Use of Coverity & Valgrind in Geant4

Gabriele Cosmo

Outline

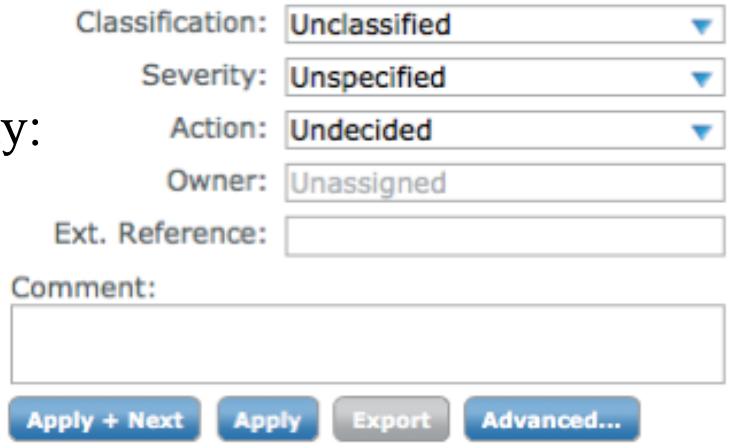
- Coverity: “triaging” defects
- Status of Static Code Analysis
- Use of Vagrind in Geant4 & DRD tool

Coverity Static Code Analysis

- Inspecting only the source code
  - Execution by replacing the compiler
- Analysis performed on all possible branches of the code
  - Complex and time consuming
- Identifies defects in the code
  - organizes by type, severity, file/module

Coverity: triaging defects

- Defects are assigned to responsible (owner)
- The owner can assess and specify the severity:
  - *Major, Moderate, Minor*
- Defects can be flagged as:
  - *Pending / False positive / Intentional / Bug*



- The owner specifies an action (performed or to be performed):
  - *Fix required / Fix submitted / Modeling required / Ignore*
  - According to the action, the defect is marked as "triaged"
  - Statistics will be updated
  - Defects not fixed will show up again at the next analysis

Coverity: defects types

- Memory corruption / illegal accesses
  - Double free, out-of-bound accesses, use-after-delete, …
- Resource leaks
  - Non virtual destructor, memory leaks, …
- Uninitialized variables, Unused pointer values, Infinite loops, Missing copy ctors, …
- API usage errors
  - Non restoring ostream format, using invalid iterator, …
- Control flow issues
  - Unused/dead code, invalid iterator comparisons, …
- Incorrect expressions
  - Self-assignment, misuse of enums, …
- Code organization / performance inefficiencies
  - Recursive headers, hidden parameters, big parameters passed by value, …
- Security best practices violations

➢ All defect kinds provided with online documentation

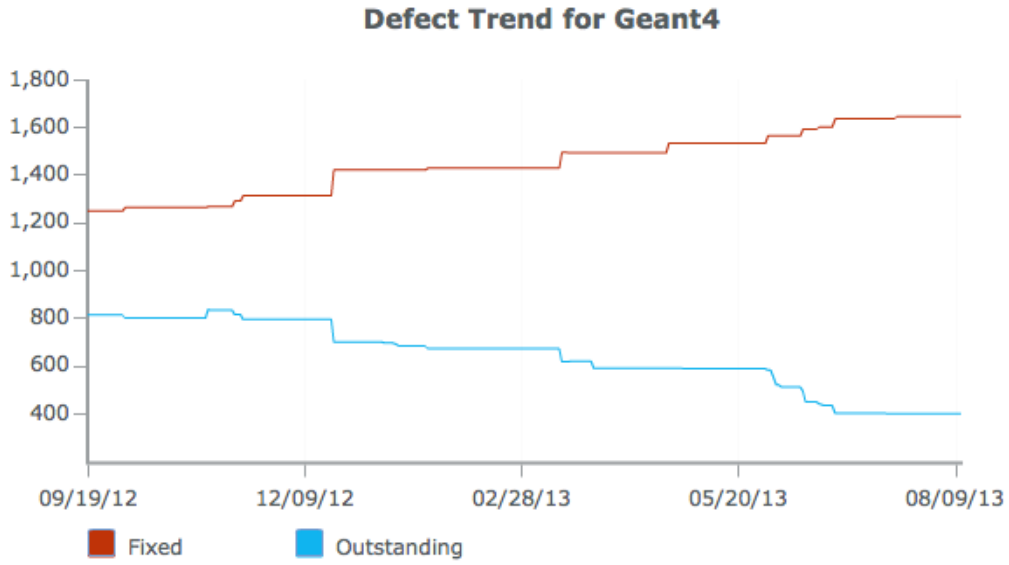
Use of Coverity in Geant4

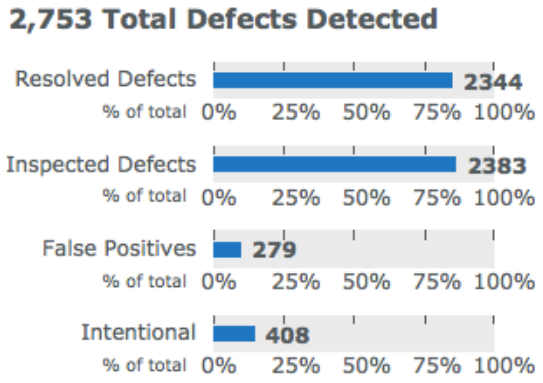
- New static analysis done after every reference tag
  - Results published immediately after
  - Notification sent to all Category coordinators *and to individual developers owning outstanding defects*
- Static analysis applied for all Geant4 libraries
  - Including all possible visualization drivers and optional modules
  - Applied to either sequential or MT mode
  - Could be automated & eventually applied as separate module also to examples/tests

Current status in Geant4

- Coverity tool introduced to the Collaboration at the 2010 Geant4 Workshop
  - First analysis (after tool upgrade) reported a total of 2753 outstanding defects
  - Status on last reference tag (09-06-ref-09):
    - Outstanding defects: 409 (of which, 370 never triaged)
    - Resolved defects: 2344

Goal: ZERO outstanding defects !!!





Valgrind

Use of Valgrind in Geant4

- Selected examples/tests executed for candidate releases and selected reference releases
  - Results published and distributed to Category Coordinators and relevant developers
- Checks for memory leaks in event loop
  - *Difficult now after introduction of MT memory model*
- Run-time errors checks
  - Planning to have them part of regular system testing
- Recently: checks for run-time race conditions in MT mode
  - Use of embedded DRD tool (*see next slides by A.Dotti*)

Valgrind DRD tool

- Data Race Detection tool
  - Included in the Valgrind suite (use with *–tool=drd*)
- Different types of errors detected:
  - Data races (unprotected possible access to shared resources)
  - Lock contention (checks Mutex locked for too long)
    - *Not used in our testing*
  - Misuse of POSIX threads
    - e.g. attempt to unlock a Mutex that was not locked,…

Valgrind: Checks for Data Race

- Tricky part:
  - Two memory operations conflict if different threads refer to the same memory location and at least one is a store
  - No issue if all conflicting operations are ordered by synchronization operations (in Geant4: barriers or locks)
- MT: all shared objects writes (by master) are protected by barriers or locks or happen before threads are spawned, workers only read shared variables
  - Verified this with FullCMS benchmark:
    - 2 threads, 4 HE pions events, FTFP_BERT
  - DRD tool extremely slow: >48 hours

Valgrind DRD results

- Few misuse of POSIX libraries (corrected)
  - Try locks of unlocked variables
  - Wrong use of signal/broadcast/wait (causing deadlocks in Mac OSX)
- Many data race conditions reported
  - Some real concerns: e.g. proper initialization of shared constants in physics models

Scoped static initialization

- In Geant4 code we have many situations like this:

```
void someFunction() {
     static G4double someConst = HeavyCalculation();
}
```

- This is not thread safe in general
  - See http://blogs.msdn.com/b/oldnewthing/archive/2004/03/08/85901.aspx
  - In reality if HeavyCalculation() is "simple" and returns always the same value, we can be safe, but cannot be guaranteed in general
  - DRD reports millions of such cases: luckily only in very few lines of codes called many many times

Conclusions

- Coverity static analyzer and Valgrind tools regularly used in our development process
  - Efforts to be made to increase automation
- Lot of progress made since first introduction of Valgrind in 2010
  - Important to keep going fixing defects and monitoring!
  - Tool also extremely useful in detecting memory leaks
- Use of Valgrind made for memory checks and MT race condition checks
  - Checks for memory leaks now hard and less effective

Thanks!