Runtime Assertion Checking and its Combinations with Static and Dynamic Analyses
Tutorial Synopsis[⋆]

Nikolai Kosmatov and Julien Signoles

CEA, LIST, Software Reliability Laboratory, PC 174
91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

Abstract. Among various static and dynamic software verification techniques, runtime assertion checking traditionally holds a particular place. Commonly used by most software developers, it can provide a fast feedback on the correctness of a property for one or several concrete executions of the program. Quite easy to realize for simple program properties, it becomes however much more complex for complete program contracts written in an expressive specification language. This paper presents a one-hour tutorial on runtime assertion checking in which we give an overview of this popular dynamic verification technique, present its various combinations with other verification techniques (such as static analysis, deductive verification, test generation, etc.) and emphasize the benefits and difficulties of these combinations. They are illustrated on concrete examples of C programs within the Frama-C software analysis framework using the executable specification language E-ACSL.

1 Introduction

Among the most useful techniques for detecting and locating software errors, *runtime assertion checking* (RAC) is nowadays a widely used programming practice [1]. *Assertions* offer one of the most convenient and scalable automated techniques for detecting errors and providing information about their locations, even for errors that are traversed during execution but do not necessarily lead to failures. More and more engineers and researchers today are interested in verification tools allowing to automatically check specified program properties at runtime.

This one-hour tutorial proposes a short survey on runtime assertion checking and focuses on combinations of this technique with other static and dynamic verification approaches (such as abstract interpretation, deductive verification, test generation, etc.). While runtime assertion checking is not so difficult to implement for simple program properties, it becomes much more complex for more evolved specification like full function contracts written in an expressive specification language. We discuss the benefits and the difficulties of runtime assertion checking for expressive specifications, and of its combinations with other analysis techniques.

[⋆] This work was partially funded by EU FP7 (project STANCE, grant 317753)

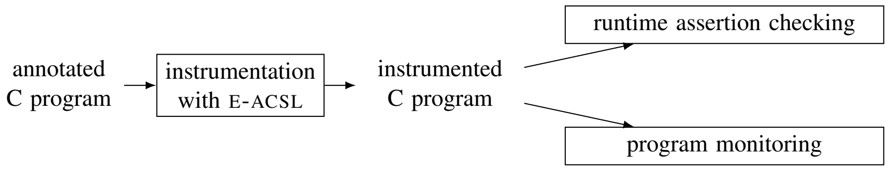

Fig. 1. Basic usages of the E-ACSL plugin translating annotations into C code

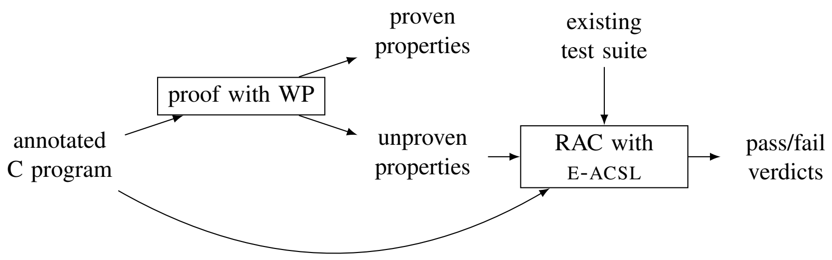

Fig. 2. Combination of deductive verification and runtime assertion checking

Tutorial examples are run in FRAMA-C[1] [2], an open-source software verification toolset, using the executable specification language E-ACSL [3,4]. E-ACSL syntax is intentionally close to C and can be easily learned on-the-fly. FRAMA-C offers various analyzers, such as abstract interpretation based value analysis plugin VALUE, deductive verification plugin WP, test generation plugin PATHCRAWLER [5]. The E-ACSL plugin of FRAMA-C translates E-ACSL annotations into instrumented C code that can be used for runtime assertion checking or program monitoring as illustrated by Fig. 1.

2 Tutorial Outline

In the first part of the tutorial, we give a historical overview of runtime assertion checking and its usage in software engineering. The second part presents runtime assertion checking for E-ACSL, an expressive specification language (including pre- and post-conditions, loop annotations, mathematical integers, quantifications, memory-related constructs, references to specific program points, etc.), and emphasizes the benefits and the issues of this kind of specifications. We also show how different types of errors, sometimes very subtle, can be efficiently detected by runtime assertion checking.

As an illustration of what kind of examples the tutorial provides, Fig. 3 shows a C function which implements binary search and contains E-ACSL annotations enclosed in special comments `/*@ ... */`. Before the function, we specify the function contract. First, the `requires` clauses define preconditions stating that each cell of the array must be a valid memory location, that the array must be sorted and that its length must be positive. The function has two behaviors: if the searched `key` exists, the result of the function is the index where the `key` is found; otherwise (if the `key` does not exist), the

[1] `http://frama-c.com/`

```
/*@ requires \forall integer i; 0 <= i < length ==> \valid(a+i);
  @ requires \forall integer i; 0 <= i < length-1 ==> a[i] <= a[i+1];
  @ requires length >= 0;
  @
  @ behavior exists:
  @ assumes \exists integer i; 0 <= i< length && a[i] == key;
  @ ensures a[\result] == key;

  @ behavior not_exists:
  @ assumes \forall integer i; 0 <= i < length ==> a[i] != key;
  @ ensures \result == -1; */
int binary_search(int *a, int length, int key) {
  int low = 0, high = length - 1;
  /*@ loop invariant 0 <= low <= high + 1;
    @ loop invariant high < length;
    @ loop invariant \forall integer k; 0 <= k < low ==> a[k] < key;
    @ loop invariant \forall integer k; high < k < length ==> a[k] > key; */
  while (low <= high) {
    int mid = low + (high - low) /2;
    /*@ assert low <= mid <= high; */
    if (a[mid] == key) return mid;
    if (a[mid] < key) { low = mid+1; }
    else { high = mid - 1; }
  }
  return -1;
}
```

Fig. 3. Function binary_search annotated in E-ACSL

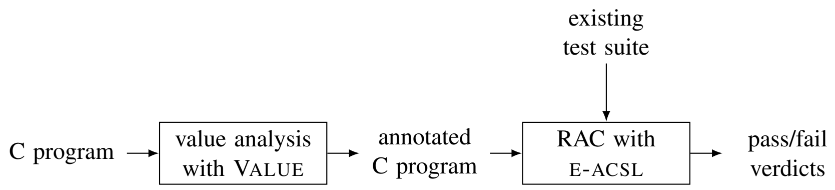

Fig. 4. Combination of value analysis and runtime assertion checking

function returns -1. The body of the function also contains several loop invariants that express invariant properties of the loop, and an assertion.

The last part of the tutorial focuses on combinations of runtime assertion checking with other analyzers. We show the benefits and the limitations of runtime verification used in combination with deductive verification where it can help to quickly check if the program respects an unproven annotation on one or several concrete executions (see Fig. 2). Combinations of abstract interpretation with runtime assertion checking can be beneficial in several ways, for example, by statically validating or invalidating some annotations, avoiding redundant or irrelevant checks to optimize runtime verification, or generating annotations for alarms to be checked (see Fig. 4). Combinations with automatic test generation can be used to check at runtime complex properties on a large test suite even when the properties are too complex to be supported by symbolic test generation techniques directly (see Fig. 5). The combinations will be illustrated on examples of C programs within the FRAMA-C verification framework and using the PATHCRAWLER test generator [5].

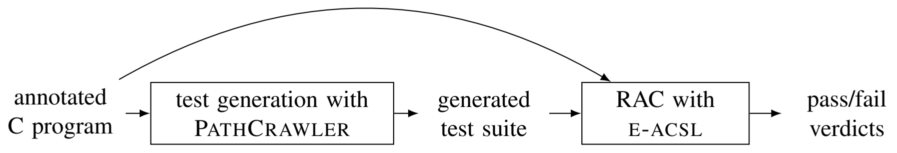

Fig. 5. Combination of test generation and runtime assertion checking

3 About the Presenters

The presenters are researchers at CEA LIST. Nikolai Kosmatov's research interests include software testing, constraint solving and combinations of various software verification techniques. Nikolai gave several theoretical courses and exercise sessions on software testing and proof of programs since 2009. He is the main author of the online testing service `pathcrawler-online.com`. Nikolai co-organized (with Nicky Williams) tutorials on software testing with PathCrawler at TAP 2012, TAROT 2012, ASE 2012 and QSIC 2012[2].

Julien Signoles is one of the main developers of Frama-C. He is also the main author of the E-ACSL plug-in of Frama-C and the E-ACSL specification language. His research focused on software security, runtime assertion checking and combination and applications of program analysis techniques. He taught various theoretical courses and exercise sessions on program specification, proof of programs, abstract interpretation and software testing since 2009. The presenters are co-authors (with Virgile Prevosto) of tutorials on proof of programs with Frama-C at SAC 2013[2], iFM 2013 and TAP 2013, and the authors of the tutorial on runtime assertion checking at RV 2013.

References

1. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. ACM SIGSOFT Software Engineering Notes **31**(3) (2006) 25–37
2. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C, a program analysis perspective. In: the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012). Volume 7504 of LNCS., Springer (2012) 233–247
3. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language. (2013) URL: http://frama-c.com/download/e-acsl/e-acsl.pdf.
4. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: the 28th Annual ACM Symposium on Applied Computing (SAC 2013), ACM (2013) 1230–1235
5. Williams, N., Marre, B., Mouy, P.: On-the-fly generation of k-paths tests for C functions: towards the automation of grey-box testing. In: the International Conference on Automated Software Engineering (ASE 2004), IEEE Computer Society (2004) 290–293

[2] tutorial materials available at `http://kosmatov.perso.sfr.fr/nikolai/`