various platforms, such as Android, Apple, or Windows. LiveCode has an English-like syntax with dynamic typing and has code expressive enough for it to be considered self-documented and suitable to be learnt from the age of 14 and above (LiveCode Ltd 2016).

4 DISCUSSION

This section presents two tables with very similar features but focused on different concepts. While Table 1 describes technologies that may also contain grammars, Table 2 describes only those that are languages and that may also form part of a tool.

4.1 Tools

As can be seen in Table 1, the main focus of these technologies is automatic source code generation or the resolution of particular domain activities. Others have a general purpose approach but integrate naturalistic elements to ease reading for programmers. It is noteworthy that many of these technologies rely on the use of predefined grammars to avoid ambiguity, using reserved words with or without data dictionaries. Some of these tools generate GPL code through code fragments in one or more programming languages. Another common feature is the use of indirect references in a very limited way, using pronouns such as *it*. As only two technologies have addressed the problem that appears when a programmer is not a native English speaker, authors promote the development of software in languages other than English. Another thing to be noted is that as many technologies are industry-focused, their features are focused on improving productivity. On the other hand, there are few tools focused on learning using natural language features, with those that do integrate them limiting their use as an efficient software development tool. For example, having no interest in efficiency, HANDS' execution can be very slow compared to other tools. Last, some tools are focused on the creation of code that is self-documented or can be translated into high-order logic.

From all the technologies reviewed in this study, automatic code generation is observed to be the main problem that authors want to solve. Pegasus is the only tool which authors define as generating automatic source code from a controlled natural language tool restricted to defining concepts in English and German through an entity dictionary. Furthermore, a future research line could look into Pegasus being supported by many other languages, such as Spanish, French or Arabic. The main focus in Knöll and Mezini (2006) was to solve development problems observed

[10]R is a statistics-focused object-oriented language influenced by both S and Scheme. It is focused on conducting statistical tests, time series analysis, classification and grouping algorithms, and linear and non-linear models (The R Foundation 2016).

Table 1. Properties of Revised Tools

Tools	Automatic code generation	Reserved words	Predefined grammars	Data dictionaries	Predefined code fragments	Multi-PL support	Multi-language support	Indirect references	Industry focused	Learning focused	Documentation focused
Heidorn	*	*	*								
NLC			*					*		*	
LDC		*	*	*				*			
TELI		*	*	*				*	*		
HyperTalk		*	*						*		
AppleScript		*	*	*			*	*	*		
ACE		*	*	*				*	*		*
NaturalJava	*	*		*	*			*	*		
HANDS		*	*					*		*	
Chong's framework	*		*			*					
Metafor	*		*		*	*		*		*	*
Spoken Java		*	*						*	*	
CPL/CPL-Lite	*	*	*	*				*			*
Vadas' prototype	*	*	*					*			
Pegasus	*		*	*	*	*	*	*	*		*
Little's prototype	*		*	*		*			*		
MOOIDE	*		*	*							
Macho	*			*	*			*	*		*
Aiaioo NLC		*	*	*					*	*	
Spoken C		*	*						*	*	
PbE NL hybrid	*				*	*					
SmartSynth	*							*			
NCLI	*	*		*	*	*		*			
Nlyze	*								*		
Desai's NL to DSL	*				*	*		*			
Total	15	14	18	12	7	7	2	15	12	6	5

by authors when there are teams whose members come from different parts of the world and who need to work together even when they have different native languages.

Macho is a Java generating tool that takes natural language text in query form for very short and specific programs, using an iterative process that can take several minutes to perform and to return source code from database-stored blocks. While Macho does generate code, it does not allow direct programming through a naturalistic language due its focus on automatic code generation

and also lacks an efficient way to treat ambiguity. Authors indicate that they are working on a major restructuring of Macho that uses machine-learning and a graphic dataflow language.

Metafor is another technology whose main focus is to help programmers describe problems through brainstorming, in which a programmer reflects a problem in Python scaffold form. Metafor requests feedback through user dialogue to strengthen the code. Similar to Pegasus, Metafor is focused on helping programmers as opposed to being a language or paradigm alternative.

As a tool whose implementation was reported several decades ago, the context in which NLC operates is different to more recent tools. Its main focus is to work solely with arrays that are shown on the display, restricting its use to a very narrow domain and, thus, controlling ambiguity when many elements are referred.

NLCI is a tool focused on translating natural language commands into executable code. As shown in previous examples, this is a domain-specific tool in which temporality is reported as a problem to be solved in future implementations. NLCI can also be implemented with several languages simply by changing the ontology.

As LDC and TELI are database management tools that learn from English instead of being a programming tool themselves, they lack several software development features. This is relevant, because natural language is used in adjective based queries, enabling complex code sentences based on phrases if used from a programming language approach.

Aiaioo NLC is a tool limited to work with operations over real numbers, where command recognition and language translation (natural languages) rules can be created, allowing imperative verbs as procedures and questions as conditionals. This tool domain is restricted to a specific domain and its domain-restricted treatment of ambiguity treatment is not mentioned.

Attempto Controlled English is a formalized subset of English and is used to document software artefacts. It was developed due to the need for an ambiguity-free English version that can be translated into a formal specification. While the main disadvantage is that it is a documentation-focused tool and there are no reports of software-generation systems, it is a good example of the importance of correct software development documentation.

4.2 Languages

Table 2 presents the natural language features of the languages reviewed here. As expected, many of these are industry-focused and have report-generation features. Due to their focus, these languages have an English-like expressiveness with some level of natural language elements. Very few are learning-focused and only one was designed explicitly for this purpose. As some languages have the ability to generate code in one or more GPL, they require data dictionaries, with the exception of the SAS language.

COBOL is a GPL that was designed by military and business companies for non-academic use, although it does have an expressive syntax. An imperative language without indirect reference support, COBOL derives from FLOW-MATIC language, meaning that both languages share many features.

CPL is a naturalistic language that resolves ambiguity using heuristics and artificial intelligence techniques. As a limitation, CPL depends on CPL-Lite, which is a formalistic language. The principal version of CPL resolves ambiguity problems with CPL-Lite, because the latter is integrated into the former. Another point is that both languages have a high expressiveness level.

The Spoken C language that is presented in Gordon and Luger (2012) is designed to be recited by programmers, making this the only natural language feature for this language due to it being imperative, like C. Another example is Spoken Java, which is a type of voice programming Java superset. Both languages are too complex to be formalized, because both prosody and the issue of

Table 2. Properties of Revised Languages

Languages	Code Generation	Data Dictionaries	English-like expressiveness	Indirect references	Industry focused	Learning focused	Report generation
FLOW-MATIC			*				
COBOL	*		*		*		*
CPL/CPL-Lite	*	*	*	*			
Spoken C				*			*
Spoken Java				*			*
HyperTalk			*	*	*		
SQL					*		
ABAP/4	*	*	*		*		*
Informix-4GL	*	*			*		*
Visual FoxPro	*				*	*	*
Nomad Software			*		*		*
SAS		*	*		*		*
OpenEdge ABL			*		*		
LiveCode			*	*	*	*	*
MOOSE Crossing	*	*	*	*		*	
Total	7	5	10	6	11	3	10

whether or not a user speaks English as a native language are factors that have a strong influence on how the code is pronounced; therefore, parsers conduct its interpretation.

HyperTalk and AppleScript are two English-based programming languages that are focused on graphic interface design, using which, inexperienced programmers are able to develop applications. While both languages provide basic indirect reference management, the main disadvantage is that they are domain-specific languages. As HyperTalk is embedded into the HyperCards tool, it does not have debugging support from its compiler.

In the case of fourth-generation languages, this study found a tendency for solving particular problems, meaning that 4GL languages lack standardization, and, moreover, are generally proprietary languages, meaning that it is hard to obtain and modify source code for a particular problem. 4GLs are an inflexible way to develop applications that are not only data-dependent, but also require several processes that language developers do not consider in language design. While many 4GLs solve this by offering or accepting libraries that users create in other languages, this means that 4GLs require other less expressive languages, a need that 4GLs were supposed to solve.

5 OPEN ISSUES

This study reviewed the state of the art of naturalistic technologies, concluding that there are several issues to be resolved to efficiently close the gap between the programmer and the computer. This section describes those issues:

(1) *Ambiguous indirect references.* The main limitation in natural languages in programming is the ambiguity that comes from using indirect references. While several tools use artificial intelligence techniques such as ontologies or anaphora resolution to find a relation between an anaphor and its referent, these techniques seek to deal with *natural language processing* rather than formal languages. Other tools avoid ambiguity by using predefined grammars and reserved words that allow them to generate indirect expressions in a very limited and controlled way. The main problem with these tools is that they limit indirect references to words such as *it*, thus ignoring concepts such as type and classification. Thus, it is necessary to strengthen the way in which indirect references are implemented in formal grammars to increase expressiveness from a human perspective.
(2) *Lack of naturalistic iterators.* As natural languages are expressive enough to describe things both sequentially iteratively, a set of steps can be described, followed by the instruction to "repeat until the task is complete," where "the task" can be implicitly described in previous (or the latter) text, requiring to use cognitive process to understand it. On the other hand, programming languages need to describe explicitly when an iteration block will be complete, while, as iterator blocks do not know about their contained instructions, coding a cycle requires a symbol to describe when the block ends (traditionally, enclosing the block into braces or tabs for the block instructions). Even though iterator blocks are an expressive and powerful means to describe repetitive tasks, they are not naturalistic enough to describe things in natural language terms. As a naturalistic iterator could provide an expressive way of describing cycles using structural anaphors and cataphors in a reflexive way, these cycles are defined in terms of "the next instruction" or "the previous ten instructions."
(3) *Lack of naturalistic bifurcations.* Natural languages describe selective actions using only the "then" course of action, thus avoiding the "else" unless it is necessary. Moreover, these structures are described in terms of "if we have permission, perform the next tasks," where "permission" is a previously defined action and "the next tasks" are the actions to be performed. Formal languages describe the *if-then-else* blocks that break down the main course of action in a similar way to exceptions in Java. A naturalistic bifurcation that works in a similar way to the naturalistic iterator described above could enable the description of instructions in terms of "the next ten elements" without using unnatural structures.
(4) *Lack of abstractions with naturalistic features.* Unlike formal languages, virtually anything can be used as descriptors in natural languages, enabling us to describe abstract concepts such as "emptiness," physical elements such as "a chair," or even grammatical elements such as "this paragraph." Furthermore, natural language elements have several features that alter state, such as quantity or gender, and which can be affected by other elements defined as adjectives. An abstraction that supports a quantity and dynamic adjective-like combination could simplify the use of lists in programming languages.

5.1 Model

This study, thus, proposes that a textual-based naturalistic language has the following features:

(1) It can enable the definition of abstractions in terms of nouns, adjectives and verbs.
(2) It can enable the definition of instructions in terms of previously defined elements.
(3) It can enable the definition of a subject and a predicate with a complexity greater than just a noun and a verb through the use of phrases.
(4) Languages must define types explicitly to use them in sentences.

Furthermore, the following elements are considered as desirable:

(1) Full support for deixis.
  (a) Endophoric reference support.
    i. Anaphors.
    ii. Cataphors.
  (b) Exophors.
    i Homphors.
(2) Property-based typing that enables the composition and generation of new abstractions.

The model described in this section allows defining naturalistic languages with formal restrictions, where ambiguity is not only solved with the grammar but also with abstractions. Therefore, a naturalistic language based on this model has both naturalistic and formalistic properties. For example, the sentence "time flies like an arrow; fruit flies like a banana" is a naturalistic description that must be resolved using clearly defined mechanisms. Its ambiguity can be resolved in a formalistic expression such as "time flies like an arrow; fruit flies enjoy a banana." In this example, the word "like" was replaced by "enjoy" to resolve ambiguity without changing semantic meaning. On the other hand, the model described in this section allows describing abstractions to solve ambiguity, where *time* is defined as a noun with a *flies like* verb with a direct object named *arrow* and defined as a noun. For the second part, and according to the context, a noun *fly* with a plural alternative *flies* and a verb *like* can be described to be used with an adjective *fruit*, or a noun *fruit* with a verb *flies like* can be described, where both verb definitions have a *banana* direct object defined as a noun.

6 CONCLUSIONS

This study analyzed various tools that use a controlled English grammar, with this controlled version then integrated with either a means of translating into a programming language or ambiguity-free documentation-generators or as an imperative language grammar component. Naturalistic programming-related concepts both from a programming and linguistic perspective have been described in this article, which proposes linguistic elements that can be considered for use in a naturalistic language, such as anaphors, temporality, and English-like formalized grammar. Furthermore, proposed abstractions have been described as a means to define and implement a general purpose naturalistic model.

According to the technologies reviewed, this study concludes that authors use natural language elements on a different scale to increase expressiveness. A drawback of this is that these elements were used as a support for the proposed solution of the problem rather than as a crucial part of the solution for the project. Disperse naturalistic features were, therefore, found in these implementations, features such as an imperative-adapted naturalistic syntax, as in COBOL, or a natural language parser implementation that automates software generation, as in Pegasus, which, in this case, is limited by its entity dictionary.

A challenge observed in most technologies is that they are focused on their own internal problems, meaning that there are several code variations resulting from said tools, which could be closed and proprietary where maintenance is performed by developers who have not used a model to support the defined language. Moreover, many of the technologies reported here are data dictionary or precompiled library-dependent, thus even further reducing their use to particular domains.

There are languages that have a high expressiveness level, which provides them with a human-like cognitive process to deal with ambiguity; however, these languages are limited by particular domains and use controlled natural language. The goal of the naturalistic paradigm is not natural language programming, nor is it to deal with a cognitive process beyond that which we use to

name identifiers, because this task is intended to be performed by a programmer in a similar way to other paradigms. Nevertheless, a naturalistic model must be capable of dealing with every domain without ambiguity problems.

This then raises the question of why programming is not undertaken using natural languages. In Bruckman and Edwards (1999), authors found 314 errors that occurred with attempts to use a natural language as described. These errors present, when source code cannot be distinguished from English, a conclusion based on analysis of children's interactions with the MOOSE Crossing tool, a study in which 16 children took part and which was based on the analysis of their errors. The authors observed these natural language errors when a child provided an incorrect English-like definition instead of a correct one. Of these errors, 147 were syntax errors, 68 were guessing errors, 58 were interpretation errors, 14 were assuming system awareness errors, and 7 were operator precedence errors. These results indicate that a clearly defined naturalistic language should be defined to avoid ambiguity.

A naturalistic language must manage indirect references regardless of whether there are temporal references such as "when a value is greater than 10," spatial references such as "the houses with garden," or reflexive references such as "the previous operation." To do this, a naturalistic language must cover not only nouns and verbs, but also their modifiers by providing more robust support for the use of phrases such as noun or verb phrases.

From the studies reviewed here, it can be concluded that a naturalistic model is reported as a conceptual proposal only in Lopes et al. (2003). Therefore, a definition of a programming model that not only enables the integration of natural language elements into programming languages, but also provides a sufficiently high level of formality in order that formal and unambiguous programming languages can be defined is required.

REFERENCES

Apple Inc. 2016. AppleScript Language Guide. Retrieved from https://developer.apple.com/library/mac/documentation/AppleScript/Conceptual/AppleScriptLangGuide/introduction/ASLR_intro.html.

Kenneth C. Arnold and Henry Lieberman. 2010a. Embracing ambiguity. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER'10)*. ACM, New York, NY, 1–6. DOI: http://dx.doi.org/10.1145/1882362.1882364

Kenneth C. Arnold and Henry Lieberman. 2010b. Managing ambiguity in programming by finding unambiguous examples. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*. ACM, New York, NY, 877–884. DOI: http://dx.doi.org/10.1145/1869459.1869531

Aubit-4gl team. 2001. X-4GL Reference Manual. Retrieved from http://aubit4gl.sourceforge.net/aubit4gldoc/4glreference/pages/4GLREFINFORMIX4GL_Reference_Help.htm.

Bruce W. Ballard. 1984. The syntax and semantics of user-defined modifiers in a transportable natural language processor. In *Proceedings of the 10th International Conference on Computational Linguistics and 22Nd Annual Meeting on Association for Computational Linguistics (ACL'84)*. Association for Computational Linguistics, Stroudsburg, PA, 52–56. DOI: http://dx.doi.org/10.3115/980491.980504

Bruce W. Ballard and John C. Lusth. 1983. An english-language processing system that "learns" about new domains. In *Proceedings of the May 16–19, 1983, National Computer Conference (AFIPS'83)*. ACM, New York, NY, 39–46. DOI: http://dx.doi.org/10.1145/1500676.1500682

Bruce W. Ballard and Douglas E. Stumberger. 1986. Semantic acquisition in TELI: A transportable, user-customized natural language processor. In *Proceedings of the 24th Annual Meeting on Association for Computational Linguistics (ACL'86)*. Association for Computational Linguistics, Stroudsburg, PA, 20–29. DOI: http://dx.doi.org/10.3115/981131.981136

Richard Bartle. 2003. *Designing Virtual Worlds*. New Riders Games.

Andrew Begel and Susan L. Graham. 2005. Spoken programs. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC'05)*. IEEE Computer Society, Washington, DC, 99–106. DOI: http://dx.doi.org/10.1109/VLHCC.2005.58

Andrew Begel and Susan L. Graham. 2006a. An assessment of a speech-based programming environment. In *Proceedings of the Visual Languages and Human-Centric Computing (VLHCC'06)*. IEEE Computer Society, Washington, DC, 116–120. DOI: http://dx.doi.org/10.1109/VLHCC.2006.9

Andrew Begel and Susan L. Graham. 2006b. XGLR: An algorithm for ambiguity in programming languages. *Sci. Comput. Program.* 61, 3 (Aug. 2006), 211–227. DOI: http://dx.doi.org/10.1016/j.scico.2006.04.003

Alan W. Biermann and Bruce W. Ballard. 1980. Toward natural language computation. *Comput. Linguist.* 6, 2 (April 1980), 71–86.

Igor A. Bolshakov and Alexander Gelbukh. 2004. *Computational Linguistics Models, Resources, Applications*. Ciencia de la computación.

Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, and A. Houston Kelli. 2007. *Object-Oriented Analysis and Design with Applications* (3rd ed.). Addison Wesley Longman, Boston, MA, USA.

Amy Bruckman. 1998. Community support for constructionist learning. *Comput. Supported Coop. Work* 7, 1–2 (Jan. 1998), 47–86. DOI: http://dx.doi.org/10.1023/A:1008684120893

Amy Bruckman and Elizabeth Edwards. 1999. Should we leverage natural-language knowledge? An analysis of user errors in a natural-language-style programming language. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'99)*. ACM, New York, NY, 207–214. DOI: http://dx.doi.org/10.1145/302979.303040

Amy Susan Bruckman. 1997. *Moose Crossing: Construction, Community, and Learning in a Networked Virtual World for Kids*. Ph.D. Dissertation. Cambridge, MA. AAI0598541.

Cohan Sujay Carlos. 2011. Natural language programming using class sequential rules. In *Proceedings of the International Joint Conference on Natural language Processing (IJCNLP'11)*. 237–245.

Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control (SIGFIDET'74)*. ACM, New York, NY, 249–264. DOI: http://dx.doi.org/10.1145/800296.811515

Stephen Chong and Riccardo Pucella. 2004. A framework for creating natural language user interfaces for action-based applications. *CoRR* abs/cs/0412065 (2004). Retrieved from http://arxiv.org/abs/cs/0412065.

Peter Clark, Phil Harrison, Tom Jenkins, John Thompson, and Rick Wojcik. 2005. Acquiring and using world knowledge using a restricted subset of english. In *Proceedings of the 18th International Florida Artificial Intelligence Research Society Conference (FLAIRS'05)*. AAAI Press, 506–511.

Peter Clark, William R. Murray, Phil Harrison, and John Thompson. 2010. *Controlled Natural Language: Proceedings of the Workshop on Controlled Natural Language (CNL'09), Marettimo Island, Italy, June 8–10, 2009. Revised Papers*. Springer, Berlin, Chapter Naturalness vs. Predictability: A Key Debate in Controlled Languages, 65–81. DOI: http://dx.doi.org/10.1007/978-3-642-14418-9_5

Anthony Cozzie, Murph Finnicum, and Samuel T. King. 2011. Macho: Programming with man pages. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*. USENIX Association, Berkeley, CA, 7–7.

Anthony Cozzie and Samuel T. King. 2011. *Macho II: Even More Macho*. Technical Report. Retrieved from https://pdfs.semanticscholar.org/367b/0b950dd76177074c9c86297ab7c188bfb2b9.pdf.

Anthony Cozzie and Samuel T. King. 2012. *Macho: Writing Programs with Natural Language and Examples*. Technical Report. University of Illinois at Urbana-Champaign.

Ferdinand De Saussure. 1916. Nature of the linguistic sign. *Course Gen. Linguist.* (1916), 65–70.

Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. ACM, New York, NY, 345–356. DOI: http://dx.doi.org/10.1145/2884781.2884786

Marc Eisenstadt. 1993. Why hypertalk debugging is more painful than it ought to be. In *People and Computers VIII*, Jim Alty, Dan Diaper, and Steve P. Guest (Eds.). Cambridge University Press, Cambridge, UK.

Norbert Fuchs and others. 1999. *Attempto Controlled English Language Manual Version 3.0*. Geneva, Switzerland: Institut für Informatik der Universitt Zürich, The address of the publisher.

Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. 2008. Reasoning web. Springer-Verlag, Berlin, Chapter Attempto Controlled English for Knowledge Representation, 104–124. DOI: http://dx.doi.org/10.1007/978-3-540-85658-0_3

Norbert E. Fuchs and Rolf Schwitter. 1996. Attempto controlled english (ACE). *CoRR* cmp-lg/9603003 (1996).

Jose Gaintzarain and Paqui Lucio. 2009. A new approach to temporal logic programming. In *Proceedings of the 9th Spanish Conference on Programming and Languages (PROLE'09)*. 341–350. Retrieved from http://www.sistedes.es/ficheros/actas-conferencias/PROLE/2009.pdf.

Benjamin. M. Gordon and George. F. Luger. 2012. English for spoken programming. In *Proceedings of the 2012 Joint 6th International Conference on Soft Computing and Intelligent Systems (SCIS'12) and 13th International Symposium on Advanced Intelligent Systems (ISIS'12)*. 16–20. DOI: http://dx.doi.org/10.1109/SCIS-ISIS.2012.6505414

Sumit Gulwani and Mark Marron. 2014. NLyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. ACM, New York, NY, 803–814. DOI: http://dx.doi.org/10.1145/2588555.2612177

Jan Heering and Marjan Mernik. 2002. Domain-specific languages for software engineering. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)*. 3649–3650. DOI: http://dx.doi.org/10.1109/HICSS.2002.994484

George E. Heidorn. 1973. An interactive simulation programming system which converses in english. In *Proceedings of the 6th Conference on Winter Simulation (WSC'73)*. ACM, New York, NY, 781–794. DOI: http://dx.doi.org/10.1145/800293.811628

Doug Hennig, Rick Schummer, Jim Slater, Tamar E. Granor, and Toni Feltman. 2005. *What's New in Nine: Visual FoxPro's Latest Hits*. Hentzenwerke Publishing.

Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-oriented programming. *J. Obj. Technol. March-April 2008, ETH Zurich* 7, 3 (2008), 125–151.

G. D. Ritchie, I. Androutsopoulos, and P. Thanisch. 1995. Natural language interfaces to databases—An introduction. *Nat. Lang. Eng.* 1 (1995), 29–81.

IBM-Corp. 2015. Informix v12.10 Documentation. Retrieved from http://www-01.ibm.com/support/knowledgecenter/SSGU8G_12.1.0/com.ibm.po.doc/informixtutorials.htm.

Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2010. Designing event-based context transition in context-oriented programming. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming (COP'10)*. ACM, New York, NY, Article 2, 6 pages. DOI: http://dx.doi.org/10.1145/1930021.1930023

Alfons Kemper, Donald Kossmann, and Bernhard Zeller. 1999. Performance tuning for SAP R/3. *IEEE Data Eng. Bull.* 22, 2 (1999), 32–39.

Won Kim. 1982. On optimizing an SQL-like nested query. *ACM Trans. Database Syst.* 7, 3 (Sept. 1982), 443–469. DOI: http://dx.doi.org/10.1145/319732.319745

Roman Knöll, Vaidas Gasiunas, and Mira Mezini. 2011. Naturalistic types. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'11)*. ACM, New York, NY, 33–48. DOI: http://dx.doi.org/10.1145/2048237.2048243

Roman Knöll and Mira Mezini. 2006. Pegasus: First steps toward a naturalistic programming language. In *Proceedings of the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'06)*. ACM, New York, NY, 542–559. DOI: http://dx.doi.org/10.1145/1176617.1176628

Tobias Kuhn. 2014. A survey and classification of controlled natural languages. *Comput. Linguist.* 40, 1 (March 2014), 121–170. DOI: http://dx.doi.org/10.1162/COLI_a_00168

Tobias Kuhn and Alexandre Bergel. 2014. Verifiable source code documentation in controlled natural language. *Sci. Comput. Program.* 96, P1 (Dec. 2014), 121–140. DOI: http://dx.doi.org/10.1016/j.scico.2014.01.002

Paul Laird and Stephen Barrett. 2009. Towards context sensitive domain specific languages. In *Proceedings of the 1st International Workshop on Context-Aware Middleware and Services: Affiliated with the 4th International Conference on Communication System Software and Middleware (COMSWARE'09) (CAMS'09)*. ACM, New York, NY, 31–36. DOI: http://dx.doi.org/10.1145/1554233.1554241

Ralf Lämmel. 1998. Object-oriented cobol: Concepts & implementation. *COBOL Unleashed. Macmillan Computer Publishing* 44 (September 1998).

Ralf Lämmel and Kris De Schutter. 2005. What does aspect-oriented programming mean to cobol? In *Proceedings of the 4th International Conference on Aspect-oriented Software Development (AOSD'05)*. ACM, New York, NY, 99–110. DOI: http://dx.doi.org/10.1145/1052898.1052907

Mathias Landhäußer, Tobias Hey, and Walter F. Tichy. 2014. Deriving time lines from texts. In *Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE'14)*. ACM, New York, NY, 45–51. DOI: http://dx.doi.org/10.1145/2593801.2593809

Mathias Landhäußer and Ronny Hug. 2015. Text understanding for programming in natural language: Control structures. In *Proceedings of the 4th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE'15)*. IEEE Press, Piscataway, NJ, 7–12.

Mathias Landhäußer, Sebastian Weigelt, and Walter F. Tichy. 2016. NLCI: A natural language command interpreter. *Automated Software Engineering* (2016), 1–23. DOI: http://dx.doi.org/10.1007/s10515-016-0202-1

Vu Le, Sumit Gulwani, and Zhendong Su. 2013. SmartSynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'13)*. ACM, New York, NY, 193–206. DOI: http://dx.doi.org/10.1145/2462456.2464443

Arthur Li. 2013. *Handbook of SAS DATA Step Programming*. CRC Press. 275 pages.

Ben Liblit, Andrew Begel, and Eve Sweetser. 2006. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the Annual Psychology of Programming Workshop*.

Henry Lieberman. 2006. The continuing quest for abstraction. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*. Springer-Verlag, Berlin, 192–197. DOI: http://dx.doi.org/10.1007/11785477_12

Henry Lieberman and Moin Ahmad. 2010. Knowing what you're talking about: Natural language programming of a multiplayer online game. *No Code Required: Giving Users Tools to Transform the Web. Morgan Kaufmann.*

Greg Little and Robert C. Miller. 2006. Translating keyword commands into executable code. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST'06)*. ACM, New York, NY, 135–144. DOI: http://dx.doi.org/10.1145/1166253.1166275

Hugo Liu and Henry Lieberman. 2005a. Metafor: Visualizing stories as code. In *Proceedings of the 10th International Conference on Intelligent User Interfaces (IUI'05)*. ACM, New York, NY, 305–307. DOI: http://dx.doi.org/10.1145/1040830.1040908

Hugo Liu and Henry Lieberman. 2005b. Programmatic semantics for natural language interfaces. In *CHI'05 Extended Abstracts from the Proceedings of the Conference on Human Factors in Computing Systems (CHI EA'05)*. ACM, New York, NY, 1597–1600. DOI: http://dx.doi.org/10.1145/1056808.1056975

LiveCode Ltd. 2016. LiveCode. Retrieved from https://livecode.com/products/livecode-platform/livecode-in-education/.

Anne Lobeck. 2007. *Ellipsis in DP*. Blackwell Publishing, 145–173. DOI: http://dx.doi.org/10.1002/9780470996591.ch22

Sebastian Lohmeier. 2011. *Shaping Statically Resolved Indirect Anaphora for Naturalistic Programming*. Bachelor's Thesis. Retrieved from http://www.monochromata.de/bachelor_thesis/.

Cristina Videira Lopes, Paul Dourish, David H. Lorenz, and Karl Lieberherr. 2003. Beyond AOP: Toward naturalistic programming. *SIGPLAN Not.* 38, 12 (Dec. 2003), 34–43. DOI: http://dx.doi.org/10.1145/966051.966058

Mehdi Manshadi, Daniel Gildea, and James Allen. 2013. Integrating programming by example and natural language programming. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI'13)*. AAAI Press, 661–667.

Daniel D. McCraken. 1980. *A Guide to Nomad for Applications Development*. Addison-Wesley Publishing Company. 220 pages.

Mariem Mefteh, A. Ben Hamadou, and Roman Knöll. 2012. Ara_Pegasus: A new framework for programming using the Arabic natural language. In *Proceedings of the International Conference on Computing and Information Technology (March, 2012)*. 468–473.

Marjan Mernik and Viljem Žumer. 2001. Domain-specific languages for software engineering. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS'01), Volume 9 (HICSS'01)*. IEEE Computer Society, Washington, DC, 9071.

Rada Mihalcea, Hugo Liu, and Henry Lieberman. 2006. NLP (Natural language processing) for NLP (Natural language programming). In *Proceedings of the 7th International Conference on Computational Linguistics and Intelligent Text Processing (CICLing'06)*. Springer-Verlag, Berlin, 319–330. DOI: http://dx.doi.org/10.1007/11671299_34

L. A. Miller. 1981. Natural language programming: Styles, strategies, and contrasts. *IBM Syst. J.* 20, 2 (June 1981), 184–215. DOI: http://dx.doi.org/10.1147/sj.202.0184

Brad Myers and John Pane. 1996. *Usability Issues in the Design of Novice Programming Systems*. Technical Report. Carnegie Mellon University.

Neelu Nihalani, Sanjay Silakari, and Mahesh Motwani. 2011. Natural language interface for database-a brief review. *International Journal of Computer Science Issues* 8 (2011), 600–608.

Sharon O'Brien. 2003. Controlling controlled english: An analysis of several controlled language rule sets. *Proc. EAMT-CLAW* 3 (2003), 105–114.

Charles Kay Ogden, Ivor Armstrong Richards, Sv Ranulf, and E Cassirer. 1923. *The Meaning of Meaning. A Study of the Influence of Language upon Thought and of the Science of Symbolism*. JSTOR.

John O'Gorman. 2010. The Aubit4GL Manual. Retrieved from http://www.og.co.nz/aubit4gl/html/index.html.

Mehmet A. Orgun and Wanli Ma. 1994. An overview of temporal and modal logic programming. In *Proceedings of the 1st International Conference on Temporal Logic (ICTL'94)*. Springer-Verlag, London, UK, 445–479. Retrieved from http://dl.acm.org/citation.cfm?id=645548.659010.

John F. Pane, Brad A. Myers, and Leah B. Miller. 2002. Using HCI techniques to design a more usable programming system. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*. IEEE Computer Society, Washington, DC, 198. Retrived from http://dl.acm.org/citation.cfm?id=795687.797801.

John F. Pane, Chotirat "Ann" Ratanamahatana, and Brad A. Myers. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Hum. Comput. Stud.* 54, 2 (Feb. 2001), 237–264. DOI: http://dx.doi.org/10.1006/ijhc.2000.0410

Les Pinter. 2004. *Visual FoxPro to Visual Basic .NET*. Pearson Education.

Oxford University Press. 2016. Stats and Analysis. Retrieved March 1, 2016 from http://www.oxforddictionaries.com/.

David Price, Ellen Rilofff, Joseph Zachary, and Brandon Harvey. 2000. NaturalJava: A natural language interface for programming in Java. In *Proceedings of the 5th International Conference on Intelligent User Interfaces (IUI'00)*. ACM, New York, NY, 207–211. DOI: http://dx.doi.org/10.1145/325737.325845

Long Qiu, Min yen Kan, and Tat seng Chua. 2004. A public reference implementation of the rap anaphora resolution algorithm. In *Proceedings of the 4th International Conference on Language Resources and Evaluation (LREC'04)*. 291–294.

John Sadd. 2006. *OpenEdge Development: Progress 4GL Handbook*. Progress Software Corporation. 844 pages.

SAP-AG. 2014. ABAP—Keyword Documentation. Retrieved from http://help.sap.com/abapdocu_740/en.

Thomas J. Schriber. 1990. *Simulation Using GPSS*. Krieger Publishing Co., Inc., Melbourne, FL.

The R Foundation. 2016. The R Project for Statistical Computing. Retrieved from https://www.r-project.org/.

Suresh Thummalapenta, Saurabh Sinha, Nimit Singhania, and Satish Chandra. 2012. Automating test automation. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE Press, Piscataway, NJ, 881–891.

Saarland University. 2014. Alice. Retrieved from http://www.ps.uni-saarland.de/alice/.

David Vadas and James R. Curran. 2005. Programming with unrestricted natural language. In *Proceedings of the Australasian Language Technology Workshop (2005)*. 191–199.

Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.* 35, 6 (June 2000), 26–36. DOI: http://dx.doi.org/10.1145/352029.352035

Richard L. Wexelblat (Ed.). 1981. *History of Programming Languages I.* ACM, New York, NY.

Kyle Wheeler. 2004. HyperTalk: The Language for the Rest of Us (2004). Technical Report. https://pdfs.semanticscholar.org/5a62/5c7a2d49a7857a880d01c851d37cda88c3a0.pdf.

Received June 2016; revised May 2017; accepted June 2017