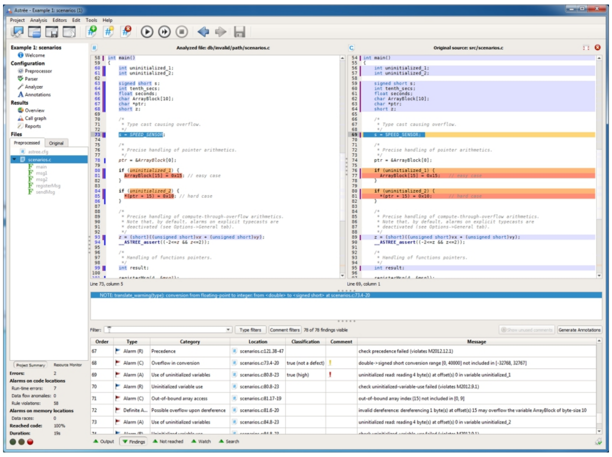
Astrée

Finding all Runtime Errors and Data Races in C/C++ Programs

Astrée is a parametric static analyzer designed to **prove the absence of runtime errors and data races** in software programs written in C/C++. Astrée is **parameterizable** and can be **specialized** to the program under analysis – key features to enable **high analysis precision**.



```
int main()
{
    int uninitialized_1;
    int uninitialized_2;

    signed short s;
    int tenth_secs;
    float seconds;
    char ArrayBlock[10];
    char *ptr;
    short z;

    /*
     * Type cast causing overflow.
     */
    s = SPEED_SENSOR;

    /*
     * Precise handling of pointer arithmetics.
     */
    ptr = &ArrayBlock[0];

    if (uninitialized_1) {
      ArrayBlock[15] = 0x15; // easy case
    }

    if (uninitialized_2) {
      *(ptr + 15) = 0x10; // hard case
    }

    /*
     * Precise handling of compute-through-overflow arithmetics.
     * Note that, by default, alarms on explicit typecasts are
     * deactivated (see Options->General tab).
     */
    z = (short)((unsigned short)vx + (unsigned short)vy);
    __ASTREE_assert((-2<=z && z<=2));

    /*
     * Handling of functions pointers.
     */
    int result;
```

Line 73, column 5

```
int main()
{
    int uninitialized_1;
    int uninitialized_2;

    signed short s;
    int tenth_secs;
    float seconds;
    char ArrayBlock[10];
    char *ptr;
    short z;

    /*
     * Type cast causing overflow.
     */
    s = SPEED_SENSOR;

    /*
     * Precise handling of pointer arithmetics.
     */
    ptr = &ArrayBlock[0];

    if (uninitialized_1) {
      ArrayBlock[15] = 0x15;     // easy case
    }

    if (uninitialized_2) {
      *(ptr + 15) = 0x10;        // hard case
    }

    /*
     * Precise handling of compute-through-overflow arithmetics.
     * Note that, by default, alarms on explicit typecasts are
     * deactivated (see Options->General tab).
     */
    z = (short)((unsigned short)vx + (unsigned short)vy);
    __ASTREE_assert((-2<=z && z<=2));

    /*
     * Handling of functions pointers.
     */
    int result;
```

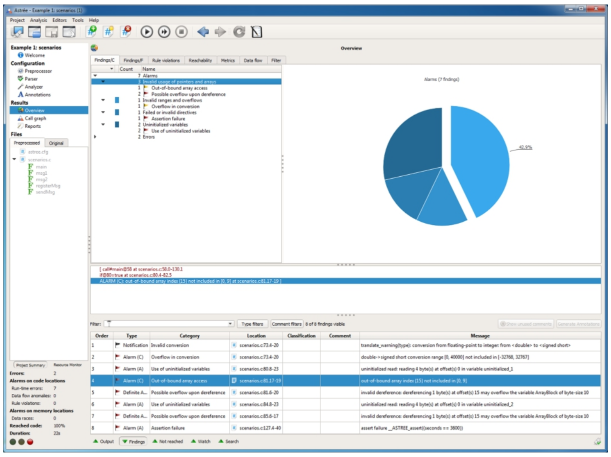
Line 69, column 1

```
NOTE: translate_warning(type): conversion from floating-point to integer: from <double> to <signed short> at scenarios.c:73.4-20
```

78 of 78 findings visible

Order	Type	Category	Location	Classification	Comment	Message
67	Alarm (R)	Precedence	scenarios.c:121.38-47			check precedence failed (violates M2012.12.1)
68	Alarm (C)	Overflow in conversion	scenarios.c:73.4-20	true (not a defect)	!	double->signed short conversion range [0, 40000] not included in [-32768, 32767]
69	Alarm (A)	Use of uninitialized variables	scenarios.c:80.8-23	true (high)	!	uninitialized read: reading 4 byte(s) at offset(s) 0 in variable uninitialized_1
70	Alarm (R)	Uninitialized variable use	scenarios.c:80.8-23			check uninitialized-variable-use failed (violates M2012.9.1)
71	Alarm (C)	Out-of-bound array access	scenarios.c:81.17-19			out-of-bound array index [15] not included in [0, 9]
72	Definite A...	Possible overflow upon dereference	scenarios.c:81.6-20			invalid dereference: dereferencing 1 byte(s) at offset(s) 15 may overflow the variable ArrayBlock of byte-size 10
73	Alarm (A)	Use of uninitialized variables	scenarios.c:84.8-23			uninitialized read: reading 4 byte(s) at offset(s) 0 in variable uninitialized_2

Errors: 2
Alarms on code locations
Run-time errors: 7
Data flow anomalies: 0
Rule violations: 58
Alarms on memory locations
Data races: 0
Reached code: 100%
Duration: 19s



Count	Name
7	Alarms
3	Invalid usage of pointers and arrays
1	Out-of-bound array access
2	Possible overflow upon dereference
1	Invalid ranges and overflows
1	Overflow in conversion
1	Failed or invalid directives
1	Assertion failure
2	Uninitialized variables
2	Use of uninitialized variables
2	Errors

```
[ call#main@58 at scenarios.c:58.0-130.1
if@80=true at scenarios.c:80.4-82.5
ALARM (C): out-of-bound array index [15] not included in [0, 9] at scenarios.c:81.17-19 ]
```

8 of 8 findings visible

Order	Type	Category	Location	Classification	Comment	Message
1	Notification	Invalid conversion	scenarios.c:73.4-20			translate_warning(type): conversion from floating-point to integer: from <double> to <signed short>
2	Alarm (C)	Overflow in conversion	scenarios.c:73.4-20			double->signed short conversion range [0, 40000] not included in [-32768, 32767]
3	Alarm (A)	Use of uninitialized variables	scenarios.c:80.8-23			uninitialized read: reading 4 byte(s) at offset(s) 0 in variable uninitialized_1
4	Alarm (C)	Out-of-bound array access	scenarios.c:81.17-19			out-of-bound array index [15] not included in [0, 9]
5	Definite A...	Possible overflow upon dereference	scenarios.c:81.6-20			invalid dereference: dereferencing 1 byte(s) at offset(s) 15 may overflow the variable ArrayBlock of byte-size 10
6	Alarm (A)	Use of uninitialized variables	scenarios.c:84.8-23			uninitialized read: reading 4 byte(s) at offset(s) 0 in variable uninitialized_2
7	Definite A...	Possible overflow upon dereference	scenarios.c:85.6-17			invalid dereference: dereferencing 1 byte(s) at offset(s) 15 may overflow the variable ArrayBlock of byte-size 10
8	Alarm (A)	Assertion failure	scenarios.c:127.4-40			assert failure __ASTREE_assert((seconds == 3600))

Errors: 2
Alarms on code locations
Run-time errors: 7
Data flow anomalies: 0
Rule violations: 0
Alarms on memory locations
Data races: 0
Reached code: 100%
Duration: 22s

Astrée is developed and distributed by AbsInt, under license from the CNRS/ENS. It has been successfully used on safety-critical software projects from various industry sectors, including aerospace, automotive, and nuclear energy.

The Challenge:

Runtime errors and data races can provoke erroneous program behavior and may even cause the software to crash. Software testing can be used to detect errors, but not to prove their absence, since usually no complete test coverage is possible. Static analysis based on Abstract Interpretation can be used to **prove the absence of runtime errors and data races**. A small number of false alarms is important to enable an **efficient validation process**.

Examples for Errors detected by Astrée:

- Out-of-bound array accesses
- Erroneous pointer manipulations and dereferencing (NULL, uninitialized and dangling pointers)
- Integer and floating-point divisions by zero
- Integer and floating-point arithmetic overflows
- Read accesses to uninitialized variables
- Violations of user-specified assertions
- Data races between concurrent threads
- Inconsistent locking and deadlocks

Astrée also reports accesses to shared variables, non-terminating loops, unreachable code, violations of MISRA C/C++ rules, and violations of CERT-C and CWE rules to prevent potential safety and security risks.

Key Features of Astrée:

- Astrée is **sound**: If the analysis does not detect any errors, the absence of runtime errors has been proven. Control and data coverage is 100%.
- Astrée is **precise**: Its state-of-the art analysis engine enables very low false alarm rates.
- Astrée is **scalable**: projects with more than 10 million lines of code have successfully been analyzed.
- False alarms can be safely eliminated by tuning the precision to the software under analysis.
- Astrée can be **seamlessly integrated** in existing development environments; plugins for **Jenkins**, **Eclipse**, and **dSPACE TargetLink** are available .
- A Qualification Support Kit is available, providing support for automatic **tool qualification** up to the highest criticality levels.
- Astrée automatically takes the **OS configuration** of ARINC653/OSEK/AUTOSAR projects into account, including the mapping of processes to cores, their resources and their priorities. **New**
- Advanced **taint analysis** supporting detection of **SPECTRE** v1/v1.1/SplitSpectre vulnerabilities, and supporting user-defined cyb ersecurity analyses. **New**
- C++ support is available as a preview version.

AbsInt Angewandte Informatik GmbH   Science Park 1   D-66123 Saarbrücken   Germany
phone: +49 681 383 60 0   fax: +49 681 383 60 20   info@absint.com   www.absint.com