The E-ACSL Perspective on Runtime Assertion Checking

Julien Signoles

Software Safety & Security Lab

VORTEX'21

July 12, 2021

- Runtime Assertion Checking (RAC)
  - verifying program assertions at runtime
- 70's: primitive `assert`
  - limited to Boolean expressions
- 80's: language `Eiffel`
  - Behavioral Interface Specification Language (BISL)

- since more than 20 years, several works about RAC
  - most about BISL
  - others about combining RAC with others techniques
  - very few about RAC as a compilation technique

- since more than 20 years, several works about RAC
  - most about BISL
  - others about combining RAC with others techniques
  - very few about RAC as a compilation technique
- this talk:
  - visiting the RAC's research area
  - emphasizing the work done on the Frama-C plug-in E-ACSL
    - BISL
    - RAC tool

Frama-C: Framework for analyses of source code written in C

http://frama-c.com

- open source: Frama-C 23-Vanadium released a week ago
- a collection of analyzers
  - each analyzer is a plug-in
  - 34 plug-ins in the latest release
  - 3 main verification plug-ins
    - Eva: abstract interpretation
    - Wp: deductive verification
    - E-ACSL: runtime assertion checking
- extensible: anyone can develop new plug-ins
- collaborative: many ways of combining plug-ins
- support ACSL, a BISL for C code

1. Behavioral Interface Specification Languages
2. Using Runtime Assertion Checking in Practice
3. Compiling Formal Assertions

1. Behavioral Interface Specification Languages
2. Using Runtime Assertion Checking in Practice
3. Compiling Formal Assertions

A Few Existing BISL

Beyond Eiffel

- BISL for mainstream programming languages
  - JML for Java
  - Spec# for C#
  - VCC, ACSL and E-ACSL for C
  - CodeContract for .Net
  - Spark2014 for Ada
  - Gospel for OCaml
- Dedicated BISL
  - Boogie
  - WhyML
- main related verification techniques:
  - RAC
  - Deductive Verification (DV), i.e. proving programs

```
/*@ requires len >= 0;
    requires \valid(a + (0 .. len-1));
    requires sorted(a, len);









   */
int binary_search(int* a, int len, int key);
```

BISL by Example

Function Contract

```
/*@ predicate sorted(int* a, int len) =
    \forall integer i, j; 0 <= i <= j < len ==> a[i] <= a[j]; */

/*@ requires len >= 0;
    requires \valid(a + (0 .. len-1));
    requires sorted(a, len);









     */
int binary_search(int* a, int len, int key);
```

```
/*@ predicate sorted(int* a, int len) =
    \forall integer i, j; 0 <= i <= j < len ==> a[i] <= a[j]; */

/*@ requires len >= 0;
    requires \valid(a + (0 .. len-1));
    requires sorted(a, len);
    assigns \nothing;




     */
int binary_search(int* a, int len, int key);
```

```
/*@ predicate sorted(int* a, int len) =
    \forall integer i, j; 0 <= i <= j < len ==> a[i] <= a[j]; */

/*@ requires len >= 0;
    requires \valid(a + (0 .. len-1));
    requires sorted(a, len);
    assigns \nothing;

    behavior exists:
      assumes \exists integer i; 0 <= i < len && a[i] == key;
      ensures 0 <= \result < len && a[\result] == key;
    behavior not_exists:
      assumes \forall integer i; 0 <= i < len ==> a[i] != key;
      ensures \result == -1;


     */
int binary_search(int* a, int len, int key);
```

BISL by Example

Function Contract

```
/*@ predicate sorted(int* a, int len) =
    \forall integer i, j; 0 <= i <= j < len ==> a[i] <= a[j]; */

/*@ requires len >= 0;
    requires \valid(a + (0 .. len-1));
    requires sorted(a, len);
    assigns \nothing;

    behavior exists:
      assumes \exists integer i; 0 <= i < len && a[i] == key;
      ensures 0 <= \result < len && a[\result] == key;
    behavior not_exists:
      assumes \forall integer i; 0 <= i < len ==> a[i] != key;
      ensures \result == -1;

    complete behaviors;
    disjoint behaviors; */
int binary_search(int* a, int len, int key);
```

BISL by Example

Code Annotation

```
int binary_search(int* a, int len, int key); {
  int low = 0, high = length - 1;









  while (low<=high) {
    int mid = low + (high - low) / 2;

    if (a[mid] == key) return mid;
    if (a[mid] < key) low = mid+1;
    else high = mid - 1;
  }
  return -1;
}
```

```
int binary_search(int* a, int len, int key); {
  int low = 0, high = length - 1;







  while (low<=high) {
    int mid = low + (high - low) / 2;
    /*@ assert 0 <= mid < length; */
    if (a[mid] == key) return mid;
    if (a[mid] < key) low = mid+1;
    else high = mid - 1;
  }
  return -1;
}
```

```
int binary_search(int* a, int len, int key); {
  int low = 0, high = length - 1;
  /*@ loop invariant 0 <= low <= high+1;
    @ loop invariant high < length;








  while (low<=high) {
    int mid = low + (high - low) / 2;
    /*@ assert 0 <= mid < length; */
    if (a[mid] == key) return mid;
    if (a[mid] < key) low = mid+1;
    else high = mid - 1;
  }
  return -1;
}
```

BISL by Example

Code Annotation

```
int binary_search(int* a, int len, 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 0 <= mid < length; */
    if (a[mid] == key) return mid;
    if (a[mid] < key) low = mid+1;
    else high = mid - 1;
  }
  return -1;
}
```

BISL by Example

Code Annotation

```
int binary_search(int* a, int len, 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;
    @ loop assigns low, high;
   */
  while (low<=high) {
    int mid = low + (high - low) / 2;
    /*@ assert 0 <= mid < length; */
    if (a[mid] == key) return mid;
    if (a[mid] < key) low = mid+1;
    else high = mid - 1;
  }
  return -1;
}
```

```
int binary_search(int* a, int len, 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;
    @ loop assigns low, high;
    @ loop variant high - low; */
  while (low<=high) {
    int mid = low + (high - low) / 2;
    /*@ assert 0 <= mid < length; */
    if (a[mid] == key) return mid;
    if (a[mid] < key) low = mid+1;
    else high = mid - 1;
  }
  return -1;
}
```

what is quite standards in most BISLs' annotatations?

- function contracts
  - behaviors: case-spliting
  - exceptional cases (`for languages with exceptions`)
- code assertions
- loop invariants and variants
- frame condition (`assigns`): what may be modified
- data invariants (e.g., object and type invariants)

what is quite standards in most BISLs' logic?

- typed first-order logic
- terms include pure expressions (i.e. side-effect free)

what is quite standards in most BISLs' logic?

- typed first-order logic
- terms include pure expressions (i.e. side-effect free)
- multi-state properties
  - refer to the pre-state's value of some data from the post-state
  - example: `ensures G == \old(G) + 1;`
  - more generally, refer to the value of some data at another program point: `\at(x, L)` (ACSL and E-ACSL)

what is quite standards in most BISLs' logic?

- typed first-order logic
- terms include pure expressions (i.e. side-effect free)
- multi-state properties
  - refer to the pre-state's value of some data from the post-state
  - example: `ensures G == \old(G) + 1;`
  - more generally, refer to the value of some data at another program point: `\at(x, L)` (ACSL and E-ACSL)
- ghost code
  - define and use code as specification
  - without interfering with the original code
  - allow to easily write stateful specifications (e.g. automata)

Main Differences Between BISL

- unbounded quantifiers
  - `\forall integer x, \exists integer y, x * 2 == y`
  - allowed in DV-oriented BISL (expressiveness, math. proof)
  - not allowed in RAC-oriented BISL (no exec. in finite time)

Main Differences Between BISL

- unbounded quantifiers
  - `\forall integer x, \exists integer y, x * 2 == y`
  - allowed in DV-oriented BISL (expressiveness, math. proof)
  - not allowed in RAC-oriented BISL (no exec. in finite time)
- pure functions
  - program's functions without side-effect
  - allowed in RAC-oriented BISL (usability)
  - not allowed in DV-oriented BISL (logic. consistency is uneasy)

Main Differences Between BISL

- unbounded quantifiers
  - `\forall integer x, \exists integer y, x * 2 == y`
  - allowed in DV-oriented BISL (expressiveness, math. proof)
  - not allowed in RAC-oriented BISL (no exec. in finite time)
- pure functions
  - program's functions without side-effect
  - allowed in RAC-oriented BISL (usability)
  - not allowed in DV-oriented BISL (logic. consistency is uneasy)
- frame conditions
  - assignable/writable terms: locations that are written
  - modified terms: locations whose values may have changed
  - below, x is assigned but not modified:

```
tmp = x;
...;      // modify [x]
x = tmp; // restore the former value of [x]
```

Main Differences Between BISL
cont'd

- mathematical numbers
  - integers (i.e., $\mathbb{Z}$)
  - real numbers (i.e., $\mathbb{R}$)
  - allowed in DV-oriented BISL (expressiveness, math. proof)
  - was not allowed in RAC-oriented BISL (no exact representations)
    - integers are now often supported

Main Differences Between BISL
cont'd

- mathematical numbers
  - integers (i.e., $\mathbb{Z}$)
  - real numbers (i.e., $\mathbb{R}$)
  - allowed in DV-oriented BISL (expressiveness, math. proof)
  - was not allowed in RAC-oriented BISL (no exact representations)
    - integers are now often supported
- undefinedness
  - what is the meaning of `1/0 == 1/0`?
  - valid in DV-oriented BISL (reflexivity of equality)
  - undefined in DV-oriented BISL (not executable)
  - strongly valid = valid **and** defined

Outline

1. Behavioral Interface Specification Languages
2. Using Runtime Assertion Checking in Practice
3. Compiling Formal Assertions

- discovering bugs upstream, when assertions are violated

Which Applications for RAC

As a Verification Technique

- discovering bugs upstream, when assertions are violated
- combining RAC with testing, e.g. fuzzing
  - RAC makes more invalid behaviors observable
  - example: a buffer overflow that overwrites some data

Which Applications for RAC

As a Verification Technique

- discovering bugs upstream, when assertions are violated
- combining RAC with testing, e.g. fuzzing
  - RAC makes more invalid behaviors observable
  - example: a buffer overflow that overwrites some data
- combining RAC with static verification techniques, e.g. DV
  - easy in verification framework
    - Frama-C, OpenJML, Spark2014, Why3, etc
  - verify most properties with DV, RAC checks the other ones
    - lower the verification effort
    - lower the runtime overhead
  - RAC helps debug specifications, find counter-examples, understand what's going on

- BISL may be seen as low-level specifications languages
  - close to the code
- compile high-level properties to BISL
  - temporal properties
  - non-interference properties
  - security automata
  - relational properties (e.g., monotony)
  - system-wide properties

Which Applications for RAC

Properties Beyond BISL

- BISL may be seen as low-level specifications languages
  - close to the code
- compile high-level properties to BISL
  - temporal properties
  - non-interference properties
  - security automata
  - relational properties (e.g., monotony)
  - system-wide properties
- make explicit properties that are otherwise left implicit
  - undefined behaviors
  - security weaknesses

1. Behavioral Interface Specification Languages
2. Using Runtime Assertion Checking in Practice
3. Compiling Formal Assertions

The Challenge of Compiling Formal Assertions

RAC is a compilation technique

- RAC compiles assertions into executable code
  - input: `/*@ assert x+1 == 0; */`
  - output: `assert (x+1 == 0);`
- "The run-time checker [of Spec#] is straightforward" [Barnet et al., 2011]

The Challenge of Compiling Formal Assertions

RAC is a compilation technique

- RAC compiles assertions into executable code
  - input: `/*@ assert x+1 == 0; */`
  - output: `assert (x+1 == 0);`
- "The run-time checker [of Spec#] is straightforward" [Barnet et al., 2011]
- always straightforward?
  - maybe not: the example above is unsound, in general
  - maybe not: "the run-time overhead [of Spec#] is prohibitive" [Barnet et al., 2011]

main challenge: being both sound and efficient

- dedicated library (GMP in C) for integers and rationals

```
/*@ assert x + 1 == 0; */
mpz_t e_acsl_1, e_acsl_2, e_acsl_3, e_acsl_4;
int e_acsl_5;
mpz_init_set_si(e_acsl_1, x);                   // e_acsl_1 = x
mpz_init_set_si(e_acsl_2, 1);                   // e_acsl_2 = 1
mpz_init(e_acsl_3);
mpz_add(e_acsl_3, e_acsl_1, e_acsl_2);   // e_acsl_3 = x + 1
mpz_init_set_si(e_acsl_4, 0);                   // e_acsl_4 = 0
e_acsl_5 = mpz_cmp(e_acsl_3, e_acsl_4); // x + 1 == 0
e_acsl_assert(e_acsl_5 == 0);                  // runtime check
mpz_clear(e_acsl_1); mpz_clear(e_acsl_2); // deallocate
mpz_clear(e_acsl_3); mpz_clear(e_acsl_4);
```

sound but not efficient

- dedicated library (GMP in C) for integers and rationals

```
/*@ assert x + 1 == 0; */
mpz_t e_acsl_1, e_acsl_2, e_acsl_3, e_acsl_4;
int e_acsl_5;
mpz_init_set_si(e_acsl_1, x);                   // e_acsl_1 = x
mpz_init_set_si(e_acsl_2, 1);                   // e_acsl_2 = 1
mpz_init(e_acsl_3);
mpz_add(e_acsl_3, e_acsl_1, e_acsl_2);          // e_acsl_3 = x + 1
mpz_init_set_si(e_acsl_4, 0);                   // e_acsl_4 = 0
e_acsl_5 = mpz_cmp(e_acsl_3, e_acsl_4);         // x + 1 == 0
e_acsl_assert(e_acsl_5 == 0);                   // runtime check
mpz_clear(e_acsl_1); mpz_clear(e_acsl_2);       // deallocate
mpz_clear(e_acsl_3); mpz_clear(e_acsl_4);
```

sound but not efficient

- not possible to be exact on real numbers
  - how to check `2 * \pi == \pi + \pi` at runtime?

- dedicated type system for being sound and efficient
  - [Kosmatov et al. @RV 2020]
- use machine bounded numbers and arithmetic whenever possible
- use GMP otherwise
- only a few GMPs integers in practice
  - very efficient in practice
- implemented in E-ACSL for integers and rationals
- adapted in Spark2014 for integers

- how to compile `x == \at(x, L) + 1` ?
  - easy: save the value of `x` at `L`

- how to compile `x == \at(x, L) + 1` ?
  - easy: save the value of `x` at `L`
- how to compile the following predicate?

```
\forall integer i, j;
  0 <= i < LEN ==> 0 <= j < i ==>
  t[i][j] == \at(t[i][j], L) + 1
```

  - issue: `i` and `j` are undefined at `L`

Compiling Multi-State Properties

- how to compile `x == \at(x, L) + 1` ?
  - easy: save the value of `x` at `L`
- how to compile the following predicate?

```
\forall integer i, j;
  0 <= i < LEN ==> 0 <= j < i ==>
  t[i][j] == \at(t[i][j], L) + 1
```

  - issue: `i` and `j` are undefined at `L`
  - "classical" solution: copy the whole array `t`
  - the classical solution is sound but not efficient

Compiling Multi-State Properties

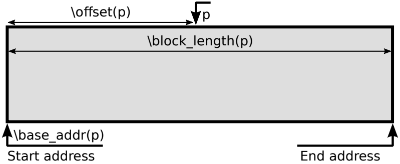
- how to compile `x == \at(x, L) + 1` ?
  - easy: save the value of `x` at `L`
- how to compile the following predicate?

```
\forall integer i, j;
  0 <= i < LEN ==> 0 <= j < i ==>
  t[i][j] == \at(t[i][j], L) + 1
```

  - issue: `i` and `j` are undefined at `L`
  - "classical" solution: copy the whole array `t`
  - the classical solution is sound but not efficient
  - better solution: copy only the necessary cells
  - knowing the "necessary cells" at compile time is undecidable
  - partial solution implemented in E-ACSL; yet to be improved

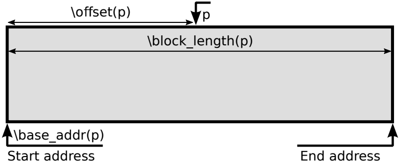
Compiling Memory Properties

- how to compile `\valid(p)` or `\initialize(p)`?
- standard solution: shadow memory
  - implemented in memory debuggers (e.g., Address Sanitizer)
  - cannot evaluate block-level properties



Compiling Memory Properties

- how to compile `\valid(p)` or `\initialize(p)`?
- standard solution: shadow memory
  - implemented in memory debuggers (e.g., Address Sanitizer)
  - cannot evaluate block-level properties



- E-ACSL's custom shadow memory [Vorobyov et al @ISMM 2017]
- issue: heavy encoding
- solution: dedicated dataflow analysis [Ly et al @HILT 2018]
  - monitor only the necessary memory locations

Conclusion and Future Work

- BISL are now well understood
  - many common standard features
  - still a few key differences between them
  - tend to be reduced over years

- BISL are now well understood
  - many common standard features
  - still a few key differences between them
  - tend to be reduced over years
- many applications for RAC
  - combined with other techniques
  - for verifying properties beyond BISL

Conclusion and Future Work

- BISL are now well understood
  - many common standard features
  - still a few key differences between them
  - tend to be reduced over years
- many applications for RAC
  - combined with other techniques
  - for verifying properties beyond BISL
- RAC is a compilation technique
  - efficient procedures for being both efficient and sound
  - still several open challenges