Project N°:	**FP7-610582**
Project Acronym:	**ENVISAGE**
Project Title:	**Engineering Virtualized Services**
Instrument:	**Collaborative Project**
Scheme:	**Information & Communication Technologies**

Deliverable D5.2.2
Envisage Virtual Collaboratory (Final Version)

Date of document: T36

Start date of the project: **1st October 2013** Duration: **36 months**

Organisation name of lead contractor for this deliverable: **UCM**

Final version

STREP Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Envisage Virtual Collaboratory (Final Version)

This document summarises deliverable D5.2.2 of project FP7-610582 (Envisage), a Collaborative Project supported by the 7th Framework Programme of the EC. within the Information & Communication Technologies scheme. Full information on this project is available online at `http://www.envisage-project.eu`.

Deliverable D5.2.2 is the final version of the Envisage Virtual Collaboratory[1] as part of the activities of Task T5.2. This task aims at making tools and technologies developed in the context of Envisage available for the general public as online services. This report supplements the prototype by documenting background work to realize the collaboratory.

List of Authors

Samir Genaim (UCM)
Einar Broch Johnsen (UIO)

[1] `http://abs-models.org/laboratory`

Contents

Chapter 1

Introduction

Deliverable D5.2.2 is a prototype documenting the final version of the Envisage Virtual Collaboratory, as part of the activities of task T5.2. This task aims at making tools and technologies developed in the context of Envisage available for the general public as online services, without any need for downloading and installing tools locally. This report supplements the prototype by documenting background work to realize the collaboratory.

We have developed a web site

`http://www.abs-models.org`,

of which the collaboratory itself is one component, which also features documentation about ABS and its associated tools, tutorials, examples, etc.

The Envisage Virtual Collaboratory is available at:

`http://abs-models.org/laboratory`.

The Envisage Virtual Collaboratory is built around a framework that we call EasyInterface. This framework allows developers to develop their applications once, and get several interfaces for free, e.g., a web-interface, an Eclipse-interface, a remote shell, etc. EasyInterface has been released as open source, and is available at `http://github.com/abstools/easyinterface`. This framework is described in Chapter 3, and its user manual is attached to this report as Appendix A.

In addition, for expert users that prefer local installation of the different tools, the GitHub repository `http://github.com/abstools/abstools` includes Vagrant and Docker configuration files for creating a Linux virtual machine with all tools installed (including an installation of EasyInterface). We will keep links in abs-models web site to pre-created virtual machines with the latest releases of all tools.

In Chapter 4 we finish this report with some concluding remarks.

Chapter 2

ABS Web Site

We develop a web site `http://www.abs-models.org` which makes the ABS tools available as a service through the Envisage Virtual Collaboratory, and additionally document the vision and approach of the Envisage project, the modeling language and tools. We have opted to build this web site around ABS rather than Envisage in order to make the language and tools less project specific. This chapter documents the current state of the web site, by a visual tour.

2.1 Arriving at the web site

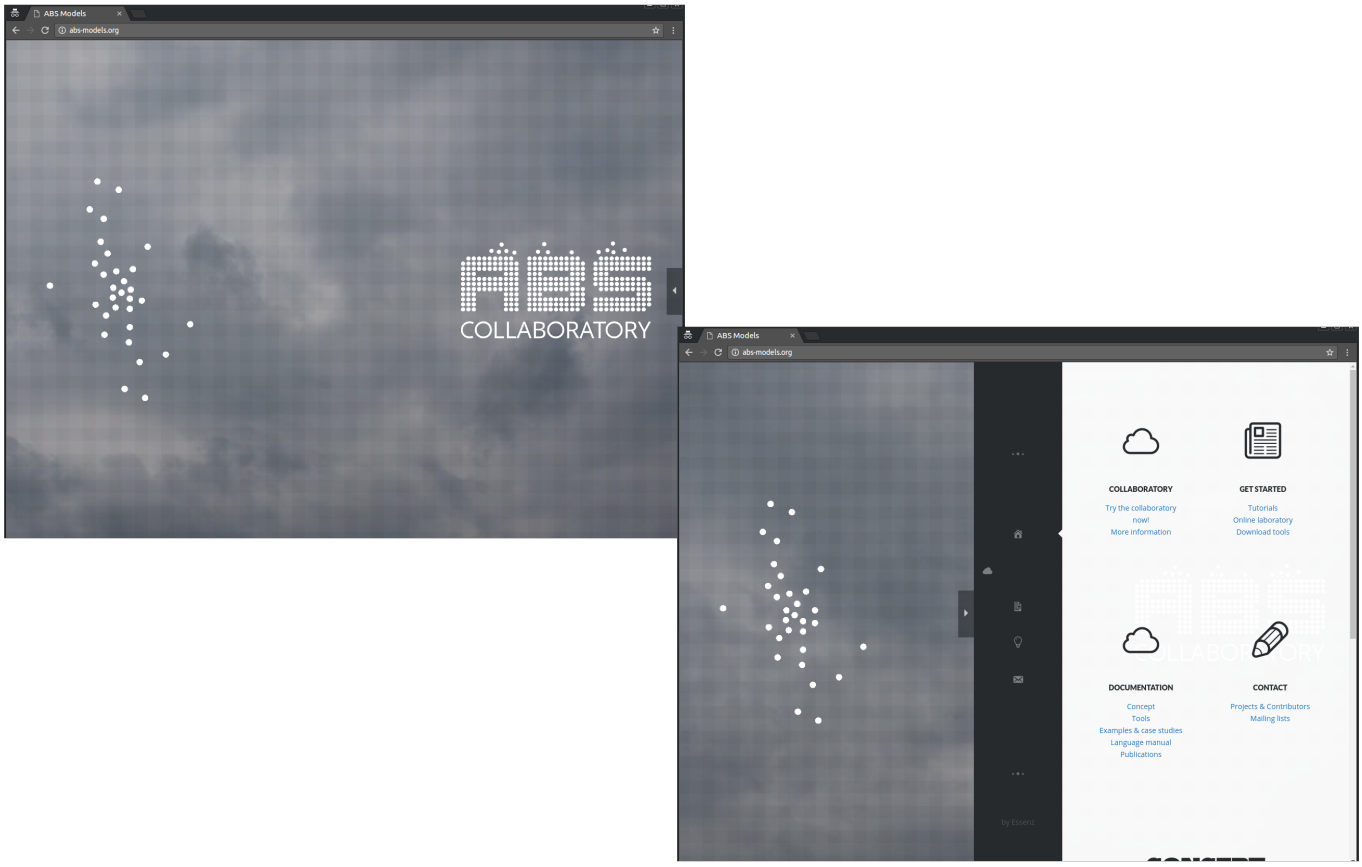

Figure 2.1: ABS web site: Compositionality in the clouds.

2.2 Vision

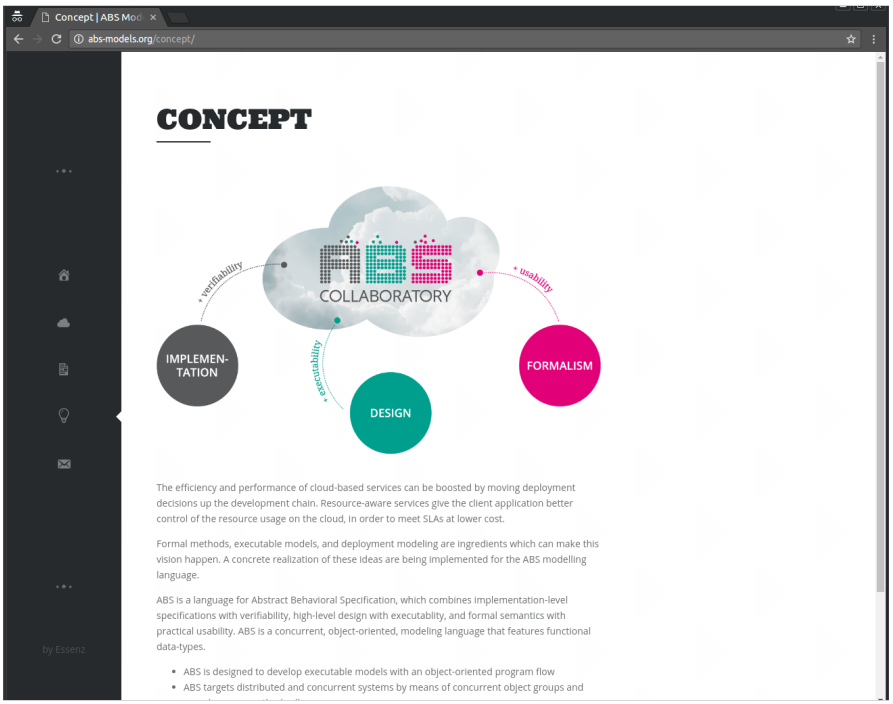

Figure 2.2: ABS web site: A vision for the clouds.

2.3 The Collaboratory

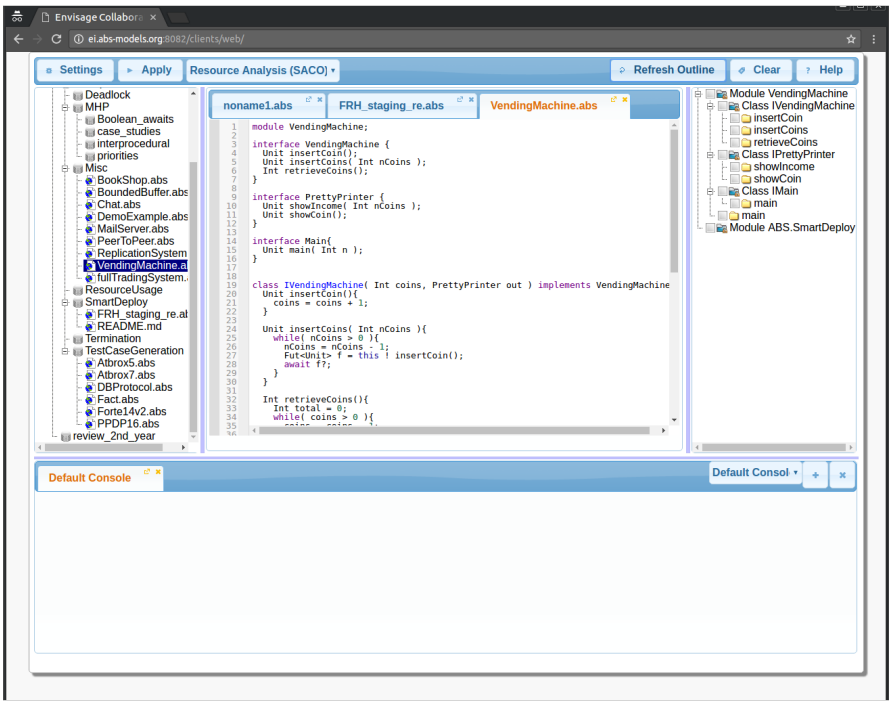

Figure 2.3: ABS web site: the Virtual Collaboratory.

2.4 Documentation

Figure 2.4: ABS web site: Documentation.

2.5 Projects & Contributors

Figure 2.5: ABS web site: overview of projects and contributors.

2.6 Mailing Lists

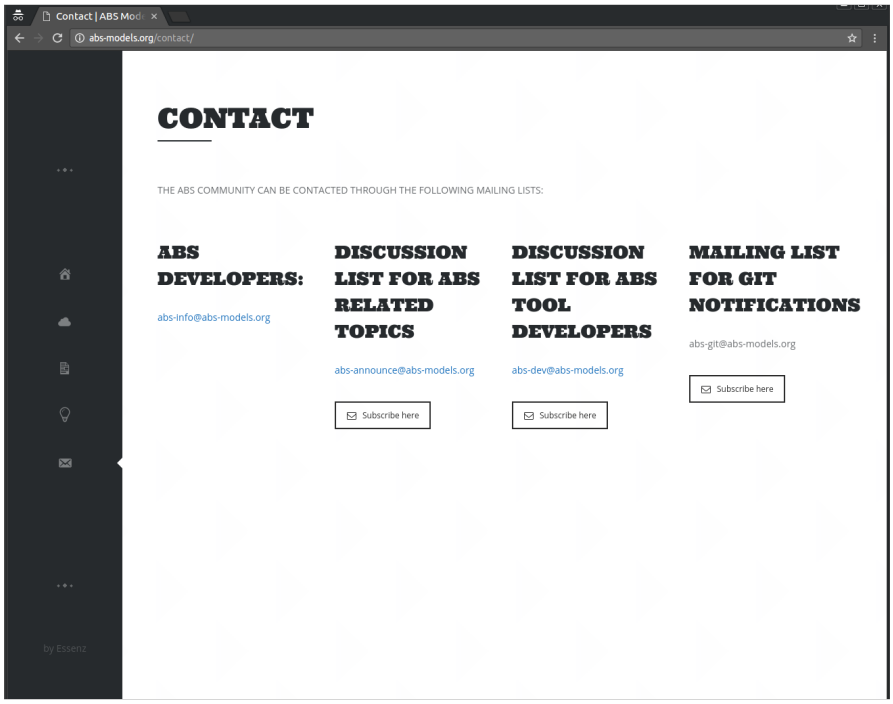

Figure 2.6: ABS web site: Mailing lists.

Chapter 3

Overview of the EasyInterface Toolkit

In this chapter we describe the overall design of EasyInterface, in particular we define its different components, their corresponding requirements and responsibilities, and how they communicate to accomplish our objectives that we state in Section 3.1. The content of this chapter is a summary of chapters 1 and 2 of the EasyInterface user manual, that is attached as Appendix A.

3.1 Objectives

Our main objective is to develop a toolkit that can be used to easily develop GUIs for research prototype tools, and, moreover, integrating them in a common environment. We focus on tools that have the following common aspects:

- A request to a tool corresponds to executing a program from a command-line, where the input is passed as command-line parameters (including file names) and the output is printed on the standard output. Note that this does not limit us to command-line tools, since, for example, if the tool runs as a server we can write a command-line wrapper that forwards the requests to the server and prints the server's response on the standard output.
- The tools are, in principle, supposed to process programs, e.g., static analysis tools. Thus, they receive as parameters (*i*) file names that represent programs; and (*ii*) possibly a list of program entities (e.g., method names, class names) which indicate some information to the tool (e.g., for static analysis tools it might indicate where to start the analysis from).
- The output of the tool mainly includes information that is related to some parts of the input program. For example, associating information to a program line, drawing graphs that represent some property of the program such as resource consumption, etc.

Note that the above aspects where not chosen arbitrarily, but they rather cover tools developed in the Envisage project. As regards the capabilities of the toolkit that we want to develop, we aim at developing one that complies with the following objectives:

O1 Using the toolkit for developing *simple* GUIs for existing tools, and integrating them in a common environment, should not take more than few minutes, and, moreover, *without* requiring any modification to the tool's code at all.

O2 Developing more sophisticated GUIs might require mild modifications to the tool's code, and by no mean deep modifications — requiring deep modifications would make it less likely that the toolkit will be used.

O3 Using the toolkit for presenting the output graphically should not require any knowledge on GUI or WEB programming. The user should describe the output in a natural language, e.g., "*highlight line*

number 10 of file ex.c" (or a structured version of such statement, e.g., using XML or JSON, to be able to parse it easily).

O4 The toolkit must provide common environments in which tools can be integrated. The most important environment is the web-based one (i.e., it runs in a web browser). In addition, the design should allow developing more environments in the future without modifying the toolkit or existing integrated tools.

O5 The common environments should be completely transparent to the integrated tools. This means that the work for integrating the tool or modifying it to produce graphical output should be done only once, and it should work in all environments equally (including future ones).

O6 The toolkit should be secure, it should not pose any security risks both to developers and users of to the corresponding tools.

To evaluate the developed toolkit, it should be used to build GUIs for the tools of the Envisage project, and, moreover, integrating them in a common environment.

3.2 The Overall Architecture of EasyInterface

In this section we describe the overall design of EasyInterface, which is driven by the objectives we have stated in Section 3.1. One of the most important objectives is O5, which aims at reducing the effort of integrating a tool in a common environment, namely, it aims at achieving a situation where a tool can be installed once, and then appears automatically in all available environments. This objective leads us to an architecture with two abstract components:

(i) the first abstract component represents an entity where tools can be installed; and

(ii) the second abstract component represents clients (e.g., development environments) that can communicate with the previous component to consult the list of installed tools, to execute a tool, etc.

This abstraction leads us to a client-server architecture as depicted in Figure 3.1, where the *server side* represents the first abstract component, and the *client side* represents the second abstract component.

The *server side* is intended to be a machine with several tools (the circles `Tool1`, `Tool2`, etc., in Figure 3.1) that can be executed from a command-line, and their output goes to the standard output (recall that our objectives are restricted to such tools – see Section 3.1). These are the tools that we want to make available for the outside world, i.e., execute them as services on the internet. The *client side* includes several clients that make it easy, to a user, to communicate with the server side to execute a tool, etc. Clients can be simple, e.g., a program that sends a very specific request, or more sophisticated development environments such as the web-based one that is included in Objective O4. Note that although Figure 3.1 includes only one server component, the overall setting allows several ones, and clients are configurable to connect to one or more servers.

The rest of this chapter is organized as follows. In Section 3.3 we explain the details of the server side; in Section 3.4 we explain the details of the client side, and finally, in Section 3.5, we explain how tools can produce output that is shown graphically in the client side (see Objective O3).

3.3 The Server Side

The problem that we want to solve at the server side can be summarized as follows:

> Provide a uniform way for remotely accessing locally installed tools as services, such that installing new tools and accessing them is straightforward.

However, this problem encapsulates different issues. In what follows, we address these issues and briefly explain how the are solver in EasyInterface.

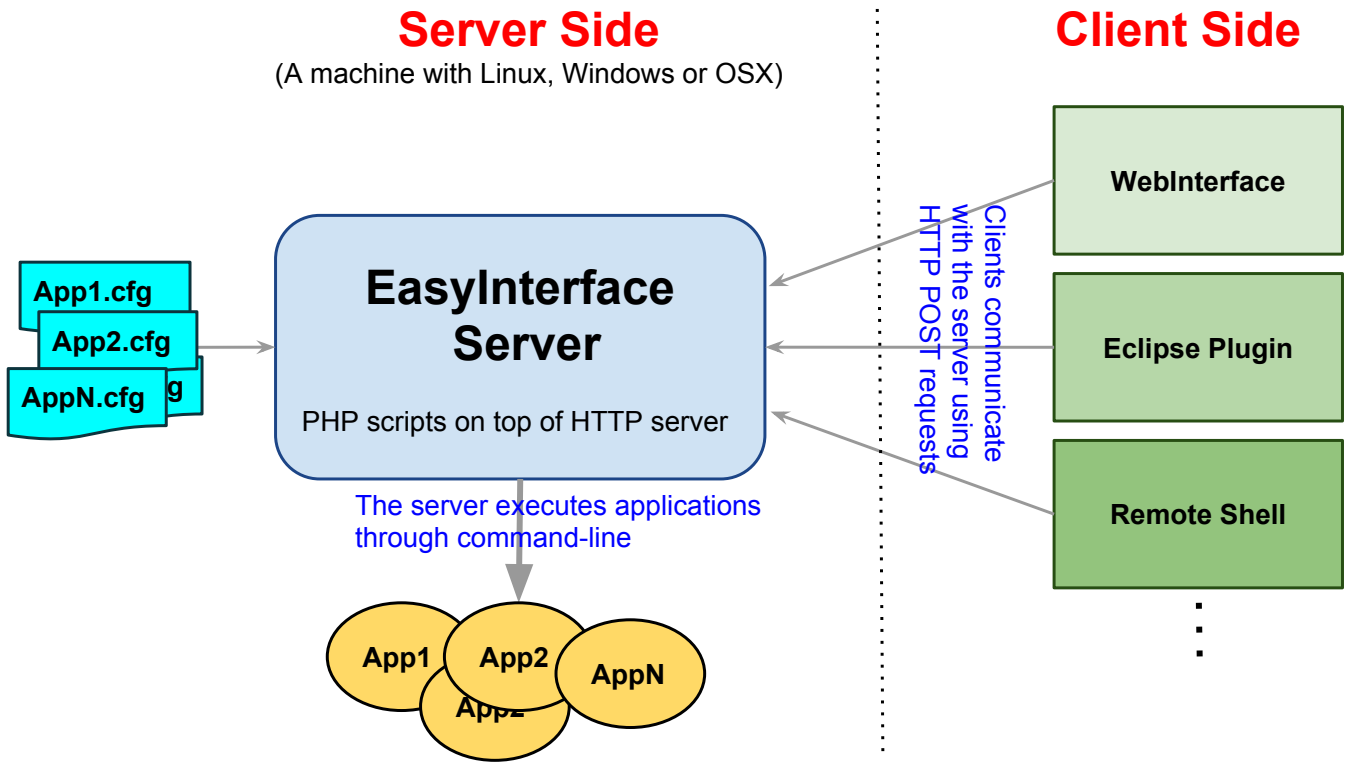

Figure 3.1: The Architecture of the EasyInterface Toolkit

3.3.1 Installing a New Tool

Developers (of research prototype tools) can make their tools available on the server in a straightforward way, in terms of simple configuration files (`Tool1.cfg`, `Tool2.cfg`, etc., in Figure 3.1). The only information they should provide is how the tool can be executed from a command-line, and which parameters it takes. Information about the parameters is crucial as clients will use it to ask users to select the desired values, etc. The following XML snippet corresponds to a tool configuration:

```
<app id="myapp">
  ...
  <execinfo>
    <cmdlineapp>/path-to/myapp.sh _ei_files _ei_parameters</cmdlineapp>
  </execinfo>
  <parameters prefix = "-" check="true">
    <selectone name="c">
      <option value="1" />
      <option value="2" />
    </selectone>
    ...
  </parameters>
</app>
```

This XML defines a tool that has a unique identifier `myapp`, which is used to refer to this tool later. The important parts in this XML are the command-line tag **cmdlineapp**, and the parameters tag **parameters**, that we explain next.

The content of the **cmdlineapp** tag is a template that describes how to execute the tool from a command-

line. Here `_ei_files` and `_ei_parameters` are *template parameters* that are replaced, by the server, with appropriate values when receiving a request to execute the corresponding tool. As expected, `_ei_files` should be replaced by a list of file names that the client passes to the tool to process, and `_ei_parameters` by a list of parameters that the client passes to the tool. The idea of using command-line templates is very convenient, because any information that the server wants to pass to a tool can be encoded in such a template, and, moreover, the template can indicate which information a tool is interested in. For example, if the server maintains sessions for the different connected clients, and the tool is interested in this information, it can use a corresponding template parameter `_ei_sessionid`.

The content of the **parameters** tag includes a list of parameters that are accepted by the tool. For example, in the above snippet, there is a parameter called "`c`" that can take one of the values 1 or 2. The `prefix` attribute can be used to indicate how the parameter is translated when passed in the command-line, e.g., one tool might require it as `-c` and another as `--c`. The `check` attribute indicates if the server should check the validity of the provided values for the different parameters, and if they are not valid to reject the client's request. Support for different kinds of parameters is provided, e.g., one value out of many, several values out of many, Boolean parameters (i.e., either they appear or not), free text, etc.

To summarize, the work-flow of the server when receiving a request to execute a tool, with some values for the parameters, is as follows: it replaces the template parameters with corresponding values; it executes the corresponding command-line; and finally forwards back the output to the client. Apart from this work-flow, we can think of the following variants:

(*i*) in some tools, generating the output might take long time, and thus we want to disconnect the connection with the client and provide a key that can be used to fetch the (partial) output when it is ready; and

(*ii*) in some tools, the generated output is very large or even not in a text format, and thus it is convenient to provide the client with a link through which this output can be fetched instead of sending it immediately.

The EasyInterface server supporting these variants through the *steaming* and *download* features (see the user manual in Appendix A).

3.3.2 Communicating with the Server

Our EasyInterface server is implemented as a collection of PHP scripts on top of an HTTP server, thus the communication is done using the HTTP POST protocol where the content of the actual request is given in JSON. For example, the following snippet corresponds to such request:

```
{
  command:  "execute",
  app_id:  "myapp",
  parameters: {
     c:  ["1"],
     ...
  },
  ...
}
```

This request indicates that we want to execute the tool identified by `myapp`, passing it some parameters as indicated in the `parameters` field, etc. The `command` field supports, apart from executing a tool, other services, e.g., fetching the list of available tools and example, etc.

3.3.3 Example Sets

Research prototype tools typically come with a default set of examples from which the user can start. The server side provides a mechanism that allows developers to specify such sets of examples, and allows clients to fetch them as well. Specifying such a sets is using an XML structure that represent a directory, where each entity that represents a file has a URL to its actual content as well. In addition, URLs to public GitHub repositories are supported.

3.3.4 Security Issues

There are different security issues that are considered at the server side, among them are the following:

- It guarantees that the client cannot manipulate the server, and make it execute a local program that is not supposed to be executed. The use of command-line templates opens a door for such attacks, and thus before executing the resulting command-line the server checks that it actually executes the desired program and nothing else.
- The server provides a way to control the resources that can be consumed by a tool once it is executed, e.g., maximum cputime.

Our EasyInterface server explicitly address the above issues.

3.4 The Client Side

Although it is now relatively easy to execute tools on the server side, as services, users would not be willing to send requests in the above format in order to execute a tool on their own input. Thus, our aim is to simplify this process further by providing graphical user interfaces that:

(*i*) connect to EasyInterface servers and ask for the list of available tools;

(*ii*) allow the user to select a tool to execute and set the values of the corresponding parameters;

(*iii*) generate a corresponding request and send it to the corresponding EasyInterface server; and

(*iv*) show the returned output to the user.

The EasyInterface toolkit provides such client which is in addition web-based, as depicted in Figure 3.2. This client has the following components with some associated functionalists:

(*i*) `Code Editor`: an area were programs can be edited. It supports editing different programming languages easily, and support features such as *search and replace* and *syntax highlighting*;

(*ii*) `File Manager`: an area that allows accessing different sets of examples, and also creating new programs. In addition, it provides a mechanism to save and load programs via GitHub repositories;

(*iii*) `Outline`: an area where the elements of the edited programs, such as class and method names, are shown. It is configurable to allow generating the outline for different programming languages; and

(*iv*) `Console`: an area where the output of a tool can be printed.

In addition, the web-client should includes a settings section were the parameters of the different tools (that are shown graphically) can be set. The web-client is configurable to include specific sets of tools and examples, from one or more EasyInterface servers.

Note that, by default, the output of a tool is shown in the `Console` area, unless it uses the EasyInterface output language (see next Section) in which case it is passed through a corresponding interpreter to convert it to graphical widgets. This interpreter is an essential part of the web-client.

3.5 The EasyInterface Output Language

Since sophisticated clients that provide a development environment, such as the web-client, are GUI based developing environments, the EasyInterface toolkit provides tools with an easy way to view their output as widgets in the corresponding environment (see Objective O3). This is done in a generic way, i.e., in principle, it should take effect in all such environments, without changing the tool to produce specific output for each one. Thus, our output language t abstracts away from the specific environment, and, in addition, it is text-based and does not require any knowledge on WEB or GUI programming. The following is a snippet of such output:

```
<highlightlines dest="/Examples_1/iterative/sum.s">
  <lines> <line from="5" to="10"/> </lines>
</highlightlines>
```

This indicates that lines 5–10 of the file `/Examples_1/iterative/sum.s` (which is opened in the editor) should be highlighted. The language also provides commands for interaction with the user such as:

```
<oncodelineclick dest="/Examples_1/iterative/sum.c" outclass="info" >
  <lines><line from="17" /></lines>
  <eicommands>
    <dialogbox boxtitle="Hey!">
      <content format="text">
        Click on the marker again to close this window
      </content>
    </dialogbox>
  </eicommands>
</oncodelineclick>
```

This indicates that when clicking on line 17, a dialog-box with a corresponding message should be opened. The language supports commands according to the needs of the tools to which we restrict ourselves (see Section 3.1), but it is easily extensible to meet the needs of new tools in the future.

3.6 Web-site of the Envisage Virtual Collaboratory

A version of EasyInterface that is deployed as the Envisage Virtual Collaboratory is available at the following address: `http://abs-models.org/laboratory`. It currently includes the following tools:

- Resource Analysis (SACO) – by UCM, it includes also the CoFloCo backend by TUD
- Resource Analysis (SRA) – by BOL
- May-Happen-in-Parallel Analysis – it is a crucial analysis of the all other analyses of SACO (by UCM), and, moreover, it is very useful for analysis of concurrent program in general, so we thought it would be useful to include it as a separate tool
- Deadlock Analysis (SACO) – by UCM
- Deadlock Analysis (DSA) – by BOL
- ABS Smart Deployer – by BOL and UIO
- ABS ErLang Simulator – by UIO
- ABS-Haskell Compiler Simulator – by CWI

- ABS Syntax/Type Checker – by UIO
- Test-Case Generation (aPET) – by UCM
- Systematic Testing (SYCO) – by UCM

In addition, the GitHub repository `http://github.com/abstools/absexamples` includes ABS examples that are automatically imported into the file-manager area of the web-client.

3.7 Source Code

The source code of EasyInterface is publicly available at the following GitHub repository: `http://github.com/abstools/easyinterface`.

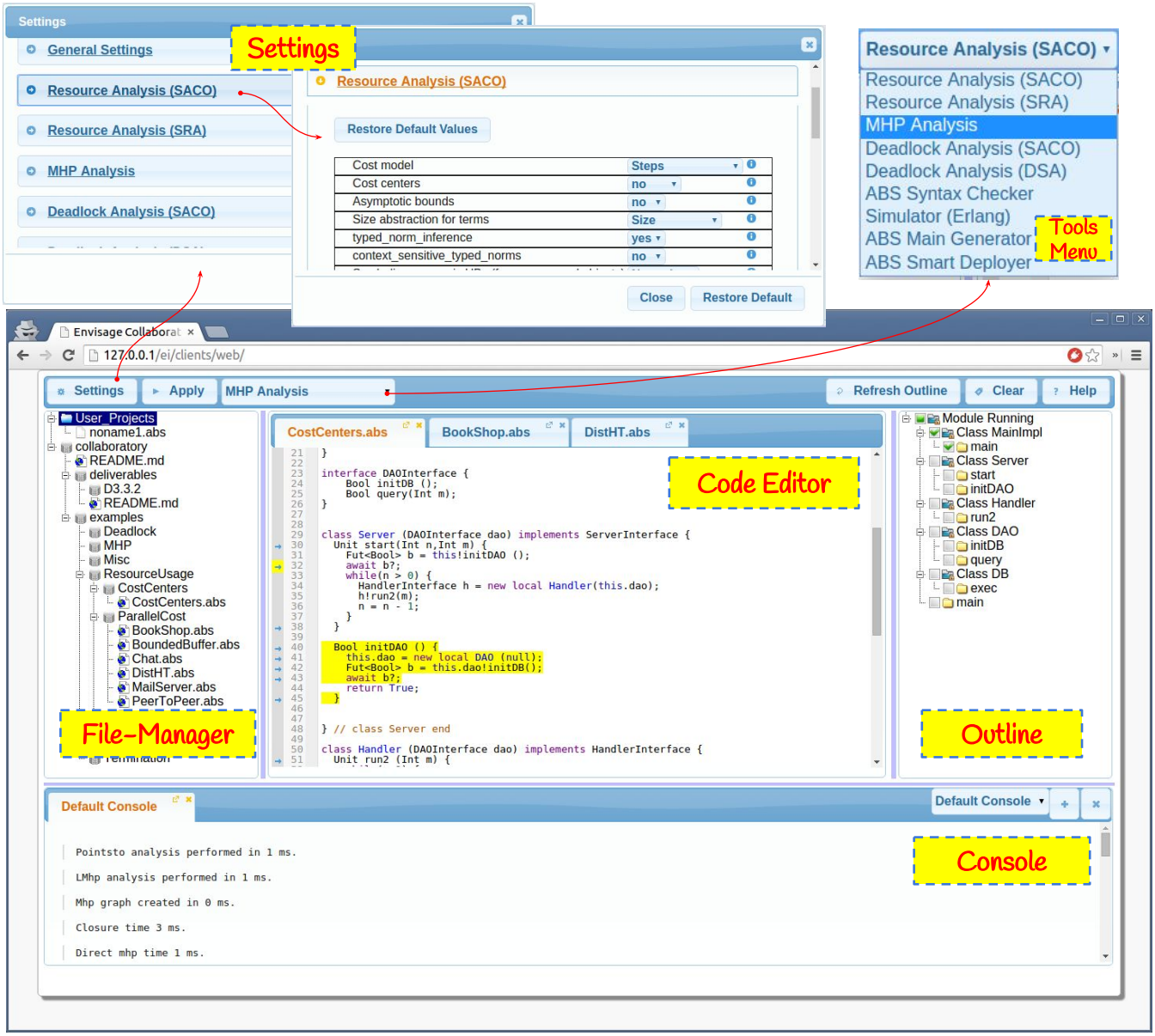

Figure 3.2: EasyInterface web-client

Chapter 4

Conclusions

During the Envisage project, the main effort of Task T5.2 has been directed to developing EasyInterface framework, which is the central component of the Envisage Virtual Collaboratory. In addition, some effort has been directed to writing documentation and tutorials for the different tools, and make them available in the `http://www.abs-models.org`.

EasyInterface is now mature enough, and includes a comprehensive user manual, which allowed Envisage's developers to integrate their tools in the collaboratory and test them through the web-client. Out plan is to keep maintaining EasyInterface as an open-source toolkit beyond Envisage, in particular to use in other projects in which the different partners are involved.

Glossary

GitHub is a powerful collaboration, code review, and code management for open source and private projects – `https://github.com`

JavaScript is a high level, dynamic, untyped, and interpreted programming language, that is commonly used for web programming – `https://en.wikipedia.org/wiki/JavaScript`

JSON (JavaScript Object Notation) is a lightweight data-interchange format – `http://www.json.org`

PHP is a server-side scripting language designed for web development but also used as a general-purpose programming language – `http://www.php.net/`

POST (HTTP) is one of many request methods supported by the HTTP protocol used by the World Wide Web – `https://en.wikipedia.org/wiki/POST_(HTTP)`

Appendix A

EasyInterface User Manual

In this appendix we attach the user manual of EasyInterface.

EasyInterface User Manual

http://github.com/abstools/easyinterface

Jesús Doménech
Samir Genaim

http://www.envisage-project.eu/

Abstract

During the lifetime of a research project, different partners develop several research prototype tools that share many common aspects. This is equally true for researchers as individuals and as groups: during a period of time they often develop several related tools to pursue a specific research line. Making research prototype tools easily accessible to the community is of utmost importance to promote the corresponding research, get feedback, and increase the tools' lifetime beyond the duration of a specific project. One way to achieve this is to build graphical user interfaces (GUIs) that facilitate trying tools; in particular, with web-interfaces one avoids the overhead of downloading and installing the tools.

Building GUIs from scratch is a tedious task, in particular for web-interfaces, and thus it typically gets low priority when developing a research prototype. Often we opt for copying the GUI of one tool and modifying it to fit the needs of a new related tool. Apart from code duplication, these tools will "live" separately, even though we might benefit from having them all in a common environment since they are related.

This work aims at simplifying the process of building GUIs for research prototypes tools. In particular, we present EasyInterface, a toolkit that is based on novel methodology that provides an easy way to make research prototype tools available via common different environments such as a web-interface, within Eclipse, etc. It includes a novel text-based output language that allows to present results graphically without requiring any knowledge in GUI/Web programming. For example, an output of a tool could be (a structured version of) "*highlight line number 10 of file ex.c*" and "*when the user clicks on line 10, open a dialog box with the text ...*". The environment will interpret this output and converts it to corresponding visual effects. The advantage of using this approach is that it will be interpreted equally by all environments of EasyInterface, e.g., the web-interface, the Eclipse plugin, etc.

EasyInterface has been developed in the context of the Envisage [5] project, and has been evaluated on tools developed in this project, which include static analyzers, test-case generators, compilers, simulators, etc. EasyInterface is open source and available at GitHub[1].

Keywords

Generic User Interfaces, Web Programming.

[1] `http://github.com/abstools/easyinterface`

Preface

How to Read this User Manual

Start with Chapter 2 in order to understand the overall architecture of the EasyInterface framework and the role of each component. Next read Chapter 3 and implement all the steps of the incremental example, after which you will probably have enough knowledge to integrate your own applications without further reading.

Contents

Chapter 1

Introduction

During the lifetime of a research project, different partners typically develop several research prototype tools that share many common aspects. For example, in the ENVISAGE [5] project, in the context of which this work was developed, several tools for processing ABS programs [32] have been developed: static analyzers, test-case generators, compilers, simulators, etc. This observation is equally true for researchers as individuals and as groups: during a period of time they develop several related tools to pursue a specific research line. For example, in the COSTA[1] group, we have developed many program analysis and test-case generation tools for several programming languages and abstract models.

Making research prototype tools available to the corresponding research communities is of utmost importance for several reasons: to promote the corresponding research; to make the community aware of the corresponding project or research group; to get valuable feedback; to increase the tools' lifetime beyond the duration of a specific project; etc. However, making tools available is not enough, they should also be easy to use since otherwise users would avoid using them, in particular if they require some technical skills to install or use them. One way to achieve this is to build graphical user interfaces (GUIs) that facilitate using the tools; in particular, web-interfaces in order to avoid the overhead/risk of downloading and installing the tools locally.

Building GUIs is a tedious task, in particular if they have to be developed from scratch. Building web-interfaces is even more difficult since web-based applications are typically more difficult to debug. Thus, the task of building GUIs for research prototype tools typically gets a lower priority in the development process, and the effort is instead directed to improving the functionality of the tools, rather than their corresponding GUIs. In addition, due to these difficulties, researchers often opt for copying the GUI of one tool and modifying it to fit the needs of a new related tool. Apart from code duplication, these tools will "live" separately, even though they might benefit from having them all in a common environment since they are related and, for example, a user who is interested in one tool will be easily exposed to other related tools when accessing the common environment.

Clearly many of the difficulties, and the corresponding effort, in building GUIs are unavoidable in general. This is because (*i*) tools produce different outputs that are presented graphically in different ways to the user; and (*ii*) the input — and the interaction with the user — is different from one tool to another. However, if we consider a set of related tools, e.g., those developed in a given research project, it is

[1] `http://costa.ls.fi.upm.es`

easy to identify many common aspects in their input and output. These common aspects can be then used as a bases for building a GUI construction toolkit that provide (*i*) an easy way to integrate tools in a common environment; and (*ii*) an easy way to present the output using a predefined set of graphical widgets that cover the common aspects of the output. Despite of the fact that such a toolkit is limited when compared to general GUI libraries, in the sense that it can be mostly used for tools that fit in the identified common aspects, researches would prefer this approach if it extremely simplifies the task of building a GUI for a new tool, e.g., if it allows building such a GUI in few hours and without the need for deep modifications to the corresponding code. Building such a toolkit is the main objective of this work.

1.1 Objectives

The main objective of this work is to develop a toolkit that can be used to easily develop GUIs for research prototype tools, and, moreover, integrating them in a common environment. We focus on tools that have the following common aspects:

- A request to a tool corresponds to executing a program from a command-line, where the input is passed as command-line parameters (including file names) and the output is printed on the standard output. Note that this does not limit us to command-line tools, since, for example, if the tool runs as a server we can write a command-line wrapper that forwards the requests to the server and prints the server's response on the standard output.

- The tools are, in principle, supposed to process programs, e.g., static analysis tools. Thus, they receive as parameters (*i*) file names that represent programs; and (*ii*) possibly a list of program entities (e.g., method names, class names) which indicate some information to the tool (e.g., for static analysis tools it might indicate where to start the analysis from).

- The output of the tool mainly includes information that is related to some parts of the input program. For example, associating information to a program line, drawing graphs that represent some property of the program such as resource consumption, etc.

Note that the above aspects where not chosen arbitrarily, but they rather cover tools developed in the Envisage [5] project, in the context of which this work was developed, and the tools (and research lines) of the Costa group to which the author of this work belongs. Later, the reader will see that our toolkit is not actually limited to tools that satisfy the above conditions, but rather it is more general.

As regards the capabilities of the toolkit that we want to develop, we aim at developing one that complies with the following objectives:

O1 Using the toolkit for developing *simple* GUIs for existing tools, and integrating them in a common environment, should not take more than few minutes, and, moreover, *without* requiring any modification to the tool's code at all.

O2 Developing more sophisticated GUIs might require mild modifications to the tool's code, and by no mean deep modifications — requiring deep modifications would make it less likely that the toolkit will be used.

O3 Using the toolkit for presenting the output graphically should not require any knowledge on GUI or WEB programming. The user should describe the output in a natural language, e.g., "*highlight line number 10 of file ex.c*" (or a structured version of such statement, e.g., using XML or JSON, to be able to parse it easily).

O4 The toolkit must provide common environments in which tools can be integrated. The most important environment is the web-based one (i.e., it runs in a web browser). In addition, the design should allow developing more environments in the future without modifying the toolkit or existing integrated tools.

O5 The common environments should be completely transparent to the integrated tools. This means that the work for integrating the tool or modifying it to produce graphical output should be done only once, and it should work in all environments equally (including future ones).

O6 The toolkit should be secure, it should not pose any security risks both to developers and users of to the corresponding tools.

To evaluate the developed toolkit, it should be used to build GUIs for the tools of the Envisage project, and, moreover, integrating them in a common environment.

1.2 Contributions

In this work we have developed EasyInterface, a toolkit for easily building GUIs for research prototype tools that comply with the objectives and requirements stated in Section 1.1. In particular, we developed a new methodology for building GUIs for research prototype tools that consists of the following:

- A server side where tools are installed. Adding a tool to the server is done by adding a configuration file (which is very easy to write) describing how to run it, etc.
- A protocol that allows connecting to the server for, among many other things that we will see later, executing a tool on a particular input and getting back the result. In some sense, this allows converting the tools installed on the server to services.
- A web-based development environment that allows users to use the tools installed on an EasyInterface servers transparently, in addition to the functionality of a normal development environment.
- An output text-based language that can be used to describe how the output should be shown graphically, and corresponding interpreters in the development environments to convert these descriptions to graphical effects.
- An extensive evaluation by using EasyInterface to integrate the tools of the Envisage project in a common development environment.

EasyInterface is open source and available at GitHub[2].

[2] `http://github.com/abstools/easyinterface`

1.3 The Structure of This Document

This document is structured as follows. In Chapter 2 we describe the overall architecture of EasyInterface and how its different components interact. In Chapter 3 we describe how to integrate a new tool in EasyInterface in details. The purpose of this chapter is to allow the reader to briefly get familiar with the concrete details before moving on to the next chapters. In Chapter 4 we describe the EasyInterface server specifications, and a corresponding implementation. In Chapter 5 we describe the available clients (web-interface, etc). In Chapter 6 we describe the syntax and semantics of the EasyInterface output language. In Chapter 7 we describe the evaluation that we have done in the context of the Envisage project. Finally, in Chapter 8, we conclude and discuss future and related work. In addition, we include the installation guide of EasyInterface as Appendix A.

Chapter 2

The EasyInterface Toolkit Architecture

In this chapter we describe the overall design of EasyInterface, in particular we define its different components, their corresponding requirements and responsibilities, and how they communicate in order to accomplish the objectives described in Section 1.1. Note that the description of each EasyInterface component is kept general in this chapter, however, in some cases we give partial specifications to convey the underlying ideas. In chapters 4-6, we suggest such complete specifications, and corresponding implementations, for each component.

2.1 The Overall Architecture of EasyInterface

In this section we describe the overall design of EasyInterface, which is driven by the objectives we have stated in Section 1.1. One of the most important objectives is O5, which aims at reducing the effort of integrating a tool in a common environment, namely, it aims at achieving a situation where a tool can be installed once, and then appears automatically in all available environments. This objective leads us to an architecture with two abstract components:

(i) the first abstract component represents an entity where tools can be installed; and

(ii) the second abstract component represents clients (e.g., development environments) that can communicate with the previous component to consult the list of installed tools, to execute a tool, etc.

This abstraction leads us to a client-server architecture as depicted in Figure 2.1, where the *server side* represents the first abstract component, and the *client side* represents the second abstract component.

The *server side* is intended to be a machine with several tools (the circles `Tool1`, `Tool2`, etc., in Figure 2.1) that can be executed from a command-line, and their output goes to the standard output (recall that our objectives are restricted to such tools – see Section 1.1). These are the tools that we want to make available for the outside world, i.e., execute them as services on the internet. The *client side* includes several clients that make it easy, to a user, to communicate with the server side to execute a tool, etc. Clients can be simple, e.g., a program that sends a very specific request, or more sophisticated development environments such as the web-based one that is included in Objective O4. Note that although Figure 2.1 includes only one server component,

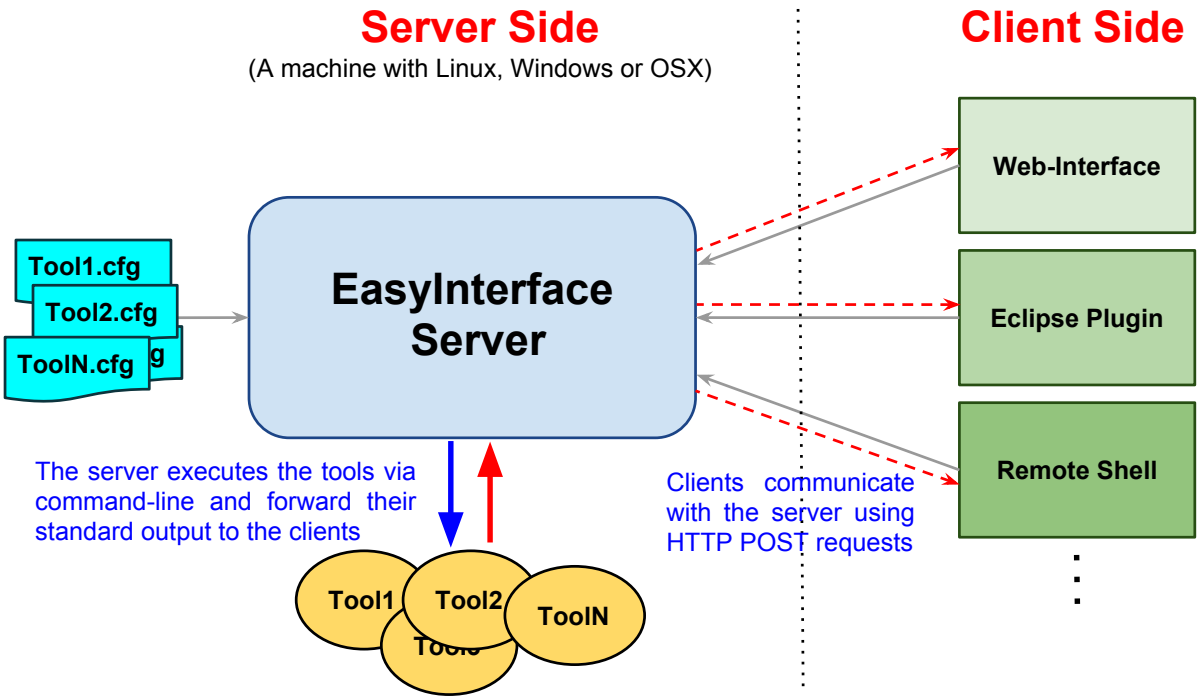

Figure 2.1: The Architecture of the EASYINTERFACE Toolkit

the overall setting should allow several ones, and clients should be configurable to connect to one or more servers.

The rest of this chapter is organized as follows. In Section 2.2 we explain the details of the server side; in Section 2.3 we explain the details of the client side, and finally, in Section 2.4, we explain how tools can produce output that is shown graphically in the client side (see Objective O3).

2.2 The Server Side

The problem that we want to solve at the server side can be summarized as follows:

> Provide a uniform way for remotely accessing locally installed tools as services, such that installing new tools and accessing them is straightforward.

However, this problem encapsulates different issues. In what follows, we address these issues and draw general guidelines to corresponding design problems. These guidelines are then used in Chapter 4 when developing the specifications of the EASYINTERFACE server.

2.2.1 Installing a New Tool

Developers (of research prototype tools) should be able to make their tools available on the server in a straightforward way, in terms of simple configuration files (`Tool1.cfg`, `Tool2.cfg`, etc., in Figure 2.1). The only information they should provide is how the tool can be executed from a command-line, and which parameters it takes. Information about the parameters is crucial as clients will use it to ask users to select the desired

values, etc. The following XML snippet suggests a format for a tool configuration (that we actually adapt and extend in Chapter 4):

```
<app id="myapp">
  ...
  <execinfo>
    <cmdlineapp>/path-to/myapp.sh _ei_files _ei_parameters</cmdlineapp>
  </execinfo>
  <parameters prefix = "-" check="true">
    <selectone name="c">
      <option value="1" />
      <option value="2" />
    </selectone>
    ...
  </parameters>
</app>
```

This XML defines a tool that has a unique identifier **myapp**, which is used to refer to this tool later. The important parts in this XML are the command-line tag **cmdlineapp**, and the parameters tag **parameters**, that we explain next.

The content of the **cmdlineapp** tag is a template that describes how to execute the tool from a command-line. Here **_ei_files** and **_ei_parameters** are *template parameters* that should be replaced, by the server, with appropriate values when receiving a request to execute the corresponding tool. As expected, **_ei_files** should be replaced by a list of file names that the client passes to the tool to process, and **_ei_parameters** by a list of parameters that the client passes to the tool. The idea of using command-line templates is very convenient, because any information that the server wants to pass to a tool can be encoded in such a template, and, moreover, the template can indicate which information a tool is interested in. For example, if the server maintains sessions for the different connected clients, and the tool is interested in this information, it can use a corresponding template parameter **_ei_sessionid**.

The content of the **parameters** tag includes a list of parameters that are accepted by the tool. For example, in the above snippet, there is a parameter called "c" that can take one of the values 1 or 2. The **prefix** attribute can be used to indicate how the parameter is translated when passed in the command-line, e.g., one tool might require it as **-c** and another as **--c**. The **check** attribute indicates if the server should check the validity of the provided values for the different parameters, and if they are not valid to reject the client's request. Support for different kinds of parameters should be provided, e.g., one value out of many, several values out of many, Boolean parameters (i.e., either they appear or not), free text, etc.

To summarize, the work-flow of the server when receiving a request to execute a tool, with some values for the parameters, is as follows: it replaces the template parameters with corresponding values; it executes the corresponding command-line; and finally forwards back the output to the client. Apart from this work-flow, we can think of the following variants:

(*i*) in some tools, generating the output might take long time, and thus we want to disconnect the connection with the client and provide a key that can be used to fetch the (partial) output when it is ready; and

(*ii*) in some tools, the generated output is very large or even not in a text format, and thus it is convenient to provide the client with a link through which this output can be fetched instead of sending it immediately.

The server should provide a mechanism for supporting such variants.

2.2.2 Communicating with the Server

Clients should be able to communicate with the server using the HTTP POST protocol [27]. The advantage of using this protocol is that one can build the EASYINTERFACE server on top of an HTTP server, and thus take advantage of the underlying machinery for serving clients concurrently. In addition, if one is not interested in building the EASYINTERFACE server on top of an HTTP server, there are numerous libraries for HTTP POST communication that one can use (this is true for the client side as well).

Apart from HTTP POST, the content of the actual request should also be in a standard structured format, e.g., JSON [6], to facilitate their processing both at the server and client sides. For example, the following snippet suggests a format for such requests:

```
{
  command:  "execute",
  app_id:  "myapp",
  parameters: {
     c:  ["1"],
     ...
  },
  ...
}
```

This request indicates that we want to execute the tool identified by **myapp**, passing it some parameters as indicated in the **parameters** field, etc. The **command** field should support, apart from executing a tool, other services, e.g., fetching the list of available tools, depending on the services that are provided by the server (according to the needs defined in the next sections).

2.2.3 Example Sets

Research prototype tools typically come with a default set of examples from which the user can start. The server side should provide a mechanism that allows developers to specify such sets of examples, and allows clients to fetch them as well. Specifying such a sets can be done, for example, using an XML structure that represent a directory, where each entity that represents a file has a URL to its actual content as well. In addition, URLs to public repositories such as GitHub should be supported.

2.2.4 Security Issues

There are different security issues [34] that should be considered in the server side, among them are the following:

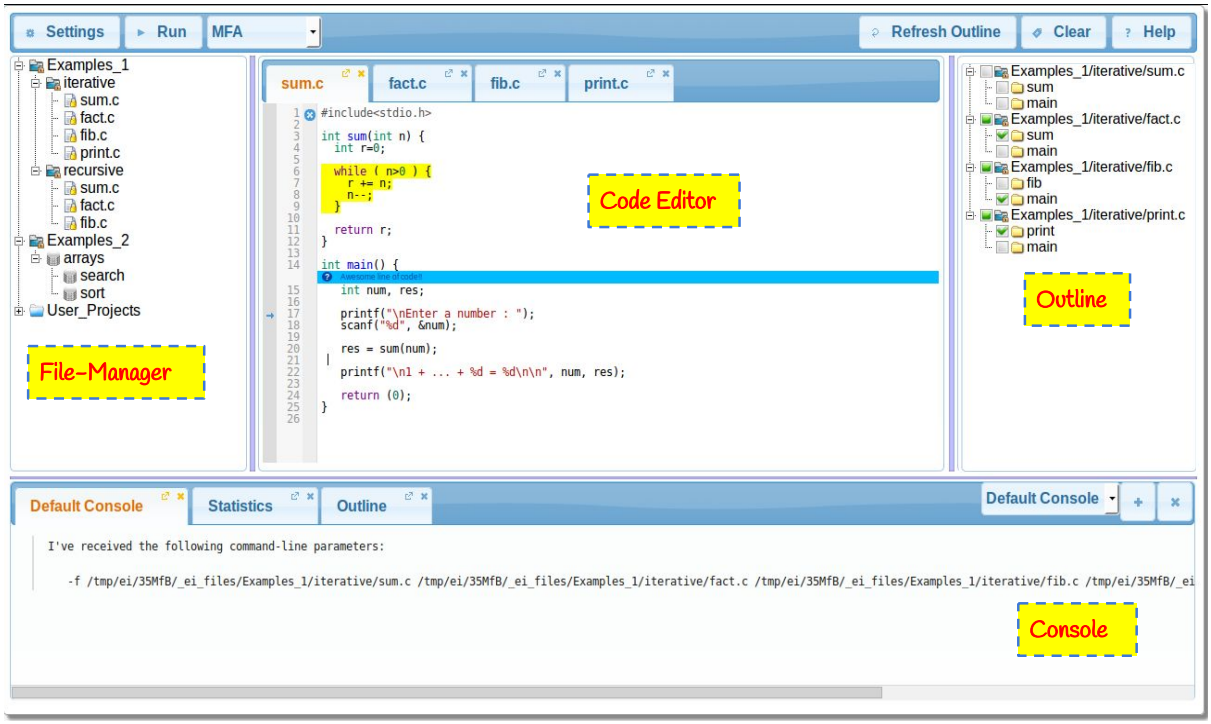

Figure 2.2: EasyInterface client example

- It should be guaranteed that the client cannot manipulate the server, and make it execute a local program that is not supposed to be executed. The use of command-line templates opens a door for such attacks, and thus before executing the resulting command-line the server should guarantee that it actually executes the desired program and nothing else.
- The server must provide a way to control the resources that can be consumed by a tool once it is executed, e.g., maximum cputime.

The server side should explicitly address the above issues, both at the specification and implementation levels.

2.3 The Client Side

Although it is now relatively easy to execute tools on the server side, as services, users would not be willing to send requests in the above format in order to execute a tool on their own input. Thus, our aim is to simplify this process further by providing graphical user interfaces that:

(*i*) connect to EasyInterface servers and ask for the list of available tools;

(*ii*) allow the user to select a tool to execute and set the values of the corresponding parameters;

(*iii*) generate a corresponding request and send it to the corresponding EasyInterface server; and

(*iv*) show the returned output to the user.

The EASYINTERFACE toolkit should provide at least one such client which is in addition web-based, as depicted in Figure 2.2 (it is screenshot of the actual EASYINTERFACE web-client – see Chapter 5). This client should have the following components with some associated functionalists:

(*i*) `Code Editor`: an area were programs can be edited. This should support editing different programming languages easily, and support features such as *search and replace* and *syntax highlighting*;

(*ii*) `File Manager`: an area that allows accessing different sets of examples, and also creating new programs. In addition, it should provide a mechanism to save and load programs, e.g., via GitHub repositories;

(*iii*) `Outline`: an area where the different elements of the edited programs, such as class and method names, are shown. It should be configurable to allow generating the outline for different programming languages; and

(*iv*) `Console`: an area where the output of a tool can be printed.

In addition, this environment should include a settings section were the parameters of the different tools (that are shown graphically) can be set. The web-client should be configurable to include specific sets of tools and examples, from one or more EASYINTERFACE servers.

Note that, by default, the output of a tool should be shown in the `Console` area, unless it uses the EASYINTERFACE output language (see next Section) in which case it should pass through a corresponding interpreter to convert it to graphical widgets. Thus, such an interpreter is expected to be an essential part of this environment.

2.4 The Text-Based GUI Output Language

Since sophisticated clients that provide a development environment, such as the web-client, are GUI based developing environments, the EASYINTERFACE toolkit is required to provide tools with an easy way to view their output as widgets in the corresponding environment (see Objective O3). This should be done in a generic way, i.e., it should take effect in all such environments, without changing the tool to produce specific output for each one. Thus, an output language that abstracts away from the specific environment must be developed, and preferably it should be text-based and does not require any knowledge on WEB or GUI programming. As a possible suggestion, the following is a snippet of such output:

```
<highlightlines dest="/Examples_1/iterative/sum.s">
  <lines> <line from="5" to="10"/> </lines>
</highlightlines>
```

This indicates that lines 5–10 of the file `/Examples_1/iterative/sum.s` (which is opened in the editor) should be highlighted. This language should also provide commands that model interaction with the user such as:

```
<oncodelineclick dest="/Examples_1/iterative/sum.c" outclass="info" >
  <lines><line from="17" /></lines>
  <eicommands>
    <dialogbox boxtitle="Hey!">
      <content format="text">
        Click on the marker again to close this window
      </content>
    </dialogbox>
  </eicommands>
</oncodelineclick>
```

This indicates that when clicking on line 17, a dialog-box with a corresponding message should be opened. The language should support commands according to the needs of the tools to which we restrict ourselves (see Section 1.1), and should be easily extensible to meet the needs of new tools in the future.

Chapter 3

EasyInterface in a Nutshell

The purpose of this chapter is to give the reader the experience of how GUIs can be developed using the EasyInterface toolkit, before moving to the specifications and implementations details of the different components in the next chapters. This, we believe, would help the reader to better understand the details of chapters 4-6.

In the rest of this chapter we develop a simple tool, integrate it in the EasyInterface server, and try it out through the web-client. The presentation in this chapter is incremental, we start with a simple tool and in each step we add more features to demonstrate the different parts of EasyInterface. In our explanation we assume that a Unix based operating system is used, however, we comment on how to do the analog operations on Windows when they are different.

3.1 Getting Started

We assume that EasyInterface is already installed and working, which can be done following the instructions in Appendix A. Let us start by trying some demo tools that are available by default in the web-client. If you visit `http://localhost/ei/clients/web`, you should get a page similar to the one shown in Figure 2.2 (see Page 9). At the top part of this page you can see a button with the label `Run`, and to its right a combo-box with several items `Test-0`, `Test-1`, etc. These items correspond to tools available in the web-client, and we will refer to it as the tools menu. To the left of `Run` there is a button with the label `Settings`, if you click it you will see that each `Test-i` has also some parameters that can be set to some values. Note that, by default, the web-client is configured to connect to the EasyInterface server at `http://localhost/ei/server` and ask for all tools, together with their corresponding parameters, that are available at that server. Note also that tool `Test-i` actually corresponds to the bash-script `server/bin/default/test-i.sh`, and that its corresponding configuration file is `server/config/default/test-i.cfg` (later we will go over the details of these configuration files).

If you select a tool, from the combo-box, and click on `Run`, the web-client sends a request to the server to execute this tool. The request includes also the current values of the parameters (those in the settings section) and the file that is currently active in the code editor area. The server, in turn, executes the corresponding program, i.e., the bash-script `server/bin/default/test-i.sh` in this case, and redirects its output back to the web-client. The web-client will either print this output in the console area, or view it graphically if it uses the EasyInterface output language. Execute the demo

tools just to get an idea on which graphical output we are talking about (e.g., highlight text, markers).

In the rest of this chapter we explain, step by step, how to add a tool to EASYINTERFACE. Note that all files that we create in the next sections are already available in the directory `docs/manual/myapp`, each step in a sub-directory named `step-i` with two files `myapp.cfg` and `myapp.sh`, so instead of creating them you can just copy them to the appropriate places.

3.2 Adding a Tool to EASYINTERFACE

When we add a tool to the EASYINTERFACE server it will automatically appear in the tools menu of the web-client (unless you have changed the configuration of the web-client already!). Let us add a simple "Hello World" tool.

We start by creating a bash-script that represents the executable of our tool (it could be any other executable), and placing it in the directory `server/bin/default` together with the `test-i.sh` scripts, however, this is not obligatory and it can be placed anywhere in the file-system as far as the server has enough permissions to access it. Create a file `myapp.sh` in `server/bin/default` with the following content:

```
1 #!/bin/bash
2
3 echo "Hello World!"
```

As you can see, it is a simple program (bash script) that prints "Hello World!" on the standard output. Later we will see how to pass input to this tool and how to generate more sophisticated output. Change the permissions of `myapp.sh` by executing the following command (on Windows this is typically not needed):

```
> chmod -R 755 myapp.sh
```

Execute `myapp.sh` (in a shell) to make sure that it works correctly before proceeding to the next step.

Next we will configure the server to recognize our tool. Create a file `myapp.cfg` in the directory `server/config/default` with the following content (we could place this file anywhere under `server/config` not necessarily in `default`):

```
<app visible="true">
  <appinfo>
    <acronym>MFA</acronym>
    <title>My First Tool</title>
    <desc>
      <short>A simple EI tool</short>
      <long>A simple tool using the EasyInterface Toolkit</long>
    </desc>
  </appinfo>
  <apphelp>
    <content format='html'>
      This is my first <b>EasyInterface</b> tool!
    </content>
```

```
  </apphelp>
  <execinfo>
    <cmdlineapp>./default/myapp.sh</cmdlineapp>
  </execinfo>
</app>
```

Let us explain the meaning of the different elements of this configuration file. The **app** tag is used to declare an EASYINTERFACE tool, and its **visible** attribute tells the server to list this tool when someone asks for the list of available tools. Changing this value to **false** will make the tool "hidden" so only those who know its identifier can use it. The **appinfo** tag provides general information about the tool, this will be used by the clients to show the tool name, etc. The **apphelp** tag provides some usage information about the tool, or simply provides a link to another page where such information can be found. The actual content goes inside the **content** tag, which is HTML as indicated by the **format** attribute (use 'text' for plain text). The most important part is the **execinfo** tag, which provides information on how to execute the tool. The text inside **cmdlineapp** is interpreted as a command-line *template*, such that when the server is requested to execute the corresponding tool it will simply execute this command-line and redirect its output back to the client. These templates are the same as those suggested in Section 2.2. Note that before executing the script, the server changes the current directory to `server/bin` and thus the command-line can be relative to `server/bin`.

Next we add the above configuration file to the server. This is done by adding the following line to `server/config/default/apps.cfg` (inside the **apps** tag):

```
<app id="myapp" src="default/myapp.cfg" />
```

Here we tell the server that we want to install a tool as defined in **default/myapp.cfg**, and we want to assign it the *unique* identifier **myapp**. This identifier will be mainly used by the server and the clients when they communicate, we are not going to use it anywhere else. Note that the main configuration file of the EASYINTERFACE server is `server/config/eiserver.default.cfg`, and that `default/apps.cfg` is imported into that file.

Let us test our tool. Go back to the web-client and reload the page, you should see a new tool named `MFA` in the tools menu. If you click on the `Help` button you will see the text provided inside the **apphelp** tag above. Select this tool and click on the `Run` button, the message "Hello World!" will be printed in the console area.

3.3 Passing Input Files to a Tool

Tools typically receive input files (e.g., programs) to process. This is why we required the web-client to provide the possibility of creating and editing such files. In this section we explain how to pass these files, via the server, to our tool when the `Run` button is clicked.

When you click on the `Run` button the web-client passes the currently opened file (i.e., the content of the active tab) to the server, and if you use the `Run` option from the context menu of the file-manager (select an element from the files tree-view on the left, and use the mouse right-click to open the context menu) it passes all files in the

corresponding sub-tree. What is left is to tell the server how to pass these files to our tool. Let us assume that `myapp.sh` is prepared to receive input files as follows:

```
> myapp.sh -f file1.c file2.c file3.c
```

In order to tell the server to pass the input files (that were received from the client) to `myapp.sh`, open `myapp.cfg` and change the command-line template, i.e, the content of **cmdlineapp**, to the following:

```
./bin/default/myapp.sh -f _ei_files
```

When the server receives the files from the client, it stores them in a temporary directory, e.g., in `/tmp`, replaces `_ei_files` by the list of their names, and then executes the resulting command-line. It is important to note that only `_ei_files` changes in the above template, the rest remains the same. Thus, the parameter "`-f`" means nothing to the server, we could replace it by anything else or even completely remove it — that depends only on how our tool is programmed to receive input files.

Let us now change `myapp.sh` to process the received files in some way, e.g., to print the number of lines in each file. For this, replace the content of `myapp.sh` by the following:

```
1  #!/bin/bash
2
3  . misc/parse_params.sh
4  files=$(getparam "f")
5
6  echo "I've received the following command-line parameters:"
7  echo ""
8  echo "  $@"
9
10 echo ""
11 echo "File statistics:"
12 echo ""
13 for f in $files
14 do
15    echo " - $f has " `wc -l $f | awk '{print $1}'` "lines"
16 done
```

Let us explain the above code. At line 3 we executes an external bash-script to parse the command-line parameters, the details are not important and all you should know is that line 4 stores the list of files (that appear after `-f`) in the variable `files`. Lines 6-8 print the command-line parameters, just to give you an idea how the server called `myapp.sh`, and the loop at lines 13-16 traverses the list of files and prints the number of lines in each one.

Let us test our tool. First run `myapp.sh` from a shell passing it some existing text files, just to check that it works correctly. Then go back to the web-client, reload the page, select `MFA` from the tools menu, open a file from the file-manager, and finally click the `Run` button. Alternatively, you can also select an entry from the file-manager and choose `Run` from its context menu, in this case all files in the sub-tree will be passed to `myapp.sh`. You should see the output of the tool in the console area.

3.4 Passing Outline Entities to a Tool

In the web-client, the area on the right is called the outline area (see Figure 2.2 on Page 9). Since EASYINTERFACE was designed mainly for tools that process programs, e.g., program analysis tools, this area is typically dedicated for a tree-view of program entities, e.g., method and class names. The idea is that, in addition to the input files, the user will select some of these entities to indicate, for example, where the analysis should start from or which parts of the program to analyze, etc. Next we explain how we can pass these selected entities to a tool.

By default the web-client is configured to work with C programs, and thus if you open such a program (from the file-manager) and then click on the `Refresh Outline` button, you will get a tree-view of this program entities, e.g., method names (if you use `Refresh Outline` from the context menu in the file-manager you will get a tree-view of program entities for all files in the sub-tree). Note that to generate this tree-view the web-client actually executes a "hidden" tool that is installed on the server, namely `server/bin/default/coutline.sh`, but this is not relevant to our discussion now (see Section 5.1.3 for more details). Note also that `coutline.sh` is limited and will not work perfectly for any C program: it simply looks for lines that start with `int` or `void` followed by something of the form `name(...)`. This script is provided just to explain how a tool that generates an outline is connected to the web-client (see Section 5.1.3 for more details).

As in the case of input files, the web-client always passes the selected entities to the server when the `Run` button is clicked, and it is our responsibility to indicate how these entities should be passed to our tool. Let us assume that `myapp.sh` is prepared to receive entities using the parameter "`-e`" as follows:

```
> myapp.sh -f file1.c file2.c file3.c -e sum.c:main sum.c:sum
```

In order to tell the server to pass the entities (that were received from the client) to our tool, open `myapp.cfg` and change the command-line template, i.e., the content of **cmdlineapp**, to the following:

```
./bin/myapp.sh -f _ei_files -e _ei_outline
```

As in the case of files, before executing the above command-line the server will replace `_ei_outline` by the list of received entities. Let us now change `myapp.sh` to process these entities in some way, e.g., printing them on the standard output. Open `myapp.sh` and add the following lines at the end:

```
1 entities=$(getparam "e")
2
3 echo ""
4 echo "Selected entities:"
5 echo ""
6 for e in $entities
7 do
8    echo "- $e"
9 done
```

This code simply prints the entities in separated lines. Again, the first line stores the list of entities in the variable **entities**.

First run myapp.sh from a shell passing it some existing text files and entities, just to check that it works correctly. Then go back to the web-client, reload the page, select some files, refresh the outline, select some entities, and finally execute the MFA tool to see the result of the last changes.

3.5 Passing Parameters to a Tool

In addition to input files and outline entities, real tools receive other parameters to control different aspects. In this section we explain how to declare parameters in the EasyInterface toolkit such that (*i*) they automatically appear in the web-client (or any other client) so the user can set their values; and (*ii*) the selected values are passed to the tool when executed.

Let us start by modifying myapp.sh to accept some command-line parameters: we add a parameter "-s" to indicate if the received outline entities should be printed; and "-c W" that takes a value W to indicate what to count in each file — here W can be "lines", "words" or "chars". For example, myapp.sh could then be invoked as follows:

```
> myapp.sh -f file1.c file2.c file3.c -e sum.c:main sum.c:sum -s -c words
```

To support these parameters, change the content of myapp.sh to the following:

```
#!/bin/bash

. misc/parse_params.sh
files=$(getparam "f")
entities=$(getparam "e")
whattocount=$(getparam "w")
showoutline=$(getparam "s")

echo "I've received the following command-line parameters:"
echo ""
echo "  $@"

echo ""
echo "File statistics:"
echo ""

case $whattocount in
    lines) wcparam="-l"
    ;;
    words) wcparam="-w"
    ;;
    chars) wcparam="-m"
    ;;
esac

```

```
26 for f in $files
27 do
28     echo " - $f has " `wc $wcparam $f | awk '{print $1}'` $whattocount
29 done
30
31 if [ $showoutline == "" ]; then
32     echo ""
33     echo "Selected entities:"
34     echo ""
35     for e in $entities
36     do
37        echo "- $e"
38     done
39 fi
```

Compared to the previous script, you can notice that: we added lines 17-24 to take the value of "-c" into account when calling wc at Line 28; and in lines 31-39 we wrapped the loop that prints the outline entities with a condition to account for the "-s" parameter.

Our goal is to show these parameters in the web-client (or any other client), so the user can select the appropriate values before executing the tool. The EASYINTERFACE toolkit provides an easy way to do this, all we have to do is to modify myapp.cfg to include a description of the supported parameters. Open myapp.cfg and add the following inside the app tag (e.g., immediately after closing the execinfo tag):

```
<parameters prefix = "-" check="false">
  <selectone name="c">
    <desc>
      <short>What to count</short>
      <long>What you want to count in each input file</long>
    </desc>
    <option value="lines">
      <desc>
        <short>Lines</short>
          <long>Count lines</long>
      </desc>
    </option>
    <option value="words">
      <desc>
        <short>Words</short>
        <long>Count words</long>
        </desc>
    </option>
    <option value="chars" >
      <desc>
        <short>Chars</short>
        <long>Count characters</long>
      </desc>
    </option>
    <default value="lines"/>
  </selectone>
  <flag name="s">
```

```
      <desc>
        <short>Show outline</short>
        <long>Show the selected outline entities</long>
      </desc>
      <default value="false"/>
    </flag>
  </parameters>
```

Let us explain the different elements of the above XML snippet. The tag **parameters** includes the definition of all parameters. The attribute **prefix** is used to specify the symbol to be attached to the parameter name when passed to the tool, for example, if we declare a parameter with name "**c**" the server will pass it to the tool as "**-c**". Note that this attribute can be overridden by each parameter. The attribute **check** tells the server to check the correctness of the parameters before passing them to the tool, i.e., that they have valid values, etc. The tag **selectone** defines a parameter with **name** "**c**" that can take one value from a set of possible ones. For example, the web-client will view it as a combo-box. The **desc** tag contains a text describing this parameter and is used by the client when viewing this parameter graphically. The **option** tags define the valid values for this parameter, from which one can be selected, and the **default** tag defines the default value. The **desc** tag of each **option** contains a text describing this option, e.g., the **short** description is used for the text in the corresponding combo-box. The tag **flag** defines a parameter with name "**s**". This parameter has no value, it is either provided in the command-line or not, and its **default** value is **false**, i.e., not provided. For the complete set of parameters supported in EASYINTERFACE see the specifications of [**PARAMETERS**] in Chapter 4.

Go to the web-client, reload the page, and click on the `Settings` button and look for the tab with the title `MFA`. You will now see the parameters declared above in a graphical way where you can set their values as well. When you click on the `Run` button, the web-client will pass these parameters to the server, however, we still have to tell the server how to pass these parameters to `myapp.sh`. Open `myapp.cfg` and change the command-line template, i.e., content of **cmdlineapp**, to the following:

```
./bin/myapp.sh -f _ei_files -e _ei_outline _ei_parameters
```

As in the case of `_ei_files` and `_ei_outline`, the server will replace `_ei_parameters` by the list of received parameters before executing the command-line. Execute the `MFA` tool from the web-client with different values for the parameters to see how the output changes.

3.6 Using the EASYINTERFACE Output Language

In the example that we have developed so far, the web-client simply printed the output of `myapp.sh` in the console area. This is the default behavior of the web-client if the output does not follow the EASYINTERFACE Output Language, which is a text-based language that allows generating more sophisticated output such as highlighting lines, adding markers, etc. In this section we will explain the basics of this language by extending `myapp.sh` to use it – for more details see Chapter 6.

An output in the EASYINTERFACE output language is an XML structure that has the following form:

```
<eiout>
 <eicommands>
    [EICOMMAND]*
 </eicommands>
 <eiactions>
    [EIACTION]*
 </eiactions>
</eiout>
```

where (*i*) **eiout** is the outermost tag that includes all the output elements; (*ii*) [EICOMMAND]* is a list of commands to be executed; and (*iii*) [EIACTION]* is a list of actions to be declared. An [EICOMMAND] is an instruction like: *print a text on the console, highlight lines 5-10, add marker at line 5*, etc. An [EIACTION] is an instructions like: *when the user clicks on line 13, highlight lines 20-25*, etc. In the rest of this section we discuss some commands and actions that are supported in the EASYINTERFACE output language, for the complete specifications see Chapter 6.

3.6.1 Printing in the Console Area

Recall that when the EASYINTERFACE output language is used, the web-client does not redirect the output to the console area, and thus we need a command to print in the console area. The following is an example of a command that prints "Hello World" in the console area:

```
<printonconsole consoleid="1" consoletitle="Welcome">
  <content format="text">
    Hello World
  </content>
</printonconsole>
```

The value of the `consoleid` attribute is the console identifier in which the given text should be printed (e.g., in the web-client the console area has several tabs, so the identifier refers to one of those tabs). If a console with such identifier does not exist yet, a new one, with a title as specified in `consoletitle`, is created. If `consoleid` is not given the output goes to the default console. Inside **printonconsole** we can have several **content** tags which include the content to be printed (in the above example we have only one). The attribute `format` indicates the format of the content. In the above example it is plain 'text', other formats are supported as well, e.g., 'html' and 'svg'.

Let us change `myapp.sh` to print the different parts of its output in several consoles. Open `myapp.sh` and change its content to the following:

```
1 #!/bin/bash
2
3 . misc/parse_params.sh
4 files=$(getparam "f")
5 entities=$(getparam "e")
```

```
whattocount=$(getparam "w")
showoutline=$(getparam "s")

echo "<eiout>"
echo "<eicommands>"
echo "<printonconsole>"
echo "<content format='text'>"
echo "I've received the following command-line parameters:"
echo ""
echo "   $@"
echo "</content>"
echo "</printonconsole>"

echo "<printonconsole consoleid='stats' consoletitle='Statistics'>"
echo "<content format='html'>"
echo "File statistics:"
echo "<div>"
echo "<ul>"

case $whattocount in
    lines) wcparam="-l"
    ;;
    words) wcparam="-w"
    ;;
    chars) wcparam="-m"
    ;;
esac

for f in $files
do
    echo " <li> $f has " `wc $wcparam $f | awk '{print $1}'` $whattocount
   "</li>"
done
echo "</ul>"
echo "</div>"
echo "</content>"
echo "</printonconsole>"

if [ $showoutline == 1 ]; then
    echo "<printonconsole consoleid='outline' consoletitle='Outline'>"
    echo "<content format='html'>"
    echo ""
    echo "Selected entities:"
    echo "<ul>"
    echo ""
    for e in $entities
    do
      echo "<li> $e </li>"
    done
    echo "</ul>"
    echo "</content>"
```

```
56      echo "</printonconsole>"
57  fi
58  echo "</eicommands>"
59  echo "</eiout>"
```

The output of `myapp.sh` is given in the EASYINTERFACE output language, because at Line 9 we start the output with the tag **eiout** which we close at Line 59. At Line 10 we start an **eicommands** tag, inside **eiout**, which we close at Line 58. Inside **eicommands** we have 3 **printonconsole** commands: the first one is generated by lines 11-17; the second by lines 19-41; and the last one by lines 44-56. Note that the first one uses the default console, while the last two use different consoles. Note also that the content in the last two is given in HTML. Execute `myapp.sh` in a shell first to check that it works correctly, and then execute the `MFA` tool from the web-client to see the effect of these changes.

3.6.2 Adding Markers

Next we explain a command for adding a marker next to a code line in the editor area. The following is an example of such command:

```
<addmarker dest="path" outclass="info">
  <lines>
    <line from="4" />
  </lines>
  <content format="text">
    text to associated to the marker
  </content>
</addmarker>
```

The attribute **dest** indicates the *full path* to the file (as received from the server) in which the marker should be added. The attribute **outclass** indicates the nature of the marker, which can be 'info', 'error', or 'warning'. This value typically affects the type/color of the icon to be used for the marker. The tag **lines** includes the lines in which markers should be added, each line is given using the tag **line** where the **from** attribute is the line number (**line** can be used to define a region in other commands, this is why the attribute is called **from**). The text inside the **content** tag is associated to the marker (as a tooltip, a dialog box, etc., depending on the client).

Let us modify `myapp.sh` to add a marker at Line 1 of each file that it receives. Open `myapp.sh` and add the following code snippet immediately before Line 58 of the previous script (i.e., immediately before closing the **eicommands** tag):

```
1  for f in $files
2  do
3    echo "<addmarker dest='$f' outclass='info'>"
4    echo "<lines><line from='1'/></lines>"
5    echo "<content format='text'> text for info marker of $f </content>"
6    echo "</addmarker>"
7  done
```

Lines 3-6 generate the actual command to add a marker for each file passed to `myapp.sh`. Execute `myapp.sh` in a shell first to check that it works correctly, and then execute the `MFA` tool from the web-client to see the effect of these changes.

3.6.3 Highlighting Code Lines

The following command can be used to highlight code lines:

```
<highlightlines dest="path" outclass="info" >
  <lines>
    <line from="5" to="10"/>
  </lines>
</highlightlines>
```

Attributes `dest` and `outclass` are as in the **addmarker** command. Each **line** tag defines a region to be highlighted. E.g., in the above example it highlights lines 5-10. You can also use the attributes `fromch` and `toch` to indicate the columns in which the highlight starts and ends respectively.

Let us modify `myapp.sh` to highlight lines 5-10 of each file that it receives. Open `myapp.sh` and add the following code snippet immediately before the instruction that closes the **eicommands** tag:

```
1 for f in $files
2 do
3   echo "<highlightlines dest='$f' outclass='info'>"
4   echo "<lines><line from='5' to='10'/></lines>"
5   echo "</highlightlines>"
6 done
```

Execute `myapp.sh` in a shell first to check that it works correctly, and then execute the `MFA` tool from the web-client to see the effect of these changes.

3.6.4 Adding Inline Markers

Inline markers are widgets placed inside the code. They typically include some read-only text. The following command adds an inline marker:

```
<addinlinemarker  dest="path" outclass="info">
  <lines>
    <line from="15" />
  </lines>
  <content format="text">
     Text to be viewed in the inline marker
  </content>
</addinlinemarker>
```

Attributes `dest` and `outclass` are as in the **addmarker** command. Each **line** tag defines a line in which a widget, showing the text inside the **content**, is added. Note that some clients, e.g., the web-client, allow only plain 'text' content.

Let us modify `myapp.sh` to add an inline marker at Line 15 of each file that it receives. Open `myapp.sh` and add the following code snippet immediately before the instruction that closes the **eicommands** tag:

```
for f in $files
do
  echo "<addinlinemarker dest='$f' outclass='info'>"
  echo "  <lines><line from='15' /></lines>"
  echo "  <content format='text'> Awesome line of code!! </content>"
  echo "</addinlinemarker>"
done
```

Execute `myapp.sh` in a shell first to check that it works correctly, and then execute the `MFA` tool from the web-client to see the effect of these changes.

3.6.5 Opening a Dialog Box

The following command can be used to open a dialog box with some content:

```
<dialogbox outclass="info" boxtitle="Done!" boxwidth="100" boxheight="100">
  <content format="html">
     Text to be shown in the dialog box
  </content>
</dialogbox>
```

The dialog box will be titled as specified in `boxtitle`, and it will include the content as specified in the **content** tag. The attributes `boxwidth` and `boxheight` are optional, they determine the initial size of the window.

Let us modify `myapp.sh` to open a dialog box with some message. Open `myapp.sh` and add the following code snippet immediately before the instruction that closes the **eicommands** tag:

```
echo "<dialogbox  boxtitle='Done!' boxwidth='300' boxheight='100'>"
echo "  <content format='html'>"
echo "   Hurray!."
echo "   The <span style='color: red'>MFA</span> tool has been applied."
echo "  </content>"
echo "</dialogbox>"
```

Execute `myapp.sh` in a shell first to check that it works correctly, and then execute the `MFA` tool from the web-client to see the effect of these changes.

3.6.6 Adding Code Line Actions

A *code line action* defines a list of commands to be executed when the user clicks on a line of code (more precisely, on a marker placed next to the line). The commands can be any of those seen above. The following is an example of such action:

```
<oncodelineclick dest="/Examples_1/iterative/sum.c" outclass="info" >
  <lines><line from="17" /></lines>
  <eicommands>
    <highlightlines>
      <lines>
        <line from="17" to="19"/>
      </lines>
    </highlightlines>
    <dialogbox boxtitle="Hey!">
      <content format="html">
        Click on the marker again to close this window
      </content>
    </dialogbox>
  </eicommands>
</oncodelineclick>
```

First note that the above XML should be placed inside the **eiactions** tag (that we have ignored so far). When the above action is executed, by the web-client for example, a marker (typically an arrow) will be shown next to Line 17 of the file `/Examples_1/iterative/sum.c`. Then, if the user clicks on this marker the commands inside the **eicommands** tag will be executed, and if the user clicks again the effect of these commands is undone. In the above case a click highlights lines 17-19 and opens a dialog box, and another click removes the highlights and closes the dialog box. Note that the commands inside **eicommands** inherit the `dest` and `outclass` attributes of **oncodelineclick**, but one can override them, e.g., if we add `dest="/Examples_1/iterative/fact.c"` to the **highlightlines** command then a click highlights lines 17-19 of **fact.c** instead of **sum.c**.

Let us modify `myapp.sh` to add a code line action, as the one above, for each file that it receives. Open `myapp.sh` and add the following code snippet immediately before the instruction that closes the **eiout** tag (i.e., after closing **eicommands**):

```
1  echo "<eiactions>"
2
3  for f in $files
4  do
5    echo "<oncodelineclick dest='$f' outclass='info' >"
6    echo "<lines><line from='17' /></lines>"
7    echo "<eicommands>"
8    echo "<highlightlines>"
9    echo "<lines><line from='17' to='19'/></lines>"
10   echo "</highlightlines>"
11   echo "<dialogbox boxtitle='Hey!'> "
12   echo "<content format='html'>"
13   echo "Click on the marker again to close this window"
14   echo "</content>"
15   echo "</dialogbox>"
16   echo "</eicommands>"
17   echo "</oncodelineclick>"
18 done
19
```

```
20 echo "</eiactions>"
```

Note that at Line 1 we open the tag **eiactions** and at Line 20 we close it. The rest of the code simply prints a code line action as the one above for each file. Execute `myapp.sh` in a shell first to check that it works correctly, and then execute the `MFA` tool from the web-client to see the effect of these changes.

3.6.7 Adding OnClick Actions

OnClick actions are similar to code line actions. The difference is that instead of assigning the action to a line of code, we can assign it to any HTML tag that we have generated. For example, suppose that at some point the tool has generated the following content in the console area:

```
<content format="html"/>
   <span style="color: red;" id="err1">10 errors</span> were found
   in the file sum.c
</content>
```

Note that the text "10 errors" is wrap by a **span** tag with an identifier **err1**. The OnClick action can assign a list of commands to be executed when this text is clicked as follows:

```
<onclick>
   <elements>
     <selector value="#err1"/>
  </elements>
  <eicommands>
    <dialogbox boxtitle="Errors">
      <content format="html">
        There are some variables used but not declared
      </content>
    </dialogbox>
  </eicommands>
</onclick>
```

It is easy to see that this action is very similar to **oncodelineclick**, the difference is that instead of **lines** we now use **elements** to identify those HTML elements a click on which should execute the commands.

Let us modify `myapp.sh` to add an OnClick action assigned to the list of files that it prints on the console. First look for the first occurrence of

```
1 echo "<ul>"
```

which should be at Line 23, and replace it by

```
1 echo "<ul style='background: yellow;' id='files'>"
```

This change will give the list of files that we print in the console (i.e., the corresponding HTML) the identifier **files**, and will change its background color to yellow. Next add the following code immediately before the instruction that closes **eiactions**:

```
echo "<onclick>"
echo "<elements>"
echo "<selector value='#files'/>"
echo "</elements>"
echo "<eicommands>"
echo "<dialogbox boxtitle='Errors'> "
echo "<content format='html'>"
echo "There are some variables used but not declated"
echo "</content>"
echo "</dialogbox>"
echo "</eicommands>"
echo "</onclick>"
```

This defines an OnClick actions such that when clicking on the list of files in the console area (anywhere in the yellow region) a dialog box is opened. Execute `myapp.sh` in a shell first to check that it works correctly, and then execute the `MFA` tool from the web-client to see the effect of these changes.

3.7 Adding Examples to the File-Manager

Consider again the web-client as depicted in Figure 2.2 (see Page 9). The file-manager on left-hand side includes a predefined set of examples. In this section we explain how to modify this set of examples. Adding a set of examples consists of two steps: (*i*) make it available on an EasyInterface server; and (*ii*) configure the client to include it in the file-manager. By default the web-client is configured to include all examples that are available on the default server (see Section 5.1.2 for details), so what is left is to define such sets on the server side. Later, in Chapter 5, we will see how to configure the examples on the client side.

A set of examples is an XML structure that represents a folder, for example, the following one is what we have in `server/config/default/examples.1.cfg`, which is actually the first set that appears in the web-client (see Figure 2.2 on Page 9):

```
<exset id="set1">
 <folder name="Examples_1">
    <folder name="iterative">
      <file name="sum.c" url="https://.../sum.c" />
      <file name="fact.c" url="https://.../iterative/sum.c" />
      ...
    </folder>
    <folder name="recursive">
      <file name="sum.c" url="https://.../recursive/sum.c" />
      <file name="fact.c" url="https://.../recursive/sum.c" />
      ...
    </folder>
 </folder>
</exset>
```

It is easy to see that it defines a folder structure, where each **file** is associated with a URL to its actual content. EASYINTERFACE provides also the possibility to associate a set of examples to a specific directory in a GitHub repository as follows (see `server/config/default/examples.2.cfg`):

```
<exset id="set2">
 <folder name="Examples_2">
     <github repo="easyinterface" owner="abstools" branch="master"
             path="examples/c/arrays" />
 </folder>
</exset>
```

To make the above two sets available on the EASYINTERFACE server we should also include them in the configuration file `server/config/default/examples.cfg` as follows:

```
<examples>
  <exset id="set1" src="default/examples.1.cfg" />
  <exset id="set2" src="default/examples.2.cfg" />
</examples>
```

Note that `examples.cfg` is also included in the main configuration file of the server `eiserver.default.cfg` by default.

Chapter 4

EasyInterface Server

This chapter describes the server side of the EasyInterface toolkit which, as explained in Section 2.2, mainly aims at providing a uniform way to access local tools, i.e., those installed on the machine where the server runs. The EasyInterface server achieves this by:

(*i*) providing a way to describe, using XML based configuration files, how to execute a local tool and which parameters it takes, as well as how to define sets of related examples; and

(*ii*) providing a JSON based protocol that can be used to request information on those tools and examples, execute tools, etc.

In the rest of this chapter we give specifications for the above two points. In particular, in Section 4.1 we describe how to configure the server, and what is the expected corresponding functionality, and in Section 4.2 we describe how to send requests to the server and how the server should react on these requests. Any implementation must respect these specifications, in particular, we describe such an implementation in Section 4.3.

4.1 Configuring the EasyInterface Server

In Section 4.1.1 we first describe the notion of command-line templates, which is crucial for describing how to execute a tool, in Section 4.1.2 we describe the work-flow of a tool, and then in Section 4.1.3 we describe the syntax and semantics of the server configuration.

4.1.1 Command-line Templates

A command-line template is a syntactic object that describes how to run a tool from a command-line, in the context of an EasyInterface server. It is, in principle, a string that corresponds to a command-line, but also includes *template parameters* that are replaced by corresponding values before executing the command-line. The following is an example to such template:

```
/path-to/app _ei_files -m _ei_outline _ei_parameters
```

In this template, anything that starts with `_ei` is a template parameter that is replaced by some corresponding value, and `/path-to/app` is the tool's executable. When the server receives a request for executing the corresponding tool, the request includes several data that should be passed to the tool. For example, the following are typical data that should be passed to a tool:

1. files to be processed (e.g., a program to be analyzed);
2. entities selected from the program outline (e.g., methods); and
3. values for the different parameters.

The server passes this data to the tool by replacing the template parameters with corresponding data as follows:

1. the files are stored locally (e.g., in `/tmp`), and `_ei_files` is replaced by a list file names (each with an absolute path, separated by a space);
2. `_ei_outline` is replaced by a list of selected entities (e.g., method names); and
3. `_ei_parameters` is replaced by the list of parameters generated from those provided in the request.

This results in, for example, the following instance of the template:

```
/path-to/app /tmp/ei_FAJw1B/a.c /tmp/ei_FAJw1B/b.c -m a.main -v 1 -d 3 -a
```

which is then executed and its output is redirected to the client.

The following is the full-list of template parameters that should be supported by the server:

- `_ei_root`: this parameter is replaced by a path to a temporal directory that corresponds to the current execution request. This directory is created by the server, and all information related to the corresponding request is stored under this directory. In addition, this directory should be writable such that the tool can use it to write temporal information as well. This directory must include the following sub-directory:
  - (*i*) `_ei_files`: it is used to store the files received from the client before passing them to the tool – see template parameter `_ei_files` below;
  - (*ii*) `_ei_download`: can be used by a tool to leave files that can be downloaded later – see sections 4.1.2 and 4.2.4;
  - (*iii*) `_ei_stream`: can be used by a tool to store partial output such that clients can fetch it later periodically – see sections 4.1.2 and 4.2.5; and
  - (*iv*) `_ei_tmp`: can be used by tools to write temporal files, i.e., it is like `/tmp` in UNIX but for this specific execution in order to avoid conflicts between different requests.
- `_ei_files`: this parameter should replaced by a list of files that are received from the client (each with an absolute path, separated by a space). The server should store the files under the directory `_ei_root/_ei_files`, using the same structure as passed by the client.

- **_ei_outline**: this parameter should be replaced by a list of selected outline entities (separated by space) that passed by the client;
- **_ei_parameters**: this parameter should be replaced by a corresponding list of parameters that are passed by the client (see [**PARAMETERS**] in Section 4.1.3 to understand how the parameters are constructed);
- **_ei_sessionid**: should be replaced by a session identifier that corresponds to a user, this makes it possible to track information of a user along several requests (see next item). The server is supposed to provide support for this sessions mechanism;
- **_ei_sessiondir**: should be replaced by a path to a temporal directory that corresponds to the session **_ei_sessionid**. Tools can use this directory to store data related to the corresponding session.
- **_ei_clientid**: should be replaced by the client identifier, i.e., **webclient**, **eclipse**, **shell**, etc.
- **_ei_outformat**: should be replaced by either **eiol** or **txt**, which indicates if the client supports the EASYINTERFACE output language or just plain text format. This value is passed by the client as we will see later. This makes it possible to provide output depending on the formats supported by the client.
- **_ei_execid**: should be replaced by an execution identifier that corresponds to the current execution of the tool, this identifier is mainly used to fetch information left in the **_ei_download** and **_ei_stream** directories (see sections 4.1.2, 4.2.4 and 4.2.5).

4.1.2 Work-flow of Tools

In this section we describe the possible work-flows of tools installed in an EASYINTERFACE server. These work-flows are different in the way the output is passed to the client, and they can be combined together as well.

Default Work-flow

The simplest work-flow of tool is the one in which once a tool is executed, anything that it prints on the standard output will be forwarded to the client who requested to execute the tool. This output could be plain text, or uses the EASYINTERFACE output language.

Download Output

In this work-flow a tool can leave files in the directory under **_ei_root/_ei_download**, and then clients can send special requests for downloading such files (see Section 4.2.4). This is particularly useful when such files are large or not in text format. Note that the tool should provide the client with the **_exec_id** (using the default work-flow) as it is required for downloading these files.

Streaming Output

In this work-flow a tool can leave processes that generate output chunks in the background, such that clients can fetch these chunks periodically later – see Section 4.2.5. The tool should adhere to the following rules:

- When the background processes are started, their *process ids* (as in the underlying operating system) should be written to the file `_ei_root/_ei_stream/pid`. This way clients can request to stop these processes later before the timeout forced by the server expires;
- Before the background processes terminate, they should create an empty file `_ei_root/_ei_stream/terminated`. This way clients can consult if the background processes have terminated already; and
- Output chunk should be written to files in the directory `_ei_root/_ei_stream`, the file names are not important, but rather their extension: the server allows retrieving these chunks (in order of creation) by their extension.

Note that the tool should provide the client with the `_exec_id` (using the default work-flow) as it is required for queries that correspond to the above points.

4.1.3 The Syntax of the Configuration File

This section describes how to configure the EASYINTERFACE server. The content of the configuration file should adhere to the [EISERVER] XML structure that is described below. Inside this tag we can define tools, examples, etc. The best way to read this XML is by following the links in the definition of [EISERVER].

General Comments about XML Structures

For the purpose of better organization of the configuration files, the server should provide a way to split them into several files and import one file into another one. Any XML structure

```
<tagname ...>
  ....
</tagname>
```

should be possible to write as

```
<tagname src=[CFGFILENAME] />
```

where the file [CFGFILENAME] includes the actual XML structure (of the first form above). It will be automatically imported when needed. However, if the XML structure (the first form) has an attribute id then it must appear as an attribute in the second form as well. This is useful for loading a partial XML structure, e.g., that refers to a tool with a specific identifier, instead of loading the whole XML and looking for that part.

The Main XML Tag of the Configuration File

EISERVER

```
<eiserver version=[VERSION]?>
  [SETTINGS]?
  [SANDBOX]?
  [EXAMPLES]?
  [APPS]?
</eiserver>
```

SINCE VERSION: 1.0

DESCRIPTION:

This XML tag is the root of the configuration file. The [SETTINGS] section is used for setting some global parameters; [SANDBOX] is used for setting some limits on resources when executing a tool; [EXAMPLES] defines which sets of examples are available on the server; and [APPS] defines which tools are available on the server. The version attribute indicates the version of the configuration syntax, which is 1.0 by default.

General Settings

SETTINGS

```
<settings>
  [SETPROP]+
</settings>
```

SINCE VERSION: 1.0

DESCRIPTION:

The purpose of this tag is to provide general settings for the server, it is mainly implementation dependent.

SETPROP

```
<setprop name=[STRING] value=[STRING] />
```

SINCE VERSION: 1.0

DESCRIPTION:

This tag describes a settings property, the actual properties are left open and will be instantiated depending on the implementation.

Sandbox Settings

SANDBOX

```
<sandbox>
  [SANDBOXPROP]+
</sandbox>
```

SINCE VERSION: 1.0

DESCRIPTION:

This tag is used for setting limits on resources when executing a tool.

SANDBOXPROP

```
<sandboxprop name=[STRING] value=[STRING] />
```

SINCE VERSION: 1.0

DESCRIPTION:

This tag describes a sandbox property, the actual properties are left open and will be instantiated depending on the implementation. However, it must include a property with name `timeout` which indicates the maximum time (in seconds) that a tool can run before it is forced to terminate by the server.

Examples Settings

EXAMPLES

```
<examples>
  [EXSET]*
</examples>
```

SINCE VERSION: 1.0

DESCRIPTION:

This tag is used to declare sets of examples that are available in the server, where each such set is defined by one [EXSET].

EXSET

```
<exset id=[EXSETID]>
  [EXELEMENT]*
</exset>
```

SINCE VERSION: 1.0

DESCRIPTION:

This tag declares a set of examples, which are defined by a collection of [EXELEMENT]

(a file, a directory, or a link to a GitHub repository). The attribute `id` is a unique identifier that is used to refer to this set when communicating with the server.

EXELEMENT

```
( [FILE] | [FOLDER] | [GITHUB] )
```

SINCE VERSION: 1.0

DESCRIPTION:

An example element, which can be a file [FILE], a folder [FOLDER], or a link to a GitHub repository [GITHUB].

FILE

```
<file name=[FILENAME] url=[URL] />
```

SINCE VERSION: 1.0

DESCRIPTION:

This tag declares a file, where the `name` attribute is its name and `url` is a link to its content. Note that `name` is not necessarily the same as the one in `url`.

FOLDER

```
<folder name=[FOLDERNAME]>
  [EXELEMENT]*
</folder>
```

SINCE VERSION: 1.0

DESCRIPTION:

This tags declares a folder with `name` as its name. The content of this tag is a list of [EXELEMENT] tags, which in turn declare the inner files, folders, etc.

GITHUB

```
<github repo=[GITHUBREPO] owner=[GITHUBUSER] branch=[GITHUBBRANCH]?
    path=[GITHUBPATH]?/>
```

SINCE VERSION: 1.0

DESCRIPTION:

Declares a reference to the public GitHub repository `repo` which is owned by the user `owner`. Optionally one can also refer to a specific `branch`, which is **master** by default, and to a specific `path` (a directory or a single file) which is the root of the

repository by default.

Tools Settings

APPS

```
<apps>
  [APP]*
</apps>
```

SINCE VERSION: 1.0

DESCRIPTION:

This tag declares a list of tools (to be added to the server). Each such tool is defined by one [APP] environment.

APP

```
<app id=[APPID] visible=[BOOL]?>
  [APPINFO]
  [APPHELP]
  [SANDBOX]
  [EXECINFO]
  [PARAMETERS]
  [PROFILES]
</app>
```

SINCE VERSION: 1.0

DESCRIPTION:

This tag defines a tool, where the meaning of the different parts is as follows:

- `id` is a unique identifier used to refer to this tool when communicating with the server.
- `visible` indicates if this tool should be listed when the list of available tools is requested — by default it is `true`. Note that even if a tool is not visible, it can be used like any other tool by those who know its `id`.
- [APPINFO] provides general information about the tool, e.g., title, logo, etc.
- [APPHELP] provides enough information on how the tool can be used, etc. It is mainly used in the help sections of the different clients.
- [SANDBOX] provides limits on resources when executing this tool. They override the ones defined directly in the [EISERVER] environment.
- [EXECINFO] defines how the tool can be executed from a command-line.

- [PARAMETERS] defines the set parameters accepted by the tool.
- [PROFILES] defines sets of default values for the different parameters. This is intended to allow users to select a predefined set of values for the parameters instead of setting them manually.

APPINFO

```
<appinfo>
  [ACRONYM]?
  [TITLE]?
  [LOGO]?
  [DESC]?
</appinfo>
```

SINCE VERSION: 1.0

DESCRIPTION:

This tag provides general information about a tool:

- [ACRONYM] is an acronym for the tool, e.g., COSTA;
- [TITLE] is the full name of the tool;
- [LOGO] is an image corresponding to the logo of the tool; and
- [DESC] is a description of the tool.

ACRONYM

```
<acronym>[TEXT]</acronym>
```

SINCE VERSION: 1.0

DESCRIPTION:

Plain text to be used as an acronym, e.g., COSTA.

TITLE

```
<title>[TEXT]</title>
```

SINCE VERSION: 1.0

DESCRIPTION:

Plain text describing a title, e.g., for a tool. It is typically more informative than an acronym (see [ACRONYM]).

LOGO

```
<logo url=[URL] />
```

SINCE VERSION: 1.0

DESCRIPTION:

A link to an image — in some standard format, e.g., `png`, `jpg` or `gif` — to be used by clients as a logo (e.g., for a tool).

DESC

```
<desc>
  <short>[TEXT]</short>
  <long>[TEXT]</long>
</desc>
```

SINCE VERSION: 1.0

DESCRIPTION:

This is a description of some entities, e.g., of a tool, a parameter, a parameter option, etc. It consists of two parts, the first one is a short description, and the second is a detailed description. In both cases it should be plain text. Clients will select one of them depending on the intended use.

APPHELP

```
<apphelp>
  [CONTENT]+
</apphelp>
```

SINCE VERSION: 1.0

DESCRIPTION:

A (formatted) text that provides enough information on how a tool can be used, etc. It can be provided in several formats, e.g., HTML or plain text, by using several [CONTENT] tags. Clients are supposed to pick the appropriate format if more than one is available. It is recommended to always include a content in plain text since it can be viewed in any client.

CONTENT

```
<content format=[TEXTFORMAT]? >
  [TEXT]
</content>
```

SINCE VERSION: 1.0

DESCRIPTION:

A text given in a specific `format`, e.g., `text`, `html`, etc. If the attribute `format` is not provided, then it is assumed to be `text` format (plain text).

EXECINFO

```
<execinfo>
 [CMDLINEAPP]
</execinfo>
```

SINCE VERSION: 1.0

DESCRIPTION:

Provides information on how to execute a tool. Currently it includes only the command-line template [CMDLINEAPP].

CMDLINEAPP

```
<cmdlineapp path=[TOOLEXEC]?> [CMDTEMPLATE] </cmdlineapp>
```

SINCE VERSION: 1.0

DESCRIPTION:

Describes how to run a tool from a command-line. It can be used in two modes:

- if the attribute `path` is not specified, then command-line template [CMDTEMPLATE] is instantiated and then executed;
- if the attribute `path` is specified, then command-line template [CMDTEMPLATE] is instantiated and passed to the program specified by `path` in the standard input.

The second possibility is more safe, since it does not allow clients to manipulate the command-line in order to execute undesired programs. However, the server should provide enough guarantees that this does not happen in the first option as well.

Tool Parameters

PARAMETERS

```
<parameters prefix=[PARAMPREFIX]? check=[BOOL]?>
 [PARAM]*
</parameters>
```

SINCE VERSION: 1.0

DESCRIPTION:

Defines a list of parameters that are accepted by a corresponding tool. Each parameter is defined by one [PARAM] environment. The prefix attribute is used to specify a string that will be attached to each parameter name when passed to the tool. For example, if prefix="--" and there is a parameter called **level** with value **X**, then "**--level X**" will be passed to the tool. The default value of prefix is "-". It can also be set to an empty string if there is no need for a prefix. The check attribute is used to indicate if the server should verify that the values of the parameters are valid (w.r.t. the specified values). The default value of check is true. The attributes prefix and check are inherited by each parameter [PARAM], which in turn can override them.

PARAM

```
( [SELECTONE] | [SELECTMANY] | [FLAG] | [TEXTFIELD] )
```

SINCE VERSION: 1.0

DESCRIPTION:

Defines a parameter accepted by a corresponding tool. There are several types of parameters supported:

- [SELECTONE] defines a parameter that takes one value from a predefined set;
- [SELECTMANY] defines a parameter that takes several values from a predefined set;
- [FLAG] defines a Boolean parameter; and
- [TEXTFIELD] defines a parameter that takes a free-text value.

SELECTONE

```
<selectone name=[PARAMNAME] prefix=[PARAMPREFIX]? check=[BOOL]? >
  [DESC]
  [OPTION]+
  [DEFAULTVALUE]?
</selectone>
```

SINCE VERSION: 1.0

DESCRIPTION:

Defines a parameter that takes a *single* value out of a given list:

- name is the name of the parameter, it must be unique among all parameters of a tool;
- prefix and check can be used to override the corresponding attributes of [PARAMETERS];

- [DESC] provides a description of this parameter;
- [OPTION]+ is a list of possible values for this parameter;
- [DEFAULTVALUE] specifies the default value. If not specified then the first [OPTION] is considered to be the default one.

SELECTMANY

```
<selectmany name=[PARAMNAME] prefix=[PARAMPREFIX]? check=[BOOL]? >
  [DESC]
  [OPTION]+
  [DEFAULTVALUE]*
</selectmany>
```

SINCE VERSION: 1.0

DESCRIPTION:

Defines a parameter that takes *several* values out of a given list. The meaning of the attributes and inner environments is as in [SELECTONE], except that in this case we can specify several [DEFAULTVALUE].

FLAG

```
<flag name=[PARAMNAME] prefix=[PARAMPREFIX]? check=[BOOL]?
      explicit=[BOOL]? trueval=[PARAMVALUE]? falseval=[PARAMVALUE]? >
  [DESC]
  [DEFAULTVALUE]?
</flag>
```

SINCE VERSION: 1.0

DESCRIPTION:

Defines a parameter that can take **true** or **false** values. The meaning of the attributes and inner environments is as in [SELECTONE]. The attribute `explicit` is used to specify how this parameter should be passed to the tool. For example, assume the parameter name is `f`, then:

- when `explicit` is `false`, the parameter is passed as "`-f`" if its value is **true** and not passed at all if its value is **false**.
- when `explicit` is `true` the parameter is explicitly passed to the tool, i.e., using "`-f X`" where `X` is the selected value. By default the possible values are **true** and **false**, however, you can redefine them (only when `explicit` is `true`) using the attributes `trueval` and `falseval`.

The default value of `explicit` is `false`.

TEXTFIELD

```
<textfield name=[PARAMNAME] prefix=[PARAMPREFIX]? check=[BOOL]?
          passinfile=[BOOL]? multiline=[BOOL]? type=[TYPE]? >
  [DESC]
  <initialtext>[TEXT]</initialtext>
</textfield>
```

SINCE VERSION: 1.0

DESCRIPTION:

Defines a parameter that can take free-text value. The **initialtext** tag includes a text to be shown in the corresponding text-area by default. The meaning of the attributes is as follows:

- `multiline` is used to specify if the free-text should be single- or multi-line. By default its values is `false`, i.e., single-line.
- `passinfile` is used to indicate that the actual value should be saved into a file, and what is passed to the tool is the file name instead of the actual text. This should be used for safety, when there is a risk that the free-text can be harmful to the command-line (although the server should do some checks to avoid this).
- `type` restricts the value that can be provided to a specific type.

The meaning of the other attributes and inner environments is as in [SELECTONE].

OPTION

```
<option value=[PARAMVALUE]>[DESC]</option>
```

SINCE VERSION: 1.0

DESCRIPTION:

Defines an option (i.e., a possible value) for a parameter.

DEFAULTVALUE

```
<default value=[PARAMVALUE] />
```

SINCE VERSION: 1.0

DESCRIPTION:

Defines a default value for a parameter.

Tool Profiles

PROFILES

```
<profiles>
  [PROFILE]*
</profiles>
```

SINCE VERSION: 1.0

DESCRIPTION:

This tag declares a list of profiles. Each such profiles is defined by one [PROFILE] environment.

PROFILE

```
<profile name=[PROFILENAME] >
  [DESC]
  [PROFILEPARAM]*
</profile>
```

SINCE VERSION: 1.0

DESCRIPTION:

This tag defines a profile which is a list of [PROFILEPARAM] parameters. [DESC] gives a description of this profile. The semantics of applying a profile is as follows: (*i*) all parameters of the tool are set to their default value; and (*ii*) the values of the parameters indicated in this profile are set to their corresponding values.

PROFILEPARAM

```
<setparamvalue name=[PARAMNAME] value=[PARAMVALUE] /> or
<setparamvalue name=[PARAMNAME]>[TEXT]</setparamvalue>
```

SINCE VERSION: 1.0

DESCRIPTION:

This tag is used to define a value for a parameter in the context of a profile. The attributes name and value refers to the parameter name and value respectively.

It has two forms that are used depending on the type of parameter as follows.

- for [SELECTONE] parameters we use the first form. The corresponding [PROFILE] should include only one setparamvalue for such parameter;
- for [SELECTMANY] parameters we use the first form. The corresponding [PROFILE] can include one setparamvalue tag for each value (since such parameters take several values);
- for [FLAG] parameters we use the first form. The value must be false or true. In addition one can use the values of the corresponding trueval or falseval attributes.

- for [TEXTFIELD] parameters we use the second form, where [TEXT] refer to the value of this parameter.

Others

TEXTFORMAT
(**text** | **html** | **svg**)

PARAMVALUE
[a-z,A-Z,0-9,-,_]+

PARAMNAME
[a-z,A-Z,0-9,-,_]+

BOOL
(**true** | **false**)

APPID
[a-z,A-Z,0-9,-,_]+

EXSETID
[a-z,A-Z,0-9,-,_]+

WIDGETID
[a-z,A-Z,0-9,-,_]+

PROFILENAME
[a-z,A-Z,0-9,-,_]+

URL
A valid `http` or `https` URL.

PARAMPREFIX
Can be any string that matches [a-z,A-Z,0-9,-,_]+, typically - or --.

TEXT
Free text.

GITHUBPATH
A path to a file or a directory in a GitHub repository (relative to the root of the repository).

GITHUBBRANCH
A valid branch name for a GitHub repository.

GITHUBUSER
A valid GitHub user name.

GITHUBREPO
A valid GitHub repository.

FOLDERNAME
[a-z,A-Z,0-9,-,.,_]+

FILENAME
[a-z,A-Z,0-9,-,.,_]+

CFGFILENAME
A path to a configuration file. Depending on the implementation, it might be restricted to relative to some configuration directory.

TYPE
(**bool** | **int** | **float** | **string** | **word**) where **word** means a string without whitespaces

TOOLEXE
A path describing an executable (of a tool).

CMDTEMPLATE
A command-line template as explained in Section 4.1.1.

4.2 Communicating with the EasyInterface Server

This section describes the protocol to be used for communicating with the EasyInterface server. The server should receive requests using the HTTP POST protocol [27]. The advantage of using this protocol is that one can build the EasyInterface server on top of an HTTP server, and thus take advantage of the underlying machinery for serving clients concurrently. In addition, if one is not interested in building the EasyInterface server on top of an HTTP server, there are numerous libraries, for different languages, for HTTP POST communication that one can use (this is true for the client side as well).

The HTTP POST request should include a single attribute called **eirequest**. The actual value of **eirequest** is a JSON record that as described in the next sections. The following is an example of how one can communicate with the server, that we have implemented, using JavaScript and jQuery [7]:

```
var req;

// here we set variable 'req' to the JSON record
// that represents the request

$.post("http://localhost/ei/server/eiserver.php",
{
  eirequest: req
},
function(data) {
  // do something with the response 'data'
});
```

The response of the server should be an XML structure of the following form:

```
<ei_response>
  <ei_server_output> ... </ei_server_output>
  <ei_output> ... </ei_output>
  <ei_error> ... </ei_error>
</ei_response>
```

Where

- ei_server_output includes messages printed by the server. These messages are not the response to the request, but rather debugging messages that can be useful when developing clients, debugging the server, etc. Most users should ignore this environment.
- ei_output includes the response to the request, i.e., if we request to execute a tool the output of that tool goes inside this tag.
- ei_error includes error messages that are related to the request (not to the tool).

Typically, ei_output and ei_error are mutually exclusive, i.e., only one can appear in the response.In the next sections we describe the format of the different requests that one can make to the EASYINTERFACE server.

4.2.1 Retrieve Information on Available Tools

To retrieve information on a given tool, or all visible tools on the server, the request should adhere to the following format:

```
{
  "command": CMD,
  "app_id": ID
}
```

where

(*i*) `CMD` can be `app_info`, `app_parameters`, or `app_details`; and

(*ii*) `ID` is either the special value `_ei_all` (i.e., all tools) or a tool identifier as specified in [APP].

A successful request will return (inside the ei_output tag) the XML structure [APPS] (that is defined in the configuration file) after filtering out some information as we explain next. First any tool that does not match `ID` is removed (if `ID` is `_ei_all` then only non-visible tools are removed). Then, for the remaining tools:

- If `CMD` equals `app_info`, it returns only the [APPINFO] of each tool;
- If `CMD` equals `app_parameters`, it returns only the [PARAMETERS] and [PROFILES] of each tool; and
- If `CMD` equals `app_details`, it returns everything except [EXECINFO] and [SANDBOX] of each tool.

Note that [EXECINFO] and [SANDBOX] are never returned as they reveal information on how to execute a tool locally, etc.

4.2.2 Execute a tool

Next we describe, by mean of an example, the form of a request for executing a tool. Suppose we are interested in executing a tool with identifier **myapp** where, in addition, we would like to pass it some values for the parameters, files to process, outline entities, the identifier of the client who is making the request and which output format it supports. Such a request has the following form:

```
{
  "command": "execute",
  "app_id": "myapp",
  "parameters": {
    "l": [ "true" ],
    "f": [ "false" ],
    "s": [ "yes" ],
    "x": [ "1", "2" ],
    "_ei_clientid": "webclient",
    "_ei_outformat": "eiol",
    "_ei_outline":  [ "ent1", "ent2", ...  ],
    "_ei_files": [
                  {
                    path: "dir1",
                    type: "dir",
                  },
                  {
                    path: "dir2",
                    type: "dir",
                  },
                  {
                    path: "dir1/file1.c",
                    type: "text",
                    content: "This is the content of the file"
                  },
                  {
                    path: "dir2/file2.c",
                    type: "text",
                    content: "This is the content of the file"
                  },
                  {
                    path: "dir2/file3.c",
                    type: "text",
                    content: "This is the content of the file"
                  }
                ]
     }
}
```

Let us explain the different parts of this request:

- The field **command** must have the value **execute**;

- The filed **app_id** should refer to the identifier of the tool that we want to execute, it can be visible or not;

- The filed **parameters** is a JSON record that includes all the information, e.g., tool parameters and files, that we want to pass over, as we explain below.

Before explaining the details of the parameters record, it is recommend that you refresh your memory with the details of the command-line template as described in Section 4.1.1. The parameters record includes the following information:

- *Tool parameters:* any field of record whose name does not start with "**_ei**" is a parameter that is supposed to be defined in the [**PARAMETERS**] environment of the corresponding tool. The value of such field is a list of elements that represent the value of the parameter. If the parameter is supposed to take a single value then the list must have a single element.

- *Files*: the field **_ei_files** represents the files that we want to pass to the tool. Its value is an array of JSON records where each record represents a text file or a directory. The **path** field of the record refers to the file or directory name, it is relative to the root of the temporary directory where the server saves these files. The **type** field indicates the type of the file. In the case of text files, the field **content** represents the actual content of the file. Note that binary files can be supported as well by encoding them to text representation, we leave this feature implementation dependent.

- *Outline entities*: the filed **_ei_outline** is a list of elements representing the selected entities from the outline.

- *Client identifier*: the field **_ei_clientid** indicates the identifier of the client who has performed the request.

- *Supported output format*: the field **_ei_outformat** indicates the supported output format, it can be **eiol** or **txt**.

4.2.3 Retrieve Example Sets

To retrieve example sets we use the following request:

```
{
  "command":  "exset_details",
  "exset_id":  ID
}
```

where **ID** is either the special value **_ei_all** (i.e., all example sets) or an examples set identifier as specified in [**EXSET**]. A successful request will return (inside the **ei_output** tag) the XML structure [**EXAMPLES**] after filtering out those example sets that do not match the value of **ID**, i.e, if **ID** is **_ei_all** then it returns all example sets, otherwise only the indicated one.

4.2.4 Download Output Files

Assuming that a tool has left a file in the directory `_ei_root/_ei_download`, we can download it later using the following request to the same server on which that tool was executed:

```
{
  "command":  "download",
  "exec_id":  EXECID,
  "file":  FILENAME
}
```

Here **EXECID** is the corresponding execution identifier (see Section 4.1.1), and **FILENAME** is the name of the file to be downloaded.

4.2.5 Manage Output Streams

Assuming that a tool has left some processes in the background which generate some output chunks into `_ei_root/_ei_stream`, we can retrieve those chunks using the following request to the same server on which that tool was executed:

```
{
  "command":  "get_stream",
  "exec_id":  EXECID,
  "extention":  EXT
}
```

Here **EXECID** is the corresponding execution identifier (see Section 4.1.1), and **EXT** is the extension of the files that represent the chunks to be retrieve – note that a tool can generate chunks with different extensions and retrieve them separately. The server responds to this request by sending back the following XML structure for each generated chunk (each chunk corresponds to one file with the extension **EXT**):

```
<ei_stream state=[STATE]>
  [CHUNKCONTENT]
</ei_stream>
```

Apart from the content of the corresponding chunk, the attribute `state` indicates the state of the background processes as follows:

- **nostream**: if there is no stream with identifier **EXECID**;
- **terminated**: if the corresponding background processes have terminated normally.
- **stopped**: if the corresponding background processes have been stopped (see **kill_stream** request below);
- **running**: if the corresponding background processes are still running.
- **empty**: if there is no new content generated for the corresponding stream. Note that the corresponding background processes are still running in this case.

- **unknown**: if some other unexpected error has occured.

In addition, clients can request to stop the background processes using the following request:

```
{
  "command":  "kill_stream",
  "exec_id":  EXECID,
}
```

where **EXECID** is the corresponding execution identifier. The response to such request is as above with a **state** **terminated** or **stopped**, but has no content.

4.3 Implementation

We have implemented an EasyInterface server as a collection of PHP programs that run on top of an HTTP server, e.g., Apache. It implements the specification as described in this chapter. It is part of the GitHub repository `http://github.com/abstools/easyinterface`, under the directory **server**.

By default the server uses **server/config/eiserver.cfg** as a configuration file, and if no such file exists it uses **server/config/eiserver.default.cfg**. The default installation comes with a default **sever/config/eiserver.default.cfg** that includes some demo tools and corresponding examples. It is recommended not to modify **server/config/eiserver.default.cfg**, but rather create your own configuration file **server/config/eiserver.cfg**. This way you can always have a correct configuration file at hand from which you can copy, etc. Note that all references to configuration files, when including them via the **src** attribute, should be relative to **server/config**.

When installing a tool, it is recommended that, instead of refereeing to the tool's executable directly, to write a bash-script wrapper that execute the tool and place this script under **server/bin**. This way we have more control on the installed tools. Note that when refereeing to these wrappers, in the configuration files, it is enough to use paths relative to **server/bin** (the server switches to the directory **server/bin** before executing the command-line).

Chapter 5

EasyInterface Clients

The aim of the EasyInterface clients is to facilitate the way users connect to EasyInterface servers, i.e., instead of directly using the protocol described in Section 4.2, they can use a (graphical) user interface that: (*i*) connects to the EasyInterface servers and retrieves for the list of available tools; (*ii*) allows the user to choose a tool to execute on some input files, and set the values of the corresponding parameters; (*iii*) generates a corresponding request and sends it to a corresponding EasyInterface server; and (*iv*) shows the returned output to the user. In addition, in some cases, clients are supposed to provide code editing capabilities so user can edit their code as well in an integrated developing environment. In Section 5.1 we describe the web-client of the EasyInterface toolkit, and in Section 5.2 we discuss other possible clients.

5.1 Web-Interface Client

The web-client of EasyInterface is a JavaScript program that runs in a web browser. It uses jQuery [7, 35] as well as some other libraries like jsTree [8] and CodeMirror [3]. It is part of the github repository `http://github.com/abstools/easyinterface`, under the directory **`clients/web`**. Once EasyInterface is installed (see Appendix A), the web-client can be accessed using `http://localhost/ei/clients/web`. A representative screenshot of the different parts of the web-client is depicted in Figure 5.1. The web-client is designed like a development environment, and it includes the following main components:

1. **`Code Editor`**: an area were programs can be edited, which also provides the functionality of an editor in general such as *search and replace* and *syntax highlighting*;

2. **`File Manager`**: an area where users can manage their own files, and also access predefined sets of examples. In addition, it supports accessing GitHub repositories;

3. **`Outline`**: an area where the different elements of the edited programs, such as class and method names, are shown. It is configurable depending on the programming languages used;

4. **`Console`**: an area where the output of a tool can be printed, which includes several tabs for better organization of the output; and

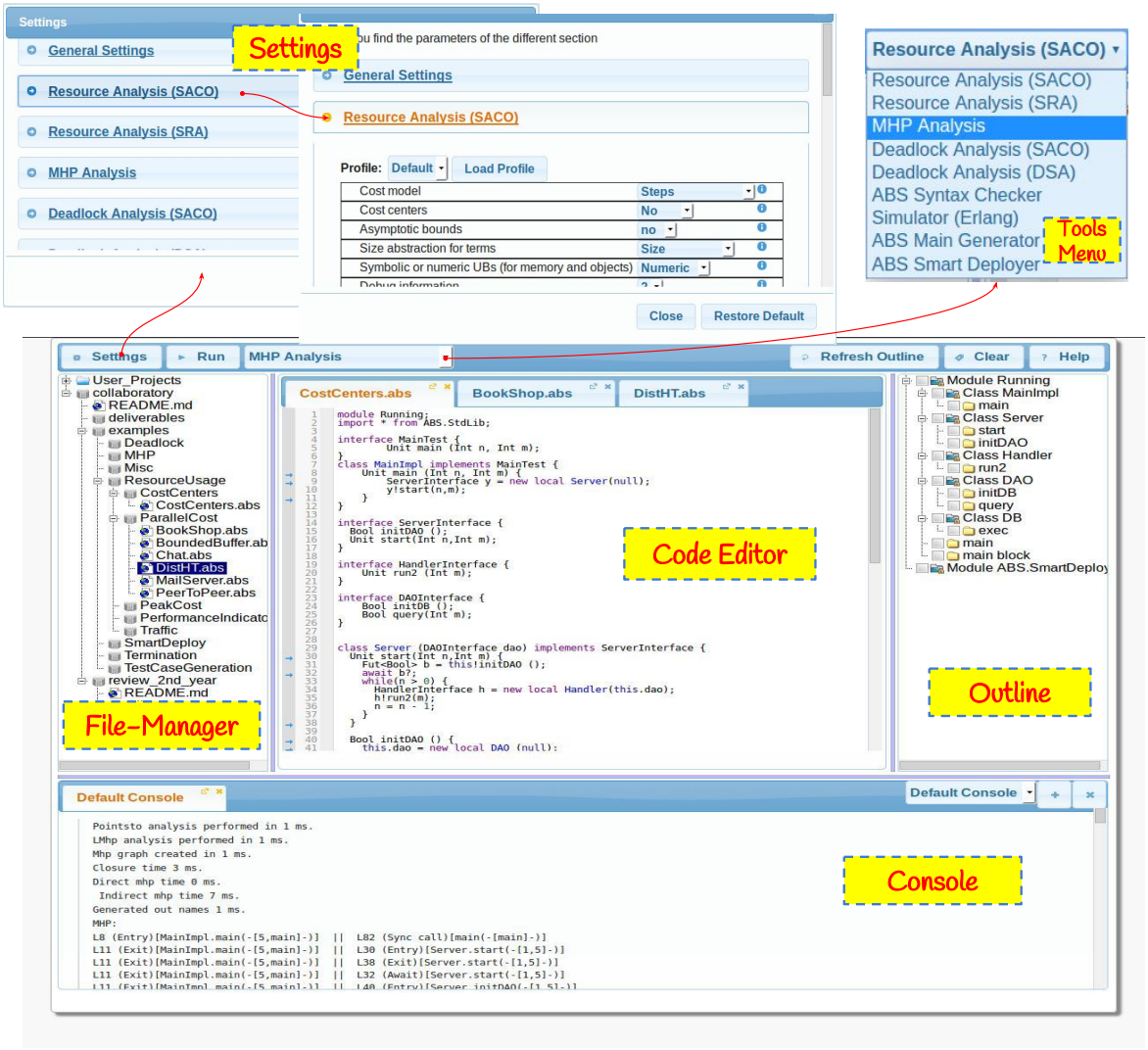

Figure 5.1: EASYINTERFACE Web Client

5. **`Settings`**: an area where the parameters of each tool are viewed graphically, allowing users to set their values, or selecting a predefined profile, before running a tool.

The interface has also a tool bar with a combo-box that includes the list of available tools, a button to execute a tool, a button to access the settings section, a button to create an outline, a button to access the help section, and a button to clear the last output generated. The web-client includes an interpreter for the EASYINTERFACE output language that is describe in Chapter 6.

The web-client can be easily configured to fit the user's needs, it has a configuration file to control, among others, the following aspects:

(*i*) the tools to include in the tools menu;

(*ii*) the examples to show in the file-manager; and

(*iii*) the tool be used for generating the outline for a set of programs – note that this is programming language dependent).

The web-client first looks for the configuration file **clients/web/webclient.cfg**, and if it does not exists it uses **clients/web/webclient.default.cfg** which is shipped by default with EASYINTERFACE. It is recommended not to directly modify **webclient.default.cfg**, but rather create a new copy into **webclient.cfg**.

Next we explain the different components of the configuration file. In the rest of this chapter, when we refer the *default server* we mean the one that is available at the same address as the web-client, i.e., if the web-client was accessed using the URL

"**http://somedomain/.../ei/client/web**",

then the URL of the default server is

"**http://somedomain/.../ei/server**".

The configuration file is a text file that includes a single JSON record with the following fields:

```
{
  title:          A title to use for the window,
  apps:           A list of tools to include in the tools menu,
  examples:       A list of examples to include in the file manager,
  outline:        Indicates if the outline area should be hidden,
  outlineserver:  The server of the tool for generating the outline,
  outlineapp:     The name of the tool to generate outline
  language:       The programming language of the edited programs,
}
```

All fields in the above record are optional, the web-client assigns default values for those that are not available. Their meaning is as follows:

- **title** is used to set the window title (see Figure 5.1), where its default value is "Easy Interface".
- **apps** is used to change the set of tools to be listed in the tools menu, the syntax is explained in Section 5.1.1.
- **examples** is used to change the set of examples that are shown in the file-manager, the syntax is explained in Section 5.1.2.
- **outline** is use to control if the **Outline** components (see Figure 5.1) is visible or not, the possible values are "**on**" and "**off**".
- **outlineserver** and **outlineapp** are used to indicate which tool to use for generating the content of the outline, the syntax is explained in Section 5.1.3.
- **language** is used to set the programming language in which programs are written (for syntax highlighting purposes), the syntax is explained in Section 5.1.4.

In the next section we give some example values for these fields.

5.1.1 Tools Menu

The tool menu, the combo-box next to the Run button in Figure 5.1, includes a list of tools that can be executed by the user. This list can be modified by setting the value of the field **apps** in the configuration file. This value is an array of JSON records of the form

```
{ server: SRV, apps: APPSLIST }
```

where **SVR** is a URL to an EASYINTERFACE server and **APPSLIST** is an array of tool identifiers (see [APP]). **APPSLIST** can also be the special value **_ei_all** which refers to all tools of the corresponding server. If this field is not provided, all tools from the default server will be included in the tools menu.

EXAMPLE 5.1. *The following is a possible value for field* ***apps****:*

```
apps: [ {server: "http://domain1/ei/server, apps: ["costa", "mhp"]},
        {server: "http://domain2/ei/server, apps: "_ei_all"} ]
```

It takes the tools identified by ***costa*** *and* ***mhp*** *from the* EASYINTERFACE *server* ***http://domain1/ei/server****, and all tools available at the* EASYINTERFACE *server* ***http://domain2/ei/server****.*

5.1.2 File-Manger

In the file-manager area, of Figure 5.1, we can see a tree-view that represents programs on which tools can be applied, etc. The one with the name **User_Projects** corresponds to programs that are created by the user; and the rest are predefined set of examples. This set of examples can be modified by setting the value of the field **examples** in the configuration file. This value is an array of JSON records of the form

```
{ server: SRV, examples: EXLIST }
```

where **SVR** is a URL to an EASYINTERFACE server and **EXLIST** is an array of example set identifiers (see [EXSET]). **EXLIST** can also be the special value **_ei_all** which refers to all example sets of the corresponding server. If this field is not provided, all example sets from the default server will be included.

EXAMPLE 5.2. *The following is a possible value for field* ***examples****:*

```
examples: [ {server: "http://domain1/ei/server, examples: ["cost"]},
            {server: "http://domain2/ei/server, examples: "_ei_all"} ]
```

It takes the example set identified by ***cost*** *from the* EASYINTERFACE *server* ***http://domain1/ei/server****, and all example sets available at the* EASYINTERFACE *server* ***http://domain2/ei/server****.*

Note that the file-manager has a context menu (use the mouse right-click to open it) with options for: creating new files; running tools; creating outline; cloning and committing to GitHub repositories, etc.

5.1.3 Outline

The outline area of Figure 5.1 includes a tree-view that represents information on some programs entities, e.g., methods, classes, etc. The actual values in this tree and its structure depend very much on the intended use of EASYINTERFACE, and thus, it is completely configurable – apart from the possibility of hiding it by setting **outline** to "**off**". The idea is that the user will select some of the entries in this tree, and then they will be passed to the tool that we run (see Section 3.4 and [CMDLINEAPP]).

The actual content of the outline is not generated by the web-client, but rather by an external tool that is installed on some EASYINTERFACE server that we refer to as the *outline tool*. It is like any other tool but typically non-visible. The exact work-flow for generating an outline is as follows:

1. The user clicks on the `Refresh Outline` button to generate an outline for the currently opened tab (in the code editor), or select the `Refresh Outline` option from the context menu of the file-manager to generate an outline for all programs in the corresponding sub-tree;

2. The web-client sends a request to execute the *outline tool*, passing it all files of interest;

3. The *outline tool* processes the input files and generates (on the standard output) some XML structure that represents the content of the outline, which is sent back to the client; and

4. The web-client converts this XML into a tree view as shown in Figure 5.1.

The fields **outlineserver** and **outlineapp** in the configuration file can be used to indicate which tool to use for generating the outline content. The default value of **outlineserver** is the default server, and the one of **outlineapp** is **coutline** which a simple example tool that comes with EASYINTERFACE to generate outlines for C programs.

As for the outline content, it must be *a sequence of XML environments* that adhere to the following syntax, each element (i.e., tree) in this sequence will be show at the root level in the outline area:

OUTLINE

```
<category version=[VERSION]? text=[NODETEXT] value=[NODEVAL]
          selectable=[BOOL]? icon=[URL]?>
    [OUTLINE]*
</category>
```

SINCE VERSION: 1.0

DESCRIPTION:

Defines a tree that represent (part of) an outline. The outer **category** tag is the root of this tree, and the inner [OUTLINE]* are its children. The meaning of the different attributes is as follows:

- `version` indicates the version of the outline structure, which is 1.0 by default.
- `text` is the text to be show for that node.

- `value` is the value to passed to a tool if that node is selected.
- `selectable` indicates if this node can be selected. Its default value is `true`. Such nodes are used to divide the tree in several logical categories. Note that, in some clients, nodes might be still selectable even if the value is `false`, however, in such case they will not be passed to the tool.
- `icon` is a URL to an alternative icon to be used for that node.

NODETEXT
A string.

NODEVAL
[a-z,A-Z,0-9,-,_,:,.]+

BOOL
(**true** | **false**)

URL
A valid **http** or **https** URL.

EXAMPLE 5.3. *The following is an example of a simple outline for an* ABS *program:*

```
<category text="Module PingPong" selectable="false">
 <category text="Class PingImpl" selectable="false">
  <category text="initPing" value="PingImpl.initPing" selectable="true" />
  <category text="ping" value="PingImpl.ping" selectable="true" />
 </category>
 <category text="Class PongSessionImpl" selectable="false">
  <category text="initPongSession" value="PongSessionImpl.initPongSession"
    selectable="true" />
  <category text="pong" value="PongSessionImpl.pong" selectable="true" />
 </category>
 <category text="Class PongImpl" selectable="false">
  <category text="hello" value="PongImpl.hello" selectable="true" />
  <category text="sessionFinished" value="PongImpl.sessionFinished"
    selectable="true" />
 </category>
 <category text="main" value="main" selectable="true" />
</category>
```

It represent a module with three classes, each with several methods. In addition, it includes a node representing the **main** *block of the module. Note that only methods have the attribute selectable set to* `true`*. The result as viewed in the web client is depicted in Figure 5.2.*

5.1.4 Code Editor

The **Code Editor** in Figure 5.1 is the area were programs can be edited. It provides basic functionality of an editor in general such as *search and replace* and *syntax highlighting*. It is implemented using the CodeMirror [3] library. The syntax highlight is configurable

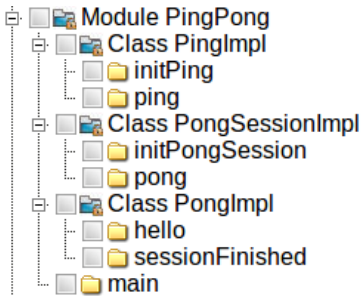

Figure 5.2: Outline example

via the field **`language`** in the configuration file. Its value should be a valid CodeMirror syntax highlighting mode which is a MIME[1] content-type attribute, for example: *text/x-csrc* (C), *text/x-c++src* (C++), *text/x-java* (Java), *text/x-csharp* (C#), *text/x-python* (Python), etc. See `http://codemirror.net/mode` for the list of available modes. By default it is *text/x-csrc*, i.e., C programs. Note that CodeMirror provides an easy way to add new rules for syntax highlighting as described in `http://codemirror.net/doc/manual.html#modeapi`. In such case, the new files should be incorporated in the local copy of CodeMirror at **`clients/web/lib/codemirror/mode`**.

5.1.5 Console

The console area is where the output of a tool can be printed, which includes several tabs for better organization of the output. It is implemented using jQuery [7].

5.1.6 Settings Section

The **`Settings`** window in Figure 5.1 includes a section for each available tool. In each section the parameters of the corresponding tool (see [**PARAMETERS**]) are viewed in a graphical way, e.g., using combo-boxes, radio buttons, etc. One can select also a profile, which automatically sets the parameters to some predefined values (see [**PROFILES**]). The *Default* profile sets all parameters to their default value.

5.1.7 Help Section

A click on the `Help` button in Figure 5.1 will open a window that includes several help sections, one for each tool. The content of each section is taken from the corresponding tool's help provided in configuration file of the EasyInterface server (see [**APPHELP**]).

5.1.8 Other Features

The web-client can be started directly with: some files (from the file-manager) opened, a tool selected from the tools menu, and the corresponding parameter set using some profile. This is useful when writing, for example, tutorials for these tools, where one can link directly to the web-client with a corresponding example opened and a tool ready to be applied. This can be done using a URL of the following form:

[1] `http://en.wikipedia.org/wiki/MIME`

```
http://domain/clients/web/?app=APPID&profile=PROFILE&file=PATH
```

where: (*i*) the value of **app** is an identifier **APPID** of the tool to be selected; (*ii*) the value of each **file** parameter (there can be several) is a **PATH** to be opened in the editor — it should be full path from the root of the file-manager, e.g., "**/Examples_1/iterative/sum.c**"; and (*iii*) the value of **profile** is a profile identifier **PROFILE** to be used for parameter values (see [**PROFILE**]).

5.2 Other Clients

Although we have presented the web-client only, which is recognized as the most important one in our objectives, other clients might be of interest as well. For example, one can think of developing an Eclipse plugin to communicate with a server within Eclipse, or a script (e.g., Python) to communicate with a server within a terminal.

Chapter 6

The EasyInterface Output Language

In this chapter we describe a text-based output language that allows tools to view their output in a graphical way, e.g., highlighting lines, adding markers, defining on-click actions, etc. The main advantage of this language is that it does not require any knowledge on GUI or WEB programming. Some clients, e.g., the web-client described in Section 5.1, are supposed to support this language. This is done by developing a corresponding interpret that renders the effect of the corresponding commands in the respective environment (e.g., a web browser).

The rest of this chapter is organized as follows: in Section 6.1 we first give a general overview of the design of this language; in Section 6.2 we describe the syntax and semantics of this language; in Section 6.3 we include some details that we left out of Section 6.2 for readability (we refer the reader to these parts in Section 6.2); and in Section 6.4 we give some examples to the different parts of the language.

6.1 General Overview

The idea behind the EasyInterface output language is that a tool should just print, on the standard output, how it wants to view the output using some high-level description, and leave the details to an interpreter that converts this description to graphical output. This way we move the complexity of constructing a GUI from the tool to the interpreter, and thus free developers of tools from mastering any WEB or GUI related libraries. To simplify the processing of such output, we should use some structured format. In our case we opt for XML, but we could use any other structured formatting. e.g., JSON.

The EasyInterface output language assumes that the environment in which it is interpreted includes:

(*i*) A "`Code Editor`" where programs can be edited, typically a tab for each file;

(*ii*) A "`File Manager`" that includes a tree-view of all user files and predefined examples;

(*iii*) A `Console` where output can be printed in different formats (it might include also several consoles, e.g., several tabs).

An output in the EasyInterface output language is an XML structure of the following form:

```
<eiout>
 <eicommands>
    [EICOMMAND]*
 </eicommands>
 <eiactions>
    [EIACTION]*
 </eiactions>
</eiout>
```

where

(*i*) **eiout** is the outermost tag that encapsulates all commands and actions;

(*ii*) [EICOMMAND]* is a list of commands to be executed; and

(*iii*) [EIACTION]* is a list of actions to be declared. Actions correspond to interactions with the user.

Typical examples of [EICOMMAND] are: *print a text on the console*, *highlight lines 5-10*, *add marker at line 5*, etc. Typical examples of [EIACTION] are: *when the user clicks on line 13*, *highlight lines 20-25*, *when the user clicks on some text*, *open a dialog box with some message*, etc.

In the next section we give a detailed specification of this language. Note that currently the language includes some commands and actions of interest, that we needed for our tools in the ENVISAGE project, however, it is design in a way that is easily extensible to include more commands an actions (interpreters on the client side should be modified to support such extensions).

6.2 Syntax and Semantics

An output in the EASYINTERFACE output language is an XML structure that adhere to the syntax of [EIOUT] that is described below. You can follow the links in its definition in order to get the definitions of its different parts. The semantics of each is fully specified, and when needed we refer the reader to clarifying examples.

IMPORTANT: note that the output must be a valid XML structure, thus, in what follows, whenever we need to include *plain text* in that output, such text should be enclosed in a `<![CDATA[ the-text-goes-here ]]>` environment (see Example 6.4).

EIOUT

```
<eiout version=[VERSION]? >
  [EICOMMANDS]*
  [EIACTIONS]*
</eiout>
```

SINCE VERSION: 1.0

DESCRIPTION:

This is the main environment of the output, it includes several lists of command environments [EICOMMANDS], and several lists of action environments [EIACTIONS]. Commands are executed first, in the given order, and then actions are executed in the given order as well. The version attribute indicates the version of the output language that is used, which is 1.0 by default.

EICOMMANDS

```
<eicommands dest=[PATH]? outclass=[OUTCLASS]? >
  [EICOMMAND]*
</eicommands>
```

SINCE VERSION: 1.0

DESCRIPTION:

A list of commands to be performed. The attribute dest is the destination file on which the command is applied (if needed), it should be the full path to that file as provided by the EASYINTERFACE server. E.g., when highlighting a line we might want to highlight a line in one file or another. If dest is not specified, then the commands will be applied to the file that is currently active, e.g., if the client includes a code editor with several tabs, one for each file, the command will be applied to the active tab. If none is active then the behavior is not specified. The attribute outclass specifies the *output class* of the commands in this environment, that is, the nature of the corresponding output generated by the commands, e.g., error, information, warning, etc. The effect of this attribute is not fully specified and it depends on the client and the actual implementation of the interpreter. All commands inside this environment inherit the values of outclass and dest, and each can override them.

EIACTIONS

```
<eiactions dest=[PATH]? autoclean=[BOOL]?>
  [EIACTION]*
</eiactions>
```

SINCE VERSION: 1.0

DESCRIPTION:

A a list of actions to be declared. An action typically executes a list of [EICOMMANDS] when the user interacts with the interface in some predetermined way, e.g., *when the user clicks on line 30, highlight lines number 12 and 16*. We say the an action is *performed* as a response to the user interaction. If the user interacts again with the interface, according to what is specified in the action, then the action is *unperformed* if possible (when the corresponding commands support the *undo* operation), e.g., in the above example if the user clicks again on Line 30 again, the highlights of lines 12 and 16 are turned off.

Before *performing* an action, the last *performed* action is *unperformed* first. This

behavior can be disabled by setting the autoclean attribute to "false". All actions inside this environment inherit the value of autoclean, and each can override it. The attribute dest and outclass are as in the case of commands (see the description of [EICOMMANDS]).

EICOMMAND

```
(
  [PRINTONCONSOLECOMMAND]
| [HIGHLIGHTLINESCOMMAND]
| [DIALOGBOXCOMMAND]
| [WRITEFILECOMMAND]
| [SETCSSCOMMAND]
| [CHANGECONTENTCOMMAND]
| [ADDMARKERCOMMAND]
| [ADDINLINEMARKERCOMMAND]
| [DOWNLOADCOMMAND]
)
```

SINCE VERSION: 1.0

DESCRIPTION:

A command in the EASYINTERFACE output language, briefly:

- [PRINTONCONSOLECOMMAND] can be used to print on the console.
- [HIGHLIGHTLINESCOMMAND] can be used to highlight lines in the code editor.
- [DIALOGBOXCOMMAND] can be used to open a dialog window with a corresponding message.
- [WRITEFILECOMMAND] can be used to add a file (and a corresponding content) to the file-manager.
- [SETCSSCOMMAND] can be used to change the CSS properties of some elements that were previously generated (e.g., content in HTML)
- [ADDMARKERCOMMAND] can be used to add a marker next to a line in the code editor.
- [ADDINLINEMARKERCOMMAND] can be used to add a line widget (an inlined marker) in the code editor.
- [DOWNLOADCOMMAND] can be used to download a file that was previously generated by a tool.
- [CHANGECONTENTCOMMAND] can be used to modify a content that has been previously generated.

EIACTION

```
(
  [ONCODELINECLICKACTION]
| [ONCLICKACTION]
)
```

SINCE VERSION: 1.0

DESCRIPTION:

An action in the EASYINTERFACE output language, briefly:

- [ONCODELINECLICKACTION] can be used to perform an action when the user clicks on a line in the code editor.
- [ONCLICKACTION] can be used to perform an action when the user clicks on a previously generated text (a DOM element in general).

PRINTONCONSOLECOMMAND

```
<printonconsole outclass=[OUTCLASS]? consoleid=[CONSOLEID]?
    consoletitle=[STRING]?>
  [CONTENT]+
</printonconsole>
```

SINCE VERSION: 1.0

DESCRIPTION:

Prints the content described by the [CONTENT] environments on the console that has an identifier consoleid. If consoleid is not specified, the output goes to the default console. If consoleid is specified but there is no console with such an identifier, the console is created and consoletitle (if specified) is used as its title. The attribute outclass is as described in [EICOMMANDS].

See Example 6.1.

HIGHLIGHTLINESCOMMAND

```
<highlightlines outclass=[OUTCLASS]? dest=[PATH]?>
  [LINES]*
</highlightlines>
```

SINCE VERSION: 1.0

DESCRIPTION:

Highlights the lines specified by [LINES] in the file dest. The attribute outclass is as described in [EICOMMANDS].

See Example 6.2.

DIALOGBOXCOMMAND

```
<dialogbox outclass=[OUTCLASS]? boxtitle=[STRING]? boxwidth=[INT]?
    boxheight=[INT]?>
  [CONTENT]+
</dialogbox>
```

SINCE VERSION: 1.0

DESCRIPTION:

Opens a dialog box with the content specified by the [CONTENT] environments. The value of boxtitle, if specified, is used as a title for the dialog box. The attributes boxwidth and boxheight can be used to set the size of the window. The attribute outclass is as in [EICOMMANDS].

See Example 6.3.

WRITEFILECOMMAND

```
<writefile filename=[PATH] overwrite=[BOOL]>
  [TEXT]
</writefile>
```

SINCE VERSION: 1.0

DESCRIPTION:

Creates a new file it in the file-manager, using the path specified by filename. The file's content is set to the value of [TEXT]. If the file exists, and overwrite is true, the content is replaced otherwise a new file is created with a new name. The default value of overwrite is false.

See Example 6.4.

SETCSSCOMMAND

```
<setcss>
  [ELEMENTS]
  [CSSPROPERTIES]
</setcss>
```

SINCE VERSION: 1.0

DESCRIPTION:

Changes the CSS properties, as specified by [CSSPROPERTIES], of all elements that match the selectors in [ELEMENTS]. There must be exactly one [ELEMENTS] environ-

ment and one [CSSPROPERTIES] environment. The elements are selected from those that were previously generated by other commands.

See Example 6.5.

CHANGECONTENTCOMMAND

```
<changecontent action=[POSITION]>
  [ELEMENTS]
  [CONTENT]
</changecontent>
```

SINCE VERSION: 1.0

DESCRIPTION:

Modifies the content of all elements that match the selectors in [ELEMENTS], using [CONTENT]. These elements are typically selected from those generated by previously executed commands. The attribute action indicates how to incorporate [CONTENT] in the current one, i.e., replace, append or prepend.

See Example 6.6.

ADDMARKERCOMMAND

```
<addmarker outclass=[OUTCLASS]? dest=[PATH]? boxtitle=[STRING]?
           boxwidth=[INT]? boxheight=[INT]?>
  [LINES]
  [CONTENT]*
</addmarker>
```

SINCE VERSION: 1.0

DESCRIPTION:

Adds a marker next to each line that is specified in [LINES]. The column information from each [LINE] in [LINES] is ignored. All markers are associated with the content given by the [CONTENT] environments, as a tooltip for example. If the client allows expanding the tooltip to a dialog window, the attributes boxtitle, boxwidth and boxheight can be used to set the properties of the corresponding window (see [DIALOGBOXCOMMAND]). If a line is already associated with a marker, then all [CONTENT] environments should be appended to the current content. The attributes dest and outclass are as described in [EICOMMANDS].

See Example 6.7.

ADDINLINEMARKERCOMMAND

```
<addinlinemarker outclass=[OUTCLASS]? dest=[PATH]?>
  [LINES]
  [CONTENT]*
</addinlinemarker>
```

SINCE VERSION: 1.0

DESCRIPTION:

Adds an inline marker (a line widget) for each line that is specified by [LINES]. All line widgets will include the content specified by the [CONTENT] environments. The column information from each [LINE] in [LINES] is ignored. If a line is already associated with a marker, then all [CONTENT] environments should be appended to the current content. The attributes dest and outclass are as described in [EICOMMANDS].

See Example 6.8.

DOWNLOADCOMMAND

```
<download execid=[EXECID] filename=[FILE] />
```

SINCE VERSION: 1.0

DESCRIPTION:

When a tool that runs on an EASYINTERFACE server X generates this command, the effect should be downloading the file specified in filename, from the EASYINTERFACE server X, using the execution identifier execid. See sections 4.1.2 and 4.2.4 for more information on downloading files from an EASYINTERFACE server.

See Example 6.9.

ONCODELINECLICKACTION

```
<oncodelineclick dest=[PATH]? autoclean=[BOOL]? outclass=[OUTCLASS]?>
  [LINES]
  [CONTENT]*
  [EICOMMANDS]
</oncodelineclick>
```

SINCE VERSION: 1.0

DESCRIPTION:

Adds (special) markers at the code lines specified by [LINES], such that when any is clicked the commands in [EICOMMANDS] are performed, and if clicked again they are unperformed (see detailed description in [EIACTIONS]). The content given by the [CONTENT] environments is associated with the markers (as a tooltip for example). If

a line is already associated with such an action, then [CONTENT] should be appended to the current content and [EICOMMANDS] should be accumulated to the current ones. The attributes dest, outclass and autoclean are as described in [EIACTIONS]. Moreover, the [EICOMMANDS] environment inherits the dest and outclass attributes of this environment.

See Example 6.2.

ONCLICKACTION

```
<onclick outclass=[OUTCLASS]? autoclean=[BOOL]? >
  [ELEMENTS]
  [EICOMMANDS]
</onclick>
```

SINCE VERSION: 1.0

DESCRIPTION:

A click on any element that matches any selector from [ELEMENTS] will execute the commands declared in [EICOMMANDS], and if clicked again they are unperformed (see detailed description in [EIACTIONS]). If the element is already associated with an action, then [EICOMMANDS] should be accumulated to the current ones. The attributes dest, outclass and autoclean are as described in [EIACTIONS]. Moreover, the above [EICOMMANDS] environment inherits the dest and outclass attributes of this environment.

See Example 6.5.

LINES

```
<lines>
    [LINE]+
</lines>
```

SINCE VERSION: 1.0

DESCRIPTION:

A group of lines, typically used to specify the lines affected by an [EICOMMAND] or an [EIACTION].

LINE

```
<line from=[INT] to=[INT]? fromch=[INT]? toch=[INT]? />
```

SINCE VERSION: 1.0

DESCRIPTION:

A region (of lines) typically used to specify the region on which the effect of an [EICOMMAND] or an [EIACTION] is applied:

- from is the start line.
- to is the end line.
- fromch is the character (i.e., column number) where the first line starts.
- toch is the character (i.e., column number) where the last line ends.

The default value of to is as the value of from. The default value of fromch is 0, and of toch is the end of the line.

ELEMENTS

```
<elements>
  [SELECTOR]*
</elements>
```

SINCE VERSION: 1.0

DESCRIPTION:

Set of selectors (of DOM elements)

SELECTOR

```
<selector value=[STRING] />
```

SINCE VERSION: 1.0

DESCRIPTION:

The attribute value must be a valid selector as in jQuery (see `https://jquery.com`). It is used to match some DOM elements.

CSSPROPERTIES

```
<cssproperties>
  [CSSPROPERTY]*
</cssproperties>
```

SINCE VERSION: 1.0

DESCRIPTION:

A set of CSS properties.

CSSPROPERTY

```
<cssproperty name=[CSSNAME] value=[CSSVAL] />
```

SINCE VERSION: 1.0

DESCRIPTION:

A CSS property. The attributes `name` and `value` should correspond to valid CSS properties.

CONTENT

```
<content format=[TEXTFORMAT]? execid=[EXECID]? ext=[STRING]?
         action=[POSITION]? refreshrate=[INT]?>
  [TEXT]
</content>
```

SINCE VERSION: 1.0

DESCRIPTION:

A text [`TEXT`] given in a specific `format`, that is supposed to be viewed to the user. If the attribute `format` is not provided, then it is assumed to be in `"text"` format (plain text). This content should be viewed to the user in the specified format. The supported formats are:

- `"text"`, for plain text;
- `"html"`, for HTML;
- `"svg"`, for Scalable Vector Graphics [26]; and
- `"graph"`, for drawing 2D graphs – see Section 6.3.1 for the exact format.

The rest of attributes can be used to associate the content with an output stream as follows (first see sections 4.1.2 and 4.2.5 to understand the notions behind streaming before you continue):

- `execid` is the execution identifier;
- `ext` is the extension of output chunk files to be retrieved;
- `action` indicates how to incorporate new output chunks to the current content (i.e., `replace`, `append` or `prepend`); and
- `refreshrate` is a time interval (in seconds) to be used for refreshing the output. If the value is not specified then refreshing the output should be on demand.

The client should stop refreshing when the background processes of the corresponding stream have terminated, or when the user asks to stop them explicitly.

CONSOLEID

([a-z,A-Z,0-9,-,_]+ | **new** | **default**)

The value **new** means a new console, we cannot refer to this console later. The value **default** means the default console of the client.

EXECID
[a-z,A-Z,0-9,-,_]+

PATH
A path to a file, including the file name. There are two forms, the first one is a full path including the temporal directory name that is created by the server, e.g., "`/tmp/easyinterfae_XYZ/_ei_files/dir1/dir2/file.c`". The client should simply ignore the prefix "`/tmp/easyinterfae_XYZ/_ei_files/`", i.e., the value in this case is "`dir1/dir2/file.c`". The second form does not include the temporal directory prefix.

FILE
A file name, without the path. Substrings `"/"`, `"\"`, and `".."` are not allowed.

VERSION
$x.y$, where x is the major version number and y is the minor one, e.g. 1.0, 1.1, etc.

OUTCLASS
(**none** | **info** | **warning** | **error**)

BOOL
(**true** | **false**)

INT
An integer

STRING
A string

TEXT
Free text.

TEXTFORMAT
(**text** | **html** | **svg** | **graph**)

CSSNAME
A valid name for a CSS property.

CSSVAL
A valid value for a corresponding CSS property.

POSITION
(**prepend** | **append** | **replace**)

6.3 Other Details

6.3.1 The Graph Format

The [CONTENT] tag, of the EasyInterface output language, supports drawing 2*D* graphs using the value graph for the format attribute. In this section we describe the syntax of such graphs.

Let us start by defining the notion of a graph we are discussing in this section. A function is a list of pairs (x, y) that defines some points in the plane, and a plot of such a function is a drawing that connects these points. A 2*D* graph is a collection of functions that share all x points, i.e., we can imagine it as drawing several functions using the same xy-axes. For the sake of compact representation, if we have n functions, every point in a graph can be represented as $(x, y_1, \ldots, y_n)$ where (x, y_i) corresponds to the i-th function.

A graph is a sequence of JSON records where each record describes a graph via the following fields:

- **"title"**: a title to be used for the graph;
- **"x-title"**: a title to be used for the x-axes (horizontal);
- **"y-title"**: a title to be used for the y-axes (vertical);
- **"f-titles"**: an array of strings, where the i-th string is a title for the i-th function.
- **"values"**: an array of points, where each point is an array $[x, y_1, \ldots, y_n]$ such that (x, y_i) corresponds to a point defining the i-th function.

In addition, since the [CONTENT] tag can define several graphs, we also provide the possibility of grouping graphs into logical groups and assigning them labels. Clients should allow viewing graphs of interest as follows: a graph is viewed if it belongs to at least one of the selected groups and has at least one of the selected labels. Defining groups and labels is done using the following fields in the corresponding JSON record:

- **"groups"**: array of groups, where each group is simply a string.
- **"labels"**: array of labels, where each label is simply a string.

See Example 6.3 for an example of 2*D* graphs.

6.4 Examples

In this section we give some examples for the different commands and actions of the EasyInterface output language. Note that they are referred to from the definitions of the corresponding commands in Section 6.2.

EXAMPLE 6.1. *The following is an example of [PRINTONCONSOLECOMMAND] using different [CONTENT] environments with different formats:*

```
<printonconsole consoleid="1" consoletitle="Welcome">
  <content format="text">
    Hello World
  </content>
  <content format="html">
    <span style="color: red;">Hello</span> World
  </content>
  <content format="svg">
    <svg height="100" width="100">
      <circle cx="50" cy="50" r="40" fill="red" />
    </svg>
  </content>
</printonconsole>
```

Its execution in the web-client generates the output depicted in Figure 6.1.

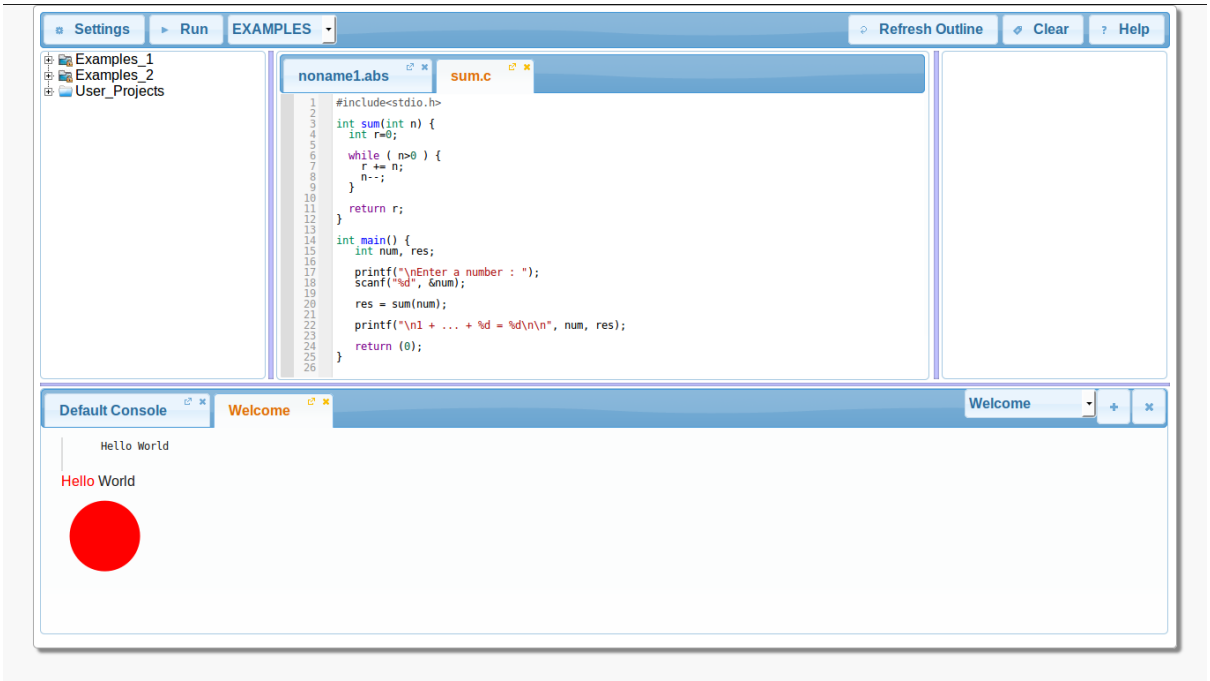

Figure 6.1: Output of Example 6.1

EXAMPLE 6.2. *The following is an example of [ONCODELINECLICKACTION] which executes two [HIGHLIGHTLINESCOMMAND], each with different* `outclass`*:*

```
<eiactions>
 <oncodelineclick>
  <lines> <line from="3" /> </lines>
  <eicommands>
   <highlightlines outclass="info">
    <lines> <line from="2" to="4" /> </lines>
   </highlightlines>
   <highlightlines outclass="warning">
```

```
    <lines> <line from="6" fromch="4" toch="8" /> </lines>
   </highlightlines>
  </eicommands>
 </oncodelineclick>
</eiactions>
```

Its execution in the web-client generates the output depicted in Figure 6.2. Note that a click on the arrow (in the left-side gutter) executes the commands and another click undo their corresponding effect.

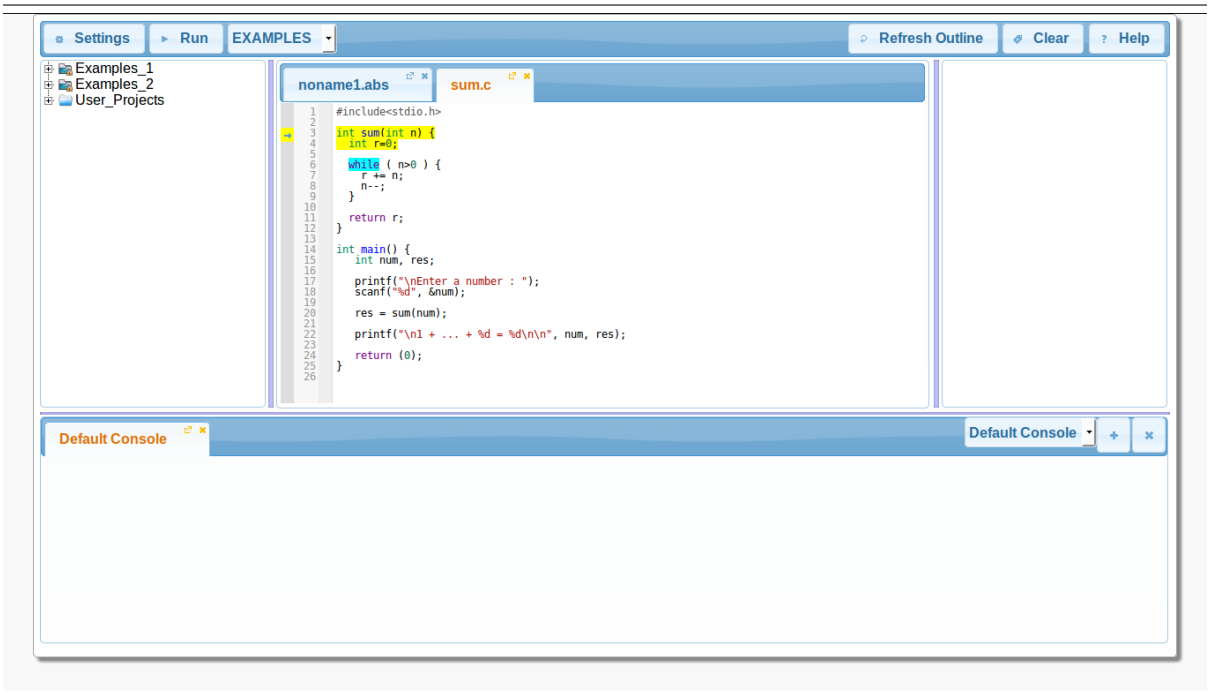

Figure 6.2: Output of Example 6.2

EXAMPLE 6.3. *The following is an example of [**DIALOGBOXCOMMAND**], using a [**CONTENT**] environment with **graph** format.*

```
<dialogbox boxtitle="CPU use" boxwidth="800" boxheight="500">
  <content format="graph">
    { "title":"BFS - Acer G5453",
      "f-titles":["Time","% CPU","% Mem"],
      "y-title":"%",
      "x-title":"Time",
      "groups":["BFS"],
      "labels":["Acer","G5453"],
      "values":[[1,22,43],[2,40,47],[3,82,88],[4,40,75]]
    }
    { "title":"BFS - Acer B12",
      "f-titles":["Timee","% CPU","% Mem"],
      "y-title":"%",
      "x-title":"Time",
      "groups":["BFS"],
```

```
      "labels":["Acer","B12"],
      "values":[[1,42,66],[2,65,47],[3,99,91],[4,68,92]]
    }
  </content>
</dialogbox>
```

Its execution in the web-client generates the output depicted in Figure 6.3.

Figure 6.3: Output of Example 6.3

EXAMPLE 6.4. *The following is an example of [WRITEFILECOMMAND], which adds a new file to the file-manager:*

```
<writefile filename="dir1/newfile.cpp" overwrite="false">
<![CDATA[
#include <iostream>
using namespace std;
int main(){
  cout << "Hello World!" << endl;
  return 0;
}
]]>
</writefile>
```

Note the of <![CDATA[...]]>, this is necessary due to the use of plain-text with special characters. Its execution in the web-client generates the output depicted in Figure 6.4.

EXAMPLE 6.5. *The following is an example of [SETCSSCOMMAND], together with an action [ONCLICKACTION] which changes the size of some text when it is clicked:*

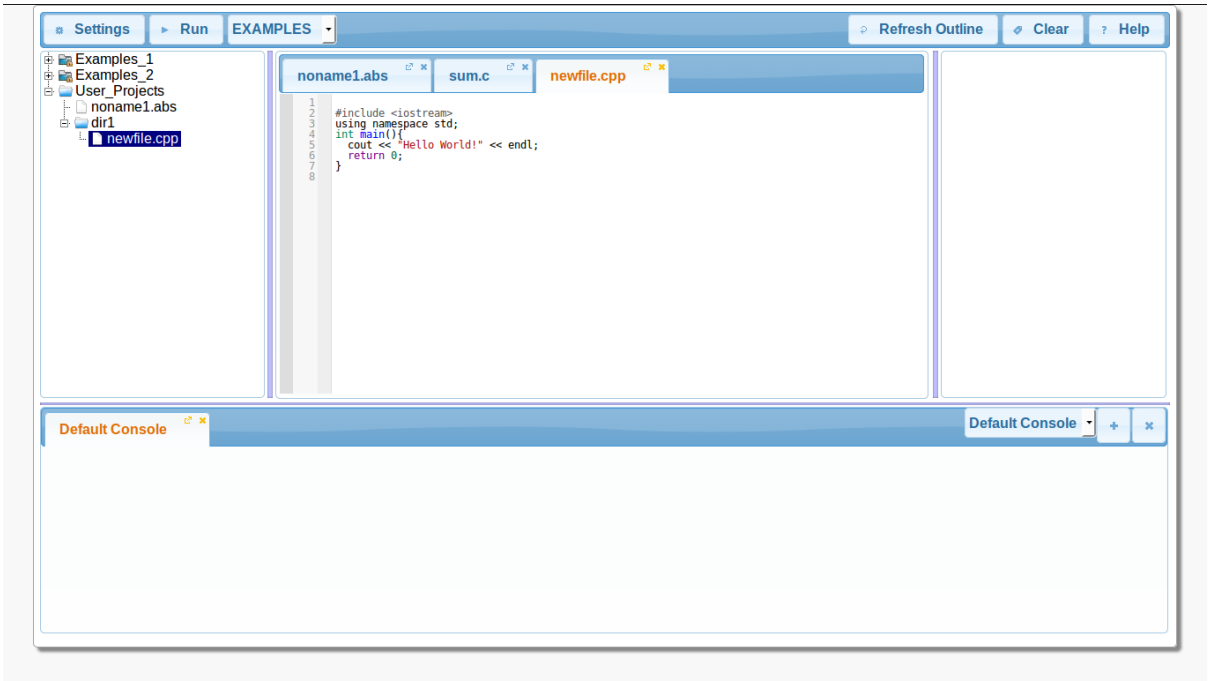

Figure 6.4: Output of Example 6.4

```
<eicommands>
 <printonconsole>
  <content format="html">
   <div id="wrap">
    <span>Some text</span><br/>
    <span id="text">Click me!</span><br/>
   </div>
  </content>
 </printonconsole>
</eicommands>
<eiactions>
 <onclick>
  <elements>
   <selector value="#text" />
  </elements>
  <eicommands>
   <setcss>
    <elements>
     <selector value="#text" />
    </elements>
    <cssproperties>
     <cssproperty name="font-size" value="30px" />
    </cssproperties>
   </setcss>
  </eicommands>
 </onclick>
</eiactions>
```

*Its execution in the web-client generates the output depicted in Figure 6.5 – it includes the result after clicking on the text "**Click me!**".*

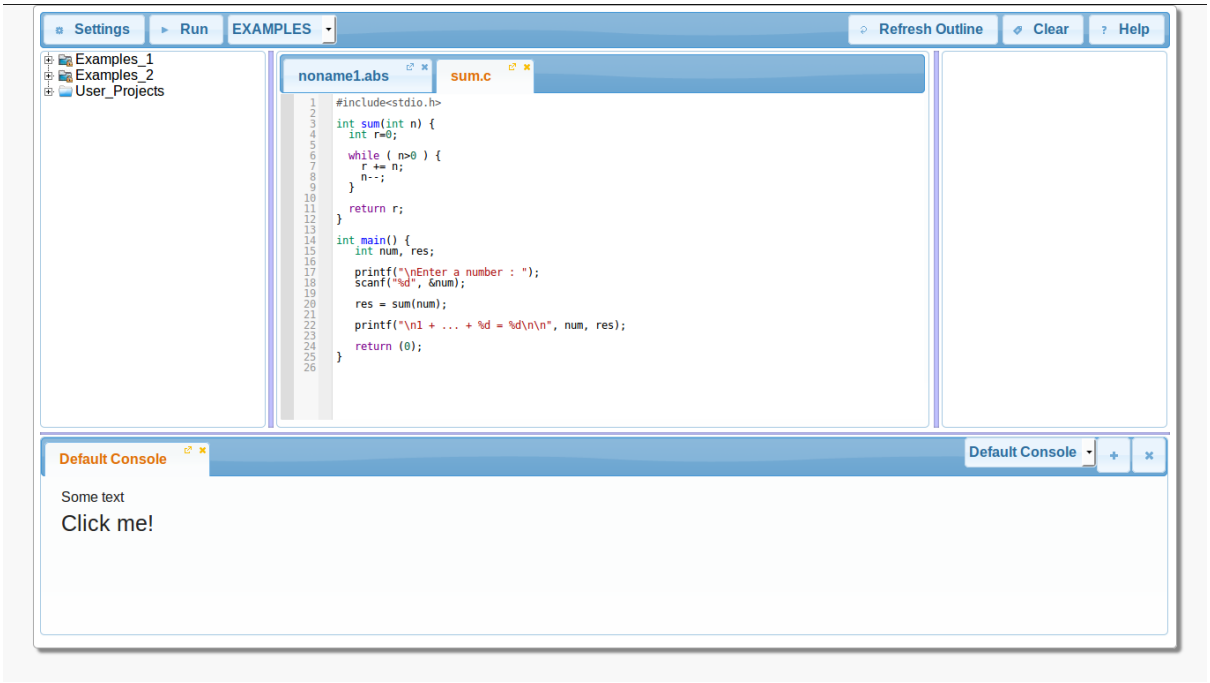

Figure 6.5: Output of Example 6.5

EXAMPLE 6.6. *The following is an example of [**CHANGECONTENTCOMMAND**]. Assuming that we add the following command to the list of commands of the [**ONCLICKACTION**] in Example 6.5, it will add some text to the HTML tag with identifier **wrap** when "**Click me!**" is clicked:*

```
<changecontent action="append">
 <elements>
  <selector value="#wrap"/>
 </elements>
 <content format="html">
  <span style="color:red;">New Text added </span><br/>
 </content>
</changecontent>
```

*Its execution in the web-client generates the output depicted in Figure 6.6 (after clicking on "**Click me!**").*

EXAMPLE 6.7. *The following is an example of [**ADDMARKERCOMMAND**], using different **outclass** values:*

```
<addmarker outclass="info">
 <lines>
  <line from="2" />
 </lines>
 <content format="text">
  Information
```

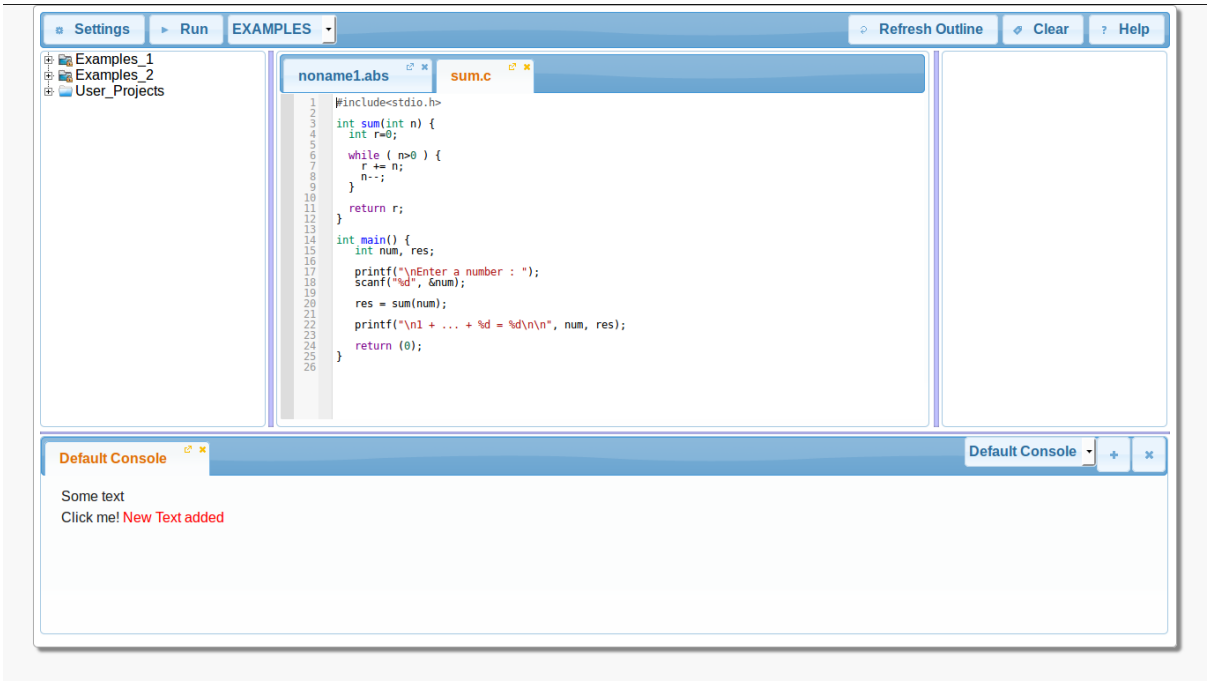

Figure 6.6: Output of Example 6.6

```
 </content>
</addmarker>
<addmarker outclass="warning">
 <lines>
  <line from="4" />
 </lines>
 <content format="text">
  Warning
 </content>
</addmarker>
<addmarker outclass="error">
 <lines>
  <line from="6" />
 </lines>
 <content format="text">
  Error
 </content>
</addmarker>
```

Its execution in the web-client generates the output depicted in Figure 6.7.

EXAMPLE 6.8. *The following is an example of [`ADDINLINEMARKERCOMMAND`] using different* `outclass` *values:*

```
<addinlinemarker outclass="info">
 <lines>
  <line from="2" />
 </lines>
```

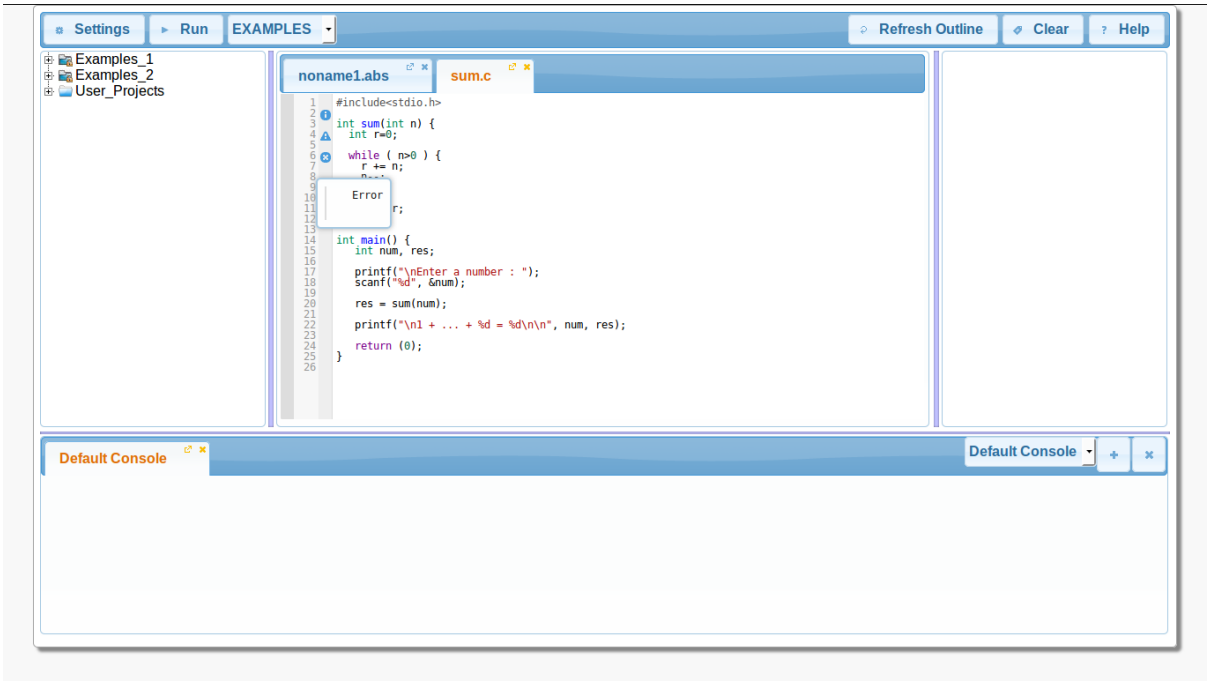

Figure 6.7: Output of Example 6.7

```
 <content format="text">
  Information
 </content>
</addinlinemarker>
<addinlinemarker outclass="warning">
 <lines>
  <line from="4" />
 </lines>
 <content format="text">
  Warning
 </content>
</addinlinemarker>
<addinlinemarker outclass="error">
 <lines>
  <line from="6" />
 </lines>
 <content format="text">
  Error
 </content>
</addinlinemarker>
```

Its execution in the web-client generates the output depicted in Figure 6.8.

EXAMPLE 6.9. *The following is an example of [**DOWNLOADCOMMAND**], to download a file called* **`download.test`** *generated by a tool execution with* execution identifier ***xV54fga****:*

```
<download execid="xV54fga" filename="download.test" />
```

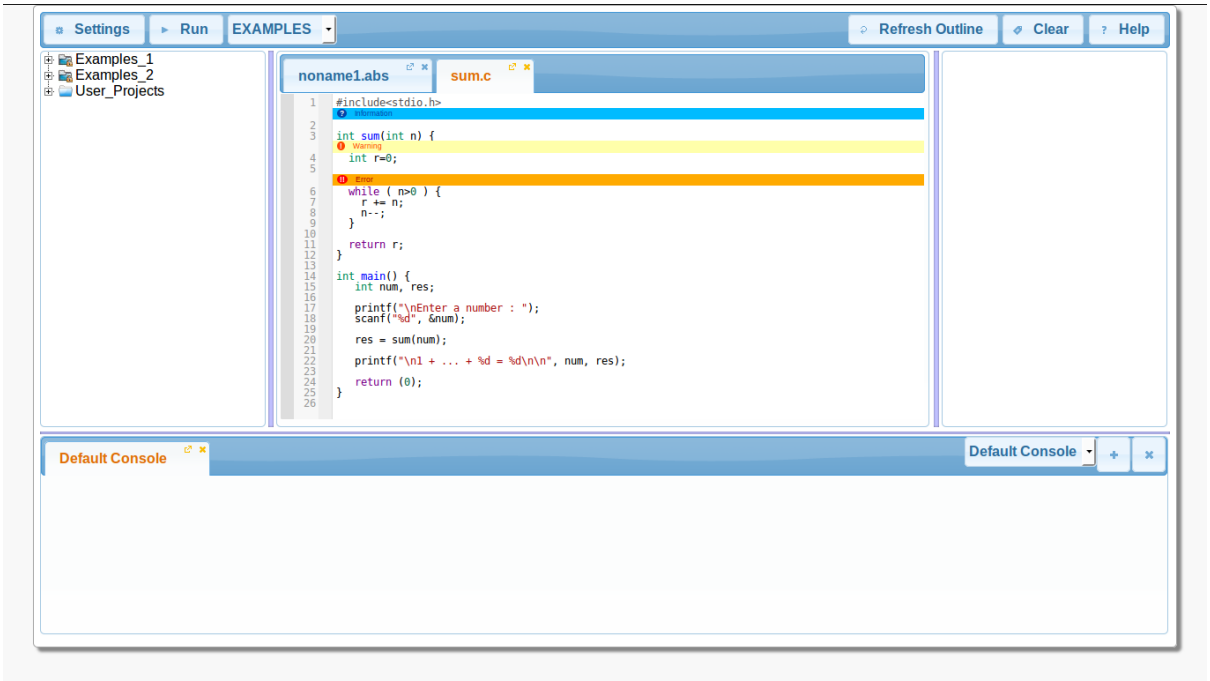

Figure 6.8: Output of Example 6.8

Its execution in the web-client generates the output as in Figure 6.9.

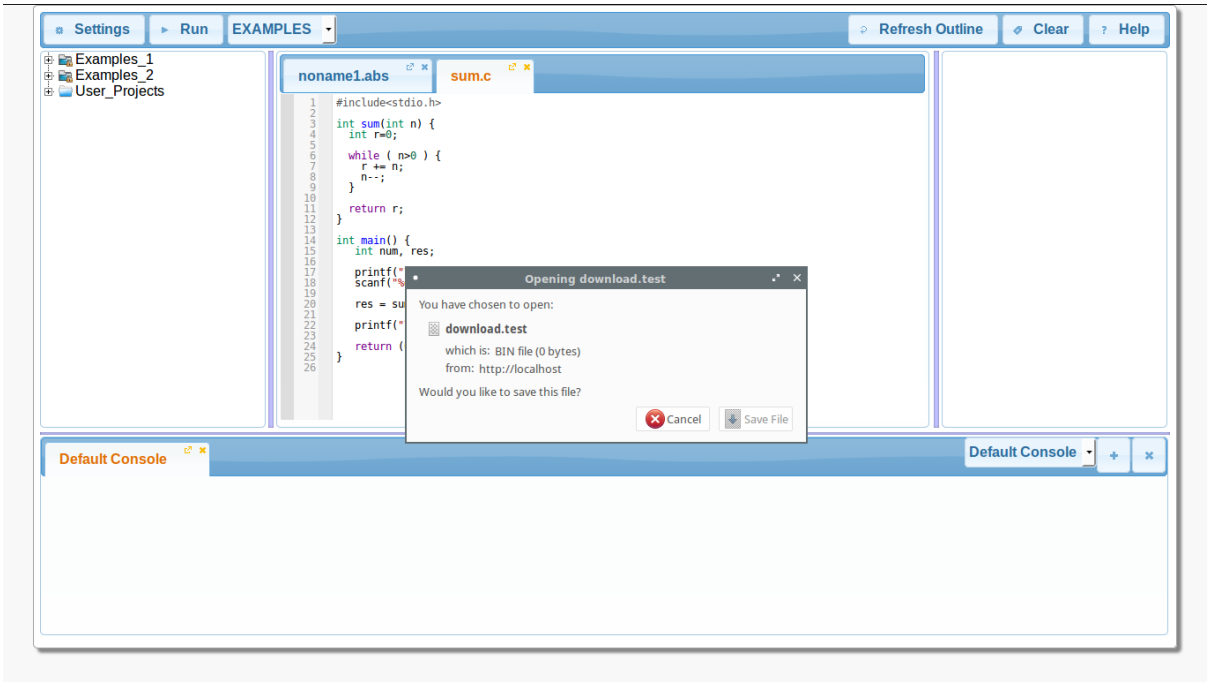

Figure 6.9: Output of Example 6.9

Chapter 7

Evaluation

The EasyInterface toolkit has been successfully applied in the context of the Envisage [5] project, where the tools developed in the project have been installed on an EasyInterface server, and they are accessible via a web-client that is available at following address: `http://www.abs-models.org`.

The integration effort was extremely low in general: a tool with a basic GUI could be integrated in few minutes. In some cases, where the EasyInterface output language has been used, the effort was higher but negligible when compared to developing GUIs from scratch. We note that in the case of Envisage, the integration was expected to be more costly since EasyInterface was developed in parallel to those tools. Using it for future projects should require even less effort.

In what follows we briefly describe each tool, and explain which parts of the EasyInterface output language were used.

7.1 Tools of the Envisage Project

Resource Analysis (SACO)

Description:

A static analysis tool for inferring the resource usage of ABS programs. It is parametric on the notion of *resource* to measure (e.g., memory, number of executed instructions, etc.), and can infer different kinds of cost (i.e., usage) such as parallel cost, peak cost, sequential cost, etc.

Further reading:

For more details see [24, 14].

Integration:

It uses the EasyInterface output language to view the output graphically, e.g., adding markers, drawing graphs using **SVG**, defining actions to view resource usage per method or per groups of objects, etc.

Resource Analysis (SRA)

DESCRIPTION:

A static analysis tool that computes upper bounds on virtual machine usage in a dialect of ABS, called vml, which has explicit acquire and release operations of virtual machines.

FURTHER READING:

For more details see [30].

INTEGRATION:

It uses the EASYINTERFACE output language to view the different parts of the result in different consoles.

May-Happen-in-Parallel Analysis (SACO)

DESCRIPTION:

A static analysis tool that infers a set of pairs of instructions that may happen in parallel (MHP) in any execution of a given ABS program. This is a very important analysis on which other analysis of concurrent programs build.

FURTHER READING:

For more details see [28, 21, 19, 20].

INTEGRATION:

It uses the EASYINTERFACE output language to view the output graphically, in particular, it defined *code line actions* such that when clicking on a line, it highlights other lines that might happen in parallel with it.

Deadlock Analysis (SACO)

DESCRIPTION:

A static analysis tool for proving deadlock freedom for ABS programs. The crux of the analysis is that it integrates the MHP information within dependency graph in order to discard unfeasible cycles that otherwise would lead to false positives. It is both precise and scalable.

FURTHER READING:

For more details see [23].

INTEGRATION:

It uses the EASYINTERFACE output language to view the output graphically, in particular, if there is a deadlock it highlight lines involved in the potential deadlock.

Deadlock Analysis (DSA)

DESCRIPTION:

A modular static analysis tool for proving deadlock freedom for ABS programs [32],

that is based on *behavioral types*.

FURTHER READING:

For more details see [29, 31].

INTEGRATION:

It makes a minimal use of the EASYINTERFACE output language to view the output or errors in the console.

ABS Smart Deployer

DESCRIPTION:

A tool that first processes the original ABS program to retrieve relevant cost annotation and deploy annotations defined in an ad-hoc domain specific declarative language. For each annotations, Smart Deployer relies on the Zephyrus2 configuration optimizer to concretely compute the objects that need to be deploy and then it generates a new ABS class that specifies the deployment steps to reach the desired target. This class can be used to trigger the execution of the deployment, and to undo it in case the system needs to downscale directly from the ABS code.

FURTHER READING:

For more details see [33].

INTEGRATION:

It makes a minimal use of EASYINTERFACE output language to view the output or errors in the console.

ABS ErLang Simulator

DESCRIPTION:

A simulator for ABS programs that is based on compiling ABS programs to corresponding ErLang programs.

FURTHER READING:

For more details see [11].

INTEGRATION:

It uses the streaming and download features. It also uses the EASYINTERFACE output language to view the output graphically, e.g., resource consumption graphs.

ABS-Haskell Compiler Simulator

DESCRIPTION:

A tool that compiles ABS programs to corresponding Haskell code, and then executes them and stream the output back to the client.

FURTHER READING:

For more details see [12].

INTEGRATION:

It uses the streaming feature.

Syntax/Type Checker

DESCRIPTION:

A tool that checks the syntax and types of ABS programs.

FURTHER READING:

For more details see [10].

INTEGRATION:

It uses the EASYINTERFACE output language to mark lines that have syntax or type errors.

Test-Case Generation (aPET)

DESCRIPTION:

A tool for static systematic testing for ABS which includes state-of-the-art partial order reduction and deadlock-guided testing techniques.

FURTHER READING:

For more details see [17, 22, 13, 18, 15, 36].

INTEGRATION:

It uses the EASYINTERFACE output language to view the output graphically, e.g., drawing execution traces using `SVG` and adding test units to the file-manager.

Systematic Testing (SYCO)

DESCRIPTION:

A tool for dynamic systematic testing for ABS which includes state-of-the-art partial order reduction and deadlock-guided testing techniques.

FURTHER READING:

For more details see [16, 22, 13, 18, 15, 36].

INTEGRATION:

It uses the EASYINTERFACE output language to view the output graphically, draw execution traces using `SVG`, etc.

Chapter 8

Conclusions, Related and Future Work

In this work, we have addressed the problem of *easily constructing* GUIs for research prototype tools, and integrating them in common environments. This is crucial since (*i*) it reduces the effort dedicated to building GUIs, which is usually tedious and complicated, and thus the main effort can be dedicated to improving the functionality of the tools; and (*ii*) it makes the tools continuously available to corresponding research communities since modifying the GUI when a tool changes, if needed, is immediate, and thus the dissemination of the corresponding research is improved as well — note that research prototype tools are expected to change continuously, for example, during a research project.

Clearly, attempting to significantly reduce the effort required for building GUIs in general is not feasible, since tools produce different outputs and receive different input. However, as we have observed in this work, this become feasible when focusing on a set of related tools that have common input and output aspects. In this work, we have focused on the tools developed in the Envisage [5] project which include: static analyzers, test-case generators, compilers, simulators, etc. We have developed a toolkit called EasyInterface that *extremely simplifies* the way GUIs are constructed for such tools, and the way they are integrated in common environments as well.

EasyInterface is a toolkit that consists of two main components: (1) a *server* where tools can be installed by providing simple configuration files, and then can be accessed as services using some protocol that we have defined; and (2) a *client*, in the form of a web-based development environment, that makes it easy to communicate with the server side to execute a tool. An important feature of the design of EasyInterface is that once a tool is installed on an EasyInterface server, it will automatically appear in all EasyInterface clients that connect to this server, without any additional effort. Thus, this design allows installing new tools in a common environment with minimal effort, i.e., in few minutes.

Another important outcome of this work is the EasyInterface output language. It is text-based language that tools can use to present their output graphically, if the corresponding client support it. Importantly, this language does not require any knowledge on GUI or Web programming. The advantage of using this output language is that it is interpreted by all EasyInterface clients equally. Thus, the tool is modified once to use this language and the effect will take place in all clients (including ones that will be developed in the future).

The EasyInterface toolkit has been successfully used in the context of the Envisage project, where the tools developed by the different partners has been integrated in a

common web-based environment. Some of the tools use the EasyInterface language as well, and others, whose output is very simple, do not.

8.1 Future Work

We have identified several future work directions, that would make EasyInterface a more powerful toolkit for building GUIs for research prototypes:

- Generating output in the EasyInterface output language is currently done by directly printing the corresponding XML on the standard output. It would be useful to provide libraries, for different programming languages, that abstract this level away. Namely, tools will not print directly but rather would use methods/objects of such libraries to generate the output, which in turn will generate the corresponding XML. Note that we have such a library for Prolog, which we use in tools marked by **SACO** in Section 7.1.

- Developing more EasyInterface clients, as the ones describes in Section 5.2, would make the toolkit adequate for different scenarios. For example, an Eclipse plugin would allow users to continue using their preferred development environment instead of learning to use a new one.

- Developing a simpler web-client that allows to inline an EasyInterface environment inside HTML documents. This is useful for writing interactive tutorials for example. Currently we have a simple support for this as described in Section 5.1.8.

- Improving the handling of sessions (in the EasyInterface server) to provide tools with and easy way to store and reload sessions. To take full advantage of this feature, the EasyInterface output language should be also extended to allow new kind of interaction that allows specialized callbacks to the corresponding tools.

We plan to continue the work on EasyInterface in the near future following the above directions, independently from the Envisage project.

8.2 Related Work

There are many powerful web-based IDEs that allow developers to develop their code online, and, in addition, some can be customized to connect external tools. In this section we overview some closely related ones.

Orion [25] is a web-based IDE developed by the Eclipse Foundation.[1] It provides some powerful features like connecting to `git` repositories, syntax-highlighting, etc. Once can use it to develop and compile code in several programming languages like C, C++ or Java. It also includes some plug-ins that can be activated like *web-tools support* for editing **HTML** and **CSS**, *JSON editor* for editing **JSON** records, etc.

[1] `http://www.eclipse.org`

Coding Ground [4] is a web-based IDE developed by TutorialsPoint.[2] It was developed with the main objective of adding exercises to their tutorials, but since then it has evolved into an IDE where developers can edit, compile, execute, and share their code. It supports more than 90 programming languages, but only one at a time. It does not allow integrating external tools.

Cloud9 [1] is a web-based IDE that supports hundreds of programming languages, and allows creating collaborative workspaces with multiple real-time edition. It has some advanced features like code completion, name refactoring, etc. It also provides support for using repositories like `git`, `mercurial` and `FTP` servers.

Codeboard [2] is web-based IDE to teach programming in the classroom. One can easily create and share exercises with students, analyze and inspect students' submissions, etc. One can customize it to connect external tools, however, their output can be shown only on the console area.

The IDEs described above are very powerful, but focused on developing code. They do not address most of the objectives that we stated in Section 1.1, in particular (*i*) they do not provide an easy way to integrate external tools; and (*ii*) they do not provide a simple way to produce output or code annotations as in our output language that is described in Chapter 6.

An exception is the tool **rise4fun** [9], which is developed by Microsoft to allow making, among others, program analysis tools available online. However, it is far simpler than EasyInterface: (*i*) tools are not integrated in a common environment, but rather each has its own page; (*ii*) the output can be shown only in a console area; and (*iii*) tools cannot easily receive parameters.

[2] `http://www.tutorialspoint.com`

Bibliography

[1] Cloud9. https://c9.io.

[2] Codeboard. https://www.codeboard.io.

[3] CodeMirror. http://codemirror.net.

[4] CodingGround. http://www.tutorialspoint.com/codingground.htm.

[5] ENVISAGE: Engineering Virtualized Services. http://www.envisage-project.eu.

[6] Introducing json, official web-site. http://www.json.org/.

[7] JQuery. http://jquery.com.

[8] JSTree. http://www.jstree.com.

[9] Rise4Fun. http://www.rise4fun.com.

[10] Syntax/Type Checker.

[11] Tutorial on ABS ErLang Simulator. http://abs-models.org/chapter/1-simulation/.

[12] Tutorial on ABS-Haskell Compiler Simulator. http://abs-models.org/chapter/9-code-generation-haskell/.

[13] E. Albert, M. Gómez-Zamalloa, and M. Isabel. Combining Static Analysis and Testing for Deadlock Detection. In *12th International Conference on integrated Formal Methods, iFM 2016*, 2016. To appear.

[14] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, German Puebla, and Guillermo Román-Díez. *SACO: Static Analyzer for Concurrent Objects*, pages 562–567. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[15] Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Test Case Generation of Actor Systems. In *13th International Symposium on Automated Technology for Verification and Analysis, ATVA 2015. Proceedings*, volume 9364, pages 259–275, 2015.

[16] Elvira Albert, Puri Arenas, Miguel Gómez-Zamalloa, and Miguel Isabel. Tutorial on Systematic Testing (SYCO). http://abs-models.org/chapter/4-systematic-testing-syco/.

[17] Elvira Albert, Puri Arenas, Miguel Gómez-Zamalloa, and Miguel Isabel. Tutorial on Test-Case Generation (aPET). http://abs-models.org/chapter/5-test-case-generation-apet/.

[18] Elvira Albert, Puri Arenas, Miguel Gómez-Zamalloa, and Jose Miguel Rojas. Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-Based Instance, and Actor-Based Concurrency. In *Formal Methods for Executable Software Models*, volume 8483 of *Lecture Notes in Computer Science*, pages 263–309. Springer, 2014.

[19] Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. May-happen-in-parallel analysis for actor-based concurrency. *ACM Trans. Comput. Logic*, 17(2):11:1–11:39, December 2015.

[20] Elvira Albert, Samir Genaim, and Pablo Gordillo. May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization. In *Static Analysis - 22nd International Symposium, SAS 2015. Proceedings*, volume 9291 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2015.

[21] Elvira Albert, Samir Genaim, and Enrique Martin-Martin. May-Happen-in-Parallel Analysis for Priority-based Scheduling. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-19)*, volume 8312 of *Lecture Notes in Computer Science*, pages 18–34. Springer, December 2013.

[22] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. SYCO: A systematic testing tool for concurrent objects. In Ayal Zaks and Manuel V. Hermenegildo, editors, *25th International Conference on Compiler Construction (CC'16)*, pages 269–270. ACM, 2016.

[23] Jesús Correas, Antonio Flores, Samir Genaim, Enrique Martín, and Guillermo Román. Tutorial on Deadlock Analysis (SACO). http://abs-models.org/chapter/2-deadlock-analysis-saco/.

[24] Jesús Correas, Antonio Flores, Samir Genaim, Enrique Martín, and Guillermo Román. Tutorial on Resource Analysis (SACO). http://abs-models.org/chapter/6-resource-analysis-saco/.

[25] Eclipse Fundation. Orion. http://orionhub.org.

[26] Ola Andersson et al. Scalable vector graphics (svg) 1.2. 2004. https://www.w3.org/Graphics/SVG/1.2/WD-SVG12-20041027.pdf.

[27] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.

[28] Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In Dirk Beyer and Michele Boreale, editors, *Formal Techniques for Distributed Systems (FMOODS/-FORTE 2013)*, volume 7892 of *Lecture Notes in Computer Science*, pages 273–288. Springer, June 2013.

[29] Abel García and Elena Gianchino. Tutorial on Deadlock Analysis (DSA). http://abs-models.org/chapter/3-deadlock-analysis-dsa/.

[30] Abel García and Elena Gianchino. Tutorial on Resource Analysis (SRA). http://abs-models.org/chapter/7-resource-analysis-sra/.

[31] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in core_ABS. *Software & Systems Modeling*, pages 1–36, 2015.

[32] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A Core Language for Abstract Behavioral Specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, volume 6957, pages 142–164. Springer, 2012.

[33] Jacopo Mauro. Tutorial on ABS Smart Deployer. http://abs-models.org/chapter/8-smart-deployer/.

[34] Ivan Ristić. *Apache Security*. Feisty Duck, March 2005.

[35] Ralph Steyer. *Learning JQuery: A Hands-on Guide to Building Rich Interactive Web Front Ends*. Addison-Wesley, 2013.

[36] Peter Y. H. Wong, Richard Bubel, Frank S. de Boer, Miguel Gómez-Zamalloa, Stijn de Gouw, Reiner Hähnle, Karl Meinke, and Muddassar Azam Sindhu. Testing Abstract Behavioral Specifications. 17(1):107–119, 2015.

Appendix A

Installation Guide

In this appendix we explain how to install the server and how to use the different clients.

A.1 Downloading EasyInterface

We assume that you have already downloaded[1] EasyInterface into a directory called **easyinterface**, and that all files inside this directory have read and execute permissions to **others** which can be done, for example, in Unix based systems by executing:

```
> chmod -R 755 easyinterface
```

The purpose of this is to make all files visible to the **Apache\ Web\ Server** on which the EasyInterface server runs. This is (most likley) not required if you are using Microsoft Windows.

A.2 Installing EasyInterface Server

The installation consists in installing an **Apache\ Web\ Server** (with **PHP** > 5.0 enabled) and then configuring it to recognize the **easyinterface** directory. If you already have Apache installed and the **easyinterface** directory is visible then no further configuration is required, simply visit the corresponding address (e.g., if it is placed in the **public_html** directory, visit `http://localhost/~user/easyinterface`). Otherwise follow the steps below depending on which operating system you are using, **Linux**, **OS X** or **Windows**.

A.2.1 Linux

Installing Apache depends on the Linux distribution you are using, for example, if you are using Ubuntu you can install it by executing the following in a shell:

```
> sudo apt-get update
> sudo apt-get install apache2
> sudo apt-get install php5 libapache2-mod-php5 php5-mcrypt
```

[1] `http://github.com/abstools/easyinterface`

```
> sudo service apache2 start
```

Once installed test that it works correctly by visiting `http://localhost` and test that PHP works correctly by visiting `http://localhost/info.php` (this address might be different from one distribution to another). Next, to make the **easyinterface** directory visible, edit **/etc/apache2/mods-enabled/alias.conf** and add the following lines:

```
Alias /ei "/path-to/easyinterface"

<Directory "/path-to/easyinterface">
   Options FollowSymlinks MultiViews Indexes IncludesNoExec
   AllowOverride All
   Require all granted
</Directory>
```

To activate this change you need to restart Apache by executing the following in a shell:

```
> sudo service apache2 restart
```

Now visit `http://localhost/ei` to check that EasyInterface works correctly. If no error message is shown, you can proceed to the next section and start using the Web client.

A.2.2 OS X

OS X typically comes with Apache installed, and all you need is to configure it to recognize the **easyinterface** directory. To do so, edit **/etc/apache2/httpd.conf** add the following lines:

```
Alias /ei "/path-to/easyinterface"

<Directory "/path-to/easyinterface">
   Options FollowSymlinks MultiViews Indexes IncludesNoExec
   AllowOverride All
   Require all granted
</Directory>
```

To activate this change you need to restart Apache by executing the following in a shell:

```
> sudo apachectl restart
```

You can also restart Apache using **System Preferences > Sharing > Web Sharing**. Now visit `http://localhost/ei` to check that EasyInterface works correctly.

A.2.3 Microsoft Windows

Apache Web Server for Microsoft Windows is available from a number of third party vendors.[2] We have tested EASYINTERFACE using WampServer.[3]

Install the WampServer, for example in **c:\wamp**, and then edit the configuration file **c:\wamp\bin\apache\apache.X.Y.Z\httpd.conf** and add the following lines to make the **easyinterface** directory visible:

```
Alias /ei "\path-to\easyinterface"

<Directory "\path-to\easyinterface">
   Options FollowSymlinks MultiViews Indexes IncludesNoExec
   AllowOverride All
   Require all granted
</Directory>
```

Next restart the WampServer by executing

```
c:\wamp\wampserver.exe -restart
```

Now visit `http://localhost/ei` to check that EASYINTERFACE works correctly. If you have permission problems when accessing this address, try to remove the file **easyinterface/.htaccess**.

By default the server is configured to execute the demo applications in a Unix based operating system, for using them in Windows you should copy the configuration file **server/config/eiserver.default.win.cfg** to **server/config/eiserver.cfg**.

The demo applications are simple bash scripts, and thus you need to install winbash if you want to use them. To do so, simply download the corresponding **zip** file and extract it in **c:\bash** (it is important to place it in **c:\bash** since the configuration files use **c:\bash\bash.exe** to execute the bash scripts).

A.3 Installing and Using EASYINTERFACE Clients

The Web client can be used by visiting `http://localhost/ei/clients/web`.

[2] `http://httpd.apache.org/docs/current/platform/windows.html#down`

[3] `http://www.wampserver.com/`