A Web Portal for the Certification of Open Source Software

Pedro Martins, João P. Fernandes, and João Saraiva

HASLab / INESC TEC, Universidade do Minho, Portugal
{prmartins, jpaulo, jas}@di.uminho.pt

Abstract. This paper presents a web portal for the certification of open source software. The portal aims at helping programmers in the internet age, when there are (too) many open source reusable libraries and tools available. Our portal offers programmers a web-based and easy setting to analyze and certify open source software, which is a crucial step to help programmers choosing among many available alternatives, and to get some guarantees before using one piece of software.

The paper presents our first prototype of such web portal. It also describes in detail a domain specific language that allows programmers to describe with a high degree of abstraction specific open source software certifications. The design and implementation of this language is the core of the web portal.

Keywords: Software Analysis, Software Certification, Open Source Software, Programming Languages

1 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. In the last century, developing software was mainly performed using programming languages and their libraries, which provided the necessary support to build software applications. Nowadays, the way we develop software is changing: 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 influences 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:

- Firstly, because there is so much OSS available in the internet it is difficult to select the right tool/library. Thus, we need an appropriate framework to support the analysis of the available alternatives.
- Secondly, because we may reuse different software artifacts, developed in different contexts, we need to integrate them into a coherent piece of software.

  – 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 may need the guarantee
    that a piece of software to be reused conforms to specific security guarantees.
    On a different context, if we if we are developing software for embedded
    systems, we may need to guarantee that a reused library implements optimal
    memory management.

In this paper we present a web portal for the analysis and certification of
Open Source Software that aims at improving on these three issues. The portal
works as a repository for tools that analyze source code. By the certification of
a piece of source code software we understand the process of analyzing the its
code while producing an information report about it.

The usage of our portal is heterogeneous in that it supports the analysis of any
programming language and distributed in the sense that it makes software anal-
ysis available in the web. Also, while already incorporating several pre-defined
certifications, the portal makes it very simple for any user to re-arrange these
certifications and to develop new ones: we designed and implemented a Domain
Specific Language (DSL) that allows portal users to define, in a high level and
abstract way, how certifications and software tools that analyze source code can
be integrated and combined. This allows the creation of personalized analysis
closely tied to the scope and nature of the necessary feedback.

This paper is organized as follows: in Section 2 we introduce the web portal
together with the software analysis scenarios it supports. In Section 3 we in-
troduce the DSL that allows the creation of new analysis suites, together with
its underlying working mechanisms. In Section 4, we describe the validations
that are ensured by the used of our combinators, and in Section 5 we introduce
implementation and usage issues of the portal. Finally, in Section 6 we conclude
the paper.

2   An Open Source Software Certification Portal

Software analysis is an interesting topic of research, whose motivations range
from the need to maintain software as easily as possible to the removal of its
bugs and the improvement of its overall characteristics [1–5]. While tools and
techniques for program analysis are very diverse, the fact is that they are often
too restrictive to gain wide acceptance. This has two main causes, in that tools
are often: i) designed for a specific programming language; ii) not flexible enough
to be tailored when the particular needs of a user differ from the built-in analysis.

In this section, we introduce the portal that we have constructed to act as
a repository for tools that are freely available and to enable the certification
of open source software based on such tools. By a Certification we mean the
execution of a software analysis tool that is capable of processing a source code
file and of producing an information report about it.

Our portal was constructed to make no distinction with respect to the pro-
gramming language or scope of the tools it hosts. Also, the tools that are hosted
can easily be combined therefore allowing the fast creation of test suites that

perfectly match the needs of different programmers. The combination possibilities, that we describe in the remaining of this section, include the possibility of analyzing software components written in multiple programming languages. To enable the definition of certifications and their composition, we have designed a domain specific language, that we describe in the next section.

The portal that is the result of our work can be found at:

<div align="center">

`http://www.cross.di.uminho.pt/`

</div>

Analyzing a single Open Source File The simplest way to use our portal is to analyze a single source code file. This simple test, that is depicted in Figure 1, may be useful to identify performance opportunities or to validate security requirements, for example.

Fig. 1. The flow of information in our portal when analyzing a single source code file.

In this illustration, a Certification is being used on a Program written in language L, leading to a Report being produced. In the context of our portal, reports are always defined as elements conforming to the following XML Schema:

```xml
<?xml version="1.0" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="report">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="file" type="xs:string"/>
   <xs:element name="name_upload" type="xs:string"/>

   <xs:element name="certification" minOccurs="1" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="description" type="xs:string"/>
      <xs:element name="result" type="xs:string"/>
      <xs:element name="image" type="xs:string" minOccurs="0"/>
      <xs:element name="html_result" type="xs:string" minOccurs="0"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>

  </xs:sequence>
 </xs:complexType>
</xs:element>

</xs:schema>
```

We have chosen an XML-based representation since it allows storing information in different formats such as charts and images. Also, XML Schemas are widely used, easily understood, and can be translated to different representations. Indeed, we show our reports as HTML pages that are produced using XSLT.

Our portal is not only suitable for one isolated certification of a program. Indeed, we show next how the same program can have several of its characteristics certified at the same time by a set of certifications, that are combined into a larger certification, while producing a single information report.

Multiple Analysis for a single Source Code File Analyzing software usually implies running a number of tests provided by a set of tools which, together, allow us to obtain information about diverse aspects of the software. Our system provides a simple interface to agglomerate and run multiple certifications while creating a single information report, as sketched in Figure 2. In this particular case, the submitted $Program_L$ is subject to three independent analyses, $Certification_1$, $Certification_2$ and $Certification_3$, but the number of certifications that can be composed is arbitrary.
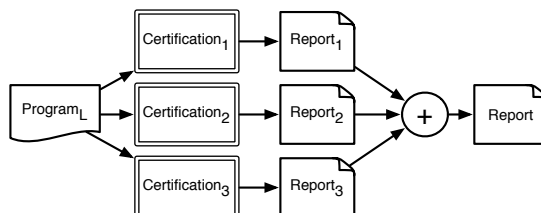

Fig. 2. Multiple analysis for one single source code file.

We observe that a set of reports is aggregated into a single final report. This is a strategy that always needs to be followed in our setting, and that we have adopted since analyzing multiple reports individually would become a tedious and confusing task, for growing numbers of such reports.

The mechanisms presented so far allow a simple analysis of source code single files. We, however, often want to certify software repositories, i.e., software artifacts that are composed of several pieces, each of which in (at least) one different file, and possibly expressed in a different programming language. Next, we show how our web portal allows analyzing a set of source code files of this kind.

Analyzing Multiple Source Files Our web portal supports the analysis of a set of different source files, and in Figure 3 we show an example of this type of analysis being performed.

In our system, uploading a set of files is achieved through an archiving format, being it ZIP, Tar or RAR. Our web portal automatically infers information from the compressed archive, extracts its regular files and parses them.

Again, the results of all certifications need to be aggregated into a single report file. In this case, the final report groups either the results of applying
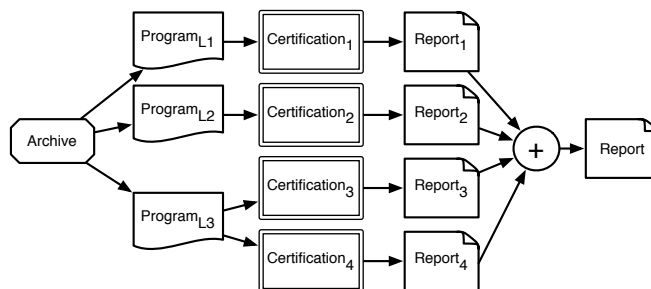
Fig. 3. A testing suite for multiple source files expressed in different languages.

the certification individually to each file, or the results provided by certifications analyzing all the source files of the same type at the same time. The produced reports will be as big as demanded by the certifications that are executed. While sometimes their information is going to be too large for manual inspection, the fact is that having them under XML files allows users to easily automate the process of analyzing the information reports that are produced. In fact, the last step of analyzing the report might be integrated into the certifications themselves, as we will see next, where we describe how to personalize and customize certifications to the individual needs of each user.

Creating Customized Certifications So far, we have seen how our web portal provides different types of analyzes for source code. These analyzes are supported by the certifications that we have already integrated in our portal, a subset of which we now present.

Certification	Input	Description
sLOCJavaFiles	`-java`	Lines of code of a Java file
sLOCCFiles	`-c`	Lines of code of a C file
sLOCHaskellFiles	`-hs`	Lines of code of a Haskell file
zips	`-zip`	CRC32 checksum, size (zipped and unzipped)
fleschKincaid	`-txt`	The Flesch/Flesch-Kincaid readability test
emptyCellSmell	`-xlsx`	The empty cell spreadsheet *bad smell* [6, 7]

While the certifications that have thus far been built-in the portal already allow for several analysis, the fact is that within the framework described so far, users do not have the possibility of combining the available certifications according to their own particular needs. Furthermore, analyzing software is not usually achievable by a single certification: it often implies using a variety of tools and techniques, whose results, unified, provide the programmer with the necessary feedback.

Ideally, we want to use the results of a certification as input to another, which requires, apart from being able of performing a series of tasks through the use of several tools, aggregating and treating the results obtained. Examining and dealing with the information that is produced is as important as the results

themselves. Just as an example, some analyses derive results that are not quantifiable, and no information can be obtained from them. It is their treatment and analysis through comparison that provides good feedback to the user[1].

To solve the limitations described above, we have created a domain specific language that we have integrated in our system. This language allows the customization of each step of the process of analyzing software, which include data extraction, treatment of the results and the analysis itself.

In Figure 4 we show how to implement a certification, $Certification_1$, that is composed of other simpler certifications.
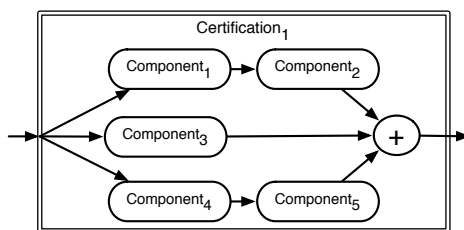

Fig. 4. An example of a certification composed by five components.

In our setting, certifications are often composed by smaller units capable of communicating among themselves to achieve a state where the overall mechanics of each unit and the flow of information among them is capable of producing quantifiable results (Certifications). A Component is one of this smaller software units. A Component is therefore a tool, capable of assessing and producing meta-data but that is not powerful enough so that a whole analysis, i.e., a Certification, is made out of it. Examples of components that we have already integrated in our portal are shown next.

Component	Input	Output	Description
readFile	`-c, -hs, -xml, -java`	`-string`	Reads a file.
text2NLines	`-string`	`-int`	Calculates the number of lines
int2Report	`-int`	`-rep`	Creates a Report from a value
text2NFors	`-string`	`-int`	Calculates the number of 'fors'
text2NIfs	`-string`	`-int`	Calculates the number of 'ifs'
text2Report	`-string`	`-rep`	Creates a Report from text

The concept of a component makes our system more powerful and configurable: not only the programmers and uploaders of tools are able of creating small software units, but also end users have access to a wide number of this specific tools that can easily be configured and adequate to concrete needs.

---

[1] An example of such case are the Halstead Complexity Measures [8].

Using existing components, our DSL provides an environment where the user has the possibility to create customized certifications. Furthermore, the mechanisms inherent to the DSL ensure that such components are isolated for errors and that the certifications are optimized, for example, by running components in parallel whenever possible. Also, it allows certifications made out of certifications themselves. In Figure 5 we show an example of a certification, $Certification_2$ whose result is composed by $Certification_4$ and by $Certification_3$, which is actually fed by a component, $Component_1$.
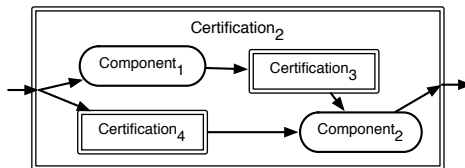

Fig. 5. A certification composed by both Components and other Certifications.

The definition of certifications based on components and other certifications allows our system to provide good testing and analysis for virtually any environment desirable, while also ensuring that the results are always on a standard format. In the next section we introduce the domain specific language in detail, together with practical and illustrative examples of its usage.

3 A Domain Specific Language for Creating Test Suites

The portal we present consists not only of an interface to analyze source code, but also as a repository of analysis tools, that we understand as Components. These tools can implement data handling and analysis or code slicing, for example, but a certification may not be constructed out of a tool if it is not able of producing a report in the required format. We could force users to upload a tool that, by itself, can analyze source code and produce a Report but this would mean forcing them to extend their solutions with features related to information input and Report generation. Also, the repository would never suit users that created a slicer, for example, since it is not an analysis technique by itself.

Figure 6 shows a context where a slicer (jSlicer) and an interface analyzer (iAnalysis) are used together to create a certification. In this figure we see that the certification implementor used a sub-certification that already existed on the system (Certification1).

In order to implement the sketched certification with the framework described so far, there exist (at least) two possible solutions: i) to manually integrate the involved tools and to compile its results to obtain a single tool that implements all this flow of information; or, ii), to run all the tools independently and to manually transfer the information among them. For reasons that we have previously discussed, both alternatives are tedious and cumbersome as they demand a significant programming/integration effort. The user would have to assume the
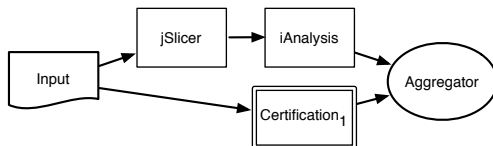
Fig. 6. An example of a user-defined certification.

responsibility of compiling and organizing all the code, which in the end would only solve a particular and specific problem.

In order to overcome this issue, we have created an interface, which was integrated in our portal, and that makes it simple to create new certifications, i.e., to organize components in a way that they provide the desired information from uploaded information. This interface is provided as a DSL embedded in Haskell as a combinator language. Before introducing our combinator DSL we present a little snippet of how it can be used to create the certification in Figure 6:

```
Input >-  (jSlicer,"-j","-i") >-
          (iAnalysis,"-i","-csv")  >|
Input >- certification1             >|> (aggregator,"-r","-r")
```

From this code, our portal automatically creates an appropriate certification. What is more, it also statically analyzes the specification and checks it for type errors, as we explain in more detail later in this section. In this particular example, the user is specifying that he/she wants component JSlicer to be called with the arguments -java and -interface in order to slice the interface out of a program written in Java. This information is then channeled to the component iAnalysis, and this chain of components runs in parallel with Certification1 that takes the same Java input that jSlicer receives. The execution results of the two parallel processes are later aggregated, by component aggregator, to form a report.

The combinator language that we propose starts by defining data-types for certifications and components. These data-types are introduced as follows, as Certification and Component, respectively.

```
data Certification = Certification Name ProcessingTree
data Component = Component Name InputList OutputList BashCall

type Name     = String      type InputList  = [(Arg, Language)]
type Arg      = String      type OutputList = [(Arg, Language)]
type BashCall = String

data Language
  = Java | C_Source | C_Header | Cpp | Haskell | XML | Report
-- .java  .c          .h          .cpp  .hs        .xml   Report XML
```

A Certification has a name and defines a particular information flow, which is represented by data-type ProcessingTree, that we introduce and describe in detail later. A Component is represented by a name, the list of arguments it

receives and the list of results it produces. These lists, that are represented by type synonyms InputList and OutputList, respectively, have similar definitions and consist of varying numbers of arguments and results. The arguments(results) that are defined(expected) for a particular component are then passed to concrete bash calls. This is the purpose of type BashCall, which consists of the name of the process to execute.

For a generic Java Slicer component that could be used in the context of the snippet previously shown, we may define the following Component.

```
Component "Java Slicer" [("-j", Java)]
          [("-i", Java), ("-s", Java)] "./jSlicer"
```

In order to represent the flow of information defined for a certification, we have defined the data-type ProcessingTree:

```
data ProcessingTree = RootTree ProcessingTree
                    | SequenceNode ProcessingTree ProcessingTree
                    | ParallelNode ProcessingList ProcessingTree
                    | ProcessCert  Certification
                    | ProcessComp  Component Arg Arg
                    | Input

data ProcessingList = ProcessingList ProcessingTree ProcessingList
                    | ProcessingListNode ProcessingTree
```

The simplest processing tree that we can construct is the one to define a certification with a single component. This is expressed by constructor ProcessComp, which also expects a name to be associated to the component and the specification of the options to run the component with. A certification can also be defined by a single sub-certification, here represented by ProcessCert. In addition to these, more complex certifications can be constructed by running processes in sequence, using SequenceNode, and in parallel, using ParallelNode. Constructor ParallelNode takes as arguments a processing list and a processing tree. The first argument represents a list of trees whose processes can run in parallel. The second argument is used to fulfill our requirement that all results of all parallel computations must be aggregated using a component. Therefore, this processing tree must always be a component that is capable of aggregating information into one uniform, combined output.

The ProcessingTree data-type can be used, for example, to describe the global information flow of a certification implementing a cyclomatic dependency analysis for Java programs while producing an information report.

```
RootTree
   ProcessComp
      Component "Cyclomatic Dependency"
               [("-j",Java)]
               [("-r",Report)]
               "./exec"
      "-j"
      "-r"
```

Having in hand the data-types that we have defined so far, we could already create, in a manual way, certifications with all the capabilities that we propose to offer. Nevertheless, manually expressing certifications would be of impractical use. This is precisely the main motivation to develop a language where simple and re-usable components can be combined into more complex ones, which themselves can grow as large as needed in order to implement extensive certifications.

With the addition of code becoming smaller, more elegant and easier to read and understand, the fact is that it will also be statically analyzed and type checked, and the script code that actually implements the certification it defines will be automatically generated. These features will be introduced in the remaining of this paper, together with the combinators that we use in our language, that we present in detail next.

The Sequence Processing Combinator For sequencing operations, we define the combinator `>-`. This combinator defines processes that are to be executed in a chain, i.e, where the output of a process serves as input to the process that follows it. When sequencing processes, it is also the case that if one of process in the chain fails the entire chain will also fail.

The use of combinator `>-` must always be preceded by the use of constructor Input, which signals the beginning of an information flow. Then, as many components and certifications as needed can be used, as long as they again connected by `>-`. Next, we show an example of a chain of events defined using `>-`.

```
Input >- (jSlicer,"-j","-i") >- (iAnalysis,"-i","-csv") >- certif
```

Combinator `>-` can be used to sequence certifications, components and other processing trees that are defined using the remaining combinators of our language. When it encounters a certification, the combinator connects the processes before it to the processing tree of the certification, and ensures that the result of this processing tree is then channeled to the processes that follow it. This is the case of the sub-certification called `certif`. When sequencing components, users need to supply to `>-` both the component and its input/output parameters. In our particular example, `jSlicer` is to be called with input parameter `-j` to state that the process accepts Java code as input and with output argument `-i` to state that it slices the interfaces out of that code.

It is worthwhile to notice that the arguments that are specified within components are very important in that they allow checking the flow of information inter-processes for correctness, as the input and output types must match when the information is channeled. When using `>-` to channel certifications, the user is constrained by the input and output types that were associated to it, and this is an information that must be carefully observed to ensure that the involved types do match.

Finally, the result of a sequence defined using `>-` is a processing tree that implements the combination of processes.

The Parallel Processing Combinator Now, we introduce the combinator that enables the parallel composition of processes, This type of composition is actually supported by two combinators, `>|` and `>|>`. The first one is responsible

for launching a varying number of processes in parallel, while the second is mandatory after a sequence of `>|` uses and chains all outputs of all processes to a component that is capable of aggregating them. An example of how these combinators work together is as follows.

```
Input >- cert3                      >|
Input >- cert1 >- cert5             >|
Input >- (jSlicer,"-j","-x") >- cert8 >|> (aggr,"-x","-r")
```

Combinator `>|` takes either a processing tree, a component, a certification or a set of processes constructed using the other combinators. The arguments of `>|` must always begin by constructor Input, to give a clear idea of the flow of information. In the case of this listing it is indeed easy to spot where the information enters a parallel distribution.

As for combinator `>|>`, it is mandatory for it to appear in the end of a parallelized set of processes. It is used to aggregate all the outputs of all the child processes into a single standard output, and it is able of combining varying numbers of parallel processes using an aggregation component.

It is worthwhile to explain further the relationship between the parallel combinators `>|` and `>|>`. In trivial cases, `>|>` can actually replace the use of `>-`. This is the case of the process `Input >|> (aggr,"-x","-r")`, which is equivalent to `Input >- (aggr,"-x","-r")`, as both processes channel the input to the aggregation component `aggr`. Finally, combinator `>|` can never appear alone in a certification, due to the constraint that all parallel processes must be aggregated.

A Combinator to Create Certifications Our combinator language includes also a combinator to create certifications and to associate names to them. This is precisely the purpose of combinator `+>`, which always combines a processing tree, given as its left argument, with a String, given to its right. It then creates a certification associating the name with the processing tree.

As an illustration of the use of `+>`, consider again the process implementations that use our sequence and parallel combinators. In both cases, the result of running the implemented code is a processing tree (i.e., an element of type ProcessingTree), that needs to be given a name to become a certification (i.e., an element of type Certification). By simply appending, for example, `+> "certification"` to the end of both codes, we would precisely be creating certifications named certification with the respective trees of processes. For the first example, this would result in the code:

```
Input >- (jSlicer,"-j","-i") >- (iAnalysis,"-i","-csv") >- certif
    +> "certification"
```

An important remark about `+>` is that it analyzes the processing tree that it receives as argument and checks its correctness. This includes testing whether the implied types match, by analyzing all the parallelized and sequenced processes for their input and output types, and see whether or not they respect the flow of information. Also, it is ensured that the processing tree produces a report which is a mandatory feature for a certification in our system. Finally, if

a certification is considered valid, a Perl script implementing its analysis is automatically generated. For the example certification just given, this overcomes the need to undergo the tedious and error-prone task of manually writing the script given in Appendix.

In the next section, we explain in detail the features that are implemented by our type analysis and how they are actually implemented under our framework.

The 'Finalize' Combinator The last combinator of our language is #>>, that combines a processing tree with a flag instructing it to either produce a script implementing that tree or to simply check its types for correctness. The following examples show the two possible uses for this combinator.

```
Input >- (comp1,"-j","-x") >- (comp2,"-k","-o") #>> 't'
Input >- cert1 >- cert2 #>> 's'
```

In the first case, we are demanding a check on the types of running component comp1 after component comp2. This means that we are interested in knowing whether the return type of comp1 is the same as the input type of comp2. With the second case, we are asking for the script that implements chaining certification cert2 after certification cert1 and also checks if the types match.

4   Type Checking on Combinators

In the previous section, we have introduced our combinator language for the development of software certifications. In this section, we introduce a set of validations that are automatically guaranteed to the users of our system.

For once, we inherit the advanced features of Haskell compilers. In particular, the powerful type system of ghc helps us providing static guarantees on the certifications that are developed. Indeed, the order in which the combinators of our language are applied within a certification is not arbitrary, and the uses that do not respect it will statically be flagged. The simplest example of this is the attempt to construct a processing tree without explicitly using constructor Input, but more realistic examples are also detected, e.g., not wrapping up a set of parallel computations with >|> as well as the application of an aggregator.

Apart from static analyzes, we have also implemented some dynamic ones: we want to analyze if the types match in the flow of information for a certification, i.e., if the input type of a process matches the output type of the process feeding it. In our setting, we perform such tests on elements of type ProcessingTree, that we use our combinators to construct. These elements are then analyzed using validations that are expressed as attribute grammars (AGs) [9]: 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 [10] and the use of higher-order attributes [11].

Our type checking structurally breaks down into analyzing the nodes of a processing tree which, in our case, correspond to the constructors of the ProcessingTree data type. The type checking is computed as the value of an attribute

called typeCheck, of type Boolean, which indicates whether or not the analyzed types are correct. Apart from this attribute two other are involved: input and output, that support typeCheck. These attributes are of type Language and for each tree node give the input and the output types of that subtree. Next, we will explain how these three attribute are calculated in each tree node.

Type Checking Component *nodes* Components are the simplest units of our processing trees, and represent simple processes without any actual flow of information. This means that their type is always correct, and that the value produced for attribute typeCheck is always True.

As for the attributes input and output, they are computed analyzing the Component against the invocation that is made for it in a ProcessingTree. Indeed, whenever a component is associated to a certification, an element such as ProcessComp comp inp out is defined. But comp itself is an element of type Component, i.e., has the form Component name inplist outlist call. So, our validation starts by detecting whether or not inp (respectively out) is an option of inplist (respectively outlist). In case it is, attribute input (respectively output) returns the Language option associated with inp (out). Otherwise an error is raised.

Type Checking Certification *nodes* Certifications are, similarly to components, simple nodes of a processing tree, and in our setting, they must always be created using combinator +>. Therefore, everytime this combinator is employed, as in `tree +> name`, we automatically inspect the value of attribute typeCheck that is computed for tree. If this value is True, we create the certification Certification name tree; otherwise, no certification is constructed and an error is raised.

Actually, it is not only necessary that a processing tree type checks in order for us to be able of producing a certification out of it. Indeed, all certifications must always produce a report within our format, and we also check if the output attribute computed for tree is Report before actually creating a certification.

Finally, when we consider sub-certifications, again we use the fact that they are created using +>. Indeed, if the construction of a sub-certification succeeds, then the tests so far described have all also succeeded. Therefore, for certifications under ProcessCert nodes, we can always ensure that they type check. Also, attributes input and output are very simple to implement: we return the value of the same attribute that is synthesized at the sub-tree of ProcessCert nodes.

Type Checking Sequence *nodes* Sequence nodes are used to have a processing tree followed by another. They channel the information from the first processing tree into the second one and returns the result of the second.

The input and output attributes for this node are very simple to compute: input is the input type of the first processing tree, and output is the output type of the second. Similarly, typeCheck is also simple to determine: apart from checking if the output attribute of the first tree is equal to the input attribute of the second one, our AG-based implementation also demands the typeCheck attribute on each sub tree individually and checks whether both have value True.

Type Checking Parallel *nodes* Parallel nodes are the hardest to type check, in that all their sub processes must have the same input type and same output

type, and this output type must be the same as the input type of the component that aggregates all the results that are computed in parallel.

Parallel nodes have two children: the second child, of type ProcessingTree, is a component that aggregates all the results of the processes that run in parallel, which are given as the first child, of type ProcessingList. The first validation we perform is to check whether the output value of the ProcessingList matches the input value of the ProcessingTree. If it does not, an error is raised; otherwise, the processes within the ProcessingList are type checked. Now, type checking processing lists is complex since it needs to analyze all the inputs of all the sub processes, which can be components, certifications or processing trees and see if they match, and to do this again for the outputs. We use the equality of the input and output attributes for the current element of the list and for the subsequent elements; by definition, a processing list must always contain at least one element so the attribute will always return a value.

The input and output attributes for parallel nodes are simple to compute: input is the input of one of the elements of the processing list, as they are all the same, and output is the output of the aggregator component. Finally, we have followed a safe approach when type checking processing lists: the input and output attributes for a processing list are given only after the type checking for the entire list is performed.

5   Implementation and Usage of the Portal

In this section, we present some implementation details about our portal and also a simple example of how it can be used in practice to create a sample Certification that outputs the number of lines of a source code file in C.

The portal has been constructed out of circa 320 lines of JavaScript and of circa 1500 lines of HTML+PHP code. In fact, from these 1500 lines, around 125 interface with a simple database for storing information related to the certifications of the portal, which itself includes 3 tables and 12 records. The DSL that the portal provides for re-arranging certifications and components was developed out of around 400 lines of Haskell code.

In order to use the portal to construct a certification that outputs the number of lines of a C program, we may rely on the existance of components readFile, that reads a file, input2Nlines, that takes an input and computes the number of its lines and int2Report, a component that takes an integer as input and produces a report. Since these components are already available in the portal, we express in our DSL a certification that arranges them as shown next:

```
Input
  >- (readFile,"-c","-string")    >- (cSlicer,"-string","-string")
     (input2Nlines,"-string","-int") >- (int2Report,"-int","-rep")
```

This certification starts with a component that reads a file in C (which is expressed by parameter -c) and returns a String (-string). A second component, executed in sequence, reads the string that is returned and filters the main procedure out of it, a text which is fed to a third component that counts the number

of its lines. Then, a final component transforms that Integer into an information report. In the portal, after defining a certification, users must give it a name and describe the analysis it implements, so that it may be reused in the future.

Once a certification for a particular programming language is available, users just need to upload a file in that language to analyze it. Having done so, our portal only presents as certification options for it the ones that match its type. This means that, for example, having uploaded a Haskell file, users will only see the certifications that are available for Haskell. Then, by choosing one particular certification, e.g., the one we have just created, the web portal will produce a report similar to the one shown in Figure 7.[2]

This report was validated by our XML Schema.

This report is an XML file transformed through XSLT. You can see the raw XML file.

See report in XML!

CROSS Certification report:

This report was generated in www.cross.di.uminho.pt on 2012-06-19.

The file uploaded was: *unarchiverC.c*

This report was produced by the Component int2report!

Description:

```
This result was produced by the Component: int2Report. This component receives an integer through STDIN
and creates a report in our XML Schema.
```

Result:

```
The result of this certificaton is: 54 .
```

End of the report!

Fig. 7. The result of a certification as it appears in our web portal.

Our approach is modular, and even if a component with a given functionality is not available, users can create it in any programming language. Indeed, as long as that component is able of receiving information via STDIN and of outputting information to STDOUT, any program is a candidate for a component. Also, we expect the number of components and certifications available on the portal to grow, which will make creating new certifications increasingly simpler.

Although we have used simple examples to illustrate our framework, this does not compromise the range or the complexity of the analyses that we may perform. For example, component input2Nlines could be replaced by one that implements a powerful pointer analysis on C code. And it would be as easy to create a

---

[2] In fact, the Figure 7 shows the HTML that corresponds to the XML that is produced.

certification using it as it was to create the one above, without programmers of the new component needing to concern with file reading or output formats.

6   Related Work

Several projects have focused on the analysis and assessment of software, being the Squale project [12], QSOS [13] and the Alitheia Core [14] important examples of this.

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.

Our solution allows a wide range of tools based on different programming languages and techniques to be imported into our portal, taken that such tools are capable of running as bash tools and that they receive information through the standard STDIN and STDOUT Unix's streams. We believe this includes a significant amount of already existing potential tools.

What is more, through the use of our DSL, virtually any tool in our portal can be connected to other tools to create a flow of information (as long as the input and output types of two chained tools match), easily allowing the introduction of software assessments and the extension of such of assessments.

In [15] an implementation of the orchestration language Orc [16] is introduced as an embedded domain specific language in Haskell. In this work, Orc was realized as a combinator library using lightweight threads. Despite the similarities on the use of Haskell combinators, this approach differs from our DSL since we do not rely on any existing orchestration language. Rather, we generate low level Perl scripts from combinators whose inputs are direct references to system processes (Components). Also, the way we manage processes does not rely on Concurrent Haskell, but rather on the parallelization features of the target system. More information about our DSL, together with examples of the scripts that it is able of generating can be found in [17].

7   Conclusions and Future Work

In this paper we present a portal for analyzing source code artifacts and providing information reports about them. Our portal supports various analysis scenarios and is able of dealing with programs expressed in different programming languages.

We have also implemented a DSL that allows manually re-arranging the certifications and components that are built-in the portal. While several analyzes are already possible, we rely on inputs from the community to extend further the certification tools that our portal hosts, and by this to increase its impact.

Although our portal has been deployed and is fully-functional, we are still incorporating in it several features such as allowing for tool developers to configure themselves how their tools are constructed and executed on our portal.

References

1. Haigh, M.: Software quality, non-functional software requirements and it-business alignment. Software Quality Control **18**(3) (September 2010) 361–385
2. Stavrinoudis, D., Xenos, M., Peppas, P., Christodoulakis, D.: Early estimation of users' perception of software quality. Software Quality Control **13**(2) (June 2005) 155–175
3. Dromey, R.G.: Software quality prevention versus cure? Software Quality Control **11**(3) (July 2003) 197–210
4. Wilson, D.N., Hall, T.: Perceptions of software quality: a pilot study. Software Quality Control **7**(1) (May 1998) 67–75
5. Chulani, S., Boehm, B., Verner, J., Wong, B.: Workshop description of 4th workshop on software quality (wosq). In: Proceedings of the 2006 international workshop on Software quality. WoSQ '06, New York, NY, USA, ACM (2006) 1–2
6. Cunha, J., Fernandes, J.P., Ribeiro, H., Saraiva, J.: Towards a catalog of spreadsheet smells. In: 12th Int. Conf. on Computational Science and Its Applications. Volume 7336 of LNCS., Springer (2012) 202–216
7. Cunha, J., Fernandes, J.P., Mendes, J., Martins, P., Saraiva, J.: Smellsheet detective: A tool for detecting bad smells in spreadsheets. In: Proceedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing. VLHCC'12, Washington, DC, USA, IEEE Computer Society (2012) (to appear).
8. Halstead, M.H.: Elements of Software Science (Operating and programming systems series). Elsevier Science Inc., New York, NY, USA (1977)
9. Knuth, D.E.: Semantics of Context-free Languages. Mathematical Systems Theory **2**(2) (1968) 127–145 Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).
10. Fernandes, J.P., Saraiva, J.: Tools and Libraries to Model and Manipulate Circular Programs. In: PEPM'07: Proceedings of the ACM SIGPLAN 2007 Symposium on Partial Evaluation and Program Manipulation, ACM Press (2007) 102–111
11. Swierstra, D., Vogt, H.: Higher order attribute grammars. In Alblas, H., Melichar, B., eds.: International Summer School on Attribute Grammars, Applications and Systems. Volume 545 of LNCS., Springer-Verlag (1991) 48–113
12. Squale: Front page. `http://www.squale.org` [Accessed in August 2012].
13. QSOS: Front page. `http://www.qsos.org` [Accessed in August 2012].
14. Alitheia Core: Front page. `http://www.sqo-oss.org` [Accessed in August 2012].
15. Campos, M.D., Barbosa, L.S.: Implementation of an orchestration language as a haskell domain specific language. Electron. Notes Theor. Comput. Sci. **255** (November 2009) 45–64
16. Kitchin, D., Quark, A., Cook, W., Misra, J.: The orc programming language. In: Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems. FMOODS '09/FORTE '09, Berlin, Heidelberg, Springer-Verlag (2009) 1–25
17. 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