A Framework for Modular and Customizable Software Analysis

Pedro Martins[1]*, Nuno Carvalho[2], João Paulo Fernandes[1,3], José João Almeida[2], and João Saraiva[1]

[1] High-Assurance Software Laboratory (HASLAB/INESC TEC), Universidade do Minho, Portugal
[2] Computer Science and Technology Center (CCTC), Universidade do Minho, Portugal
[3] Reliable and Secure Computation Group ((rel)ease), Universidade da Beira Interior, Portugal
{prmartins,narcarvalho,jpaulo,jj,jas}@di.uminho.pt

Abstract. This paper presents a framework for the analysis of software artifacts. We revise and propose techniques that aid in the manipulation and combination of target-language specific tools, and in handling and controlling the results of such tools. We also propose to integrate under our framework techniques that are capable of performing language independent analyses.
The final result of our work is an analysis environment that is modular and flexible and that allows easy and elegant implementations of complex analysis suites.
We finally conduct a proof of concept for our framework by analyzing a well-known, widely used open-source software package.

Keywords: Software Analysis, Software Certification, Combinator Languages

1 Introduction

Building software has historically always been considered a challenging engineering task. And this is particularly true nowadays, where programming involves not only reusing libraries that are provided by our programming language of choice, but also trusting and reusing libraries that have been built by other programmers and that are available on the Internet as open source software.

While software reuse has evident benefits such as rapid development, one often needs to make sure that the reused libraries satisfy certain properties. In our context, we refer to analyzing a property of piece of software as its certification

* This work is partly funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within projects FCOMP-01-0124-FEDER-010049, and FCOMP-01-0124-FEDER-022701.

and indeed choosing between many available libraries with the same purpose is often influenced by the properties that each one holds.

This paper consolidates our ongoing effort to provide a customizable framework for the certification of (reusable) software packages under the Certification and Re-engineering of Open Source Software (CROSS) project[4]. Indeed, when one tries to come up with a solution for dealing with all packages, under any programming language and using potentially any analysis tool, multiple issues arise:

i) how can we provide a setting for users to easily and elegantly combine different tools, each one providing a concrete and desirable analysis for the same package?
A solution for this problem was achieved by us with a combinator language that allows programmers to describe at an abstract level how software tools can be combined into powerful software certification processes [1].

ii) with such a language at hand, we turned our focus into providing a globally accessible framework where users could define and store their customized certification processes and that could work as a repository for a wide range of analysis tools that would otherwise need to be installed locally.
With this in mind, we have developed a web portal that relies on the domain-specific language of *i*) as its core [2], and that provides a common storage location for multi-purpose analysis tools.[5]

iii) finally, we have realized the need to customize the results produced by different certifications. Indeed, in our context certification results may assume different formats ranging from simple text to complex images or charts, and producing impactful results (or reports) again may require manual customization.
In order to satisfy this need, we have also developed an embedded domain-specific language for combining reports [3].

In this paper we now propose to build on these results and to improve on them. In particular, we make the following contributions:

1. we propose a single and coherent framework that elegantly integrates the combinator languages described in *i*) and *iii*). This is a framework that is currently under integration in the portal provided in *ii*);
2. while the tools that have already been integrated in our web portal allow for language-specific analyses only (i.e., they target one specific language such as C or Java and they focus on one specific characteristic of it), we now propose the integration of a set of analyses that are language independent. This analyses include inspecting elements other than source code that must be available on any software ecosystem, such as README files or even comments within the source code itself. This further extends the potential practical interest of our certification environment;

[4] `http://twiki.di.uminho.pt/twiki/bin/view/Research/CROSS/`

[5] While the portal already stores a significant number of analysis tools, still we rely on further inputs from the community to enlarge this set.

3. we provide a detailed case study that fully illustrates, one by one, the steps to undergo in order to certify a realistic software package. By this, we also hope to demonstrate the expressive and practical power of the global certification scenario that we envision in this paper.

This paper is organized as follows. In Sections 2 and 3 we revise the combinator languages described previously in items *i*) and *iii*), respectively. In Section 4, we propose and describe the set of analyses for elements other than source code that we have considered extending our certification environment with, i.e., we describe contribution 2 of the paper. In Section 5 we refer to contribution 1 of the paper, i.e., we describe how we have integrated in a single and coherent framework all the independent pieces that are necessary to provide customizable software certifications. The case study that we have used to demonstrate the power of the framework that we finally obtain, in the line of contribution 3 is described in Section 6. Finally, in Section 7 we conclude the paper and point some possible directions for future research.

2 Combining Software Analysis Tools

We start by revising the combinator language that is used in our web portal to allow the creation and customization of analysis schemas, that we call Certifications. Such combinator language, which has been proposed and thoroughly explained in [1], allows an easy and modular implementation of flows of information through different analysis tools that are integrated in our web server. Furthermore, the results of these tools always end up being collected and transformed into a report, contained on a XML file to which the users also have access.

In order to briefly introduce the reader to the combinator language of our portal, we present next a simple yet illustrative example of all the combinators in the language, using them in the construction of a concrete Certification. The example shown is a snippet of Haskell [4] code, which is the programming language that we have used to implement our combinators.

Example of a Certification defined with combinators:

```
certification =
    Input >- (slicer,"-j","-jpg")
              >- (jpeg2Report,"-jpeg","-r")  >|
    Input >- (memoryCheck,"-j","-r")         >|>
       (aggregator,"-r","-r")
       +> "Certification"
```

This example certification is called certification, and has two main flows of information, both started by the primitive Input. After Input is used, the user can create a flow of information by intercalating tools with the primitive `>-`. As long as he/she continues to use this primitive, the initial information will be consumed by the first tool, whose result will be consumed by the second one, and so on. In this particular example, we have one sequence: `Input >- (slicer,"-j","-jpg")  >- (jpg2Report,"-jpeg","-r")`, where the initial input is processed by the tool 'slicer', which fuels information to jpeg2Report.

The combinators >| and >|> are used for parallel computation: the first one appears between linear flows of information, and splits the certification into parts that are autonomous and can run in parallel. The combinator >|> appears only once per parallel computation, and marks a point where all the results of all the parallel processes are combined using one tool whose responsibility is only to aggregate all these results.

In this particular example, we have two flows of information both started with the primitive Input. These are split with >-, meaning they will constitute two sets of tests that will run in parallel, and terminated with >|>, meaning that their result will be aggregated with the tool aggregator.

The last combinator is +>. The only purpose of this combinator is to create a Certification, by giving it a name, which in this case is "certification", and making it available on our web portal.

Our Certifications web portal is a collection of bash tools created and maintained by any user, with the only limitations of being capable of running in an UNIX-based shell and using the the standard UNIX streams, STDIN and STDOUT to process and return information. Our combinator language works on top of this standard ambient and provides, through the web portal, a set of primitives that channels information through tools.

This combinators language is powerful enough to allow parallel chains of analysis, for example, when a user wants to integrate into the report two different results from two different types of analysis which are in no way related, while isolating them (failures in one chain do not necessarily imply the failure of the whole analysis, as the system tries to isolate errors). This is achieved through a meta-script that is generated by these combinators and controls systems calls and the flow of information through the tools that integrate the analysis.

Another important feature of our analysis combinators language is type checking. Tools by themselves are not type-constrained - they are just bash tools consuming and returning information through standard streams, but on practice tools have specific limitations according to the analysis they perform: some might work on Java or C, others on Haskell or even on XML, so there is a need to constrain these specifications on our combinators language.

In the previous example, we can see that tools are called with parameters, such as in: (slicer,"-j","-jpeg"). This parameters are used to indicate the types this tool will deal with. In this case, the tool will read Java code and produce a JPEG.

To avoid potential type errors, we force tools to have, in our web portal database, a set on input and output types, which can be triggered by the arguments when calling the tools. When creating a Certification, the user is forced to

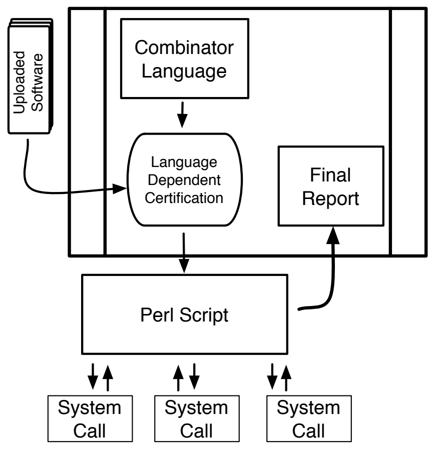


Fig. 1. The combinator language for tools.

say explicitly the types of information one expects the tools to handle, and the machinery inherent to this language checks if the tools support the given types and if the types match between the flow on information throughout the tools.

Figure 1 shows an overview of the whole process: the user defines an analysis schema on the web portal using the combinators primitives, which is transformed into a Perl script, which does a set of systems calls to the tools that constitute the analysis and produces a report that the user can visualize on the web portal.

Through this combinator language one can easily create analysis suits on our web portal, which are self-contained but modular and can be further used to create more analysis. In [1] we explain in detail the creation, the usage and the inherent mechanisms that support this combinators, such as our Attribute Grammar-based type checker.

3 Report Combinators

In the previous section we have reviewed a combinator language that allows a user to easily create test suits using combinator primitives that specify an underlying script responsible for making controlled system calls and transform them into a report.

We have showed how powerful and simple a combinator language can be for controlling and creating test suits, but there is still one important part of that analysis - the report - where the user has no control whatsoever. This might not seem like a big problem for small analysis, but for situations when the user wants to perform a huge set of analysis, being able to organize their results is very important for being able to understand the analysis itself.

In this section we will review a combinator language that allows precisely this: through it, the user is capable of organizing and customizing the layout of the report and even personalizing it with custom titles and notes. This language, together with its implementation as well as the algorithms that support it is further analyzed in [3].

Next, we present an example of a report specified with our combinator language for reports. Similarly to the previous example, it is also written in Haskell.

Example of a report defined with combinators:

```
report = Init >| ("Memory Tests",
                     (beginSubsection $ cert1)
                   )
                >- ("Usability Tests",
                      ( beginSubsection $ cert2)
                        >-- ("Result of Cert3", cert3)
                        >-- ( cert4)
                     )
```

These combinators are used to specify reports, so through them the user can defined wether a certain certification should fit into a section, or a subsection or if should have a customized title. In the particular case of report, it is composed by 4 different certifications, here named cert1 to cert4 for simplicity, whose results are organized through sections and subsections.

The report starts with >|, whose only purpose is to start a report. This combinator is mandatory and its single usage represents the smallest report possible, composed by one section only.

In this particular example, the combinator is immediately followed by the string "Memory Tests", which represents the name of this section, and by another primitive, beginSubsection that, as the name clearly states, creates a subsection. This subsection is unnamed, because it is followed directly by a certification, and not by a string and a certification. The machinery responsibly for the implementation of these combinators gives standard names such as "Subsection 1" in cases like these.

The primitive >| can be followed by an infinite number of >-. Each of these create a new section, with the exact same rules we have seen: it's name can be customized by writing a string immediately after and it can be followed by the primitive beginSubsection to further structure the report. In this case, report is composed by two sections, named "Memory Tests" and "Usability Tests" respectively.

There is also the option to create an infinite number os subsections for each section in the report. In this case, the results is cert3 and cert4 are both integrated into subsections of "Usability tests", using the combinator >--. It is important to note that in the case of the result of cert3, a custom name is given to the subsection that integrates its result: "Result of Cert3". The programmer chose not to customize the title of the result of cert4.

Next, we present the XML file created by the combinators that implement report.

Example of a report generated with combinators:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<section title="Memory Tests">
    <subsection>
        c1_result
    </subsection>
</section>
<section title="Usability Tests">
    <subsection>
        c2_result
    </subsection>
    <subsection title="Result of Cert3">
        c3_result
    </subsection>
    <subsection>
        c4_result
    </subsection>
</section>
```

As stated earlier, reports in our web portal environment are represented by XML files. The user is presented with an HyperText Markup Language (HTML) report in our portal, but it is just an transformation using eXtensible Stylesheet Language Transformations (XSLT) of the XML file, to which the user always has access. This also holds for the tool combinators presented in the previous section.

4 Non Source Code Software Analysis

Besides source code, another fundamental source of information about open source software lies in documentation, and other non source code files, like README, INSTALL, or HowTo files, commonly available in the software ecosystem. These documents, written in natural language, provide valuable information during the software development stage, and also in future maintenance and evolution tasks.

The CROSS research project aims at developing software analysis techniques that can be combined to assess open source software projects. Although most of the effort is spent analyzing source code, non-source code content found in packages can have a direct impact on the overall quality of the software. Forward *et al* survey [5] about the general opinion of software professionals regarding the relevance of documentation and related tools, highlights the general consensus that documentation content is relevant and important. It also highlights that software documentation technologies should be more aware of professionals' requirements, opposed to blindly enforce documentation formats and tools.

Documentation analysis is also relevant in other research areas. Program Comprehension is an area of Software Engineering concerned with gathering information and providing knowledge about software, to help programmers understand how a program works in order to ease software evolution and maintenance tasks [6]. Many of the techniques and methods used rely on mappings between program elements and the real world concepts these elements are addressing [7]. Non-source code content included in software packages can provide clues and valuable information to enhance the creation of these mappings. Program maintainers often rely on documentation to understand some key aspects of the software [8].

DMOSS[6] is a toolkit designed to systematically assess the quality of non source code text found in software packages. The goal of the toolkit is to provide a systematic approach to gather metrics about this content and assess its quality. It starts by gathering content written in natural language found in the package, process this content to compute metrics, and finally reason about these metrics to draw conclusions about the overall software quality. The toolkit handles a software package as an attribute tree, and the major engines for processing a package are implemented using tree transversal techniques. The specific metric calculations are made using a specialized set of plugins, that are responsible for: (1) analyzing a specific chunk of text and produce a metric, (2) reduce and aggregate sets of metrics to produce intermediate and final results, and (3) use templates for creating report snippets. Adding features to the analysis workflow is just a matter of adding a new plugin. This approach has allowed the development of a modular and pluggable toolkit, easy to maintain and extend. The toolkit can process any package, regardless of programming language used, but the text extracting tool (from files) can require update for some specific archiving technologies or documentation formats.

Assessing software quality for any given definition of quality is not easy [9] mainly due to subjectivity. DMOSS evaluates the non-source code files included in a software package. This set of files can include README files, INSTALL files, HTML documentation pages, or even UNIX man(ual) pages. Instead of trying to come up with a definition for quality, we select three main traits that we are concerned about. We envisage that these characteristics have a direct impact in the overall documentation quality regardless of the degree of individual subjectivity.

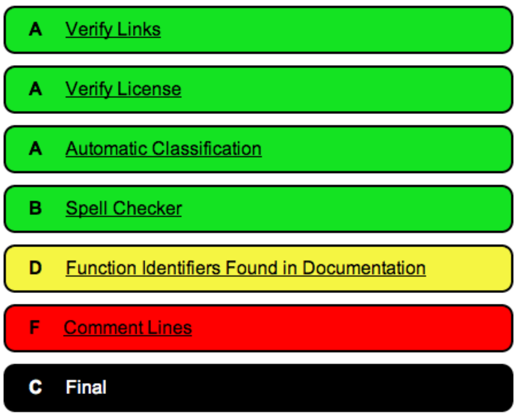
- Readability: text readability can be subjective, but there are linguistic characteristics that generally make text harder to read. Some of them can even be measured, as for example, the number of syntax errors or the excessive use of abbreviations;
- Actuality: this is an important feature of documentation and other textual files, they should be up-to-date, and refer to the latest version of the software;

[6] Documentation Mining Open Source Software

– Completeness: this trait tells us how much the documentation is complete, and if it addressees all the required topics.

DMOSS processes a software package to gather information about specific metrics that are related with these traits.

The dmoss-process tool provided by the toolkit is used to process a package. The result of processing a given package is a tree, decorated with attributes storing the calculated set of features. Another tool provided by the toolkit is dmoss-report, that uses the result of the previous tool to create a report in HTML format. An example report, created for the tree[7] software package (version 1.5.3) is illustrated in Figure 2.



A	Verify Links
A	Verify License
A	Automatic Classification
B	Spell Checker
D	Function Identifiers Found in Documentation
F	Comment Lines
C	Final

Fig. 2. Screenshot of a HTML report produced using DMOSS[8].

This report shows metrics that are used to grade key features about the package. For example, many documents in software packages contain links to official websites or discussion forums, one of the plugins included in the toolkit validates that these link are still working. If all links included in the documentation are working this feature is graded A. Another example is the number of comment lines in order to the total source code lines. In this specific case the percentage of comment lines per number of line codes is below 20%, which graded this feature of documentation with grade F. Some of these features are based on thresholds, that can be configured and adapted to specific contexts or packages. By clicking on each specific feature in the HTML report, more information is shown regarding each specific metric. A final grade is given to the package (C in this report), which is the features' grade average.

For more details about the DMOSS toolkit please refer to [10].

[7] Available from `http://mama.indstate.edu/users/ice/tree/`.

[8] Figure requires colored printing for optimal visualization.

5 Improving Software Analysis in CROSS

We have revised in Sections 2, 3 and 4 different technologies that aid in software analysis by implementing techniques that are applied in the analysis customization, in its resulting data and in the verification of important meta-information orthogonal to most software systems.

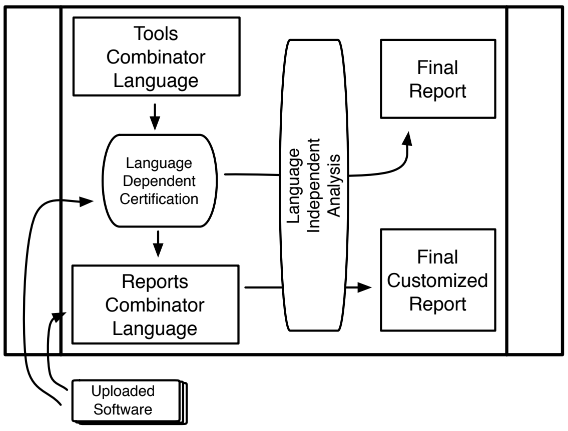

Tools Combinator Language
Language Dependent Certification
Reports Combinator Language
Language Independent Analysis
Final Report
Final Customized Report
Uploaded Software

Fig. 3. A framework for software analysis.

In this section we describe how the integration of different technologies creates an inter-dependent ecosystem that is capable of producing important artifacts whose information can help in understanding, improving and expanding a huge domain of programs written in various programming languages.

In Figure 3 we present the overall analysis framework that constitutes our analysis environment. We can clearly see three technologies in action and how they are interconnected:

- The tool combination language is being used to create language-dependent certifications, which can, alone, evaluate and analyze software artifacts. These certifications are built upon software tools existing in the web portal, and represent controlled flows of information through these tools, until a desired result is obtained.
- The report combination language is built upon the certifications created with the tool combination language. Working in a similar fashion, this time we are not controlling the analysis itself, but rather the data it provides. With the introduction of this technology, different certifications can be structured to create powerful and customized reports.
- The language-independent analysis works as a layer providing contextual information for uploaded software resources. It is independent of the type of

analysis, being it a simple certification or a complex multi-certification set, and provides important results about the uploaded software.

The analysis framework suggested in this paper is the result of the integration of all these technologies into a setting that uses the main advantaged of which one of them to support powerful software analysis.

5.1 Integrating the Report Combinators

The technology to customize certifications by creating flows of information across heterogeneous tools in already integrated in our web portal [2], and is an important way in which software can be analyzed and studied in our environment. The structure of the reports that results from that analysis, on the other way, was predetermined and its structured was steady.

The machinery presented in Section 3 was explicitly created to solve this issued, with its integration designed to be natural to our environment and its usage intuitive and similar to the usage of other CROSS mechanisms, both for analysis of specifications.

Specifying a report in our web portal works in parallel with creating a certification with the tools combinators: all certifications can be used and combined into a custom report, but this task is done in two steps: first the user creates a set of certifications he wants to compose into an analysis suite (or uses the ones that already exist in the portal's database) and then he specifies how this suite of certifications is composed into a structured report.

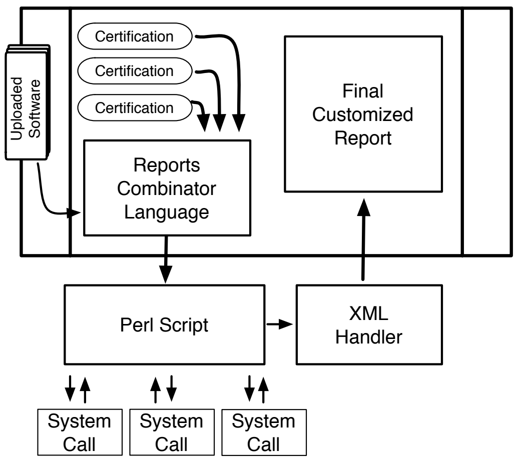


Fig. 4. The combinator language for reports customization.

Figure 4 shows how this machinery integrates into our web portal. It starts with report customization based on previous existing certifications. Since certifications represent by themselves analysis suites, these are applied in order until all their results is obtained. The next step is to pass the results through an XML Handler, whose responsibility is to arrange them according to the user specification.

Similarly to the architecture sketched in Figure 1, the final report is presented in HTML format, although it exists as raw XML to which the user has access. The analysis engines also try to isolate errors and apply the analysis even if there are problems with its constituting certifications.

With the introduction of this technology, we believe our analysis environment becomes more powerful while maintaining it easy to use, by facilitating the important step of analyzing the final data through which qualifiable results are obtained.

The access to the XML file that results from analyzing software is also an important feature since users might want to apply further automated analysis to the results produces by our framework, being XML an optimum medium for this analysis. This automation can even be performed by uploading the XML report into capable certifications specified in the framework itself.

5.2 Integrating Language-Independent Analysis

Since a language-independent analysis is an important technology to be integrated in any software analysis, the mechanics of our framework imply this techniques constitutes a layer which, by being orthogonal to any software artifact, is also orthogonal to any analysis.

The modular nature of our framework implies this analysis was easily integrated, and its results are as customizable as the results of any other certification, aiding in the quality and in the data of the final report.

6 Case Study

In this section, we apply our software quality framework to a software artifact of practical interest: we analyze the VLC media player (VLC)[9]. This is a free, open source and cross-platform multimedia player whose capability of playing various multimedia encoded files and various streaming protocols make it a well-known and widely used tool. We have used the source code from VLC version 2.0.5[10] that is available from SourceForge[11], a well-known, web-based source code repository that hosts more than 300000 projects. The VLC version that we have used is of size 18.4 MBytes and contains more than 3500 files.

To test VLC, we envisioned a test composed of:

1) Our default language-independent analysis to produce generic results regarding the software documentation and the overall quality of its source code.

[9] `http://www.videolan.org`

[10] `http://sourceforge.net/projects/vlc/files/2.0.5/vlc-2.0.5.tar.xz/download` (accessed in 2013-2-14).

[11] `http://sourceforge.net`

2) A certification to compute the number of C source files (in particular, this certification searches for *.c files).
3) A certification that identifies C source-code files in which function strcat() is used. Unless care is taken, this function can cause memory overwriting, and in extreme cases allows hacking of the target machine through buffer overflow[12]. The use of this function in a program does not necessarily imply that it is unsafe, but may raise improvement concerns.
4) A certification that produces the number of C++ lines of code throughout the entire project to give a general overview of the amount of functionality that is implemented in this language.

Certifications 1) to 4) are composed using our combinator language for reports, with the results of certifications 1), 2) and 4) being shown in the first, second and third sections of the report, respectively. The result of certification 3) is intended to be shown as a subsection of the second section, providing the overall perspective illustrated as follows:

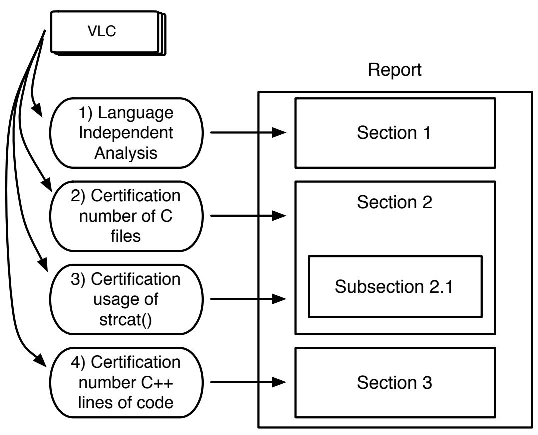

VLC
Report
1) Language Independent Analysis
2) Certification number of C files
3) Certification usage of strcat()
4) Certification number C++ lines of code
Section 1
Section 2
Subsection 2.1
Section 3

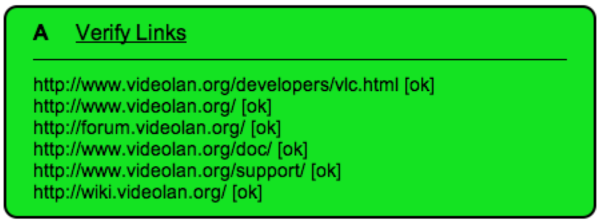
Fig. 5. An example of an analysis.

The analysis that we have implemented is straightforward: VLC feeds a serious of certifications that analyze different aspects of it while producing a structured report. While certifications larger in number and in complexity and more structured reports are possible within our framework, we have opted to maintain our running example as simple and clear as possible.

In Figure 7 (page 16) we see the final resulting report that was produced after analyzing VLC. The first thing to notice is the structure of the report: it is easy to read and to understand, and we see the subdivision specified in

[12] `http://en.wikibooks.org/wiki/C_Programming/C_Reference/string.h/strcat` (accessed in 2013-2-14).

Figure 5. It is important to remember that the final report is always an XML file, which we do not show here due to size constrains, and the layout presented is the HTML file we choose to generate from the original XML report. The user is free to personalized this transformation as he finds better suits his/her needs, or even analyze the XML file directly.



```
http://www.videolan.org/developers/vlc.html [ok]
http://www.videolan.org/ [ok]
http://forum.videolan.org/ [ok]
http://www.videolan.org/doc/ [ok]
http://www.videolan.org/support/ [ok]
http://wiki.videolan.org/ [ok]
```

Fig. 6. One example of the results provided by the language-independent analysis[13].

The language-independent analysis is shown as a series of colored lists. This lists represent the various fields that are analyzed, and that go from licenses verifications to spell checking. These fields are colored depending on how critical the results are, and are fully expandable to show details about each specific analysis. Figure 6 shows an example of the expanded results of the field: "Verify Links". These are green, meaning the results are good, and show each link individually and its status, indicating if they are either online or offline. All the fields of the language-independent are expandable to show details about its analysis.

Another interesting result is the number of files our analysis found which use the function strcat(). We found thirteen files in the source code of VLC that use this function, indicating good points for possible optimizations of the software. It should be noticed that due to the high modularity of our framework, the user could easily change this certification to show simpler results, such as only indicating if it found any usage of strcat(), or more complex ones, where the exact line and column of each file is presented in the report. Making these modifications would imply only very simple transformations to the certification that searches for these parameters.

By looking at the results presented in Figure 5 we can obtain interesting information about a random software package, VLC, even though we choose a very simple analysis made by simple certifications. We believe such interesting results obtained by this simple analysis suite prove the potential of our framework to support complex examinations and, more important, to produce important information from software artifacts.

[13] Figure requires colored printing for optimal visualization.

7 Conclusion

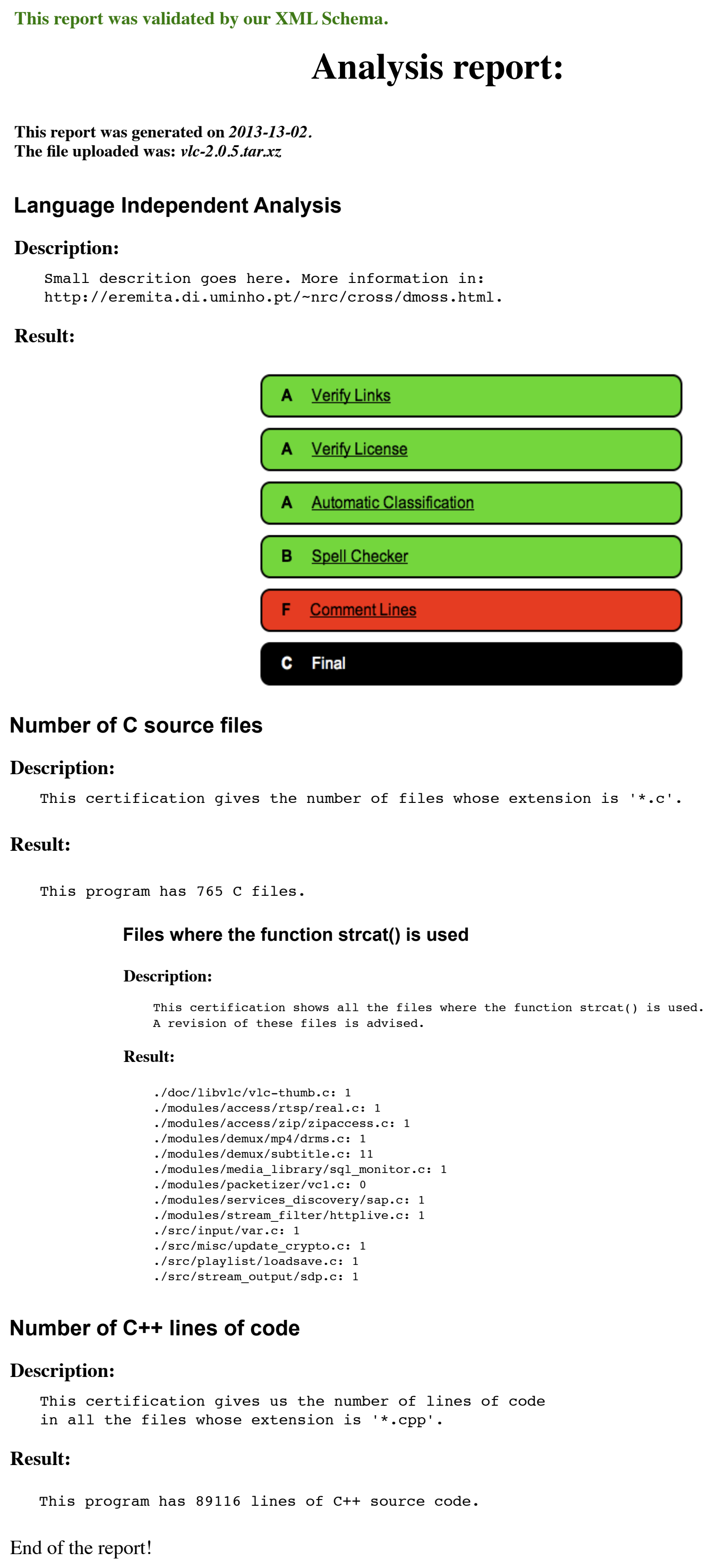
In this paper we have revised different technologies that aid in software analysis, and which we have combined into an analysis framework that is modular and flexible, allowing an easy implementation of software analysis both by integrating new techniques and by re-organizing existing ones.

We have furthermore applied one example of such an analysis to a well known, medium sized but realistic software product from the open source community, whose results are promising and prove the potential of our framework.

As for future work, we will be focusing our attention on flexible and effective ways of spreading the analysis results other than simply showing them on a browser.

References

1. Martins, P., Fernandes, J.P., Saraiva, J.: A purely functional combinator language for software quality assessment. In: Symposium on Languages, Applications and Technologies (SLATE '12). Volume 21 of OASICS., Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012) 51–69
2. Martins, P., Fernandes, J.P., Saraiva, J.: A web portal for the certification of open source software. In: 6th International Workshop on Foundations and Techniques for Open Source Software Certification (OPENCERT '12) (to appear). LNCS, Lecture Notes in Computer Science (2012)
3. Martins, P., Fernandes, J.P., Saraiva, J.: A combinator language for software quality reports. International Journal of Computer and Communication Engineering **2** (2013)
4. Simon, Hughes, J., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C., Wadler, P.: The Haskell 98 Report (1999)
5. Forward, A., Lethbridge, T.: The relevance of software documentation, tools and technologies: a survey. In: Proceedings of the 2002 ACM symposium on Document engineering, ACM (2002) 26–33
6. Nelson, M.L.: A survey of reverse engineering and program comprehension. CoRR (2005)
7. Rajlich, V., Wilde, N.: The role of concepts in program comprehension. In: Proceesing of the 10th International Workshop on Program Comprehension, 2002., IEEE (2002) 271–278
8. Thomas, B., Tilley, S.: Documentation for software engineers: what is needed to aid system understanding? In: Proceedings of the 19th annual international conference on Computer documentation, ACM (2001) 235–236
9. Kitchenham, B., Pfleeger, S.: Software quality: the elusive target [special issues section]. Software, IEEE **13**(1) (1996) 12–21
10. Carvalho, N.R., Simões, A., Almeida, J.J.: Open source software documentation mining for quality assessment. In: WorldCIST'13 - World Conference on Information Systems and Technologies Proceedings (to appear). (2013)



This report was validated by our XML Schema.

Analysis report:

This report was generated on *2013-13-02*.
The file uploaded was: *vlc-2.0.5.tar.xz*

Language Independent Analysis

Description:

```
Small descrition goes here. More information in:
http://eremita.di.uminho.pt/~nrc/cross/dmoss.html.
```

Result:

A Verify Links
A Verify License
A Automatic Classification
B Spell Checker
F Comment Lines
C Final

Number of C source files

Description:

```
This certification gives the number of files whose extension is '*.c'.
```

Result:

```
This program has 765 C files.
```

Files where the function strcat() is used

Description:

```
This certification shows all the files where the function strcat() is used.
A revision of these files is advised.
```

Result:

```
./doc/libvlc/vlc-thumb.c: 1
./modules/access/rtsp/real.c: 1
./modules/access/zip/zipaccess.c: 1
./modules/demux/mp4/drms.c: 1
./modules/demux/subtitle.c: 11
./modules/media_library/sql_monitor.c: 1
./modules/packetizer/vc1.c: 0
./modules/services_discovery/sap.c: 1
./modules/stream_filter/httplive.c: 1
./src/input/var.c: 1
./src/misc/update_crypto.c: 1
./src/playlist/loadsave.c: 1
./src/stream_output/sdp.c: 1
```

Number of C++ lines of code

Description:

```
This certification gives us the number of lines of code
in all the files whose extension is '*.cpp'.
```

Result:

```
This program has 89116 lines of C++ source code.
```

End of the report!

Fig. 7. The report produced after analyzing VLC.