D3.2
Preliminary Version of the Platforms

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:	Other
Deliverable reference number:	DS-01-731453 / D3.2/ V1.0
Work package contributing to the deliverable:	WP 3
Due date:	August 2018 – M20
Actual submission date:	30th August, 2018

Responsible organisation:	Fraunhofer FOKUS
Editor:	Jens Gerlach
Dissemination level:	PU
Revision:	1.0

Abstract:	In this document we describe the requirements and a preliminary implementation of a verification service for deductive verification of IoT and other applications.
Keywords:	Internet of Things, deductive verification, Frama-C, verification as a service,

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

Editor

Jens Gerlach (FOKUS)

Contributors

Jens Gerlach, Jochen Burghardt (FOKUS)

Disclaimer

Executive Summary

The goal of this document is to outline the current state of the implementation of the so-called *verification service* for modular deductive verification. We discuss the application context of modular deductive verification with Frama-C/WP and formulate basic requirements for the implementation of such a service. We then discuss our preliminary implementation, highlight current limitations and formulate ideas for further developments.

Contents

Chapter 1 Introduction

Fully formal static analyses usually require a considerable amount of computing power, both for abstract interpretation and Hoare-style deductive verification.

Tools that rely on abstract interpretation, e.g. the EVA plug-in of Frama-C, can process the source code of complete programs. They are limited, however, to the verification of a predefined set of (important) properties, such as, absence of numeric overflows and illegal memory accesses. On the other hand, abstract interpretation tools don't need a lot of manual preparation of the source code. Altogether this makes abstract interpretation tools a good choice for the *system-wide verification* of said properties.

Deductive verification tools, such as the WP plug-in of Frama-C and VeriFast can verify more elaborate, user-defined properties of the source code. These tools require, however, that *formal function contracts* are provided that state the intended functional behaviour. Depending on the properties that are to be verified this can lead to a substantial amount of work. At the same time, deductive verification can be relatively easily applied in a modular fashion. In short, deductive verification tools are best applied on the level of *individual components*.

Chapter 2 Requirements for a verification service

2.1 Computational demands of verification with Frama-C/WP

Frama-C/WP analyses the source code of a function (including its contract and other formal annotations) and transforms it according to the rules of the weakest precondition calculus into so-called *verification conditions (VC)*.

These verification conditions are first order formulas that are further simplified and can then be submitted to various theorem provers (both automatic and interactive provers are supported). If all verification conditions have been proven, then the function satisfies its contract.

Natively, Frama-C/WP supports the automatic prover Alt-Ergo[1] and the interactive prover Coq[2]. More provers can be used when combining Frama-C/WP with the Why3[3] deductive verification platform.

Using automatic provers is of course preferable but, due to theoretic limitations, in general not feasible for all VCs of a given function. In order to avoid, as much as possible, the use of interactive provers, it is highly recommended to use several automatic provers. Experience shows that the capabilities of automatic provers sometimes complement each other.

The number of verification conditions generated by Frama-C/WP depends both on the complexity of the source code and the formal contract. Even for a C function of a few lines more than hundred verification conditions can be generated. Exploring their validity with one or more automatic provers is the main reason for the high computational demands on deductive verification.

2.2 Workflows of deductive verification

Deductive verification of a function usually follows an *iterative* workflow. Often it takes many attempts to identify and verify the right formal code annotations (loop invariants, assertions and/or statement contracts) that help showing that the function satisfies its contract. This means that Frama-C/WP and its associated provers have to be executed again and again. This can be quite time consuming, even though Frama-C/WP allows selecting subsets of verification conditions in order to speed up this task.

Moreover, changes in the tool chain, e.g., deploying newer versions of Frama-C/WP, Why3 or of the theorem provers necessitate the re-verification of an already verified function.

[1] http://alt-ergo.lri.fr
[2] https://coq.inria.fr
[3] http://why3.lri.fr

In short, deductive verification is usually not a one-shot activity but requires the repeated application of the verification tool chain.

2.3 The potential for parallel verification

Once Frama-C/WP has generated the verification conditions they can be verified independently from each other. This applies even more to the parallel application of different provers to a given verification condition.

Frama-C/WP has supported the parallel execution of theorem provers for a long time. However, as of now it is necessary to run Frama-C/WP, Why3 and the provers on a single, parallel machine with shared memory.

2.4 The idea of a verification service

The basic idea of a verification service for Frama-C/WP is to separate the task of *generating* the verification conditions from the task of *invoking the provers* on the generated verification conditions.

There are two main advantages of using a verification service.

- The first advantage is to speed up the process of discharging the verification conditions. This, of course, relies on the fact that there is not too much overhead involved with sending verification conditions to the server and receiving the respective results.
- The second advantage has more to do with organizational issues. Selecting, installing and updating theorem provers is sometimes a nuisance since these tools are developed independently from the Frama-C tool chain and often in academic contexts (meaning provers appear and vanish in unexpected ways and that support is scarce).

2.5 The two meanings of verification service

In a narrow sense a verification service is a tool that accepts one or more verification conditions from a client, invokes theorem provers on them and reports the results back to the client. In this sense a verification service is similar to distcc[4], which can be used to speed up the compilation of source code by using distributed computing resources.

In a wider sense a verification service is an infrastructure built around such a program that provides additional services such as

- scheduling verification requests for various clients
- storage of verification conditions for further exploration by provers (including newer versions of the provers or variation of prover parameters)

[4] https://github.com/distcc/distcc

- possibly billing for the used computing resources

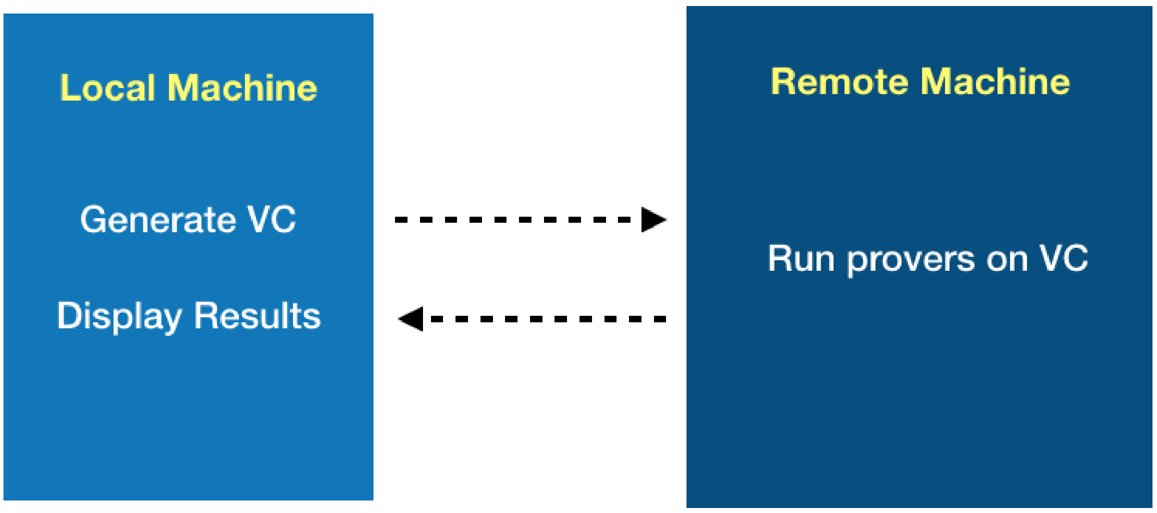
In the first sense, a verification service can be easily deployed by whatever party that needs its capabilities. In the second sense, questions of data privacy and trade secrets can easily arise when the service is managed by a third party.

Chapter 3 A preliminary implementation

The basic idea of a verification service is to enable to execution of the various components of Frama-C/WP's verification tool chain on different machines. Before we explain our approach we look at the core activities of Frama-C/WP when verifying a function. These consist of

- generating VCs out of annotated source code
- invoking provers on these VCs
- gathering the results into a verification report
  - optionally: displaying results in the Frama-C graphical user interface

The following figure shows our simple approach to execute the provers in parallel on a remote machine.



Here is a more detailed explanation of our approach.

- Frama-C/WP processes locally (on the client machine) the annotated source code and generates the verification conditions
- the generated verification conditions are then transferred in bulk to a remote machine
- on the remote machine Why3 invokes the various provers in parallel on the verification conditions
- the results are transferred back in bulk to the client machine
- on the client machine the results are gathered into a verification report

3.1 Details of the preliminary implementation

The verification service allows one to run the Frama-C/WP back end provers on a remote computer, preferably on a fast and powerful multi-core computer. To this end, the user calls the script vs.sh with an optional *mode specifier* (see below), a *source file* argument and any additional Frama-C/WP command-line parameters.

The script vs.sh

- runs Frama-C/WP locally to create proof obligations only (option -wp-gen),
- sends them to the remote computer (except those that Frama-C/WP's built-in simplifier Qed could already decide)
- runs the script vs_remote.sh on the remote computer
- transfers its results back to the local computer
- prints some statistics on stdout.

The remote script vs_remote.sh, in turn,

- receives the proof obligation files and stores them in a temporary directory
- generates a list of proving tasks (see below)
- schedules prover processes as appropriate to work out that list
- sends the prover outputs back to the local computer.

3.2 The task list

The task list generated by script vs_remote.sh depends on the mode that was specified on the command-line of script vs.sh. In parallel mode (option -p), each of the specified provers is run on each of the specified proof obligations. In daisy chain mode (default, or option -d), on each proof obligation, the specified provers are run in the specified order until one succeeds.

As an example, assume Frama-C/WP generates two proof obligations p and q, and the user specifies the prover list

```
-wp-prover a -wp-prover b -wp-prover c -wp-prover d
```

In daisy chain mode, the task list is

```
p:a/b/c/d q:a/b/c/d
```

while in parallel mode, the task list is

```
p:a p:b p:c p:d q:a q;b q:c q:d
```

Tasks are separated by blanks. To *accomplish a task* means to run, in the given order, each prover from the list until one succeeds.

Thus, in daisy chain mode, prover a is run on obligation p, and we assume for sake of explanation, it fails. Then prover b is run on p, which fails, too. Then prover c is run on p, and succeeds. In this case, prover d needn't be run on the task p. Similarly, the obligation q is handled. Note that the obligations p and q can be handled in parallel.

In parallel mode, we have 8 = 2x4 tasks from the start, one for each possible combination of obligation and prover. All of them can be handled in parallel.

Chapter 4 Limitations

We list here a couple of limitations of our preliminary implementation. While some of them are relatively easy to implement, others require additional support from the Frama-C framework.

4.1 Implementing a verification daemon

As of now the script vs_remote.sh is directly executed by the user through secure shell invocation on the remote machine. This assumes that the user has control over the remote machine. It would be more convenient and, more importantly, safer if the functionality of this script would be implemented as a *daemon*, that is, as a *background process* that runs on the remote machine.

4.2 Display of verification results

As of now, it is not possible to display the remote verification results with the Frama-C graphical user interface. This would require a substantial re-implementation of the GUI. Requirements for such a re-implementation have been gathered in the context of Task 3.4.

4.3 Better support for Frama-C options

The script supports only few Frama-C/WP's command-line options. In particular, the option -wp-prop is currently not fully supported. For example, when -wp-prop=-@lemma is given, none of the lemmas are to be proven, but all of them may be used, hence all of them are translated to Why3 language. Since the script vs_remote.sh collects the proof goals from the Why3 files, it has no information about which lemmas are to be proven. (Cf. https://bts.frama-c.com/view.php?id=2371.)

4.4 Improvement of task scheduling

The script vs_remote.sh detects the number of available CPUs and runs up to that number of parallel task processes (using the xargs command). In most cases only few proof obligations will remain to be tried by the provers at the end of the prover list. As a consequence, not all available CPUs are used in the final phase of vs_remote.sh task work-out. Using a more sophisticated scheduling program, user-time could be saved by running several provers in parallel.

4.5 Implementation of a proof cache

During a session with Frama-C/WP to get a piece of code 100% verified, often an *edit-verify-edit* cycle is used where alternatingly small changes to source code or/and ACSL annotations are made and Frama-C/WP is run. Therefore, many obligations are proven repeatedly, since they didn't change since the previous edit. In order not to waste time by proving them again and again, a caching mechanism could be used. Such a *session mechanism* is available within Why3 and it should be investigated how it can be best used.

Chapter 5 Conclusions

In this document we have motivated the need for a *distributed discharge* of verification conditions that occur in a modular deductive verification approach. We have discussed the application context and have presented the current state of the implementation of a verification service that uses computation resource from the cloud. We will continue working to overcome the limitations mentioned in Chapter 4.