D3.5
Enhanced version of the tools

Project number:	731453
Project acronym:	VESSEDIA
Project title:	Verification engineering of safety and security critical dynamic industrial applications
Start date of the project:	1st January, 2017
Duration:	36 months
Programme:	H2020-DS-2016-2017

Deliverable type:	Others
Deliverable reference number:	DS-01-731453 / D3.5 / 1.0
Work package contributing to the deliverable:	WP3
Due date:	December 2019 - M36
Actual submission date:	15th of January, 2020

Responsible organisation:	CEA
Editor:	Virgile Prevosto
Dissemination level:	PU
Revision:	1.0

Abstract:	This report describes the software tools that compose deliverable D3.5
Keywords:	Frama-C,system verification,high-level specification, verification server

The project VESSEDIA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453.

Editor
Virgile Prevosto(CEA)

Contributors
Jens Gerlach (FOKUS)
Malte Brodmann (FOKUS)
Marko Fabiunke (FOKUS)

Executive Summary

Deliverable 3.5 consists in the software tools that have been developed in tasks T3.1 (modular reasoning for system validation and verification), T3.3 (verification service for formal static analysis), and T3.4 (enhancements of the GUI). The present document describes each tool included in this deliverable.

Contents

1 Introduction **1**

2 Diversity **2**

3 Frama-C **3**

3.1 RPP plugin 3

3.2 MetACSL plugin 4

4 Verification Server **6**

4.1 Design 6

4.1.1 Theorem provers as microservices 6

4.2 Implementation 7

5 GUI **11**

5.1 Frama-C GUI 11

5.2 Server plugin 12

5.3 Dome framework 14

5.4 Dive 14

5.5 Ivette 14

6 Conclusion **19**

Chapter 1

Introduction

The present document is part of D3.5 of VESSEDIA, which gathers the tools that have been developed or enhanced within WP3, and more specifically T3.1 (modular reasoning for system validation and verification), T3.3 (verification service for formal static analysis), and T3.4 (enhancements of the GUI).

As indicated in report D3.1, T3.1 has primarily consisted in binding together the Diversity tool, dedicated to model analysis with Frama-C, for verifying that the scenarios identified by Diversity at model level were indeed feasible within the implementation. For that, Diversity didn't target directly ACSL, the primary formal specification language of the platform, but took advantage of a Frama-C plugin called RPP, able to accept specifications in a format more suitable to Diversity. Finally, in the last year of the project, another Frama-C plugin, MetACSL, has been designed and developed in order to facilitate the specification of another kind of higher-level properties. More precisely, ACSL is basically restricted to properties that must be verified at a specific program point (when entering a function, returning from a function, at each step of a particular loop, or at given statement). MetACSL offers the possibility to write once a property that must be verified at many places, scattered all along the code base, thereby facilitating the expression of system-level properties that must be verified by the implementation. Diversity is described in more detail in chapter 2. RPP and MetACSL are presented in chapter 3, respectively in section 3.1 and 3.2.

Task T3.3 was dedicated to build a verification server that would be able to launch many automated theorem provers over the proof obligations generated by WP. Since, at first glance, these proof obligations are all independent from each others, and their number can grow quickly as the code and its specification become longer, deferring them to a server that can handle them in parallel seems indeed interesting. A first version of the server has been made available as D3.2. Chapter 4 describes this new version of the server.

Finally, various enhancements have been done to Frama-C's GUI as part of T3.4, and are presented in chapter 5. First, the work done on the current GUI of Frama-C is described in section 5.1. Then, the new server plugin, which is at the heart of the client/server architecture of the future GUI is presented in section 5.2. The first client of the server, the Dive plug-in for exploring dependency graphs between memory location, is detailed in section 5.4. Then, the early prototype of a more generic client of Frama-C server, Ivette, is presented in section 5.5.

Chapter 2

Diversity

Diversity is a tool aimed at generating test scenarios from the description of a system in terms of input/output automata. In particular, It can take as input UML sequence diagrams describing the intended interactions between the various components of a system. This is then converted into Diversity's internal format *xLIA*, and Diversity performs a symbolic execution in order to generate the constraints on the inputs of the system that will lead to the execution of the various possible branches. These constraints will then be used to generate different test sequences that will be fed to the actual implementation in order to check if it reacts according to the model. Diversity is based on the Eclipse IDE, and is part of the Eclipse Formal Modeling project in the Eclipse Foundation (`https://projects.eclipse.org/projects/modeling.efm`).

Within VESSEDIA, Diversity has been extended in order to be able to take into account actual function calls in the model of the system, in order to gather constraints on the arguments that are fed to these functions, as well as to the results they are supposed to return to their callers. Since these constraints are accumulated along each execution path explored by the symbolic execution, they cannot readily be translated as function contracts for each function encountered on the path. Instead, the constraint as a whole can be seen as a *relational property* tying together the results of these calls. Relational properties are not handled natively by Frama-C, but there exists a dedicated plug-in for them, RPP, described in section 3.1.

The archive included in the deliverable (`diversity4inference.7z`) contains a windows64 distribution of the framework including the Vessedia extensions as well as a small tutorial on how to use Diversity in general.

Chapter 3

Frama-C

The Frama-C platform offers a set of tools for verifying properties about code written in C (with an experimental C++ front-end developed within WP2 of VESSEDIA, whose latest version is part of D2.5). A certain number of Frama-C plug-ins are the subject of work done within WP3 and are gathered in this deliverable. Furthermore, the main Frama-C distribution is included as well. As the server plug-in (see section 5.2) is part of the main distribution and Ivette (see section 5.5) depends on requests that were implemented after the latest stable release of Frama-C (20.0 Calcium), the version included in this deliverable is a development one, available through the public git repository at `https://git.frama-c.com/pub/frama-c/-/tags/Vessedia-D3.5`.

Frama-C main distribution is a pre-requisite before installing the external plug-in that are included in this deliverable as well. the `INSTALL.md` file of the Frama-C archive contains detailed installation instructions, but basically the easiest route is to use the `opam` package manager. With respect to the instructions for installing the stable version, installing the version corresponding to this deliverable differ only by the need of specifying a custom target for installation (`pin` operation in `opam`'s vocabulary). Assuming `opam` is installed, the following sequence of commands should thus install the appropriate Frama-C version:

```
opam init # only if not done before
opam install depext # only if not done before
opam depext
opam pin add frama-c https://git.frama-c.com/pub/frama-c.git#Vessedia-D3.5
```

3.1 RPP plugin

The Relational Property Prover (RPP) plug-in is aimed at handling so-called *relational properties*, that is properties relating the behavior of different function calls. A simple example of relational properties can be given by considering a comparison function

```
int compare(T x, T y)
```

that is supposed to return `-1` if `x` is less than `y`, `1` of `x` is greater than `y` and `0` if they are equivalent.

Two basic properties of such a comparison function are in fact relational properties. The first one is the antisymmetry if two calls to `compare` are made with the arguments swapped, the result must be opposite from each others, or in other words:

```
\forall x, y; compare(x, y) == -compare(y,x)
```

The second is the transitivity: if `x` is less (resp. greater) than `y` and `y` is less (resp. greater) than `z`, then `x` must be less (resp. greater) than `z`:

```
\forall x, y, z; compare(x,y) == compare(y,z) ==> compare(x,y) == compare(x,z);
```

Such properties cannot readily be expressed in the standard specification language of Frama-C, ACSL, whose function contracts can only describe what is supposed to happen in a single call at a time. RPP has thus introduced a specific syntax to write relational properties, and then performs a transformation, known as *self-composition*, to obtain a program with plain ACSL specifications whose validity imply the validity of the relational property over the original program. Another transformation is also available in order to take advantage of the relational properties that have been proved in subsequent verification activities.

In the context of VESSEDIA, RPP has been extended to facilitate the production of relational properties from the constraints gathered by Diversity (see chapter 2). The archive included in this deliverable is meant to be compiled against the development version of Frama-C mentioned above, with the usual sequence of commands:

```
make
make install
```

The main option of the plugin is `-rpp` that will activate the transformation of the relational properties in plain ACSL contracts over wrapper functions

3.2 MetACSL plugin

The MetACSL plug-in is aimed at expressing in a concise way properties that are supposed to hold at many points in the program under analysis, with an emphasis towards confidentiality and integrity properties. More precisely, a meta-property, as handled by MetACSL consists of three parts:

- its perimeter, i.e. the set of functions where it is active.
- its context, i.e. the kinds of program points where it must be verified
- its body, i.e. the property itself

Depending on the context, the body of the meta-property can use one or more meta-variables, that will be instantiated at each point relevant for the meta-property. For instance, a meta-property active for the `\writing` context, which thus must be verified for each writing access, can refer to the `\written` meta-variable, which represents the memory location that is being written to. With this meta-variable, it becomes easy to give integrity properties resembling to

```
AUTHORIZED==0 ==> \separated(\written, confidential_array[0 .. length - 1]);
```

The line above indicates, in ACSL terms, that unless `AUTHORIZED` is not `0`, no write access to any cell of `confidential_array` should occur.

Meta-properties are expressed at global level and are thus interesting for reflecting system-wide properties that are supposed to be verified by the implementation. MetACSL proceeds by instantiating each meta-properties into a set of plain ACSL annotations, at each program point specified by the perimeter and the context of the meta-property. It is then up to the classical analyzers plug-ins of the platform (in particular WP and Eva) to check whether the generated ACSL properties hold.

The MetACSL archive included in the deliverable is meant to be compiled against the development version of Frama-C mentioned above. Once the main Frama-C distribution is installed, installing MetACSL on top of it is merely a matter of issuing the following commands in MetACSL's top directory:

```
make
make install
```

Chapter 4

Verification Server

A first version of a verification server for parallelizing proof tasks coming from the WP plug-in of Frama-C has been introduced in deliverable D3.2 and is the subject of task T3.3. The present deliverable contains an implementation of the verification server that is meant to be deployed on the AWS Lambda cloud computing platform. Due to the high degree of parallel execution, the system enables a high acceleration compared to the conventional execution on the (local) system.

4.1 Design

4.1.1 Theorem provers as microservices

In a so-called microservice architecture, an application is developed as a set of small independent services. These so-called microservices run in their own processes and can communicate via defined interfaces.

The verification service should enable the parallel and independent execution of different automatic theorem provers. These are addressed and executed as separate services, each with its own interface. In principle, the verification service can therefore be understood as a set of different microservices, whereby each microservice covers one theorem verifier. These can then be integrated by a verification application in which the microservice is contacted via the corresponding interface. The way in which the execution of these different microservices is coordinated, the so-called microservice orchestration, is taken over by the verification application. A microservice itself is not directly involved in this coordination, as it is only responsible for the one-time execution of a theorem proving device. Thus, more complex applications and different use cases can be realized on the basis of the independently executable Microservices.

The independent microservices are implemented on a separate cloud-based infrastructure. In this case, AWS Lambda is used, but the approach should probably be adaptable to other cloud infrastructures. AWS Lambda is a so-called serverless computing platform offered by the cloud computing provider Amazon Web Services (AWS). Serverless computing is a programming model in which the application developer does not have to worry about providing and operating physical infrastructure but has to develop his own application code. The execution of the program code and the reservation of resources is handled by a single vendor. Thus, AWS Lambda offers the abstraction of stateless functions, so-called Lambda functions, for which a developer can write any application code, which is then executed by the provider. AWS Lambda is therefore also an example of the so-called Function as a Service (FaaS) model. In serverless computing, the execution of program code for a function is often event-based, for example after a request has been made to an HTTP interface.

A lambda function is a (usually small) program written in one of the programming languages supported by AWS Lambda. The program code must have an implementation of a special function which is used as an entry point for executing the Lambda function. If, for example, the Java programming language is used, one way of creating a lambda function is to define a class that implements the `RequestStreamHandler` interface. To do this, it must define a method called `handleRequest`, which is then used as the entry point when the lambda function is started. The `handleRequest` method also has an argument of type `InputStream`, an argument of type `OutputStream` and an argument of type `Context`. The arguments of type `Input`- and `OutputStream` serve to communicate with the Lambda function externally and can be used to integrate the Lambda function into a Web API. The argument of type `Context` can be used within the lambda function to access some information concerning the context of the lambda function. An example of a simple lambda function written in the Java programming language can be found in the following listing. The function shown here stores the received input data in a buffer and sends this data back via the `OutputStream`.

```
public class Echo implements RequestStreamHandler {
  public void handleRequest(
    InputStream is,
    OutputStream os,
    Context c)
    throws IOException {
      byte[] buffer = new byte[16384];
      int length = is.read(buffer,0,buffer.length);
      os.write(buffer,0,length);
  }
}
```

A completed Lambda function can be uploaded to the AWS Lambda platform via a so-called provision package (e.g. as a zip file). After uploading, AWS Lambda performs the lambda function. In the case of Java, such a provision package contains the compiled program code of the lambda function, resources (data that the program code can access during execution) and program libraries on which the lambda function depends (usually JAR files).

The implementation of the theorem provers as a microservice is done with the help of AWS Lambda by designing a corresponding lambda function that models the processing of a request, the execution of the corresponding theorem proving program and the response to the request. For each theorem verifier to be used, only the program code of the corresponding Lambda functions has to be developed. After this has been provided in AWS Lambda, the execution and automatic scaling is taken over by AWS. Based on the automatic scaling of the lambda functions, the verification service can execute many theorem provers in parallel and accelerate the verification with Frama-C/WP.

4.2 Implementation

In order to feed the microservices, a verification application must run on the client machine, where Frama-C/WP is also located, in order to act as a gateway between Frama-C (with WP generating the proof obligations as usual) itself and the provers microservices. The verification application is a single executable program implemented in the Go programming language. The main reason for using Go was that Go enables efficient parallel execution of many processes. This is particularly important for the parallel execution of requests to the verification service. In its current form, the verification service consists in 5 lambda functions for the five provers that are used.

The aim of the verification application is (only) to outsource the execution of the theorem provers to the verification service. Frama-C/WP normally executes the entire verification process from the generation of proof obligations to the execution of the theorem provers without interruption on one computer. To implement the exemplary (parallel) verification application, the verification process must therefore be broken off by terminating the execution of Frama C/WP after generation of the proof obligations. Since Frama-C/WP does not support interfaces for the transmission of proof obligations, all information required for verification must be extracted from the files that were generated until Frama-C/WP was aborted.

The starting point for formal verification with Frama-C/WP using the verification service is ACSL-specified C source code. In the current implementation of the application, Frama-C version 18.0 was used to generate the proof obligations. The first step within the verification application is to run Frama-C/WP to generate proof obligations from the specified source code. A simple call of the program in the following form is sufficient:

```
frama-c -wp [Options] -wp-prover why3 -wp-gen [C files]
```

Options is a set of configuration options (see Frama-C/WP Manual for how to use the configuration options of Frama-C/WP) and C files are the paths to the specified C files to be verified by Frama-C/WP. Frama-C/WP generates text files in `.why` format when executed in the described form. These contain all the information needed to prove the proof obligations contained therein by means of theorem provers. The execution of Frama-C/WP is aborted after the generation of the proof obligations, so no theorem provers (local) are called/executed.

Since Frama-C/WP does not support interfaces for the transfer of proof obligations, the verification application must now use the generated files in the following step to manually identify all proof obligations to be proven. There are two types of proof obligations in Frama-C/WP: Theories and Lemmas. However, once all proof obligations have been identified, this distinction is of no further significance in the context of the verification service. All generated proof obligations, both theories and lemmas, must be proven. To identify all generated proof obligations, the generated .why files are searched for definitions of theories and lemmas.

With the help of the proof obligations determined, so-called tasks are now generated in an application-internal representation. A task symbolizes a task to be performed by the verification service and corresponds to a combination of a burden of proof and a theorem prover who is to try to prove the burden of proof. In addition, a task captures the name of the file in which the burden of proof is defined and the result of the processing by a theorem verifier for later evaluation.

Depending on the use case used, the number of tasks and also the times at which the tasks are created differ. In the most simple usage, there is exactly one task per proof obligation after all proof obligations have been identified. The theorem verifier to be used is the first theorem verifier in the order used. If it does not succeed, a new task is created for the second prover in the list and so on. In another setting, in order to benchmark provers, tasks for all possible combinations of proof obligation and theorem verifier are created from the beginning.

Before the tasks can be sent as requests to the verification service for further processing in the next step, they must first be serialized. To do this, the task is transferred to a JSON representation. This contains the name of the proof obligation and the name of the file that contains the proof obligation. In addition, it contains a coded version of all .why files that are needed by theorem provers to perform a proof and must therefore be sent as part of the request.

After creating the JSON representation of a task, it is sent as a request to the verification service. The verification application sends an HTTP post request to the corresponding interface (API endpoint) of the theorem verifier that is to be executed. The JSON representation of the task is located in the message body of the HTTP request. The system then waits for the response from the service. For all tasks available to the verification application, the generation and execution of the corresponding queries takes place in parallel. This is possible because there are no dependencies between them. In this way, the requests are sent to the verification service simultaneously and as quickly as possible. In particular, the fast sending of a request is useful for the fast processing of all tasks, since the verification service scales with the number of proof obligations. Therefore, it is not necessary for the verification application to delay a request.

In the next step, the requests are received by the verification service, the theorem provers are executed, and the results are sent back to the verification application. The verification service is implemented with the two AWS services Amazon API Gateway and AWS Lambda. The AWS Lambda service is only responsible for the execution of the theorem provers. The Amazon API Gateway service is used for communication between the verification application and the Lambda functions. This enables the integration of the lambda functions, which are executed by the theorem provers, by providing API endpoints. In this way, interfaces are provided through which the verification application can effect the execution of a lambda function and the associated theorem prover.

Using the Amazon API Gateway, an API endpoint is defined for each of the theorem provers used. This API endpoint can also be configured to trigger a lambda function when an HTTP mail request is received at one of these endpoints. Therefore, each of the five defined API endpoints is linked to the lambda function of the associated theorem verifier. Upon receipt of a request, the Amazon API Gateway service ensures that an instance of the respective lambda function is started. It is quite possible that not only the lambda functions of different theorem provers are started and executed in parallel, but also several instances of a single prover's lambda function. In this way, the automatic scaling of the executed Lambda functions with the number of API requests received is made possible. Furthermore, the Amazon API gateway transmits the message body of the HTTP request or the JSON representation of the task to the started Lambda function.

A difficulty with the realization of the automatic theorem provers as Lambda functions is that for their execution the Why3 platform is needed. This platform is responsible for the start and the execution of the theorem provers. The AWS Lambda function of a theorem verifier is therefore created as a wrapper function. A wrapper class written in Java implements the necessary interfaces to integrate the service into the AWS Lambda platform (Web Services). The actual verification is passed on from the wrapper class to an instance of the Why3 platform running in the cloud platform.

Amazon Web Services uses so-called MicroVMs to execute Lambda functions. These virtualize the existing computer resources. The operating system that is used as part of a lambda function is an Amazon Linux specially adapted by Amazon. Within a lambda function, the `\tmp` directory provides a location for storing files in the file system. In order to run the Why3 platform and the theorem verifier, they must be stored in the `\tmp` directory from within the deployment package. A problem that occurs here is that the Why3 platform usually attempts to create or access various configuration files in user-specific directories. Also, the Why3 platform cannot locate the theorem prover in the `\tmp` directory by default, since the theorem provers are usually stored in other directories. To get a version of the Why3 platform that works as expected in AWS Lambda, some small changes have been made to the source code of the Why3 platform. In this way a slightly modified version of the Why3 platform was created. The compilation of the modified source code was done on Amazon Linux to ensure better compatibility.

The resulting executable version of the Why3 platform consists of several executable files as well as driver and text files. Since AWS Lambda limits the size of a deployment package to 50 MB, any files not required to run a theorem proving program had to be identified and removed to prevent the deployment package from becoming too large. These steps ultimately resulted in a light version of the Why3 platform running within AWS Lambda. This is included in the deployment package as a zip archive file together with the executable file of the corresponding theorem verifier. The deployment package thus contains the executable versions of the Why3 platform and the used theorem verifier as well as the compiled program code of the lambda function and libraries on which the execution of the lambda function depends.

At the beginning of the execution of a lambda function, the .why files required for processing the proof obligation are first stored in the file system of the current lambda function. For this purpose, the JSON representation of the task is read via the InputStream and transferred to an internal format. The individual .why files are then extracted from this and written to the `\tmp` directory. Likewise the used theorem prover and the Why3 platform from the supply package are stored in the `\tmp` directory and made executable. Then the theorem prover can be executed. The necessary information, such as the name of the proof obligation, is also taken from the received request. The call of the Why3 platform takes place in the following form:

```
why3 prove [why file] -t 10 [options] \
     -T [proof obligation] -P [theorem prover]
```

The call described here has been simplified, for example by omitting various options. Important parameters are above all the theorem prover to be used and the proof obligation to be processed. Further information on the use of the Why3 platform can be found in the Why3 Manual. Finally, the result of the theorem prover execution is returned in a JSON representation using the OutputStream. This contains the name of the proof obligation and the result obtained after the theorem prover has been executed.

After the execution of a lambda function is complete, the result of the execution is sent by the Amazon API Gateway in response to the HTTP request received from the verification application. After receiving such a response, the verification application extracts the result from the received JSON representation and stores it in the associated task. This concludes the processing of this task.

After all tasks have been processed, i.e. the results of the processing by the verification service are available for all tasks, the results can be evaluated. The number of proof obligations that have been proven at least once and the number of proof obligations proven per theorem prover used are determined and issued.

The `VerificationService.tar.gz` archive contained in this deliverable provides the prover microservices.

Chapter 5

GUI

Task T3.4 was dedicated to enhance the graphical user interface (GUI) of Frama-C, in order to make the tool easier to use. Some work has been devoted to the current GUI of the platform, based on the well-known `GTK` library and is described in section 5.1. In parallel, a new architecture, based on a client/server approach, has been designed. The server is included in the main Frama-C distribution and is presented in section 5.2. Two external clients are also included in this deliverable, both based on the Electron JavaScript framework. Dive (section 5.4) is dedicated to explore data dependencies between memory locations, in particular for investigating the root causes of an alarm emitted by Eva, while Ivette (section 5.5) is a prototype of a generic GUI for verification activities, with a view dedicated to Frama-C itself.

5.1 Frama-C GUI

An important task from a technical point of view has been to move away from the obsolete GTK 2 toolkit to use the slightly newer GTK 3. This was prompted by the support of GTK 3 in OCaml, the language in which the Frama-C platform is written, in the form of the `lablgtk3` library containing the bindings with the C-based GTK 3. Support for `lablgtk3` appeared in Frama-C 19 Potassium, released in May 2019, and current versions of Frama-C can be built against either GTK 2 or GTK 3.

More user-visible changes to the GUI include notably the development of an interactive mode for guiding the automated provers in resolving proof obligations stemming from WP. This module, called TIP, appeared in Frama-C 15.0 Phosphorus, in May 2017, and has been continuously improved ever since.

The other main analyzer of the platform, the abstract interpretation-based Eva plugin, has also seen notable extensions for displaying its results in Frama-C's GUI. In particular, it is now much easier to unfold the abstract values computed for the content of an array at a given program point, and more generally for observing the abstract state computed by Eva. Another addition was the so-called *Red Alarms* panel, dedicated to display alarms emitted by Eva which have been found to be totally invalid on a least one of the execution paths explored by Eva. Indeed, such alarms are much more likely to be true alarms than other emitted alarms (which might only be artefacts of the overapproximations made by Eva in order to guarantee the termination of the analysis). Hence the user will probably want to focus on such alarms first and examine the others later. Finally, an helper plug-in, called Studia, has been developed in order to be able to trace back in the GUI the assignments that might contribute to the value of a given memory location at a given program point (typically where an alarm has been raised by Eva in order to better understand where the issue might come from).

The current Frama-C GUI is included in the main distribution and is installed by default if the appropriate dependencies (i.e. lablgtk2 or lablgtk3) are available on the system, which is the case with the opam-based installation procedure mentioned in chapter 3.

5.2 Server plugin

In Frama-C 20.0 Calcium, a new plug-in `server` has been released, whose aim is to respond to requests from external clients, either to get some information about the internal state of Frama-C and its plug-ins, or to set the configuration of some analyzers and execute them. More precisely, the `server` plugin provides a remote procedure call (RPC) interface to foreign applications. The protocol is organized in three logic layers, organized as follows:

1. Many external entry points, based on various networking and system facilities
2. A generic logic run-time responsible for scheduling the requests coming from the various entry points
3. The Frama-C implementation of requests handler, at the kernel or plug-in level

The intermediate, logic layer, is responsible for adding a small bit of parallelism upon the intrinsically synchronous behavior of Frama-C. This makes Frama-C resembling an asynchronous RPC server.

The externally visible layer is only focused on transporting external requests to the logic layer, and transporting back the results to the caller. The only requirement for an entry point is to be able to transport a sequence of 1-input message for 1-output message over time.

The concrete layer is implemented by the Frama-C kernel and its plug-ins. All requests must be registered via the Frama-C Server OCaml API in order to be accessible from the entry-points. Some parts of this documentation are automatically generated from the registered requests.

From a functional point of view, logical requests are remote procedures with input data that reply with output data. Each request is identified by a unique name. Input and output parameters are encoded into JSON values.

To adapt the internal synchronous Frama-C implementation with the external asynchronous entry points, requests are classified into three kinds:

`GET` to instantaneously return data from the internal state of Frama-C

`SET` to instantaneously modifies the state or configure Frama-C plug-ins

`EXEC` to start a resource intensive analysis in Frama-C

During an `EXEC` request, the implementation of the all resource demanding computations shall repeatedly call the yielding routine `!Db.progress()` of the Frama-C kernel to ensures a smooth asynchronous behavior of the Server. During a yield call, the Server would be allowed to handle few `GET` pending requests, since they shall be fast without any modification. When the server is idled, any kind of requests can be started.

From the entry points layer, the asynchronous behavior of the Server makes output data and input data to be dispatched into different messages. However, from the Client side, we still want to have one response message for each incoming message. However, answer messages might contains output data from potentially any previously received requests.

When the client has no more requests to send, but is simply waiting for pending requests responses, it must periodically send polling requests to simply get back the expected responses.

To implement those features, the Client-Server protocol consists of a sequence of paired intput messages and output messages. Each single input message consists of a list of commands:

Commands	Parameters	Description
`POLL`	-	Ask for pending responses, if any
`GET`	id,request,data	En-queue the given `GET` request
`SET`	id,request,data	En-queue the given `SET` request
`EXEC`	id,request,data	En-queue the given `EXEC` request
`KILL`	id	Cancel the given request or interrupt its execution
`SHUTDOWN`	-	Makes the server to stop running

Similarly, a single output message consists of a list of replies, listed in table below:

Replies	Parameters	Description
`DATA`	id,data	Response data from the identified request
`ERROR`	id,message	Error message from the identified request
`KILLED`	id	The identified request has been killed or interrupted
`REJECTED`	id	The identified request was not registered on the Server

The logic layer makes no usage of identifiers and simply pass them unchanged into output messages in response to received requests. At the transport message layer, input and output data are made of a single JSON encoded value. Requests are identified by string, and request identifiers can be of any type from the entry-points.

Remark the `GET`, `SET` or `EXEC` behavior of a request is actually defined by the request implementation, from the Frama-C internal side. The Server will silently ignore the request kind from the incoming messages and use the actual internal one instead. The distinction still appears in the transport protocol only for a purpose of information, as clients shall know what they are asking for.

Implementations of entry points layers shall provide a non-blocking fetch function that possibly returns a list of commands, associated with a callback for emitting the paired list of replies. The Server main loop is guaranteed to invoke the callback exactly once.

The Server plug-in implements two entry-points, however, other Frama-C plugins might implement their own entry-points and call the `Server.Main.run fetch ()` function to make the server starting and exchanging messages with the external world.

It is the responsibility of Frama-C plug-ins to implement and register requests into the Server to make them accessible via any entry point. Whereas data is encoded into JSON structures at the transport layer, requests are processes with well typed OCaml types from the internal side.

Hence, the requests implementations also requires data encoder and decoders to be defined. Some predefined data types are provided by the Server plug-in, but more complex types can be defined and shared among plug-ins via the Server.Data module factory.

Registration of requests, data encoder and decoders always comes with their markdown documentation thanks to the Markdown library provided by the Frama-C kernel. Hence, a full documentation of all implemented requests with their data formats can be generated consistently at any time.

5.3 Dome framework

Dome is a JavaScript library based on Electron and React in order to provide the basic blocks for creating desktop applications based on these libraries, and relies also on Yarn to install additional JS libraries if needed. It forms the basis of the two clients described below, Dive and Ivette.

5.4 Dive

Dive is the first client developed against Frama-C Server. It plays a role similar to the Studia plugin mentioned in section 5.1: it is meant to explore data dependencies between the memory locations of the program, in order to find the origin of imprecisions in Eva analysis. It consists in two parts. First, there is a Frama-C plug-in Dive, that can be installed against Frama-C main distribution with the standard sequence of commands.

```
./configure
make
make install
```

The plugin can inspect the abstract state computed by Eva in order to compute data dependencies and registers the appropriate requests with the server.

As mentioned above, the client part is based on the Dome framework.Additionally, it uses the Cytoscape graph visualization library. It is named `dive-electron` and started by typing `make dev` in the directory of the application. It will then listen for Frama-C Server on port `9000` of `localhost`. Hence the server must be started with

```
frama-c [eva-options or -load state.sav] -server-zmq tcp://127.0.0.1:9000
```

`dive-electron` will then let the user navigate in the call graph of the application, displaying the dependencies of the memory locations of the currently focused function.

5.5 Ivette

Ivette, the Integrated Verification Environment Toolkit for Trusted Execution, is a prototype GUI for program verification. It is also based on javascript and the electron framework and uses the yarn package manager to gather all its external dependencies. The compilation is done by

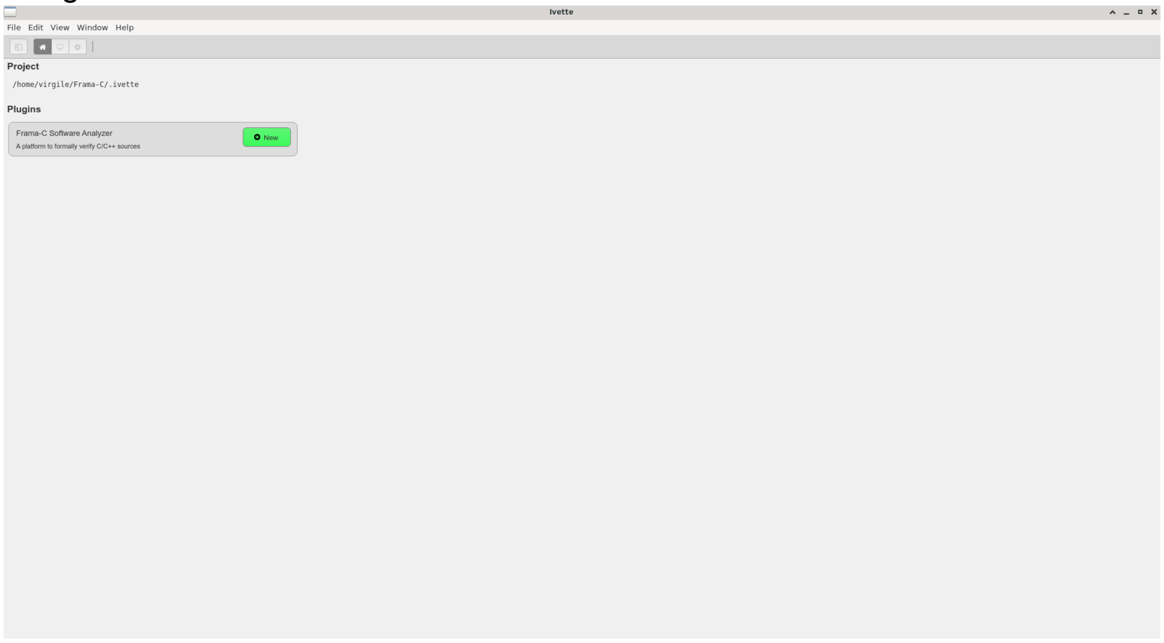
```
./configure
make app
```

However, for convenience, a linux version of the application is readily available as the `dist/linux-unpacked/ivette` executable in the `ivette` archive.

The process for creating a Frama-C view for Ivette is currently a bit cumbersome, and the view does not propose all the features of the classic Frama-C GUI: basically only viewing the source code of functions and displaying the status of ACSL annotations and emitted alarms is supported, by loading a Frama-C state previously saved with

```
frama-c [other options] [C files] -save <file.sav>
```

Once the `ivette` executable has been launched, the main window should look like the following:



Project

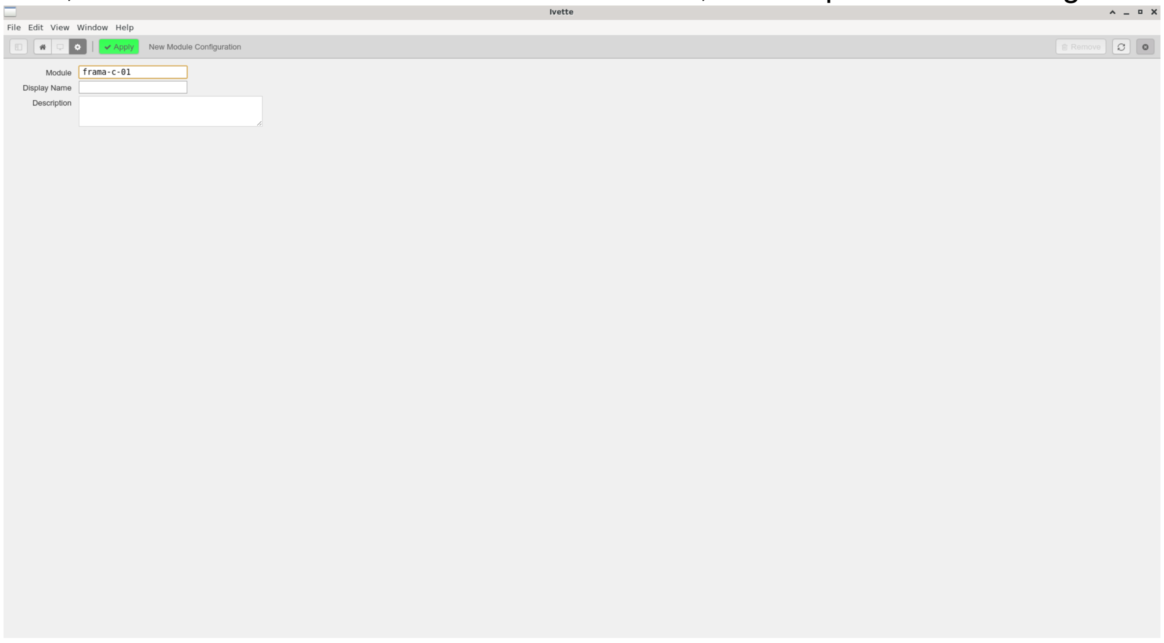
/home/virgile/Frama-C/.ivette

Plugins

Frama-C Software Analyzer

A platform to formally verify C/C++ sources

First, one must create a new module for Frama-C, which opens the following screen:

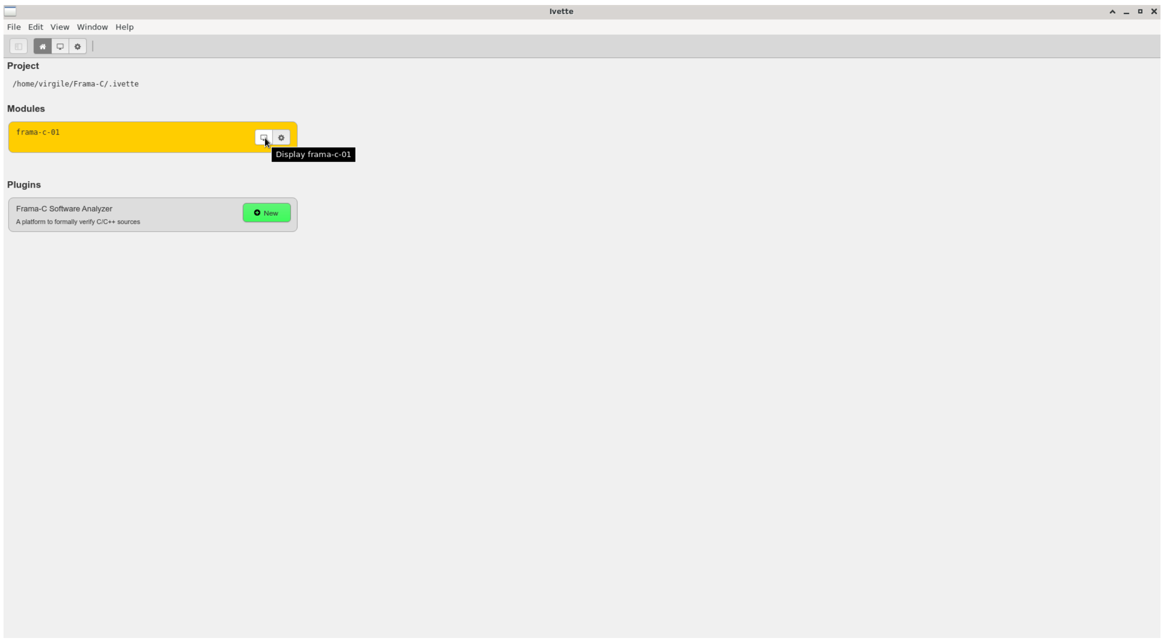


Module frama-c-01

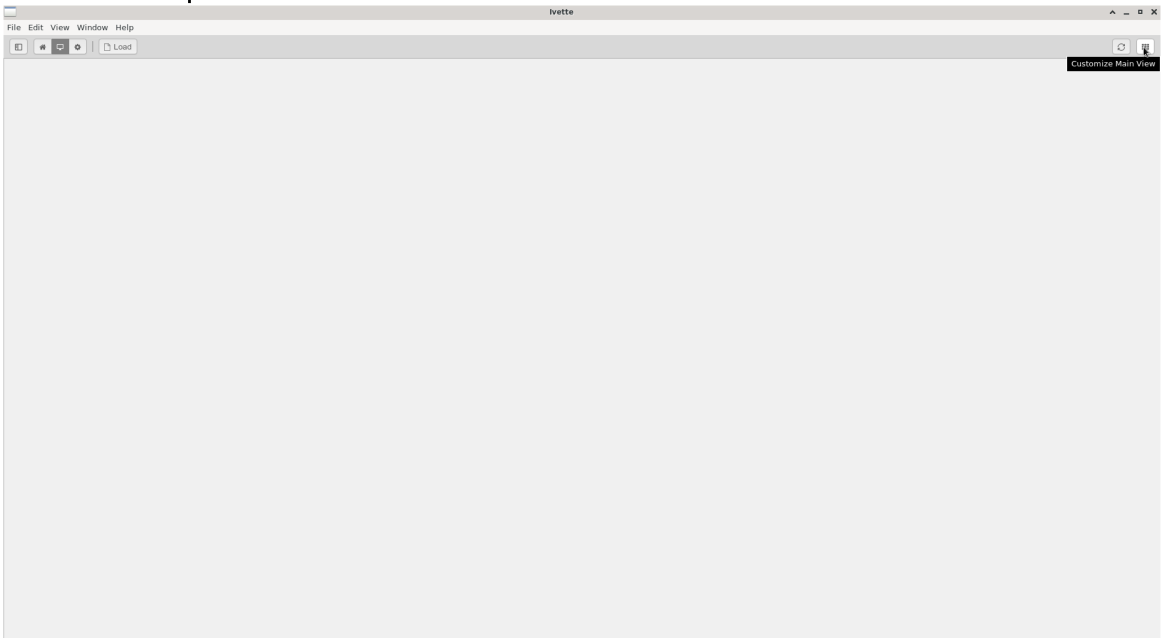
Display Name

Description

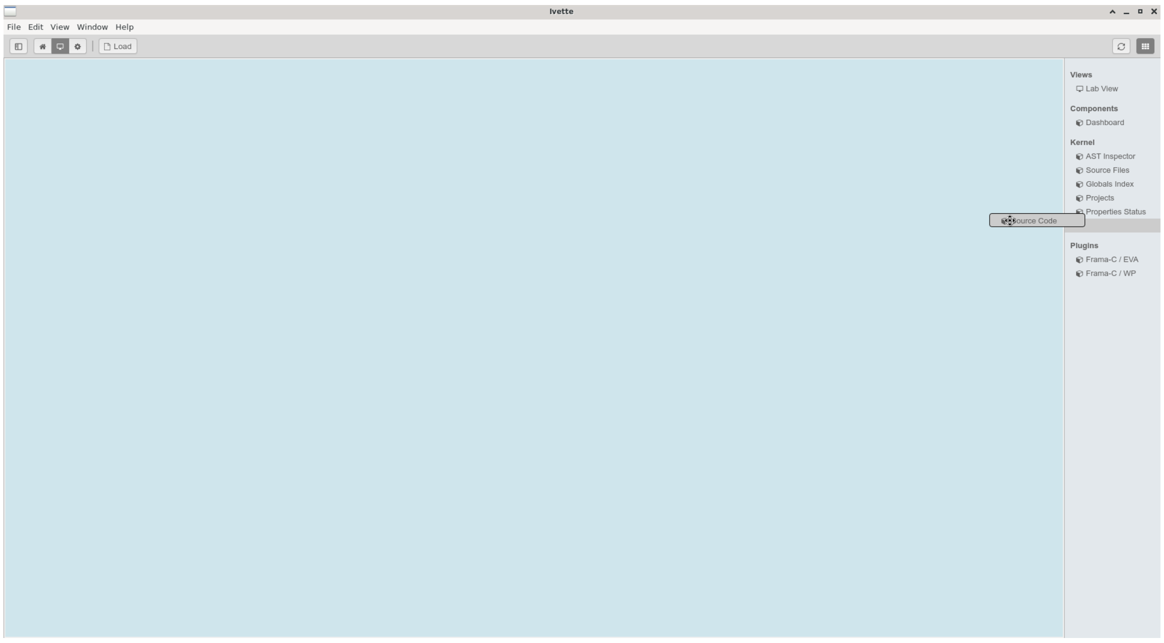
The name can be freely changed, but cannot include spaces. Once the module is created (by applying the changes), we are back to the main screen, but with the possibility of displaying our new module:


Ivette
File Edit View Window Help
Project
/home/virgile/Frama-C/.ivette
Modules
frama-c-01
Display frama-c-01
Plugins
Frama-C Software Analyzer
A platform to formally verify C/C++ sources
New

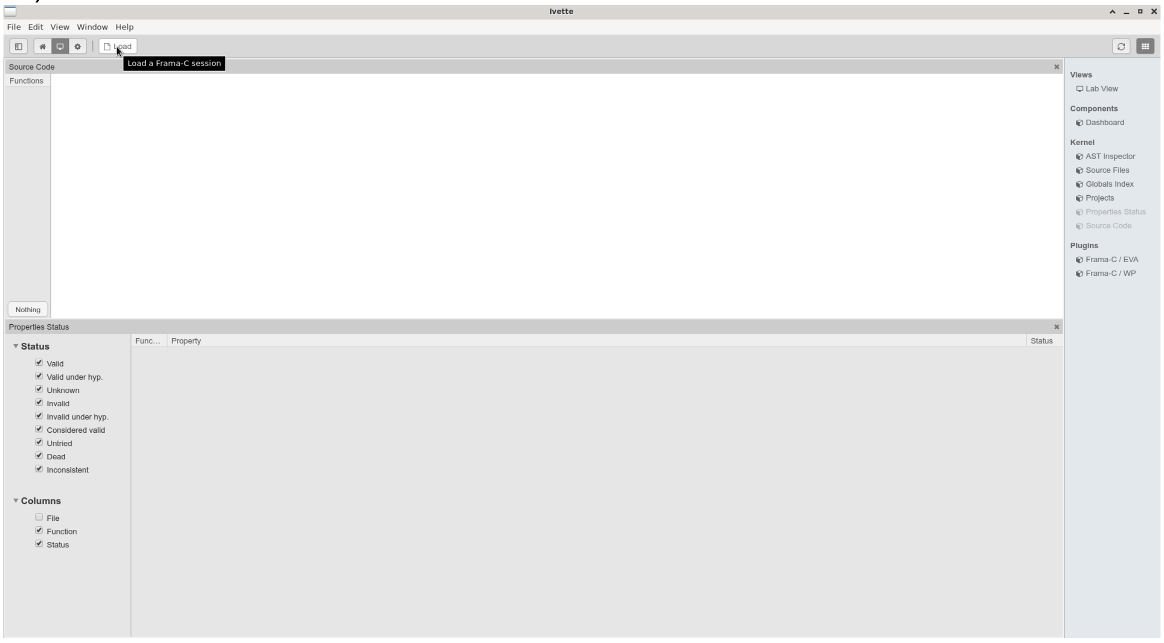
The view of the module is initially empty. In order to populate it, we must open the menu of available components:



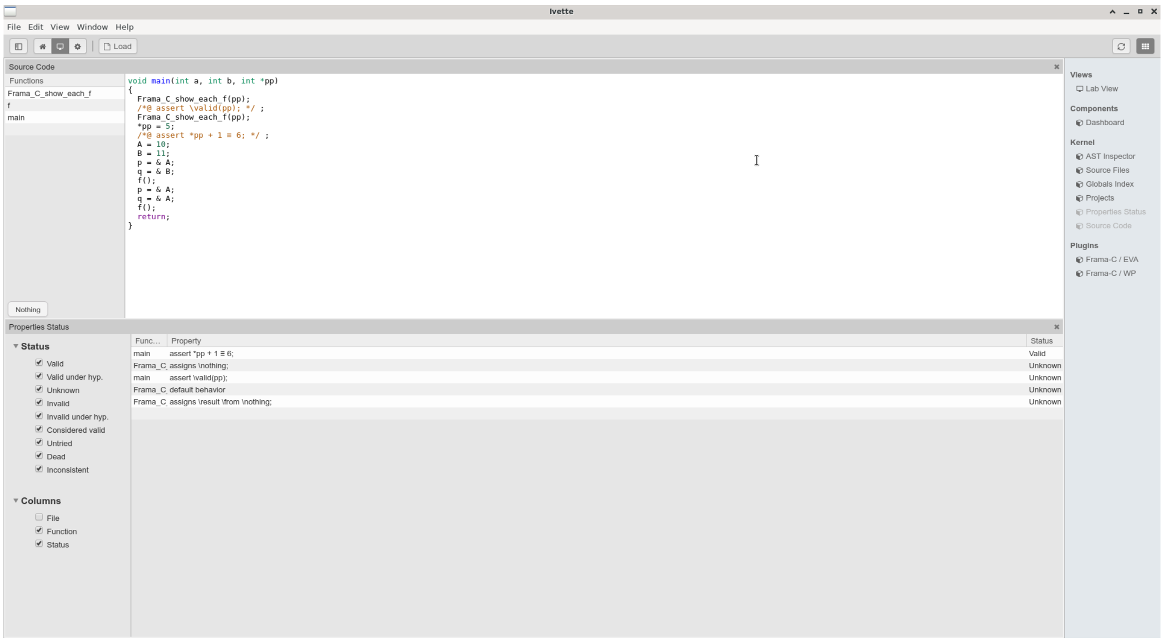
This is done by dragging and dropping the components of interest into the main view. Note that currently only `Source Code` and `Properties Status` are really supported. The relative size of each component can be changed afterwards.



Then, it is possible to load the Frama-C state (it must have been created by the Frama-C version contained in this deliverable, whose executable must be in the `PATH` when launching Ivette).



Then, we can see the properties and the source code of the functions:



```
void main(int a, int b, int *pp)
{
  Frama_C_show_each_f(pp);
  /*@ assert \valid(pp); */ ;
  Frama_C_show_each_f(pp);
  *pp = 5;
  /*@ assert *pp + 1 ≡ 6; */ ;
  A = 10;
  B = 11;
  p = & A;
  q = & B;
  f();
  p = & A;
  q = & A;
  f();
  return;
}
```

Func…	Property	Status
main	assert *pp + 1 ≡ 6;	Valid
Frama_C	assigns \nothing;	Unknown
main	assert \valid(pp);	Unknown
Frama_C	default behavior	Unknown
Frama_C	assigns \result \from \nothing;	Unknown

Chapter 6

Conclusion

This report presented the software packages that compose the deliverable D3.5 of VESSEDIA, which is the result of the work done in tasks T3.1, T3.3 and T3.4. As a short summary, it consists in

- the Diversity tool with its extension to deal with function calls
- the Frama-C tool, with its GUI and its server plugin (distributed in the main archive)
- the Frama-C plugins Meta, RPP and Dive (distributed externally)
- the java-script based clients of Frama-C's server Dive-Electron and Ivette
- the AWS-Lambda based verification services

All of these tools have shown promising results within VESSEDIA and have good perspectives of future developments in the near future.