D2.3
Basic Analyzers - Final Release

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:	Report
Deliverable reference number:	DS-01-731453 / D2.3 / 1.0
Work package contributing to the deliverable:	WP2
Due date:	August 2019 - M32
Actual submission date:	3rd of September, 2019

Responsible organisation:	KUL
Editor:	Bart Jacobs
Dissemination level:	PU
Revision:	1.0

Abstract:	This report refers to, and briefly describes, the final versions of the front-ends and specification libraries developed in Task 2.1, the improved analyzers developed in Task 2.2, and the prototypes of new analyzers developed in Task 2.3.
Keywords:	software verification, software analyzers

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

Editor
Bart Jacobs(KUL)

Contributors
David Cok (2-CEA)
Balázs Berkes (4-SLAB)
Allan Blanchard (6-INRIA)
Bart Jacobs (8-KUL)
Oscar Llorente (9-FD)

Executive Summary

This document reports on the work done as part of Tasks 2.1, 2.2, and 2.3. The work was centered around the following five technologies:

- The FlowGuard data-flow integrity enforcement technology for C and C++ programs
- ACSL++ and Frama-Clang, which extend the ACSL specification language and the Frama-C verification framework for C programs with support for C++ programs
- An extension of Frama-C with support for *ghost code*
- The VeriFast tool for modular formal verification of C and Java programs
- The SecuRate metric for prioritizing verification alarms

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's software industry faces the considerable challenge of delivering software for powering IoT devices and applications that meets both the market's stringent cost and time-to-market constraints, and users' expectations of privacy and security. Indeed, a number of well-publicized cases of IoT devices being hacked, allowing attackers to spy on users or even inflict physical damage, indicate that current industry practice fails to meet these goals.

The VESSEDIA project aims to address this challenge by developing methodologies, metrics, and tools that enable industry actors to more cost-effectively assess, verify, and certify the security of the IoT software they develop. Work Package 2, in particular, collects the VESSEDIA activities concerning verification tool development. Within Work Package 2, Tasks 2.1, **Programming Languages Artefacts**, 2.2, **Extensions of Existing Analyses**, and 2.3, **Runtime Vulnerability and Attack Protection through Data and Control Flow Integrity Enforcements** are concerned with the development of individual verification tool building blocks, whereas Tasks 2.4, **Composite Analysis Tools**, and 2.5, **Integration into the Software Development Lifecycle**, are concerned with composing building blocks to achieve advanced functionalities. The present deliverable reports on the former; Deliverable 2.4 reports on the latter.

Since, in the current state of the art of verification tools, no tool is capable of offering the strongest security guarantees without any user intervention, all currently proposed tools make a trade-off between the strength of the security properties verified, and the amount of user intervention required. Choosing the right tool then depends on weighing the cost of security issues versus the cost of verification in the context of a particular software development project: for a project where the risk of residual security vulnerabilities remaining in the delivered product is relatively tolerable, tools that require little or no user intervention are preferrable, whereas for a project where such residual vulnerabilities would be catastrophic, tools that require significant user expertise and effort but provide high assurance of security are justified.

To cover this spectrum and in this way support all types of software development projects, VESSEDIA verification tool development has focused on the following four tools, supporting three modes of operation:

- The *FlowGuard* tool is a *dynamic analysis tool*. Rather than verifying the absence of vulnerabilities ahead of deployment, it monitors program execution at run time and detects ongoing attacks that attempt to exploit vulnerabilities to manipulate a program's data flow. It requires little or no user intervention.

- *Frama-C Eva* is an *abstract interpretation tool*, built on the *Frama-C* framework for verification tools for the C programming language. It analyzes a program's source code

for particular types of security properties, by computing an abstraction that overapproximates the program's set of reachable states. In contrast to dynamic analysis, if successful it can guarantee the absence of certain types of vulnerabilities in all possible executions of the program. The user intervention required is moderate: it is limited to tuning the abstraction mechanism. Its verification power is limited by the fact that it supports a fixed set of built-in abstractions, which means that it supports only the security properties that can be expressed using those abstractions and the programs that can be analyzed using those abstractions.

- *Frama-C WP* is a *deductive verification tool*, also built on Frama-C. In contrast to abstract interpretation, it is *modular*: it verifies, for each of the program's modules separately, that the module satisfies the security properties specified by the user. Compared to abstract interpretation, it can provide stronger guarantees and support more complex programs, at the cost of requiring the user to supply a specification for each module, as well as other user-defined abstractions such as loop invariants and data structure specifications.

- *VeriFast* is another deductive verification tool. Whereas Frama-C WP is based on classical Hoare logic, VeriFast is based on an extension of Hoare logic called *separation logic*. Separation logic forces the user to specify *ownership* of all memory locations and other resources used by the program, further increasing the amount of user intervention required; in return, this allows VeriFast to support more advanced programming constructs such as, in particular, concurrency. VeriFast supports the C and Java programming languages.

Whereas the FlowGuard tool was developed from scratch as part of the VESSEDIA project, the Frama-C tools[1] and VeriFast[2] are relatively mature tools that have existed for a long time; the development done as part of VESSEDIA has consisted in making significant improvements to these tools, enhancing their applicability to IoT verification use cases in important ways.

The structure of the remainder of this report is as follows:

- Chapter 2 reports on the **FlowGuard** technology, implemented as a plugin for the industry-standard GCC C compiler, for hardening programs against attacks that manipulate a program's variables without hijacking their control flow. Such attacks are therefore not prevented by control-flow integrity protections.

- While many IoT applications are written in C, at least as many are written in C++. Chapter 3 reports on the work done to extend the Frama-C framework to also support the C++ programming language. This involves in particular developing ACSL++, an extension of Frama-C's ACSL notation for C program annotations, as well as Frama-Clang, a Frama-C module that translates ACSL++-annotated C++ programs to ACSL-annotated C programs. The latter is based on the parser and typechecker of the popular Clang C++ compiler.

- Chapter 4 reports on work done to add support for a program annotation construct known as ***ghost code*** to **Frama-C**. This construct underlies many important verification techniques, enabling verification of additional properties and program patterns.

[1] `https://frama-c.com/`
[2] `https://github.com/verifast/verifast`

- Chapter 5 reports on work done in the context of the **VeriFast** tool for modular formal verification of C and Java programs. Specifically, it reports on progress made in verifying deadlock-freedom of programs that use condition variables, and in verifying the I/O behavior of programs.
- All of the VESSEDIA verification tools generate so-called *alarms* when potential security issues are detected. An important challenge for users is to decide which alarms to investigate and address first. Chapter 6 reports on work implementing **SecuRate**, a metric for prioritizing verification alarms based on the number of affected products and the number of affected devices.

Chapter 2

FlowGuard

2.1 Introduction

FlowGuard implements the Data-Flow Integrity (DFI) enforcement for C and C++ code. DFI emerges as a prevention mechanism for non-control data attacks, which exploit vulnerabilities to overwrite security critical data without subverting the intended control-flow in the program.

2.1.1 Non-Control-Data Attacks

A non-control-data attack differs from a control-data attack (i.e., code-injection, code-reuse) in that it does not affect the control-flow of a program. Instead, it follows legitimate control-flow transfers and modifies program logic or decision-making data. The security-critical data that may be subjected to these attacks includes configuration data, user input data, user identity data (e.g., UIDs and GIDs), decision-making data (e.g., a boolean value indicating authentication status), passwords and private keys, randomized values (e.g., canaries, randomized addresses), and system call parameters.

As an example, the OpenSSL *Heartbleed* vulnerability [10] allowed a remote attacker to expose sensitive data, such as private keys, using a non-control-data attack. On the heartbeat request/response protocol, an attacker could request a heartbeat using a legitimate payload but providing a payload length field larger (up to 65,535 bytes) than the real payload. Then, the response protocol would craft a response by copying the original payload in a buffer allocated with size equal to the payload length field. Since the payload length field was not correctly verified against the length of the real payload, a memory leakage was possible.

2.1.2 Data-Flow Integrity

The DFI enforcement generates the Data-Flow Graph (DFG) of the program leveraging static analysis. Concretely, it computes a DFG that contains a set of definitions, assigns an identifier to each definition, and maps those identifiers to instructions. This way, the DFG shows the instructions that assigned a value to each used variable.

Secondly, it instruments the program introducing data-flow integrity checks that ensure, before every variable use, that its definition is within the statically generated reaching definition identifiers.

Finally, it enforces at runtime that the data flow of the program is allowed by the DFG, if not, the data-flow integrity property does not hold, and the program is terminated.

2.2 Prototype Details

FlowGuard has been implemented as a plug-in to the popular GCC compiler for C and C++ languages. As a prototype, its intended usage is for research purposes only. It is distributed under the GNU GPL open source license.

To install the tool, the user can proceed as follows:

1. Install dependencies: `./dependencies.sh`
2. Install GCC version 5.4.0: `./getgcc.sh`
3. `cd flowguard; make plugin`

2.3 Usage

To use the tool, follow these steps.

1. First, make sure the tool is properly installed. See the installation instructions above.
2. To use this tool we need to generate the static Data-Flow Graph of the program, and to instrument the program based on that DFG, to then at runtime ensure that the program follows the statically defined DFG. When a program is compiled the tool generates the DFG and injects the required instrumentation. Then we compile the runtime library and insert this static DFG which is the one that will be checked against at runtime. Once the lib is compiled with the static DFG in it, we link the instrumented program to the runtime lib.
3. There is a Makefile that automates the compile and link process. `make test` is the easiest way to compile a program, translate the DFG into object files, compile the runtime instrumentation lib and link the program against the runtime lib.

2.4 Changes

The following changes were made to correct and enhance the functionality of the plugin:

1. During the compilation process, there was an issue that, at first, looked like a problem that affected variables stored in static storage (when obtaining the first operand of the expression via `TREE_OPERAND(XXX, 0)`) leading to a segmentation fault and preventing the compilation to finish. In the end, we discovered that it was a more general problem that affected every expression node that had 0 operands, and thus the `TREE_OPERAND` operation would keep providing an invalid value. To solve it, the plugin now first computes the number of operands in an expression node before accessing its operands.
2. Fixed a problem that caused the plugin to not load the statics for C programs because it could not find the main function of the program and consequently could not enforce the Data-Flow Integrity.

3. A compilation error was triggered when handling structures that contain a bit-field that immediately follows another bit-field. This happens because bit-fields can overlap at RTL level. We have corrected the way in which these cases are handled, in a similar way as other sanitizers do.

4. Programs compiled with the plugin showed a runtime error that was triggered because, under certain conditions, instrumentation code was inserted to check if a definition was within the previously statically generated identifiers before inserting the call to set the corresponding definition identifier. In the new version, it is ensured that the corresponding “set definition” call is inserted at some point before a “check definition” call.

5. When inserting instrumentation to check the definition of PHI function expressions that dereference a pointer that is incremented or decremented in a previous different path, the plugin crashed due to an incorrect handling that resulted in a compilation error when expanding from GIMPLE to RTL. These cases triggered an error because an invalid expression that contained multiple consecutive operators was built in consequence. The new version detects these conditions and builds expressions correctly.

2.5 Changelog

- 20190810: release beta version
- 20190401: release alpha version
- 20180615: release pre-alpha version

Chapter 3

ACSL++ and Frama-Clang

While many IoT applications are written in C, at least as many are written in C++. This chapter reports on the work done to extend the Frama-C framework to also support the C++ programming language. This involves in particular developing ACSL++, an extension of Frama-C's ACSL notation for C program annotations, as well as Frama-Clang, a Frama-C module that translates ACSL++-annotated C++ programs to ACSL-annotated C programs. The latter is based on the parser and typechecker of the popular Clang C++ compiler.

For this chapter, the following three artifacts are included in this deliverable by reference:

- The document *ACSL++: ANSI/ISO C++ Specification Language*
- The document *Frama-Clang User Manual*
- The *Frama-Clang* software distribution

They are described below.

3.1 ACSL++

One goal of the VESSEDIA project is to advance the definition and capability of the ACSL++ (ANSI/C++ Specification Language) language and tools. There has been for many years a definition of a specification language for ANSI-C, namely the ANSI-C Specification Language (ACSL) [1]. This language definition underpins the Frama-C tool suite that enables performing a variety of tool-supported program analysis and deductive verification tasks. The goal of the ACSL++ effort is to extend the ACSL language to support C++ and to enhance the Frama-C tool suite to accept C++ programs as input.

Incorporated by reference within this deliverable is the stand-alone document *ACSL++: ANSI/ISO C++ Specification Language*. This document is a substantial enhancement of the ACSL document [1] (which it incorporates). Both of these are examples of Behavioral Interface Specification Languages [5], as are JML [3, 12] for Java, SPARK [7] for Ada, Spec# for C# [13], Dafny [6], Eiffel [8], VeriFast [11] for separation logic, Vercors [2] for concurrent programs, and so on.

3.1.1 Enhancements added in ACSL++

Though some of the work of supporting C++ in Frama-C was begun in earlier projects, under VESSEDIA the definition of ACSL++ was crystallized and documented. In particular, this

required defining how specifications would operate for C++, an object-oriented language, in conjunction with specifications for C, a low-level language. Key features of ACSL++ are these:

- logic functions and predicates within classes and as member functions or static or virtual
- class invariants
- specifications of templates
- specifications appropriate to functional programming and lambda expressions
- name scoping in classes and namespaces
- inheritance of specifications, including multiple inheritance
- specifications for C++'s for-range loop
- exceptions and exception specifications
- dynamic type information and delegation
- attributes such as `[[noreturn]]` and noexcept
- C++ casts
- pure functions usable in specifications
- type inference: C++'s auto and decltype features
- first-class boolean literals and types

3.2 Frama-Clang

Frama-Clang is a module for the Frama-C framework that translates ACSL++-annotated C++ programs into ACSL-annotated C programs; the resulting programs can then be analyzed using other Frama-C components (known as *plugins*) such as Frama-C Eva and Frama-C WP. Like ACSL++, the development of Frama-Clang was started before the start of VESSEDIA. During VESSEDIA, the pre-existing Frama-Clang prototype was enhanced to parse ACSL++ and, as far as is currently supported by the Frrama-C kernel, to represent the C++ and ACSL++ features in a form that can be used by Frama-C and its multitude of existing plug-ins.

3.2.1 Technical work

Preprocessor

Prior to this work, text written within ACSL++ annotations was not subject to standard C/C++ preprocessing in a uniform way; only a partially implemented, custom preprocessor had been written. Under VESSEDIA, the design and implementation were changed to use the popular Clang C++ compiler for preprocessing and lexing the source files. Clang was already being

used to parse the C++ portion of source files, so this step further advanced the integration with and use of Clang.

There is still a custom lexer that is used in the following workflow:

- First the file is broken into tokens without preprocessing in order to identify preprocessing directives that are not permitted in ACSL++ annotations (the details of these design decisions are described in the user manual). Illegal preprocessing directives result in errors and then are elided from the input so processing can continue, if possible, with the remaining text.
- The text, possibly modified, is then submitted to Clang and a sequence of preprocessor tokens is retrieved. Clang does all of the preprocessing steps as described in the C/C++ standards.
- These Clang tokens are then converted to ACSL++ tokens. ACSL++ tokens are used for two reasons. First, the lexer tokens for ACSL++ are slightly different than the C/C++ preprocessor tokens, so some conversion is needed. For example, ACSL++ includes keywords that begin with a backslash, which is not a legal single token in C++. Second, by using ACSL++ tokens we can reuse much of the infrastructure that was already present for handling ACSL++ source text and ASTs.
- The ACSL++ token stream is then parsed and ASTs are generated and passed on to the Frama-C kernel.

Parser

The second significant technical accomplishment was a rewrite and enhancement of the parser. Although the work on the scanner and preprocessor described above was intended to meld with the existing scanner, this goal proved not the best long-term solution. In fact as addional ACSL++ features were added to the software implementation the original parser was found to be too costly to maintain, fix, and expand. It had been written as a handgenerated Bison-like parser but with state and actions encoded in function pointers, rather than in a tool-generated table. As a result the grammar implemented by the tool was implicit in the actions and difficult to correctly fix or change.

The decision was made to replace this parser with a recursive descent parser in which non-terminals of the grammar are explicitly clear as named function calls in the implementation. The difficulty of such an implementation is that it is best suited for LL grammars, which C++ is not. However, ACSL++ mostly is. The areas in which it is not are described in the user manual, along with the design solutions that were adopted in the new implementation.

3.2.2 User manual

A user manual has been written to accompany the software described in the previous section. Once installed along with Frama-C, the Frama-Clang front-end mostly acts invisibly to the user: C++ files are processed by Frama-Clang without user intervention. The user manual describes the various options available to modify the behavior of the front-end and the limitations of the current implementation. The user manual includes a technical summary of some implementation details, useful to future maintainers and developers.

Chapter 4

Ghost code support in FRAMA-C

Formal verification of source code often relies on the use of ghost code to make explicit some information that is otherwise implicit. Basically, it consists in adding some code that can be seen by the tool used for verification. In order to assure that the verification is correct, the code must not modify the behavior of the program (otherwise we would verify a program that has a different semantics). Hence, ghost code can read the variables of the program but not modify them, it must always terminate and it must not change the control flow of the program.

Currently, FRAMA-C does not provide the ability to write certain kinds of ghost code and does not check that the ghost code is non-interferent with the original code. In the context of Task 2.2 of VESSEDIA, we have improved the support for ghost code in FRAMA-C. This prototype version of FRAMA-C:

- accepts ghost arguments to functions,
- keeps the ghost status of ghost functions,
- checks that ghost code is not interferent,
- accepts a ghost qualifier for types.

The support for the ghost arguments and ghost functions status has been added by modifying the FRAMA-C kernel and more precisely the parsing and typing phases and is thus an improvement of the platform itself. The verification of the non-interference of the ghost code and the ghost qualifier are provided by a plugin called MEDIUM.

4.1 Example code

Figure 4.1 presents an example of code using the ghost syntax for parameters (see for example lines 9, 15, 37), functions (lines 11–12, 24–34) and function calls (line 38) as well as the `_ghost` qualifier to specify that some memory belongs to the ghost world (lines 9, 12, 15, 26, 37).

In FRAMA-C 19 Potassium, this code cannot be parsed by the kernel, and the ghost code is not verified to be non-interferent. Our prototype verifies all of this[1]. For example, one can slightly modify the code above, for example by declaring the local variable `l` on line 16 as a

[1]Note that one particular aspect of non-interference, that is termination, is not verified by MEDIUM but can be verified using other plugins. The documentation associated to this protoype tells more about it.

```
1  #define NULL ((void*) 0)
2
3  struct list {
4    struct list *next;
5    int k;
6  };
7
8  struct list * list_pop (struct list ** list)
9               /*@ ghost (struct list * _ghost * array, int index, int n) */;
10
11 /*@ ghost
12     void array_pop (struct list * _ghost * array, int index, int n) ; */
13
14 struct list * list_pop (struct list ** list)
15                /*@ ghost(struct list * _ghost * array, int index, int n) */ {
16   struct list *l = *list;
17   if(*list != NULL) {
18     //@ ghost array_pop(array, index, n) ;
19     *list = (*list)->next;
20   }
21   return l;
22 }
23
24 /*@ ghost
25
26 void array_pop (struct list * _ghost * array, int index, int n) {
27   int i = index;
28   while (i < index + n - 1){
29     array[i] = array[i+1];
30     i++;
31   }
32 }
33
34 */
35
36 void caller_code(struct list ** list)
37       /*@ ghost (struct list * _ghost * array, int index, int n) */ {
38   list_pop(list) /*@ ghost (array, index, n) */ ;
39 }
```

Figure 4.1: A code sample using the new ghost support

ghost variable, or adding a line modifying the field `array[i]->k` between lines 28 and 29, to see that when it is done the plugin detects the violations.

An archive containing the source code for the modified and extended version of FRAMA-C, as well as a document describing the result in greater detail, are attached to this deliverable.

Chapter 5

VeriFast

VeriFast is a research prototype of a tool for modular formal verification of correctness properties of single-threaded and multithreaded C and Java programs. It takes as input the source code files for a Java or C program module, annotated with specifications that express the module's intended correctness properties, expressed in a variant of separation logic, as well as any necessary proof hints, such as loop invariants. It then, without further user interaction and usually in a matter of seconds, reports either "0 errors found" or the source location of a verification failure. If it reports "0 errors found", this means that all possible executions of the provided module are safe (i.e. do not crash and do not access unallocated memory or overrun any buffers) and comply with the provided specifications. Otherwise, the user can inspect the failed symbolic execution path in a debugger-like GUI.

KUL has been developing VeriFast since 2008. It is available as open source at `https://github.com/verifast/verifast` under an MIT license.

During Period 1 of the VESSEDIA project, we have extended VeriFast's support for I/O verification, crypto verification, automation, and verification of deadlock-freedom of programs that use monitors. For details on this work, see D2.1.

During Period 2, we have developed further VeriFast's support for verifying deadlock-freedom of concurrent programs that use monitors or channels. This has led to a publication at ECOOP 2019 [4], and a submission to the ACM TOPLAS journal (under review). These results will also appear in Jafar Hamin's PhD thesis (expected to be defended in September 2019). As part of this work, we have further developed the machine-readable formalization and machine-checked correctness proof of the theory underlying our proposed approach; it can be found at `https://github.com/jafarhamin/deadlock-free-monitors-soundness`.

Additionally, we have done further work on verifying I/O behavior with VeriFast, leading to a publication at FTfJP 2019 [9].

Chapter 6

SecuRate

A description of this result can be found in a separate document attached to this deliverable.

Bibliography

[1] P. Baudin, J. C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C Specification Language, version 1.10 (2013). `http://frama-c.cea.fr/acsl.html`.

[2] Stefan Blom and Marieke Huisman. The vercors tool for verification of concurrent programs. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, pages 127–131, Cham, 2014. Springer International Publishing.

[3] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.

[4] Jafar Hamin and Bart Jacobs. Transferring obligations through synchronizations. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom.*, volume 134 of *LIPIcs*, pages 19:1–19:58. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

[5] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-01, University of Central Florida, School of EECS, Orlando, FL, March 2009.

[6] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] J. W. McCormick and P. C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.

[8] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[9] Willem Penninckx, Amin Timany, and Bart Jacobs. Specifying I/O using abstract nested Hoare triples in separation logic. In *21st Workshop on Formal Techniques for Java-like Programs (FTfJP 2019)*, 2019. Available at `https://people.cs.kuleuven.be/bart.jacobs/ftfjp19.pdf`.

[10] US-CERT. Openssl 'heartbleed' vulnerability (cve-2014-0160). `https://www.us-cert.gov/ncas/alerts/TA14-098A`.

[11] Frédéric Vogels, Bart Jacobs, and Frank Piessens. Featherweight verifast. *Logical Methods in Computer Science*, 11(3), 2015.

[12] Many papers regarding JML can be found on the JML web site: http://www.jmlspecs.org.

[13] The Spec# web site gives code, documentation and papers: http://research.microsoft.com/SpecSharp/.