A Combinator Language for Software Quality Reports

Pedro Martins, Jo ão P. Fernandes, and Jo ão Saraiva

***Abstract*—Quality assessment of open source software isan important and active research area. One of the reasons for this permanent interest is a consequence of Internet popularity. Nowadays, programming involves looking within a large set of open source libraries and tools that may be reused when developing our software applications. In order to reuse such open source software artifacts, programmers not only need the guarantee that the reused artifact is certified, but also that independently developed artifacts can be easily combined into acoherent piece of software. In this paper we describe a domain specific language that allows programmers to describe in an abstract level how software artifacts can be combined into powerful software certification reports. This domain specific language is an important system of a web-based, open-source software certification portal. This paper introduces the embedding of such a domain specific language as a combinator library.**

***Index Terms*—Process management, combinators, attribute grammars, functional programming.**

I. Introduction

The advent of the Internet is changing our lives. Not only is it changing the way we live, but also the way we develop our software. While in the last century building software applications was mainly supported by programming languages and their libraries, which provided the necessary support to build software applications, nowadays, the way we develop has changed: programming languages still offer supporting libraries, but there are many more resources available in the internet. These wide set of resources can be other powerful off-the-shelf reusable libraries and tools, usually available as Open Source Software (OSS).

This fact influence the way we program since developing a particular software tool/library may be, in most cases, a matter of looking for the right (open source) software/libraries solutions already available. Indeed, the Internet encourages sharing our software. This new style of developing software, however, needs to handle three important issues:

1) Firstly, because there is so much OSS available in the Internetit is difficult to select the right tool/library. Thus, we need anappropriate framework to support the analysis of the available alternatives.
2) Secondly, because we may reuse different software artifacts, developed in different contexts, we need to integrate them into acoherent piece of software.
3) Thirdly, because we are reusing OSS, we may need to guarantee that it satisfies certain properties before reusing it. For example,when developing software that handles credit card information we mayneed the guarantee that a piece of software to be reused conforms tospecific security guarantees. On a different context, if weare developing software for embedded systems, we may need toguarantee that a reused library implements optimal memory management.

In the past, we have presented a web portal for the analysis and certification of Open Source Software (CROSS) that aims at improving on these three issues [1]. The portal works as a repository for tools that certifies source code. By the certification of a piece of software we understand the process of analyzing its source code while producing an information report about it.

In this paper we continue on developing this heterogeneous and distributed analysis system, focusing on the creation of reports. On a web portal that manages software analysis by applying a sequence of pre-chosen, individual and self-contained tools, managing their results implies dealing with a huge amount of heterogeneous information, both in their type and context. For example, the result of tool *A* can be *HTML* code, while the result of tool *B*can is tabular data in the form of comma-separated values (CSV). And while these are plain text, they have different contexts and meanings and should, therefore, be managed differently. Furthermore, other tools can produce Scalable Vector Graphics (SVG), showing dependency graphs, or simply JPEG or PNG figures with statistics information.

This document is organized as follows: in Section II we provide an overview of the motivation and potential challenges this work faces. In Section III we introduce our combinator language together with small examples of its usage. In Section IV we present works that relate to ours, and in Section VI we conclude.

II. Motivation

The techniques for analyzing source code to produce in the context of our web portal for the analysis and certification of Open Source Software (CROSS) [1], should, either individually or combined with others, result in the production of reports called *Certifications*. *Certifications* are often composed by smaller units that are capable of communicating with each other in order to achieve a state where the overall mechanics of each unit and the flow of information among them is capable of producing quantifiable results. In the remaining of this paper, we will address ourselves to these smaller units that contribute to a general

Manuscript received November 4, 2012; revised January 31, 2013. This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Govern- ment through FCT - Foundation for Science and Technology, project ref. PTDC/EIA-CCO/108995/2008.

Pedro Martins and Jo ão Saraiva are with the HASLab / INESC TEC, Universidade do Minho, Portugal (e-mail: prmartins@di.uminho.pt, jas@di.uminho.pt).

Jo ão P. Fernandes is with HASLab / INESC TEC, Universidade do Minho, Portugal (e-mail: jpaulo@di.uminho.pt), and he is also with Universidade da Beira Interior (e-mail: jpf@di.ubi.pt).

goal as Components.

In detail, a Component is a bash tool that is capable of accessing and producing meta-data via the standard *UNIX* communication channels (the standard input, *STDIN*, and the standard output, *STDOUT*). Also, a component must be able of receiving arguments that define the type of the information that is received via STDIN and the type of the information that is to be channeled through *STDOUT*.

Such tools can have functionalities that range from the need to maintain software as easily as possible to the removal of its bugs and the improvement of its overall characteristics [2]–[7].

What is more, different, heterogeneous and distributed teams often develop components independently, and their development and integration in more complex *Certifications* closely follows the philosophy of open source software development itself.

In Fig. 1 we sketch the flow of information that has been implemented in order to produce a sample *Certification* called *Certification 2*. This is a *Certification* that expects *Java* programs and that analyzes them according to three distinct sub-processes that are independent with respect to each other and therefore can be executed in parallel. One of these processes chains a series of software units, namely *Interface Analysis* and*csv2Report*. *Java Metrics* is composed by a simple, singular tool and the other, which is itself a Certification called*Certification1*, implements a *Cyclomatic Dependency* analysis while producing an information report by itself.

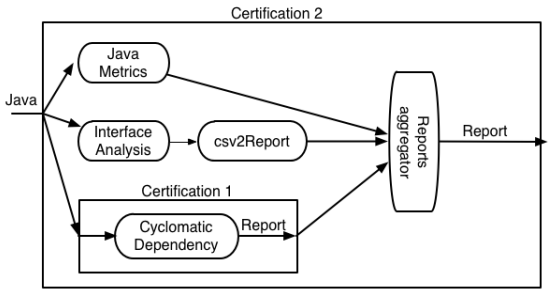


Fig. 1. The flow of information implemented in Certification 2.

We have already addressed in [8] the problem of defining the flow of information through various Components using a Domain Specific Combinator Language for Process Management. Through this DSL, the user is capable of specifying the flow of information through a number of tools, both through sequential or parallel processes, while type checking and process management and isolation are automatically guaranteed.

The main issue with dealing with the flow of information among these tools is that they will inevitably produce different information, and by different we mean completely diverse in character and content, while our web portal must always produce a standard-type report. Although this technique elegantly solves the problem of creating multi-process Certifications, there is no control support on the output they produced. In fact, the final report will be composed by the union of all sub-results and one tool, *Interface Analysis*, had to be supported through the implementation of a translatorthat ensured that it would be capable of producing information that can be concatenated with all the other results.

The DSL that we present in this paper ensures that all the information can be composed onto a final report, which is an *XML* file, but forces the sequential computations to also produce an *XML* report, forcing their programmers to either implement that feature or to implement translator tools. What is more, there is no support, on the final report, for the usual subdivision and structural support documents have. The final report produced by our web portal should be composed of results that are sub-parts of bigger results, similar to chapters and sections usually found on documented information.

In Fig. 2 we show a possible definition for the result of *Certification 2*.

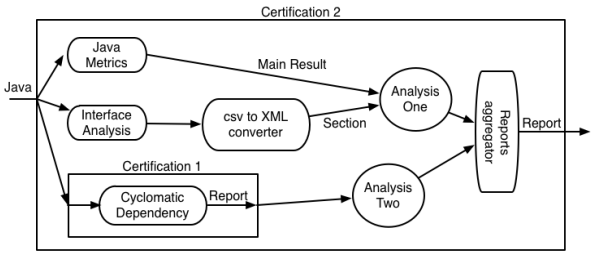


Fig. 2. Report specification in Certification 2.

While the flow of information is kept unchanged from Fig. 1, the report is now subdivided and structured. The three initial processes that compose this Certification do not generate three different reports. Instead, *Interface Analysis* outputs a sub-result of *Java Metrics*. In this context, *Certification 2* produces the second part of the report, which is structurally independent from the other two processes. Also, we introduced a special set of combinators, which take common formats and wrap them around *XML* files in order to be united to produce a final report.

We achieve the combination of outputs by forcing the final result shown to the user to be an *XML* file whose schema we have previously defined. All the tools and analysis will necessarily be a part of this *XML* file. To achieve this, we have defined a set of combinators that take common data types and transform them into pieces of our *XML* report and that guarantee formats conformity.

We believe the combination of our process management DSL together with this work will create an optimum environment for software analysis. Not only management is easily controlled to create powerful analysis, through the work we present in this paper this analysis is a document which is customized, easy to read and to understand and therefore has the exact characteristics the user wants the final report to have.

III. A Combinator Language for Report Management

The combinator language we is written in Haskell [9] and propose helps controlling the final report structure and content by providing a set of constructors that allow the easy manipulation of such information. What is more, its complexity is scalable, as the user can simply define the order and position of the information to create a report that is easy to read, add contextual information between results or even tune the name of a section to create expressive software reports.

A. Datatypes

We start by presenting the datatypes that support our combinator language. These are used to support the intermediate structures between the reports specified with the combinators and the final XML document that our framework creates.

These datatypes are somehow similar to the structure of XML documents themselves, with a header that carries information related to the XML version and the encoding, and a set of data constructors that behave like XML attributes, on a tree-based structure, and that carry information of another constructors. The datatypes we have defined are presented next:

```
data XML = XML Header [Section]
         | Init

type Header = String

data Section
  = NoTitleSection Result
  | TitleSection Title Result
  | TitleWithSubSections Title [SubSection]
  | NoTitleWithSubSections [SubSection]

data SubSection
  = NoTitleSubSection Result
| TitleSubSection Title Result

type Title = String
```

XML contains a header and a list of `Sections`. Sections can contain results or a set of SubSections, with both Sections and SubSection giving the option to customize the name of that part of the report, although this is not mandatory.

Together with these datatypes we present a set of combinators that aid in creating XML reports and in creating information within the XML datatype. We also provide functions that perform the transformation from this XML datatype to concrete XML text documents.

B. XML Begin Combinator

First, we present the combinator that represents the beginning of a report. This combinator is mandatory, and it is the only one that must always appear in a report description. Using this combinator a user can specify both a very simple report, that represents only the information of a Tool, and a very complex report with differentSections and SubSections and with customized names.

Next, we present an example of the simple usage of this combinator:

```
Init >| (“Custom Title”, imageTranslator
tool1)
```

This is a very simple example, where only one tool constitutes the report. Init is just a starting flag, designed to help the user initiate the report. Although it is mandatory in all reports, it has no semantic purpose and exists only to provide syntactic sugar to our language.

When the Begin combinator receives as argument only a tool, it automatically creates a Section in the XML report. In fact, our reportr will always have at least one or more Sections and 0 or more SubSections.

Similarly to other combinators, the user can always provide a custom title for the report, as shown in the next example.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<section title="Custom Title">
   Result of tool1
</section>
```

The resulting XML produced by this report is shown next:

As stated earlier, a report will always have at least one Section that we can clearly see in the XML document. In this case, only one result is presented, which is produced by “tool 1”. Another important note is that our framework automatically indents the XML report to make it easier to read it or to change it, if necessary.

C. Section Combinator

The Section combinator is mandatory only when the user pretends to have two or more Sections in his/her report. As we have seen earlier, it is possible to create one report without using this combinator, as long as the user only wants to have the result of one tool.

When it is desired to create more than one Section, this combinator can be used to sequence information, which may have customized Section titles (later we will see that this combinator can also be used to sequence SubSections).

Next, we present a report with three Sections, which represent the results of tool1, tool2 and tool3:

In this example the user chose to customize a few Sections, as it is the case with the second Section, which has the title`Result of Tool2`, and the case of the third Section, which has the title: `Result of Tool3`. The first Section is left with a static name. The XML report created with this sequence of combinators is shown next:

```
Init >|(imageTradutor tool1)
>- ("Result of Tool2", pdfTranslator tool2)
>- ("Result of Tool3", htmlTranslator tool3)
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<section>
Result of Tool1
</section>
<section title="Result of Tool2">
Result of Tool2
</section>
<section title="Result of Tool3">
Result of Tool3
</section>
```

The produced XML Report has the expected three Sections defined with the combinators, where the last two have a “title” attribute that carries the customized title the user chose to give these Sections. The Section left without a name has no attribute. By doing so, the tool that analyzes and transforms the XML Report has the responsibility to do whatever the user wants with these “empty titles” Sections. In the context of our website, for example, we chose to set default names, such as Section I, Section II, etc by defining so on an“EXtensible Stylesheet Language” (XSL) document that takes the XML reports and applies them to our web page, but different users with different needs might need to hand these differently.

D. SubSection Combinator

Similarly to the Section combinator, SubSection is not mandatory, as the user can chose to have a report composed only of Sections. Nevertheless, this combinator is important

because it increases the expressiveness of reports which can be composed of various groups of information in Sections, with the SubSections holding the actual results.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<section title="Memory Tests">
<subsection>
        pdf_path
</subsection>
</section>
<section title="Usability Tests">
<subsection>
        This is just text
</subsection>
<subsection title="Result of Tool2">
        pdf_path
</subsection>
<subsection>
        csv_path
</subsection>
</section>
```

This combinator is composed of two primitives: one primitive that constitutes the beginning of a sequence of SubSections, `beginSubsection`, and which can be used alone and is mandatory whenever the user wants to add SubSections, and another primitive, `>--` that combines SubSections.

The simplest example of an application of a SubSection is one where the user wants to have a report with only one Section and only one SubSection, as shown next:

```
t6 = Init >| ("Section with Subsections",
(beginSubsection $ textTranslator tool6))
```

In this example, the user used the Begin combinator, and applies the SubSection immediately after. This procedure automatically creates a Section to contain the SubSection. It is also important to notice that the user chose to give the Section a name, Section with Subsections but leave the SubSection anonymous. As is common with our combinator framework, the user can always choose to give custom titles to Sections or SubSection. Next, we present the XML document generated:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<section title="Section with Subsections">
<subsection>
        This is just text
</subsection>
</section>
```

The SubSection combinator can be used in more practical examples, such as to set a group on results in the report. The example shown next presents a usage of this combinators that shows this particular case where the report is splitted into groups of results:

```
Init
    >| ("Memory Tests",
(beginSubsection $ pdfTranslator tool2))
    >- ("Usability tests",
    (beginSubsection $ textTranslator
  tool6)
    >-- ("Result of Tool2", pdfTranslator
    tool2)
```

This example is a little more complex, and shows the creation of a report with two groups of tests, in two Sections named, respectively, `Memory` Tests and Usability tests. The first Section contains only one SubSection, with the result of tool 2. The second Section, which contains the group of results labeled Usability Tests, is more complex and contains a set of three SubSections, each one with the result of one analysis tool. Next, we present the XML document produced by this report:

The XML report clearly shows this distinction between two groups of tests into two sections. Also, the three SubSections in the second Section are also pretty clear. We believe our framework presents not only an elegant method to write documentation, but also produces an elegant, easily to understand and manipulate XML document.

E. Translators Combinator

We have seen, throughout the examples presented previously, that we usually use primitives like "imageTranslator" or "csvTranslator", which we apply to tools in order to create results to produce in our reports.

These translators are necessary because, due to the nature of our web portal, we use a very wide and heterogeneous set of tools, which must always be translated into a XML report. One simpler option would be to force the tool programmers to write tools that comply with a specific output format, but we believe this is a burden to programmers and it will make producing tools for our web portal harder. Therefore, we have created a set of translators which have the responsibility of taking common formats of information, such as "CSV" or "PDF" files, and make them available in our report. To some formats, these translators might do things as simple as insert the path of the resulting file, being the responsibility of presenting them to the user carried by whatever mechanism is used to handle the report. For example, in the context of our web portal, we translate the XML reports to HTML code, where files like images or "PDF"s are displayed using HTML primitives.

F. Creating an XML Document

We have seen and presented a set of combinators that allow the definition of XML structured documents. This combinators are not translated directly into an XML document. Instead, they are translated into an intermediate datatype, presented in Section A, which represents an abstract definition of our final XML report. This intermediate datatype is important, because it facilitates the extensibility of our combinators and the introduction of new ones: as long as the result belongs to this intermediate datatype, we always guarantee its traduction to an actual XML document. What is more, if static analysis needs to be performed, it is easier to do it in this intermediate data type rather than on the actual XML document (although both options are available to the user).

Despite its usefulness, this intermediate, abstract representation of a report must be translated into an actual XML text document.

Next, we present the function responsible for such transformation:

This function is presented as a Class in Haskell, with an instance for every constructor from our intermediate data types.

We have built this function so that it creates indented reports, to make them easier to understand by the user, but also to be easily configurable.

```
indent = 4

sectionElement = "section"
subsectionElement = "subsection"

class ToXML a where
  toXML :: Int -> a -> String

instance ToXML XML where
  toXML 0 (XML h body) = h ++ "\n" ++ (toXML 0 body)
instance ToXML [Section] where
  toXML i l = concat $ map (toXML i) l
instance ToXML Section where
  toXML i (NoTitleSection r) = spaces i ++ "<"++sectionElement++">" ++ (toXML (i+indent) r) ++
spaces i ++ "</"++sectionElement++">\n"
  toXML i (TitleSection t r) = spaces i ++ "<"++sectionElement++" title=\""++t++"\">\n" ++
(toXML (i+indent) r) ++ spaces i ++ "</"++sectionElement++">\n"
  toXML i (NoTitleWithSubSections s) = spaces i ++ "<"++sectionElement++">\n" ++
(toXML (i+indent) s) ++ spaces i ++ "</"++sectionElement++">\n"
  toXML i (TitleWithSubSections t s) = spaces i ++ "<"++sectionElement++" title=\""++t++"\">\n"
++ (toXML (i+indent) s) ++ spaces i ++ "</"++sectionElement++">\n"
instance ToXML [SubSection] where
  toXML i l = concat $ map (toXML i) l
instance ToXML SubSection where
  toXML i (NoTitleSubSection r ) = spaces i ++ "<"++subsectionElement++">\n" ++
(toXML (i+indent) r) ++ spaces i ++ "</"++subsectionElement++">\n"
  toXML i (TitleSubSection t r ) = spaces i ++ "<"++subsectionElement++" title=\""++t++"\">\n" ++
(toXML (i+indent) r) ++ spaces i ++ "</"++subsectionElement++">\n"
instance ToXML Result where
  toXML i (IMAGE r) = spaces i ++ r ++ "\n"
  toXML i (PDF  r ) = spaces i ++ r ++ "\n"
  toXML i (CSV  r ) = spaces i ++ r ++ "\n"
  toXML i (DOT  r ) = spaces i ++ r ++ "\n"
  toXML i (HTML r ) = spaces i ++ r ++ "\n"
  toXML i (TEXT r ) = spaces i ++ r ++ "\n"

spaces :: Int -> String
spaces i = take i aux
  where aux = ' ':aux
```

As we can see by the code above, it is very easy to customize the final, generated XML document. The elements for Section and SubSection have flags that allow the user to easily customize the generated XML document, in case he wants to, and even the indentation, i.e., the number of spaces used for pretty printing the report, is something easy to change.

We believe this translator from our abstract datatype to an XML report is not only powerful enough to create easy to read information, it's easy customization allows the adaptation of this framework to whatever needs the final user has, either in the context of our web portal or in any other contexts analysis reports must be easily produced.

IV. Related Work

Several projects have focused on the analysis and assessment ofsoftware, being the *Squale* project [10] *QSOS* [11] and the *Alitheia Core* [12] important examples of these.

In comparison with our work, we believe that potential users of these systems see their extensibility and improvement limited by custom schemas of information or domain-specific languages for plug-ins development. This is either because these projects are based on assessment models for OSS, or because they create unified storage systems or even because they imply the usage of frames of reference to create an evaluation that often depends on axis of criteria. What is more, none of these systems has an option to customize its results: the user is always stuck with pre-defined information representations with no or very little customization.

V. Future Work and Conclusion

In this paper we have presented a combinator language for software quality reports. Through it a user can easily define software reports structurally by organizing its contents into groups of information composed by sections and sub-sections and even customizing their titles.

We believe the advantages of our system are two fold: the combinators not only create an intuitive, simple and powerful environment to create *Certification* reports, but it also supports processes outputs management in general and is upgradable to create not only *XML* information, as it is the case, but also any type that suits the users needs.

Also, because we use intermediate data types and defined functions in order to make they perform differently easy, we believe our framework has a good potential to be adapted and used in other information and analysis site who force into the user pre-defined representations information, both on their context and on their syntax.

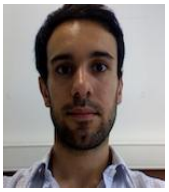
One potential analysis technique that could be applied to this Combinatory language is the analysis and validation of our intermediate structure by expressing them as Attribute Grammars (AGs) [13] i) for once, we are analyzing tree-based structures, for which the AG formalism is particularly suitable; ii) secondly, because AGs have a declarative nature which in our context contributes to intuitive implementations that are easy to reason about and to further extend. In fact, we believe that it would be simple to integrate in our framework advanced AG-based and well studied techniques such as the detection of circular dependencies [14] and the use of higher-order attributes [15].

References

[1] P. Martins, J. P. Fernandes, and J. Saraiva, "A web portal for the certification of open source software," in *Proc. of the 6th International Workshop on Foundations and Techniques for Open Source Software Certification Conference*, LNCS (to appear).

[2] M. Haigh, "Software quality, non-functional software requirements and it-business alignment," *Software Quality Control*, vol. 18, no. 3, pp. 361-385, September 2010.

[3] D. Stavrinoudis, M. Xenos, P. Peppas, and D. Christodoulakis, "Early estimation of users' perception of software quality," *Software Quality Control*, vol. 13, no. 2, pp. 155-175, June 2005.
[4] R. G. Dromey, "Software quality prevention versus cure?" *Software Quality Control*, vol. 11, no. 3, pp. 197–210, July 2003.
[5] D. N. Wilson and T. Hall, "Perceptions of software quality: a pilot study," *Software Quality Control*, vol. 7, no. 1, pp. 67–75, May 1998.
[6] S. Chulani, B. Boehm, J. Verner, and B. Wong, "Workshop description of 4th work- shop on software quality (wosq)," in *Proc. the 2006 international workshop on Software quality*, New York, NY, USA, ACM, 2006, pp. 1–2.
[7] J. Cunha, J. P. Fernandes, J. Mendes, P. Martins, and J. Saraiva, "Smellsheet detective: A tool for detecting bad smells in spreadsheets," in *Proc. the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing*, Washington, DC, USA, *IEEE Computer Society*, 2012, pp. 243-244.
[8] P. Martins, J. P. Fernandes, and J. Saraiva, "A purely functional combinator language for process management," in *Proc. the 1st Symposium on Languages, Applications and Technologies*, Braga, Portugal, pp. 51-69.
[9] S. P. Jones, *Haskell 98 Language and Libraries: the Revised Report*, 2003.
[10] Squale: Front page. (August 2012). [Online]. Available: http://www.squale.org
[11] QSOS: Front page. (August 2012). [Online]. Available: http://www.qsos.org
[12] Alitheia Core: Front page. (August 2012). [Online]. Available: http://www.sqo-oss.org
[13] D. E. Knuth, "Semantics of context-free languages," *Mathematical Systems Theory 5*, vol. 1, pp. 95-96, 1971.
[14] J. P. Fernandes and J. Saraiva, "Tools and libraries to model and manipulate circular programs," in *Proc. the ACM SIGPLAN 2007 Symposium on Partial Evaluation and Program Manipulation*, ACM Press, 2007, pp. 102–111.
[15] D. Swierstra and H. Vogt, "Higher order attribute grammars," *International Summer School on Attribute Grammars, Applications and Systems*. *LNCS*, Springer-Verlag, vol. 545, pp. 48–113, 1991.



Pedro Martins is a Ph.D student at the Department of Informatics, University of Minho, Portugal. He obtained his Master's degree in 2001, where he developed language extensions to the Matlab programming environment. Currently he is researching software quality methodologies for Open Source Software.

João F. Paulo has graduated in Mathematics and Computer Science from the University of Minho, in 2004 (best of class), where he conducted his graduate thesis.Later, in March 2009, he received his Ph.D. degree from the same university, following his work on the *Design, Implementation and Calculation of Circular Programs*. In his research he pursues rigorous ways to reason about programming, which he has successfully been able to apply in the context of functional programming, spreadsheets, language engineering and bidirectional transformations, and in the context of several research projects.

João Saraiva studied systems and software engineering (Licenciatura em Engenharia em Sistema e Informáticos, 1986-1991) and Computer Science (MSc. defended in July 1993) at the Department of Informatics at Minho University, after which he went to Utrecht University, The Netherlands, where he worked under Prof. Dr. Doaitse Swierstra supervision on his Ph.D. thesis. He defended the thesis in Utrecht in December 1999, and, then, returned to Minho University where he is now an auxiliar professor at the Department of Informatics.