View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Universidade do Minho: RepositoriUM

A Visual DSL for the Certification of Open Source Software

Tiago Carção and Pedro Martins*

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

Abstract. Quality assessment of open source software is becoming an important and active research area. One of the reasons for this recent interest is the consequence of Internet popularity. Nowadays, programming also involves looking for the 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 a coherent piece of software.

In this paper we improve over previous works and describe a visual language that allows programmers to graphically describe how software artifacts can be combined into powerful software certification processes. This paper introduces the visual language and describes how its elements are available to the user through an intuitive interface.

Keywords: Software Analysis, Software Certification, Open Source Software, Programming Languages, Process Management, Web

1 Introduction

The advent and massive use of the Internet not only changed how people communicate, for example, via social networks, but also how software developers build their large and complex software systems. For software developers one of the main results of the Internet was the creation of large open source software repositories, such as *sourceforge*, where we may find libraries and tools for an immense variety of problems.

As a consequence, developing software nowadays not only involves reusing libraries provided by the underlying programming language, but also reusing libraries and tools that have been built by other software engineers and that are available as open source software.

In order to reuse such open source software, developers need to trust that software and, as a consequence, they often need to be certified that a reused

* This work is co-financed by the North Portugal Regional Operational Programme (ON.2 O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), within project NORTE-07-0124-FEDER-000058.

software artifact does satisfy certain properties. For example, a developer may need to be sure that the copyrights involved in the reused software allow its usage. In our context, we refer to analyzing a property of piece of open source software as its certification.

In this paper we extend our work on developing a customizable web portal [1] for the certification of reusable, open source software packages. One of the main features of our web portal for the certification of open source software is the Domain Specific Language (DSL), introduced in [2]. This DSL allows the users of the web portal to define/combine new certifications as the combination of several different software tools. This DSL is implemented in the web portal as an Embedded DSL (EDSL): a DSL embedded in the Haskell programming language. Although this approach is powerful and did allow a quick development of the web portal, it contains two main issues: First, to define a new certification in that EDSL, the software developer needs to be an expert in the Haskell programming language. Secondly, and most important, the EDSL provides both poor syntax and error reporting since they are provided by the host language.

The purpose of this paper is two-fold:

- Firstly, we introduce a Visual Domain Specific Language (VDSL) where it is easy to specify the reuse and combination of (open source) software tools. Moreover, we build a proper compiler for such VDSL which reports errors in the web portal domain. This compiler translates the visual DSL to our former textual Haskell-based EDSL.
- Secondly, we present a case study where a tool to monitor energy consumption is specified in our visual DSL. Thus, we show how the widely used (open source) graph visualization tool *GraphViz* tool is instrumented in order to produce a report showing the energy consumption per its functions.

This paper is organized as follows. In Section 2 we provide an overview of the motivation and potential challenges this work faces. In Section 3 we introduce our visual language together with small examples of its usage. In Section 4 we present the process of creating a certification. The case study that we have used to demonstrate our VDSL is described in Section 5. In Section 6 we provide an overview of related works, and finally in Section 7 we conclude.

2 Motivation

The CROSS portal developed encloses the composition of tools to certify software. 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.

These certifications can be composed out of various elements which can be arranged in a way that produces the wanted analysis, through a textual DSL presented in the portal. This DSL combines components, which are simple processes which together compose a certification.

In the same certification multiple analysis can be done, as in the DSL we have a notion of flow of information: from the program the user sequences components

that transform inputs and produce new results which are themselves fueled to other components. In each certification, multiple flows can be implemented and then joined by a special type of component, an aggregator. Figure 1 shows a certification illustrating an analysis to a program with five different components.

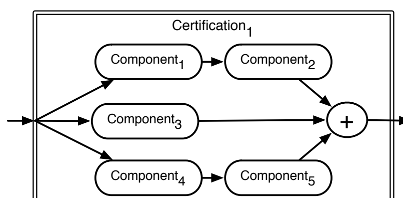


Fig. 1: An example of a certification composed by five components.

Despite the powerful characteristics of the DSL, and the powerful analysis that can be implemented with it, the fact that this is a textual language embedded in hosting languages creates some disadvantages in this environment.

Firstly, having a textual representation forces the user to learn a new language. It is defined by constructs and primitives that user has to be aware of, as well as a set of syntactic and semantic rules mandatory to the creation of certifications. A visual language is much more intuitive, with drag and drop interfaces and real-time visualization of the flow of information, the learning curve is greatly diminished. Secondly, one great disadvantage of embedded DSLs is that error messages are related to the hosting language, not to the domain. This means that in the example of the web portal CROSS, whenever a user accidentally makes an error he is presented with error messages that have no relation whatsoever with the domain of software analysis or processes composition. This can make using the DSL hard for users not familiar with the hosting language.

A visual language make everything easier to handle: graphical representations of process relation are always visible, some errors can be avoided just by forbidding specific, invalid certifications on the graphical environment and errors are targeted to the specific domain of software analysis, making debug of certifications much easier and faster.

3 A Visual Language for Certifications

In this section we will introduce the visual language developed to allow easy implementation of processes in our portal. To desing such VDSL, we started by identifying all syntactic elements of the textual language; and by understanding how to recreate them visually. Their graphic representation would allow a more intuitive use of the language and reduce its associated learning curve.

In order to create a new certification, users always need to specify a flow that the certifying programs must follow. This flow starts in the inserted input and

goes through a series of components to generate a report. From the same input, multiple customized flows of information can be used, but they must eventually be explicitly aggregated to generate a report. These elements are linked together by a connection that symbolizes the notion of the program flow between them.

The existing elements in the textual language which should be represented in the visual language are: *input*, *connection*, *components* and *aggregators*, which we shall present next.

The components of our visual language have been implemented in a system that is not restrictive on their organization and layout. This allows users to rearrange positions and connections with little effort. The fact that this is a flexible environment but with controlled actions, unobtrusively enforces the correct construction of certifications.

The elements which were identified in the textual language and implemented in the visual language are explained next in detail.

3.1 Input

The input represents the program to certify, and from here a certification can be composed. Each added component will indicate the action to be applied to the input. The textual representation can be seen in Listing 1.1.

Listing 1.1: Input specified in the textual language.

```
Input
```

The graphic design of the input element, as is shown in Figure 2, is similar to the UML input for flow diagrams.

Fig. 2: The Input graphical representation.

3.2 Component

A component is an element of the flow of information throughout a certification. The functionality of this element can be see as a single process to which information is fed and producing a desired result. All components have at least on input type and one input type, but some have various which have to be defined by the user.

Each component has a standard format: a parallelogram with unique and useful visual information. From the textual language, presented in Listing 1.2, we can see that each component has a name (which by definition is unique within

all components), and the description of its input and output type.

Listing 1.2: A Component specified in the textual language

```
(readFile, "-j", "-s")
```

Our visual language allows easy navigation and selection of the available components in the portal. For each component, relevant information is shown: the description of the component, and the input/output types it supports. To choose a component and add it to the certification creation process, it is necessary to select the component from the set of available components and to customize it by choosing its input and output types.

In the example of Figure 3, the user selected the component *readFile*, with *Java* as the input type and text as its output type.

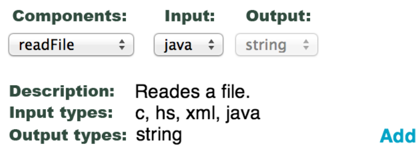


Fig. 3: Information presented to the user, associated with the available components.

The graphical representation of a component, as can be seen in Figure 4, shows its name and the chosen languages for input and output separated by a symbol of transformation `->`. Also indicated is a green circle representing the point of input where one can connect the flow either from the initial input, from other component or from an aggregator. The blue circle is the output point.

Fig. 4: A Component specified in the visual language.

3.3 Aggregation

An aggregator is a specific type of component that aggregates various flows of information. This element can be seen as a special type of component and there-

for the graphical representation of an aggregator is similar to the representation of a component.

In our textual representation, an aggregator would be represented as seen in Listing 1.3.

Listing 1.3: Textual version of an aggregator that takes the number of lines of multiple sources and produces a report.

```
>|> (Aggregator, "-i", "-rep")
```

Adding an aggregator, in our visual environment is similar to adding a typical component: the user has to choose from a list of available aggregators. As other components they where they can see detailed information about it, such as its functionality and type.

The graphical representation of an aggregator, which can be seen in Figure 5, has a name that is unique within all aggregators and the input and output language separated by a symbol of transformation `->`.

Fig. 5: Example of an aggregator, that takes the number of lines of multiple sources and produces a report.

Since there are differences between how an aggregator and a typical component handles input information, with an aggregator being an element that will receive information from multiple sources, we changed the aspect of the input point to be represented as a green square meaning it can receive multiple flows. In component the input point, as shown in Subsection 3.2, is a green circle. Since the output works like a normal component, the graphical output representation is a blue circle.

3.4 Connection

The connection is the element linking two components or a component to an aggregator. Each connection has an arrow that showing of the flow and a label displaying the types that flow will handle, as can be seen in Figure 6 represented by the yellow arrow.

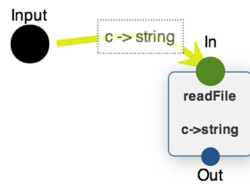


Fig. 6: Connecting two components.

4 Creating Certifications

With the visual language elements, defined in the previous section, we shall now describe how different types of certifications with multiple components and layouts can be defined in our setting.

4.1 Sequential Flow

Our visual language allows a wide type of certifications, based on flows of information, to be implemented. One of these possibilities is the sequential flow.

Listing 1.4: Sequential flow of information in a textual form.

```
Input
    >- (readFile, "-c", "-s")
    >- (text2NLines, "-s", "-i")
    >- (int2Report, "-i", "-rep")
```

This flow is a connection of tools in a sequential order, allowing the analysis of a software program all the way from the Input to the generation of a final report, chaining different components throughout the process.

One example of a certification is one where we want to analyze the number of lines in a C program. In the textual language we have to write the sequence of components linked by the combinator >- and make sure that the types and their representation is correct, as can be seen in Listing 1.4.

In the visual language, this process is faster and more intuitive. The user only needs to select the tools and its types, and link them with connections. Since our visual setting is restricted by invalid connections, the user does not have to worry about the correctness of the language syntax, since these are simply not allowed.

An example of the implementation of the certification to analyze the number of lines in a C program can be seen in Figure 7.

4.2 Parallel Flow

In addition to the sequential flow, our setting also supports parallel flows of information obtained when combining the components in two or more process

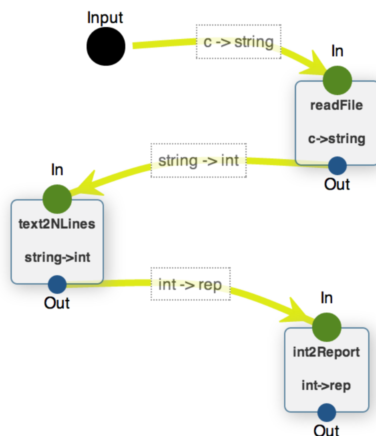


Fig. 7: Sequential flow of information.

chains. This flow, can have multiple paths, where each of these paths is sequential and joined by the element aggregator.

For instance, let us imagine that we want to analyze a tool, and know how many *ifs* conditions and *for* loops it has. This can be done by implementing a certification with two paths, where one counts the number of ifs conditions and the other counts the number of *for* loops. Afterwards, the results are joined by an aggregator producing a report.

Listing 1.5: Parallel flows of information described textually.

```
Input
    >- (readFile, "-c", "-s")
    >- (text2NFors, "-s", "-i")     >|
Input
    >- (readFile, "-c", "-s")
    >- (text2NIfs, "-s", "-i")      >|>
    (Aggregator, "-i", "-rep")
```

This certification can be expressed in the textual language by building two different paths. For each path, the user has to define the flow from the `Input` and include the combinator `>|` in the end to give an indication that the returned information will be aggregated. This can be seen in Listing 1.5.

In order to build this certification we need to add two paths. We need to start each of these paths with the readFile component. In order to count the number of *for* loops, in the left path, we add the text2NFors component. We do the same in the right path but this time to count the *if* loops, using the element text2NFors component. These two components produce integers in their analysis, which are funneled to an aggregator which generates a report with the relevant informations. The graphical representation of this process is in Figure 8.

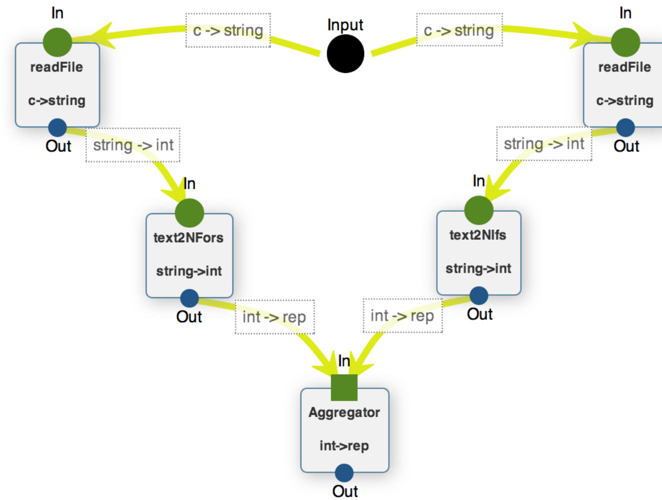


Fig. 8: Parallel flows of information.

4.3 Verification

As explained before, creating certifications in our settings has some rules, related to type correctness and to the structure of the computation chain that composes the certification. One important rule comes from the connection of a `component1` to a `component2`, where the first must have the same output type as the input type of the second.

These rules, in the textual language, can only be verified when the certification is fully constructed. Only after the user describes the certification can the the syntax and semantics of the language be analyzed.

In our visual language, this verification is done in real time. When one tries to link two components and their type do not match, an alert notification appears and the system simply forbids the creation of this connection. This is exemplified in Figure 9.

Another important rule when defining a certification is that one path must be continuous and sequential, i.e., all components in the certification must be linked together in a sequential manner. Elements of a certification are have to be fed an input to produce their result. This verification is automatically performed and all errors found are signaled and presented to the user, as we can see in Figure 10.

In Figure 11 we see yet another example of how our visual language aids in certification construction. In this example there are multiple paths which should be joined by an aggregator. This, as we saw in Figure 10, is also verified and any problems found are presented to the user..

Due to its interactive nature, a visual language improves over a previous textual approach as problems are easier to detect and are presented to the user

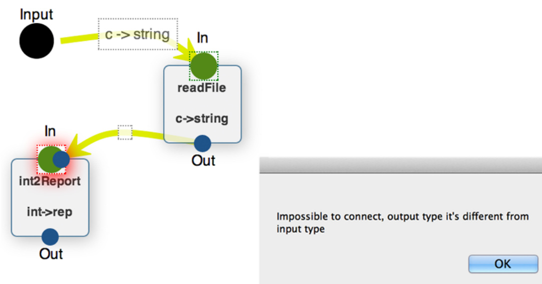


Fig. 9: Trying to establish a connection between two incompatible components.

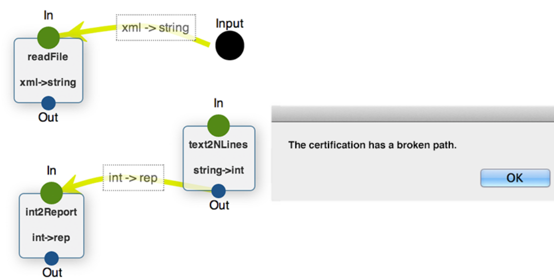


Fig. 10: Trying to create a certification with a broken path.

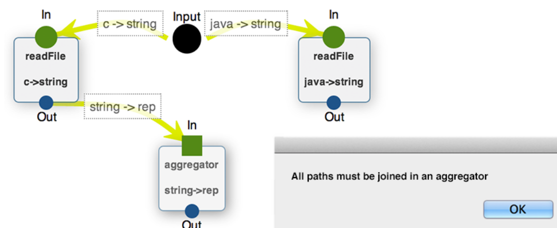


Fig. 11: Trying to create a certification with multiple paths not joined by an aggregator.

in a setting that is easier to understand and to correct. Furthermore, some problems, typical of a textual setting, are simply inexistent here as the visual environment denies certain compositions of elements.

4.4 Technical Details

In order to facilitate familiarization with the language and to improve user-experience when dealing with it, we were influenced by existing visual elements of

the Unified Modeling Language (UML) to represent some of its items. In Figure 12, for example, we see a graphical representation of the Input very similar to the one found in UML. Another example of similarities is in the flow presented in activity diagrams, with boxes containing important nodes in the flow of information and arrow relating these boxes.

Fig. 12: Start point of UML activity diagrams.

This language was conceived to be embedded in an open source analysis web portal - CROSS [1][1]. Since we are dealing with a web environment, the chosen techniques relate to the ones widely used when developing web applications.

The visual language was developed using the programming language JavaScript. Alongside JavaScript, we used a well-known framework for the language, jQuery[2]. This framework simplifies the use of JavaScript and adds the possibility of using plugins to further aid in the development.

One plug-in used was jsPlumb[3], a plug-in that allows an environment where one can connect elements in a UI. This environment can be used to represent state machine or activity diagrams and user-specified diagrams. All the source code that implements the visual language can be can be consulted in the web portal, at `www.cross.di.uminho.pt`.

5 Case Study

The visual language for certifications presented in this work was designed to be used with different components and in different contexts, covering a wide range of possible analysis.

In this section we will present an example of how the functionality of the web postal can be used to analise a software program, written in the C programming language, and produce a report with information regarding energy consumption.

To do so, we use a work that has been developed in the context of power consumption in software [3]. In this particular work, techniques were developed to measure the impact of software design in energy consumption, a topic of high relevance with the current wide usage of smartphones and other mobile devices.

The analysis that we intend to make, regarding power consumption, goes through different phases. Initially it is necessary to instrument all the software modules so that each function can produce an output in order to know how much energy was spent on processing when the function was called. Once the software has been instrumented, it must be compiled to machine code.

[1] `www.cross.di.uminho.pt`
[2] `http://jquery.com`
[3] `http://jsplumbtoolkit.com/`

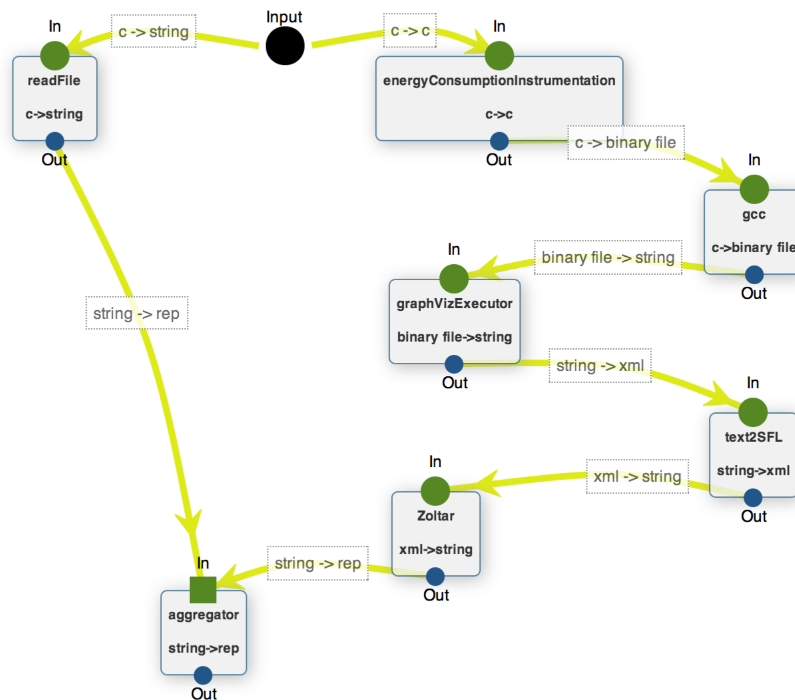


Fig. 13: Certification of energy consumption analysis for GraphViz

One of the known methods we can use is the Spectrum-based Fault Localization (SFL) [4]. This method uses the information of the running spectrum of the program and, along with the information about the input and the expected output, can indicate what are the faults in the software.

Adapting an SFL-based algorithm to the energy analysis consumption creates an adapted SFL model for energy consumption, with which we can use the energy consumption information and build a matrix with the spectrum of the program consumption and, again, apply SFL techniques to obtain information from a matrix. All this can be seen in [3].

This analysis of the energy consumption can be made with a certification using a combination of various tool and components. Doing so, aids the user in implementing the analysis as he/she does not have to fiddle with the typical textual language constructs we have seen before.

As input for the study case we chose the open source tool GraphViz[4]. GraphViz is a software tool that allows textual representations to be presented in a visual format, such as graphs. This tool has different modes of presentation and is widely configurable.

[4] www.graphviz.org

In order to allow the representation of every phase of the energy consumption analysis in the certification creation process it is necessary to choose which tools must be available. The instrumentation of the ¸software modules can be made by using a tool developed which uses clang to instrument the software.

The software compilation is made with gcc[5]. To execute Graphviz with different inputs we have used a tool that contains a textual graph sample and runs Graphviz with different flags, generating different visualizations. We have also used a tool that collects the outputs of power consumption of each function and builds the SFL matrix.

The tool to analyze the SFL matrix and to present the functions with problems was Zoltar [5] as. Thus, a tool that gathers the Zoltar output and generates a report describing software consumption using a graph bar which is explained in [3] was also used. This tool has the additional function of pretty printing the software with information about functions' energy consumption.

This certification implemented in our visual language can be seen in Figure 13. In this certification we can see that there are two paths, one of the paths represents the flow of code instrumentation, gathering results, and analyzing the SFL information. In the other path we have the flow that will allow the pretty printing. Both paths are joined in an aggregator that will assimilate and produce a report which is the conjunction of the results of both paths.

The certification produces a report with the GraphViz consumption values, as can be seen in Figure 14. Here we can see that in the first section we have a summary of the total functions analyzed and the total power consumed. In section two, we have two graph bars. In both graphs the yy axis represents the power consumption. The xx axis represents the modules and functions in graph one and two respectively. Each xx value has 6 series - the 6 series represent the 6 different inputs to which GraphViz was tested.

In this section we presented a case study that analysis a source code of a regular programming language. Our VDSL, however, can define certifications for other software artifacts, by composing both simple tools, like for example HaLeX for reason about regular expressions[6], and complex software systems, like the GuiSurfer framework to reason about intercatice specfications [7, 8], or the LRC attribute grammar system [9]. By combining such tools we are able to define powerful software certifications.

6 Related Work

This works improves on previous ones, related to both the construction of a language for process management and the implementation of a web portal to certify software. In [2] we present a textual language for process management. This language is based on the functional language Haskell and uses zipper-based techniques, as the ones found in [10] to develops attribute grammar techniques to defines semantic analysis on the language [11]. Because we express our VDSL

[5] `http://gcc.gnu.org`

in an AG setting, we get for free well-kown techniques to analise and optimize our visual programs/specifications, namely the detection of circularities [12], the optimization of such circularities [13–16], and the incremental execution of our programs [17]. In [1] and [18] we presented the portal and develop a framework that allows not only the certifications and analysis of open source software, but we also introduce novel language-independent techniques to further aid generating information about computer programs submitted to our portal.

There are works on languages for process management. Of relevant reference is [19], an implementation of the orchestration language Orc [20] is introduced as an embedded domain specific language in Haskell. In this work, Orc was realized as a combinator library using the lightweight threads and the communication and synchronization primitives of the Concurrent Haskell library [21].

7 Conclusions and Future Work

In this paper we presented a visual language that allows the creation of software certifications in the CROSS web portal. This language enables the user to create certifications to analyze software without needing to deal with the typical disadvantages of an embedded DSL.

For this visual language we created different visual elements and implemented the notion of sequential and parallel flows of information. We also worked on the validation associated with the construction of certifications. We also present a full example through a case study which analyzes energy consumption on a open source tool, GraphViz. We show how this certification can be implemented in our visual language and the type of results we can produce with our environment.

As a future work, we intend to extend the visual language to also allow reports generation and customization, by allowing specific results to be agglomerated into chapter, sections and subsections of the report.

References

1. Martins, P., Fernandes, J.P., Saraiva, J.: A Web Portal for the Certification of Open Source Software. In Cerone, A., et al., eds.: SEFM Satellite Events. Volume 7991 of Lecture Notes in Computer Science., Springer (2012) 244–260
2. 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 (2012) 51–69
3. Carção, T.: Spectrum-based Energy Leak Localization. Master's thesis, University of Minho, Portugal (2014, in preparation)
4. Abreu, R., Zoeteweij, P., van Gemund, A.: On the accuracy of spectrum-based fault localization. In: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques – Mutation (Mutation'07). (2007) 89–98
5. Janssen, T., Abreu, R., van Gemund, A.J.: Zoltar: A spectrum-based fault localization tool. In: Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution @ Runtime. SINTER '09, ACM (2009) 23–30

6. Saraiva, J.: HaLeX: A Haskell Library to Model, Manipulate and Animate Regular Languages. In: ACM Workshop on Functional and Declarative Programming in Education. University of Kiel - TR 0210 (September 2002) 133–140
7. Silva, J.C., Saraiva, J., Campos, J.C.: A generic library for gui reasoning and testing. In: Proceedings of the 2009 ACM Symposium on Applied Computing. SAC '09, New York, NY, USA, ACM (2009) 121–128
8. Silva, J.C., Silva, C., Gonçalo, R.D., Saraiva, J., Campos, J.C.: The guisurfer tool: Towards a language independent approach to reverse engineering gui code. In: Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems. EICS '10, New York, NY, USA, ACM (2010) 181–186
9. Kuiper, M., Saraiva, J.: Lrc - A Generator for Incremental Language-Oriented Tools. In Koskimies, K., ed.: 7th International Conference on Compiler Construction (CC/ETAPS). Volume 1383 of LNCS., Springer-Verlag (1998) 298–301
10. Martins, P., Fernandes, J.P., Saraiva, J.: Zipper-based attribute grammars and their extensions. In: Procs. of Brazilian Conference on Programming Languages (SBLP). Number 8129 in LNCS, Springer-Verlag (2013) 135–149
11. Swierstra, S.D., Alcocer, P.R.A., Saraiva, J.: Designing and implementing combinator languages. Lecture Notes in Computer Science **1608** (1999) 150–206
12. Kastens, U.: Ordered attribute grammars. Acta Informatica **13** (1980) 229–256
13. Saraiva, J., Swierstra, D.: Data Structure Free Compilation. In Stefan Jähnichen, ed.: 8th International Conference on Compiler Construction, CC/ETAPS'99. Volume 1575 of LNCS., Springer-Verlag (March 1999) 1–16
14. 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
15. Pardo, A., Fernandes, J.P., Saraiva, J.: Shortcut fusion rules for the derivation of circular and higher-order programs. Higher-Order and Symbolic Computation (2011) Springer, 1–35
16. Fernandes, J.P., Pardo, A., Saraiva, J.: A shortcut fusion rule for circular program calculation. In: ACM SIGPLAN Haskell Workshop. Haskell'07, New York, NY, USA, ACM (2007) 95–106
17. Saraiva, J., Swierstra, S.D., Kuiper, M.F.: Functional incremental attribute evaluation. In Watt, D.A., ed.: 9th International Conference on Compiler Construction, CC/ETAPS'00. Volume 1781 of LNCS., Springer (2000) 279–294
18. Martins, P., Carvalho, N., Fernandes, J., Almeida, J., Saraiva, J.: A framework for modular and customizable software analysis. In: Proc. of Computational Science and Its Applications ICCSA 2013. Volume 7972 of LNCS. Springer (2013) 443–458
19. Campos, M., Barbosa, L.: Implementation of an orchestration language as a haskell domain specific language. Elect. Notes Theor. Comput. Sci. **255** (2009) 45–64
20. Kitchin, D., Quark, A., Cook, W., Misra, J.: The orc programming language. In: Proc. of Joint Conf. FMOODS/FORTE '09, Springer (2009) 1–25
21. Peyton Jones, S., Gordon, A., Finne, S.: Concurrent haskell. In: 23rd Symposium on Principles of programming languages. POPL '96, ACM (1996) 295–308

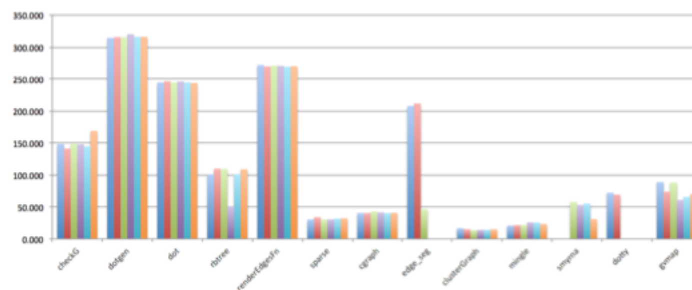
Energy consumption analysis

March 14, 2014

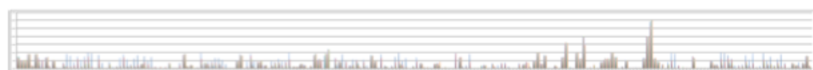
1 Information

- **Number of functions analyzed** - 753
- **Total energy consumed** - 3000mW

2 Consumption by modules



3 Consumption by functions



1

Fig. 14: The report generated in the certification for GraphViz