Evidential and Continuous Integration of Software Verification Tools

Tewodros A. Beyene(✉) and Harald Ruess

fortiss — An-Institut Technische Universität München, Munich, Germany
beyene@fortiss.org

1 Introduction

The complexity of embedded software and increasing demands on dependability, safety, and security has outpaced the capabilities of current verification and certification methods. In particular traditional verification and certification methods based on manual reviews, process constraints, and testing, which are mandated by current safety standards such as DO-178C [1] and DO-278A [2] for airborne systems and air traffic management systems, ISO 26262 [11] in the automative domain, and IEC 61508 for industrial domains including factory automation and robotics are proving to be overly time- and resource-intensive. For example, costs for developing certification evidence in *safety cases* according to the DO-178C standard have been shown to range between \$50 to \$100 per executable line of code, depending on the required safety level [15]. Unless mission-critical embedded software can be developed and verified with less cost and effort, while still satisfying the highest dependability requirements, new mission-critical capabilities such as autonomous control may never reach the market.

In this short paper, we present an overview of our approach for automating the process of creating certification evidence for mission-critical software. This framework supports the *integrated verification* of a wide range of complementary approaches to software verification, including automated tools and methods such as model checking and static analysis [3,7], and manual and consensus-driven approaches such as code review processes [8]. A *workflow pattern* for a given verification activity (e.g., analysis, review or test) specifies which methods to be used and how the methods are integrated in the given verification activity [10]. Our framework takes a single *workflow pattern* as an input to perform such an integrated verification task.

The framework is said to be *evidential* as verification evidence, which form the basis for certification processes, are automatically generated from pre-defined *workflow patterns*. This is done by chaining evidence from combination of the formal software analysis methods [5] as given in the *workflow patterns* [12]. The framework also supports *continuous* verification by executing verification and by generating corresponding evidence during each iteration of an agile development process. For this, the framework is technically based on the widely used *Jenkins CI* [6]. Therefore, our framework supports integrated verification, where verification evidence are automatically generated and updated continuously as software development progresses.

K. Havelund et al. (Eds.): FM 2018, LNCS 10951, pp. 679–685, 2018.
https://doi.org/10.1007/978-3-319-95582-7_45

Our tool integration framework is inspired and also closely related to SRI's evidential toolbus (ETB) [4,14,16], which is a distributed workflow-based tool integration framework for constructing claims supported by evidence. A main difference is our choice of basing our integration framework on an widely used continuous integration framework. This design choice allows us to seamlessly integrate our verification framework into a large number of industrial software development infrastructures. Our prototype implementation uses *Jenkins CI*[1] as it provides the following crticial services for our integration framework: distributed computing capabilities, notion of analysis evidence, and Interaction mechanism with humans.

2 Verification Activities, Workflows and Patterns

The creation of assurance cases as the basis for certification is labour-intensive and largely manual. This process usually starts by developing a verification plan for determining adequate verification methods and tools together with acceptance criteria for successful verification runs. These verification plans are executed and verification results are, more or less manually, compiled into an assurance case as the basis for certification.

More generally, verification planning may be viewed as defining workflows for the selected verification activities such as analysis, review and test. A *verification workflow* is a sequence of steps applying verification methods and tools with the aim of ensuring that a system under verification satisfies its specification. Verification workflows, together with their verification methods and tools, are identified and defined during the verification planning phase of a project [10].

```
workflow CodeReview
  input
    prog $P$ of type SourceCode
  tools
    infer of type StaticAnalysis
    cppCheck of type StaticAnalysis
    CBMC of type ModelChecking
  begin
    $R_1 :=$ infer($P$)
    $R_2 :=$ cppCheck($P$)
    $R_3 :=$ Merge($R_1, R_2$)
    $R_4 :=$ Refine(CBMC, $R_3, P$)
    return $R_4$
  end
```

Fig. 1. A code review workflow

```
workflow pattern CodeReview
  input
    prog $P$ of type SourceCode
  parameter
    tool set $T$ of type StaticAnalysis
    tool $M$ of type ModelChecking
  begin
    $R := \{\}$
    for each $t \in T$ do
      $R_t :=$ ApplyStaticAnalysis($t, P$)
      $R :=$ Merge($R, R_t$)
    done
    $R :=$ Refine($M, R, P$)
    return $R$
  end
```

Fig. 2. A code review workflow pattern

[1] https://jenkins.io.

An example code review workflow, which is inspired and extends Holzmann's [8,10] portfolio approach, is provided in Fig. 1. This code review workflow takes a program source code P as input and produces a review report R with potential defects. The workflow applies static analysis using *Infer*[2] and *cppCheck*[3] (Lines 1 & 2), and merges analysis results (Line 3). The merged result is further refined (e.g., false positives from static analysis are detected and therefore excluded from the review report) by calling the function Refine, which employs the CBMC [13] model checker (Line 4).

These kinds of workflows are usually instantiations of given *verification workflow patterns*. An example workflow pattern for a code review verification process is specified in Fig. 2. Like the workflow in Fig. 1, the workflow pattern takes a program source code P as an input, and produces a review report R as an output. However, the specification of specific verification tools are not required in the definition of the workflow pattern. Instead, it is parameterised by a set of static analysis tools T and a model cheker M. The workflow pattern applies each static analysis tool over the source code (Lines 2–5), and collects the analysis results (Line 4). It also refines the collected report further by tagging, for example, false positives (Line 6).

The advantage of using *verification workflow patterns* is twofold: (1) choice of specific verification tools implementing verification methods in the workflow pattern can be made during the actual verification execution phase of the project, and (2) a given workflow pattern can be flexibly instantiated with different verification tools resulting in completely different verification workflows. In this way, our tool integration framework provides a flexible way of instantiating verification workflow patterns according to the heterogeneous needs and requirements of different industrial software development environments and supporting tool infrastructures. For example, the code review workflow pattern in Fig. 2 can be instantiated in a number of different ways by assigning different values for the parameters T and M. Consider, for example, the following three possible instantiations: $\{(T = \{\text{infer, cppCheck}\}, M = \text{CBMC}), (T = \{\text{infer, coverity}$[4]$\}, M = \text{CBMC})$, and $(T = \{\text{infer, coverity}\}, M = \text{SPIN}$ [9]$)\}$. The first instantiation is actually equivalent to the workflow in Fig. 1.

As verification methods form an integral part of *verification workflow patterns*, each verification method must be defined in terms of inputs and outputs. For example, the verification method *Static Analysis*, which is used in the *verification workflow pattern* of Fig. 2, can be defined as taking a *program source code* as an input and producing a *set of errors* as an output. Any verification tool implementing a given verification method must agree on inputs and outputs with the verification method.

[2] http://fbinfer.com.
[3] http://cppcheck.sourceforge.net.
[4] https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html.

3 Structure of the Integration Framework

Our tool integration framework, as shown in Fig. 3, is built on top of a CI framework. It contains three additional components, namely *Patterns Database*, *Tools Server* and *Integration Engine*, for instrumenting and configuring the framework with the specific verification needs and available resources of a given software development project.

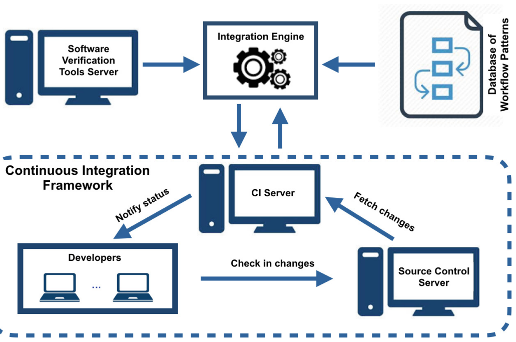

Fig. 3. The Integration Framework

One instrumentation deals with adding each verification tool, which implements certain verification method, to the *Tools Server* component. Another instrumentation deals with adding *workflow patterns*, which are created for the planned verification activities of the project, to the *Patterns Database* component. These instrumentations make the framework ready for integrated verification.

Users can perform an integrated verification by executing the framework with the appropriate *workflow pattern* and tools to instantiate the verification methods specified in the given *workflow pattern*. The *Integration Engine* is responsible for instantiating verification methods with the proper verification tools (as provided by the user), collecting outputs of each tool, and composing these outputs into high-level verification evidence.

4 Illustration

Let us run our framework with the *verification workflow pattern* in Fig. 2, parameter instantiations $T = \{\text{infer}, \text{coverity}\}$ and $M = \text{SPIN}$, and with the input program listed in Fig. 4. The generated review report is listed in Fig. 5. Let us consider two of the error entries in the report:

Error E1 is initially reported during static analysis by *Infer* and *Coverity* as a possible *dereference of null pointer on line 50 of the source file*. Then, the refinement procedure of our framework (implemented as function REFINE, and whose refinement logic depends on the type of error) tries to refute the error

```
 6:  struct BankInfo{ ... };
             ...
14:  int fileNotClosed(int fd){
15:    fd = open(csvFile, O_WRONLY | O_CREAT | O_TRUNC, 0600);
16:    if (fd != -1) {
17:      char buffer[256];
18:      write(fd,buffer,strlen(buffer));
19:    }
20     return fd;
21: }
               ...
46:  int main(){
47:    int fd;
48:    fileNotClosed(fd).close();
49:    BankInfo *PostB = 0;
50:    return PostB->Revenue;
51:  }
```

Fig. 4. An example source code fragment

No	Error Type	Location	Status
E1	*null dereference*	example.c (line 50)	**verified error**
	pointer PostB could be assigned null on line 49 and is dereferenced at line 50. [*code analysis tools: infer, coverity, model checking tool: SPIN*]		
E2	*resource leak*	example.c (line 20)	**false positive**
	resource acquired to fd at line 18 is not released. [static analysis tools: {*infer, coverity*}, model checking tool: *SPIN*]		
E3	*memory leak*	example.c (line 39)	**not refined**
	memory acquired by getMem at line 33 is never released. [*code analysis tools: infer, coverity*]		
E4	*division by zero*	example.c (line 43)	**verified error**
	variable count could be assigned 0 on line 41 and is used in division operation at line 43. [*code analysis tools: infer, coverity, model checking tool: SPIN*]		

Fig. 5. Report generated by our framework

claim by adding the assertion '$max \neq$ NULL' on line 50 and running SPIN. Since SPIN proves the assertion does not hold and provides a counter-example, the process concludes that the error is a real violation. The counter-example from SPIN further supports the initial error claim. These kinds of additional evidence are added to the error report as the basis for further investigation, for example, in a code review meeting.

Error E2 is initially reported during static analysis as a possible *resource leak error* on line 20. The refinement procedure now encodes the eventual release of the file resource in LTL and applies SPIN. The model checking by SPIN actually succeeds as the resource is later released in the main function, i.e., the potential error does not actually materialize. Therefore, this error is marked as *false alarm* in the final review report. The set of applied tools as well as verification outputs will be kept as verification evidence for every verification decision made by the framework.

We have applied our framework with the code review workflow pattern on the Toyota static analysis benchmarks.[5] Although the refinement procedure of our framework is not defined for all type of possible defects addressed in this benchmark, the framework at least is able to refine many false positives for the type of errors it can handle (such as null dereference and division by zero errors). We have also used the framework in a project within the Airbus Group as a front end for a tool-based code review solution for Ada programs. In this project, the framework is able to integrate Ada code analysis tools, such as GNATProve and AdaCtl, with an in-house developed Ada model checker.

5 Conclusion

We have presented a tool integration framework for supporting the automated and continuous verification of mission-critical software. First, the framework supports *integrated verification* as it applies a workflow-based combination of complementary software analysis methods. Second, the framework is *evidential* as verification evidence, which form the basis for certification, are automatically generated from pre-defined *verification workflow patterns* by chaining results

[5] https://github.com/regehr/itc-benchmarks/.

from the integrated software analysis tools. Third, the framework is *continuous* as it is aimed at executing verification and generating corresponding evidence during each iteration of an agile development process.

We are in the process of launching and applying this tool integration framework in a number of industrial verification efforts for safety- and security-related software in the automotive and the aerospace domain. The ultimate goal here is to automatically generate assurance cases according to sector-specific safety standards such as ISO 26262, DO178C, or ECSS. In the future, we also plan to use this integrated verification framework for the on-line generation of certification evidence during operation, for example, for adaptive and learning-based control systems.

Benefits of using our tool integration framework include: (1) flexible and seamless integration into agile industrial software development processes, (2) integration of a number of complementary automated software verification tools with more process-oriented methods such as code review as mandated in industrial safety standards, (3) formal and automated process from verification planning down to producing corresponding verification evidence during development and in accordance with industrial safety standards; (4) considerable reduction of certification effort by means of automated generation of verification evidence; (5) instantaneous and up-to-date verification evidence and corresponding assurance cases as the basis for guiding agile development processes.

References

1. RTCA DO-178C Software Considerations in Airborne Systems and Equipment Certification. RTCA Standard, December 2011
2. RTCA DO-278A Software Integrity Assurance Considerations for Communication, Navigation and Air Traffic Management (CNS/ATM) Systems, December 2011
3. Ábrahám, E., Havelund, K.: Some recent advances in automated analysis. Int. J. Softw. Tools Technol. Transf. **18**(2), 121–128 (2016)
4. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool integration with the evidential tool bus. In: VMCAI (2013)
5. Denney, E., Pai, G.: Evidence arguments for using formal methods in software certification. In: Software Reliability Engineering Workshops (ISSREW), Nov 2013
6. Duvall, P., Matyas, S.M., Glover, A.: Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley Professional, Boston (2007)
7. Groce, A., Havelund, K., Holzmann, G., Joshi, R., Xu, R.-G.: Establishing flight software reliability: testing, model checking, constraint-solving, monitoring and learning. Ann. Math. Artif. Intell. **70**, 315–349 (2014)
8. Havelund, K., Holzmann, G.J.: Software certification: coding, code, and coders. In: EMSOFT 2011. ACM, New York, NY, USA (2011)
9. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual, 1st edn. Addison-Wesley Professional, Boston (2011)
10. Holzmann, G.J.: SCRUB: a tool for code reviews, December 2010
11. ISO: Road vehicles - Functional safety (2011)
12. Kelly, T.P., McDermid, J.A.: Safety case construction and reuse using patterns. In: Software Reliability Engineering Workshops (ISSREW) (1997)

13. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26
14. Moura, L.D., Owre, S., Ruess, H., Rushby, J., Shankar, N.: Integrating verification components. Theories, Tools, Experiments. In: Verified Software (2005)
15. RTI Real-Time Innovations: DDS for Safety-Critical Applications (2014)
16. Rushby. J.: An evidential tool bus. In: Proceedings of ICFEM (2005)