A Design-Oriented Specification Language for defining software requirements

Lerina Aversano, Angelo Ciampa, Fabio Giovine, Giovanni Moriano, Antonio Pirozzi and Maria Tortorella

Department of Engineering, University of Sannio, Benevento, Italy

Abstract—The most common difficulty of requirements elicitation is to define all components and aspects of a project by using the natural language without ambiguities, weak phrases and useless descriptions which make project outlines chaotic. In this paper, we present the Design Oriented Specification Language (DOSLang) that aims at reducing the gap existing among the project stakeholders, which are involved into the requirements specification and comprehension activities. The DOSLang language provides a free-form syntax with the introduction of constructs helping project stakeholders to reducing the ambiguity during definitions, descriptions, todos, actions, constraints and all the other aspects related to requirement definition, without sacrificing the ability of specifying loops and conditional constructs, small sets of data type and multiplicity between 'entities'. The DOSLang compiler is designed to be able to create a common baseline, which the project stakeholders can use during the first phases of the project elicitation. DOSLang is suitable for any kind of project that requires a precise description of all of its aspects. Moreover the language was created keeping the possibility to be used by all people working in a project development: from business people to programmers, including customers.

Keywords—Programming Language; Requirements Engineering; SRS; Projects; Stakeholders; Baseline; Compiler; Abstract Syntax Tree; PKI; Signature; Quality Metrics.

I. INTRODUCTION

The first phase to be executed during the development of a new project concerns the definition of its requirements that will describe its functionality, components, behaviors and wished user interface. The requirements are usually expressed in a natural language and do not consider the language that will be used for the software system development. Therefore, they could contain a high level of ambiguity due to the grammatical constructions of sentences, use of weak words and so on. These problems could introduce a large set of "defects" in the documentation describing the project, undermine the meaning of a requirement and heavily compromise what the designer really wants to describe and obtain. In addition, if we consider the business field of the project, it is common that the requirements are expressed by stakeholders that could not have the technical competences or right domain knowledge for correctly formalizing them by using the formalism of languages, such as the UML diagrams or software programming languages. Furthermore, even if the requirements are expressed by technical people, it is not always obvious that other technical professional figures, such as programmers, designers, project managers, have different perceptions and understanding of the project blueprint. Therefore, nevertheless the big quantity of shared information, the result can be that the people involved in the realization of a project give a different meaning to the requirements and he way they should be implemented. In addition, there is the risk that what is already realized is not conform to the project outlines specified by the stakeholders.

In this paper, we present DOSLang – Design-Oriented Specification Language – a project born with the aim of reducing the difficulty of project stakeholders when they need to define and share requirements, ideas, rfc, todos and all the other aspects related to a project. The DOSLang has a built-in compiler which takes in input a dosl program describing the requirements and produces several deliverables starting from the implementation. After the compilation process, the DOSLang builds a set of output files in a target language of choice. These files form a baseline related to the
project that contains code from which it is possible to continue the development of the system to be implemented. In order to evaluate the quality of a requirement specification and measure the presence/absence of ambiguity and other properties of the dosl program, the DOSLang team has implemented a taxonomy based on the same categories considered by the ARM Tool [2] during the words classification.

The applicability of DOSLang is proved by using it during the requirements specification of a software project concerning a software engineering application. Nevertheless, the considered example, DOSLang is suitable for all kinds of projects that require the definition of clear and unambiguous requirements.

In the following, the DOSLang solution and architecture are described. In particular, Section 2 presents some related works proposed in the field of requirement elicitation using natural speaking while Section 3 shows the proposed solution introducing the DOSLang specification language in terms of syntax and constructs, characteristics, features and produced deliverables along with its objectives and main benefits. Section 4 describes the architecture in a component view. Section 5 discusses an example to show how DOSLang can be applied, and final remarks are given in the last section.

II. RELATED WORK

In the field of requirements engineering, we can find many proposed solutions aiming at helping stakeholders in formalizing requirements while reducing the ambiguity. Most of these give the possibility to write down specifications using natural language, like DOSLang.

L. Rosenberg et al. expose the importance of tools that support the requirement definition since the early stages of a project design process [1]. Their work is focused on tools, in particular the NASA ARM (Automated Requirement Measurement) Tool, and methodologies helping in writing qualitatively valid requirements.

Umber et al. propose a First Order Logic-based automated approach, SBVR (Semantics of Business Vocabulary and Rules), aiding to minimize the ambiguity of natural language inside a SRS by using mathematical foundation and business vocabulary [3].

In et al. [4], Charles Consel, propose DiaSuite, a tool suite implementing a software design approach to drive the development process. It relies on a domain specific design language based on Sense/Compute/Control then, it is suitable in the field of telecommunications, building automation, robotics and avionics. J. Mylopoulos et al. introduce a requirement-driven methodology, Tropos [5], based on the notions of actors and goals taken from E. Yu's i* modelling framework [6] for the agent-oriented programming platforms. Smialek et al. introduce ReDSeeDS(Requirements Driven Software Development System) [7], a tool able to automatically translate UML use case and domain models into code using a Requirements Specification Language, a precise constrained language grammar.

In the following, other solutions proposed in literature are summarized. They have been chosen among others due to their similarity with DOSLang. Cucumber [http://cukes.info/] is a framework focused on creating suites of automated acceptance tests through the definition of the feature files along with related step files by using a GIVEN-WHEN-THEN approach [8]. In the Cucumber environment, a feature file is a plain text file which contains structured natural language descriptions of the scenario that needs to be described/tested. Step files, on the other side, contains the execution directives obtained by translating features into actions that run on the system under test. The feature files could be written by all stakeholders while, typically, only developers write the code in the step files. The parser behind Cucumber is the Gherkin [https://github.com/cucumber/gherkin] which is written in Ragel and offers a syntax which is minimal and simple to read and understand.

JBehave [http://jbehave.org/] is a BDD (Behaviour-Driven Development) framework focused on Java platform. It focuses on specification rather than verification of a requirement. The framework offers the ability to work on file containing text written in a spoken language (story) and to define the scenarios using the GIVEN-WHEN-THEN pattern in the same way as offered by the Gherkin language. After that, the framework works on POJO containing directives JUnit-like used to perform the tests.

Spec2 [http://etorreborre.github.io/specs2/] is a library for writing executable software specifications for SCALA programming language according to the developer. In specs 2, it is possible to write the specifications in two ways: class (unit specifications) and full system (acceptance specifications). The former permits to specify a single class by interleaving specification text and specification code. The latter provides text that stands as one and the implementation code is elsewhere. It is generally used for acceptance or integration scenarios.

Taste [http://taste.tuxfamily.org/] is a tool-chain targeting heterogeneous embedded systems using a model-driven development approach. It is useful during prototyping, for software quality evaluation and review during the system development life cycle. It relies on existing language such as Abstract Syntax Notation to capture the software design and data, vhdl and simulink to specify the system behaviours and message sequence charts and Python for testing. A typical Taste application involves:
• Description of the logical structure of the system with ASN.1 or AADL
• Generation of the skeletons
• Writing code and models based on skeletons
• Verification of the models
• Building the system and deploying the models on it
• Monitoring of the system and run-time interactions

NatSpec [http://nat-spec.com/] is a commercial solution with which it is possible to describe requirements involving stakeholders using plain language and transform them into executable tests. The tests are suitable to be used during unit testing stage of system development life cycle. Similar to the others, the language uses minimalistic syntax with keywords such as Given, With and Assume. After the definitions in NatSpec language, the framework generated a JUnit test case with annotations and directives.

As it can be evicted from the description, the discussed solutions and related features are mainly suitable for test-driven scenarios defined during a Behaviour Driven Development (BDD). The DOSLang is a language suitable for requirement definition and is focused on Requirement Engineering process instead of testing. Some deliverables, like Graphical AST, could give immediate and direct support during requirement elicitation phases. Moreover, with the public key infrastructure (PKI) DOSLang signature validation, it offers a way to automatically achieve the common agreement among stakeholders without the needs for them of defining a protocol with which validate the requirements during each modifications.

III. THE SPECIFICATION LANGUAGE

This section describes the syntax of the DOSLang specification language and sections that forms a dosl listing. From the earlier stage of the DOSLang definition, the goal was to obtain a free-form constructs with very few constraints while ensuring the possibility of describing the scenarios in all the possible ways the stakeholder expects.

DOSLang was defined with a simple but effective idea: facilitate the requirements comprehension during a project design and their intercommunicability between a group of stakeholders, with the aim of minimizing the ambiguity and inaccuracies introduced by the spoken language. Then, the definition of DOSLang considered the creation of a unique environment where each stakeholder could formally verify the requirement through a PKI (Public Key Infrastructure)-based signed compilation process and participate at each stage of the Development Process.

While considering the needs arising during the requirements specification phase of any system development life cycle, we moved towards the idea of "verifiable requirements", understandable at the same time by stakeholders, customers, people, in an heterogeneous context composed of needs, attributes and constraints not necessarily bounded to a particular degree of comprehensibility and "true distinctions" factor.

In the following subsections, a description of the DOSLang features and syntax will be given.

A. Features

The main features of the DOSLang are the following:

• to specify stakeholders through RSA key: useful during the "Common Agreement" evaluation stage;
• to focus on the main elements of a project requirement: using the constructs, keywords and atomic units offered by the language;
• to involve relational aspect among entities;
• to describe directly the action related to entities and project subjects.

The only limit of the preparation of a dosl listing regards the cognitive aspects, perceptions and descriptive abilities of the stakeholders. Moreover, the keywords and sections the language introduces, can induce the dosl writer to focus his attention on the most important aspects of a requirement, but with the possibility of enriching its definition with todos and descriptions.

B. Syntax

The DOSLang is composed of a set of sections that forms the mandatory sections and a section that is not mandatory during the earlier stage of requirements elicitation, but can be added to the dosl during the subsequent phases by either the stakeholder or another subject who takes part in the project design during its lifecycle.

The mandatory sections are the following:

• INIT Stakeholder
• INIT ENTITY DECL
• INIT BEHAVIOR
• INIT REL DECL

The section order is not casual: it is the one expected by the DOSLang compiler. The not mandatory section is INIT BEHAVIOR entity. It describes the behavior of the entities declared in the INIT ENTITY DECL section.

Generally speaking, the mandatory sections, that are the first four INIT sections, are declared just once unless an error at compile time is issued. Section BEHAVIOR entity can be declared more than once. The mandatory
sections can be considered as the DECLARATIVE part of a dosl; while the (set of) declared BEHAVIOR entity section(s) represent its EXECUTIVE part. The mandatory sections precede the optional one and keywords are written in the uppercase notation. Each section is composed of a set declarations. There is no constraint in the order declarations.

Then, it is possible to interleave them and there is no limit in the number of the declared elements. DOSLang listing starts with the INIT STAKEHOLDER section which contains the keywords KEY, PKEY, PUB KEY. These keywords are equivalent and are followed by the RSA public key of the stakeholder. In particular, each KEY identifies a stakeholder which participates in the project requirements elicitation process. The dosl program must be signed with the corresponding private key in order to be validated correctly by the compiler. All the public keys specified will be used by the compiler during the validation stage of the dosl. DOSLang relies on a well-formed RSA key with the correct format and expected characters. The section following the STAKEHOLDER is INIT ENTITY DECL. In this section, an entity can directly be identified by its significant identifier (IDENTIFIED BY), or its structure or properties can be anticipated with data type (COMPOSED BY). The subsequent section is INIT BEHAVIOR. It describes aspects of a requirement that provide behaviors related to concepts inside the object oriented modeling technique. In particular, the STANDALONE keyword states that the indicated entity does not share any of its behavior (no inheritance or polymorphism); while SHARED implies that the listed entity shares something with other entities. DEPENDENT indicates that the indicated entity name inherits its behavior from superior entity (implies inheritance). These aspects will have a direct correspondence with the baseline produced during the Target Language Production Phase.

INIT REL DECL is the last block of the mandatory part of dosl. It permits to specify the multiplicity in an analogous manner of the RDBMS multiplicity. For example, lines of REL DECL in Listing 6 of Table 3 in the Example section, indicates the users that can use multiple clients and activate multiple accounts, or that just one user can belong to lab staff.

The executive section INIT BEHAVIOR entity is the last part of a dosl listing. The name may be misleading but this section should not be confused with the BEHAVIOR described above. The indentation is not mandatory (the compiler is TAB-unaware) but it helps to increase the dosl readability. In this executive section, one can find the actions representing the DOSLang statements and may contain familiar constructs such as IF statements (with or without ELSE), FOR statements and a particular statement called TEXT, which is one of the most important atomic units of DOSLang. The TEXT statements contains the description that, together with the other instructions, can form the core of a requirement and can be composed of sentences written in natural language, but with the possibility of including WEAK, OPTIONAL and STRONG keywords as the one specified in NASA ARM Tool word classification [1][2]. The use of these keywords is not mandatory, but they can help during the requirement specification by avoiding superfluous indications. In addition, DOSLang is able to evaluate quality metrics regarding the quality of dosl, but this is out of the scope of this paper. Each ACTION can be viewed as a function or method to be implemented and will be present in the baseline produced at the end of compilation process. If there is the necessity to define more than one action it is possible to use the multiple statement constructs delimited by <<>>.

The other aspects of the language regards the definition of comments, todos, descriptions, offered data type and logical operators. In particular:

- **DataType:** they are STRING, INTEGER, SET(array), FLOAT. These data types are used during the COMPOSED BY declaration in the ENTITY definition.
- **Logical operators:** they are used in conditionals and can be: LT (less than), GT (greater than), EQ (equal to) or DIFF (different from).
- **Comments:** they can be specified using the delimiters \** ... **/
- **Todos:** they start with \**TODO and end with TODO**/. Each of them may contain the list of todos related to a particular requirement or the dosl section.
- **Descriptions:** they start with n \**DESCRIPTION and end with DESCRIPTION**/. Each of them may contain useful descriptions related to a particular requirement or section.

Comments, Todos and Descriptions are ignored during the preprocessing phase of the dosl but can contribute in the evaluation of the cited quality metrics. In fact, their content is extracted during the quality metrics estimation phase for evaluating the ambiguity. Their semantics are different and induce the stakeholder in using them as appropriate as possible. The contents of comments, todos and descriptions will be propagated as they are in the produced baseline.
IV. THE ARCHITECTURE

DOSLang is characterized by a modular architecture (Fig. 1) in which each module has its own specific aim and produce the expected deliverable. This section presents each module together with its deliverables and in particular:

- to navigate the parse tree
- the graphical abstract syntax tree (ADT).
- the baseline produced after the dosl compilation process (TLG)
- the Common Agreement process.
- the quality metrics that the compiler evaluates by extracting information and keywords from the dosl code (DEM)

As shown in the architecture presented in Fig. 1, the compiler can produce the expected output through two main layers:

- Abstract DataType Builder
- Visitor Layer

The first has its logic in part wired directly into the parser, which is able to build the data structures useful for the lower layer. It performs this building operation during the validation and expansion of the rules regulating the syntax constructs.

Using the data structures built with the cooperation of the ADT builder, DOSLang is able to create the low-level tree representation of the dosl code with which the Visitor layer can provide its functionality to the other modules by applying the Visitor Pattern used for navigating the tree. All the other modules using the functionality offered by the upper layer Visitor, are able to perform their operations and produce their own deliverables. The Visitor is one of the core component of DOSLang. it uses the Visitor Pattern adapted from a design-oriented approach and provides a simple interface to other DOSLang components. This component is used to retrieve complex data among the DOSLang tokens offering at the same time a simple interface. Every method returns a subtree of the requested node/section in a breadth first manner using recursive methods. In particular, the implemented Visitor model gives to the other DOSLang modules the possibility of retrieving the whole dosl tree, a subtree with specific tokens like DESCRIPTION, TODO, TEXT and so on, together with all the other subtrees. Each of them is related to the specific DECLARATIVE or EXECUTIVE section. In this way, the modules of the lower layer can directly access the information content of their competence and navigate the tree in an appropriate way.

The DOSLang n-ary tree is an abstract data structure built after the compilation process, when all the data structures are ready to be manipulated: it is a structural representation of the program source code written in dosl. The nodes are the sections of the dosl program, such as ENTITIES, RELATION, etc., while the arcs represent the father-child relationship. The VisitorGraph class belonging to the ADT, adopts the dot syntax to generate a dot Intermediate Language deliverable that can be compiled with a dot compiler to generate the DOSLang tree in the image format. The hierarchical representation is made possible by a structure of 4-arity: each tree node is defined like a symbol node object, composed by pointers to other SymbolNode objects each of which is defined as follows: ID, FATHER, R BROTHER, LCHILD.

The DOSLang tree main advantage is that it can be fully considered as a documentation artifact itself and could be shared among stakeholders and used as annotation canvas during the requirement engineering stages. Some examples of the graph AST are given in Section 5.

The translation in the target language is a stage of the build process that is driven by the visitor. The idea is based on three key concepts:

1. to navigate the parse tree;
2. to use of the constructs associated in the specific target language for each visited node;
3. to produce the output _les in the specific target language.

To facilitate the use of DOSLang, a default target language translation engine within the compiler with a native support is made available. It is enough to extend a specific interface and extend the visitor VisitorImpl implementing the method extract(). In particular, the language could be enhanced with the support of the other target languages besides the default one.
The Lazy approach is the most direct and simple approach. It provides translation into the available default target language (i.e., Java) and gives the stakeholder the possibility of adapting the obtained baseline and making it executable by renaming the output obtained after the compilation. The native support is composed of the standard and advanced approaches. The advanced approach is the technique with which one can implement a new interface and create a native implementation of Target Rules, File Generator and Visitor Pattern. The Fig. 2 shows the order to start the translation process. An example of the output file obtained as target is shown in Listing 5 of Table 2.

The Common Agreement is the process adopted by the DOSLang in order to implement a mechanism of validation/acceptance and non-repudiation of the dosl code through a PKI-based schema. With this process it is guaranteed that each stakeholder agrees with the requirement specification implemented in the dosl code. It induces stakeholders to cooperate each other for obtaining clear requirement specified in the dosl by improving the code, adding more elements, describing more scenarios and so on.

The process is composed of the following steps:

1. Import RSA pub. keys
2. Sign dosl with RSA private keys
3. Check the signatures with the keys specified in dosl
4. Iff the number of keys matches the signatures goto 0 else goto 1
   0 Common Agreement succeeded and compilation process produces deliverables
   1 An error occurred at compilation time and there are no deliverables produced until all stakeholders agree with dosl

The stakeholders participating in the dosl implementation, must generate a RSA key pair and import the public one into the dosl inside the STAKEHOLDER section. Therefore, the stakeholder can add/check/edit/evaluate the dosl code and submit his changes by signing the dosl with his private key.

In a second stage, if changes are made, the stakeholder resign the code. At this point he cannot be able to compile the dosl anymore. He must submit the dosl to ALL the other stakeholders listed in the section, which must approve the dosl by signing it again if they wants to make compilation process successful and obtain the deliverables. Obviously, the signature process undertaken by each stakeholder, is performed if the code is considered valid/compliant in relation to the information contained in the listing. The Common Agreement implicitly creates a cyclic process in which each stakeholder, in turn, becomes a requirement reviewer, its developer or its contributor.

The last DOSLang module, performing a parsing of the dosl code and using information extracted from the symbol table produced during the compilation process, is able to evaluate the requirements specified in the code by calculating its ambiguity percentage through the application of the DOSLang Evaluation Metrics. DOSLang provides a templating system with which it's possible to customize the evaluation of KEYWORDS and their CATEGORIES, giving the possibility to create a best _t with the semantics of the information contained in the dosl.

V. EXAMPLE

This section presents a simple DOSLang application regarding the definition of a project requirements specified to design an InternetCafe software system. You can think of it as the project requirements snapshot implemented in dosl at an intermediate stage of the project lifecycle.

You can refer to Tables 1 and 2 for DOSLang code examples discussed here. The first section in the DOSLang source code is the INIT STAKEHOLDER section in which we define all the Stakeholders of the software system.

The Stakeholders are identified by their respective RSA PubKey. In the listing 1 of Table 1, we define 3 stakeholders of this project. The public keys are trimmed to ensure readability. We also define a general description of the whole System between the section delimited by DESCRIPTION and TODO. The DESCRIPTION and TODO sections are also used to provide useful descriptions of a specific requirement.

Now, we have to define all the Entities involved in the System. For example, we define one of the main entity of the SIC system like 'users' identified by: id, password, information field and account type. In the listing 3 of Table 1, we also define guest, clients, students, and the lab staff, and all the models which compose the whole system, like the sic server, sic clients, and the accounts.

We also provide a description regarding the monitor component and sysop role using DOSLang keywords in order to describe in a natural-like language a particular
scenario or additional requirement specification. Note that the description is inside the Entity section in order to emphasize its information on the specific entity it refers to. After that, we have to define all the 'behaviours' of the system.

Table 1. SmartICafe dosl

| Listing 1: smartICafe stakeholders |
|------------------------------------|
| /* START DECLARATION SECTION */    |
| INIT JAVA33333333;                |
| PUBLIC KEY A8G6C9SgBhC3D2R8 ...; |
| KEY MGMA8G6C9SgBhC3D2R8 ...;    |
| END JAVA33333333;                 |

| Listing 2: smartICafe descriptions and todos |
|-----------------------------------------------|
| /* DESCRIPTION */                             |
| Smart Internet Cafe (sic): open source OS for secure Internet surfing |
| Paradigm: Cl1/Srv                             |
| DESCRIPTION **/                               |
| /** TODO */                                   |
| - define/separate UI interfaces for Colleges and Internet Cafe |
| - expand monitor entity                       |
| TODO **/                                     |
| /** DESCRIPTION */                            |
| Main targets: Colleges Labs. and Internet cafes |
| **/                                          |
| /** TODO */                                   |
| - define/separate UI interfaces for Colleges and Internet Cafe |
| - expand monitor entity                       |
| TODO **/                                     |
| /** DESCRIPTION */                            |
| Main targets: Colleges Labs. and Internet cafes |
| **/                                          |

For example, 3 behaviors are defined: user, clients and accounts. These behaviors, will correspond to the implementation of a particular responsibility in the target Language. We also define another behavior called 'lab staff' and all the inheritance relation among the behaviors. Then, we define the relations, i.e. 1 monitor can be used by one or more sysop.

Figure 3. Part of the Behavior users.

Listing 3: smartICafe entities

```java
INIT ENTITY DECL;
ENTITY users COMPOSED BY [STRING: userid, STRING: pwd, STRING: info, SET: account_type];
ENTITY accounts IDENTIFIED BY accounts;
ENTITY clients IDENTIFIED BY clients;
ENTITY labstaff IDENTIFIED BY labstaff;
ENTITY students IDENTIFIED BY students;
ENTITY guest IDENTIFIED BY guest;
ENTITY sic_server IDENTIFIED BY sic_server;
ENTITY sic.clients IDENTIFIED BY sic.clients;
ENTITY sic IDENTIFIED BY sic;
/** DESCRIPTION */
The monitor MUST be a sysop. The sysop MUST be able to monitor and log all users' activities as appropriate.
It is connections AWARE.
/** ENTITY monitor COMPOSED BY [INTEGER: traceid, STRING : logged_activity];
END ENTITY DECL;
```

Listing 4: smartICafe behaviors

```java
INIT BEHAVIOR;
STANDALONE monitor;
SHARED users;            // /* DESCRIPTION */
SHARED clients;          // /* DESCRIPTION */
SHARED accounts;         // Shared accounts
DEPENDENT clients<>users; // /* DESCRIPTION */
DEPENDENT guest<>clients; // /* DESCRIPTION */
DEPENDENT users<>accounts; // /* DESCRIPTION */
DEPENDENT labstaff<>users; // /* DESCRIPTION */
DEPENDENT students<>clients; // /* DESCRIPTION */
DEPENDENT users<>accounts;
END BEHAVIOR;
```

Figure 4. SmartCafe java baseline users.

After the declarative section, we need to implement all the use case of the system, assumed that we want to adopt a use case driven approach for the actions and activities related to the SIC software system.

In the listings 7 and 8 of Table 3, the stakeholder tries to define aspects related to authentication, monitoring and other operations, such as those related to the 'check system', 'check connection', and 'checkbrowsers' use cases, where each client must check that the proper browser is installed.
The DOSSLang syntax specification is flexible because you can specify a simple description of the requirements also in all the cases where a particular requirement was, for example, not completely defined yet or, as in the case of verify connected or check user type, we have a more complete vision of a requirement by defining also its conditional control flow.

In the listing 8 of Table 3, the action behavior defined with the name ‘check user type’ must implements the logic controlling the type of the user connecting to the system in three way:

- if the user is a client, the account type is set as client;
- if the user is a lab staff member, the method set the account type to admin;
- account type become guest otherwise.

All these checks are mandatory for the system to be implemented. After the compilation process, the DOSSLang generated: the graphical AST; the baseline in Java language; and the quality metrics related to the dosl code implemented. A part of the obtained deliverables is shown in Figure 3. It represents an extract of the Graphical AST obtained after the compilation process, and includes some actions related to particular project requirements, such as the checking authentication operation which must validate the users’ authentication and permit access if credentials are correct.

It is important to note that the portions of trees showed in Figure 3, contains many TEXT nodes: these nodes represent the descriptions and indications associated to the specified action. In the figures these nodes are generic TEXT in order to maintain the readability of the tree, but the real reference is to the related action as indicated in the dosl code.

Even if this is out of the scope of this paper, for sake of completeness, we cite one of the metrics estimated by the compiler. It is the ambiguity level that characterizes the dosl. An example of the output is: It indicates that the level of ambiguity is at 22% and that

### Table 2. TLG output

| Listing 5: TLG output example |
|---|
| public entity Users { |
|   public check_user_type { |
|     if [users == clients] { |
|       // MANDATORY set user account_type as client |
|     } else { |
|       if [users == labstaff] { |
|         // MANDATORY set user account_type as admin |
|       } else { |
|         // MANDATORY set user as guest |
|       } |
|   } |
| }} |

### Table 3. SmartCafe dosl 2

| Listing 7: SmartCafe actions |
|---|
| /* ACTIONS Section: adopt doping metrics used et cetera */ |
| INIT BEHAVIOR; |
| ACTION check_connection {} | \* MANDATORY check client's connection \* | |
| ACTION check_browsers FOREACH clients {} | \* MANDATORY check all browsers \* | |
| ACTION check_server {} | \* MANDATORY check server \* | |
| ACTION authenticate {} | \* MANDATORY authenticate user \* | |
| ACTION set_operations {} | \* MANDATORY set all available operations \* | |
| END BEHAVIOR; |

| Listing 8: SmartCafe other actions |
|---|
| ACTION check_user_type IF [users EQ clients] THEN |
| { \* MANDATORY set user account_type as client \* } |
| ELSE |
| { \* MANDATORY set user account_type as admin \* } |
| END |

| INIT BEHAVIOR; |
| --- |
| /* DESCRIPTION TOBE_DESCRIBED */ |
| ALGORITHM; |
| END BEHAVIOR; |
only few parts of the dosl implementation needs to be improved in order to increase the comprehension of each requirement specified for all stakeholders.

VI. CONCLUSIONS

We presented the DOSLang, a specification language built to allow stakeholders to specify the requirements of a project during its lifecycle.

Together with its free-form syntax, constructs and deliverables obtained, the main features offered by the language and its compiler are the possibility to use specific keywords in order to decrease the ambiguity, the possibility to share the project requirements implemented in the dosl code and the use of the Common Agreement process to validate and approve them. After the compilation procedure, the DOSLang is able to create a common baseline in a target language. This offers the possibility of realizing the requirements specified in terms of software implementation.

REFERENCES

[1] L.H. Rosenberg, T.F. Hammer, L.L. Huffman, "Requirements, Testing and Metrics" - In 15th Annual Pacific Northwest Software Quality Conference, 1998

[2] L.H. Rosenberg, T.F. Hammer, "A Methodology for Writing High Quality Requirement Specifications and for Evaluating Existing Ones", NASA Technical Reports Server (NTRS), 1999

[3] A. Umber, I.S. Bajwa, "Minimizing Ambiguity in Natural Language Software Requirements Specification" - Digital Information Management (ICDIM), 2011 Sixth International Conference, 2011

[4] B. Bertran, J. Bruneau, D. Cassou, N. Loriant, E. Ballard, C. Consel, "DuSuite: a Tool Suite To Develop Sense/Compute/Control Applications", Science of Computer Programming, Fourth special issue on Experimental Software and Toolkits, 2012

[5] J. Mylopoulos, J. Castro, "Tropos: A Framework for Requirements-Driven Software Development", Information Systems Engineering: State of the Art and Research Themes, Lecture Notes in Computer Science, 2000

[6] E. Yu, "Modelling Strategic Relationships for Process Reengineering", Ph.D. thesis, Department of Computer Science, University of Toronto, 1995

[7] M. Smialek, T. Straszak, "Facilitating Transition from Requirements to Code with the ReDSeeDS Tool", Requirements Engineering Conference (RE), 20th IEEE International, 2012

[8] A. Hellesoy, Matt Wynne, "The Cucumber Book: Behaviour-Driven Development for Testers and Developers", Pragmatic Bookshelf, 2012, p.313.

[9] G.M. Christel, Kyo C. Kang, "Issues in Requirements Elicitation", Technical Report CMU/SEI-92-TR-012 ESC-TR-92-012, 1992

[10] Leite, P.J. Cesar, "A Survey on Requirements Analysis", Advanced Software Engineering Project Technical Report RTP-071, 1987