Static Analysis on the Tartan Racing Code Base

17-754 Analysis of Software Artifacts

Matthew McNaughton

Abstract

The DARPA Urban Challenge held on November 3, 2007, pitted autonomous robotic passenger vehicles against each other in a head-to-head race to see which one could drive the most efficiently and safely on suburban roads with real traffic, while following all of the rules of the road. Tartan Racing of Carnegie Mellon University won with Boss, a modified Chevrolet Tahoe with multiple laser and radar sensors, GPS, a rack of computers, and over 400 000 lines of C++ code to run the show. The team followed a formal quality assurance regime, undertaking frequent whole system tests, code inspections, and bug tracking. Static code analysis tools were not used, partially due to mistrust of and unfamiliarity with tools capable of analyzing C++ code. In this project we evaluated two C++ static analysis tools on the Tartan Racing code in an effort to determine how helpful they would have been to the project. One of these tools was Coverity Prevent (the other has been excised from this document due to confidentiality constraints). This project was undertaken with the generous support of Coverity, who offered their tools for this evaluation.

1. Coverity Prevent

Coverity Prevent gave a good impression in terms of its appearance, documentation, cleaner and simpler build process. It has support for tracking multiple analysis runs on an evolving code base and keeping track of the same issues within the code even as the code evolved. The build and analysis steps both ran very quickly. Prevent also maintains a database of internal representation so that analyses can be re-run without going through the build step again.

The reported philosophies of the differ with regard to false positives and false negatives. Coverity claims that they prefer to deliver no false positives at the risk of giving false negatives. Prevent reported only 173 issues on the entire code base. However, many of these issues were at least interesting, if not dangerous.

It was not possible to determine the severity of all issues found by static analysis, but I was able to guess.

Problem Type	# Instances	# Major	# Minor
Uncheck return value	18	0	2
Bad virtual method override	1	0	2
Bad character I/O - assign int to char	3	0	3
Dead code	13	0	10
Use of possibly null pointer	38	0?	17?
Missing return	0		
Return unsigned into from function declaring negative return value	7	0?	2?
Null return value from function	3	0	3
Overrun dynamic array	0		
Overrun static array	1	1 (in boost library)	0
Resource leak	7	0	3
Return reference to local variable	0	0	0
Use pointer and check if it’s null later	15	0	0
Use numerical value and check if it’s negative later	1	0	0
Uninitialized value	46	2	10?

Problem Type	# Instances	# Major	# Minor
Unused value	3	0	0
Use after free	0		
Misuse of varargs	0		

Issues unaccounted for in either the Major or Minor columns were deemed not an issue at all, either because they were intentional or Prevent was confused.

An variety of issue reported frequently by Prevent looked like the following:

```
if( dynamic_cast<Bar *>( foo ) ) {
   Bar *bar = dynamic_cast<Bar *>( foo );
  // Prevent thinks bar could be null
}
```

This code is actually acceptable. One might think there could be a race condition with another thread changing foo, but objects can't change their type at run-time, so this should not be a problem. If there is a race condition and foo is freed by another thread, the problem is really not local to this code.

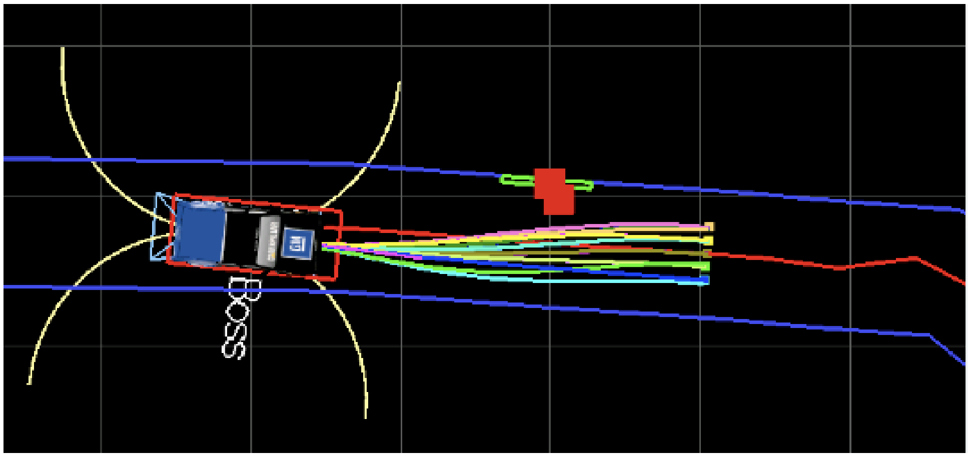
A very common issue about potential uses of null pointers was code following this pattern:

```
void f (SuperType *obj) {
  if( obj->typecode == SuperType::SUBTYPE ) {
    SubType *obj2 =dynamic_cast<SubType*>(obj);
    // obj2 could be null…
  }
}
```

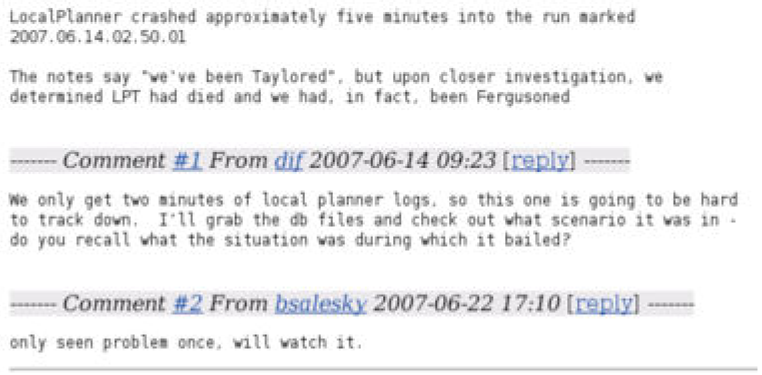
The robotics application may have many classes of, for example, moving obstacles that require slightly different handling. Both the format and operation on these classes may change frequently during the development effort. Therefore, fitting all classes into a coherent type hierarchy with a well-factored set of polymorphic methods is too expensive given the modifiability demands. We engaged in the unclean practice of switching on type codes, which gave rise to numerous complaints by Prevent like the above. This is a valid complaint for it to make, and if we had used Prevent in the project we would have silenced it by changing the syntax of this idiom.

Local Planner

The LocalPlanner process is responsible for finding an open path down the lane ahead of the vehicle that balances priorities such as staying in the lane and avoiding obstacles. In the following figure, the vehicle is driving from left to right. The red boxes ahead and to its left are obstacles it must avoid, while the blue lines running above and below are the nominal boundaries of the lane it is driving in. The red line between the blue lines is the middle of the lane which the robot is trying to drive along. The squid tentacles emanating out of the front of the vehicle represent paths the Local Planner is considering taking, and the dark blue one near the bottom is the one chosen as the best balance of priorities within the constraints.



The Local Planner was one of the most important tasks in the system. The vehicle travels at up to 30 mph. If it dies, the vehicle will screech to a halt, but possibly not before hitting something. Finding crash bugs in the process is a high priority. The last crash bug encountered during the development effort happened a month before the race. I chose five crash bugs reported for the Local Planner from the project's Bugzilla bug tracking software, to see if Prevent could provide any clues. The following is a screenshot of the comments on the bug. We see that the Local Planner has been identified as the culprit in a robot misbehavior, but there are not enough logs to determine the problem. The problem was not observed again after a week and it is still not known if it was resolved by some other bug fix applied later.



```
LocalPlanner crashed approximately five minutes into the run marked
2007.06.14.02.50.01

The notes say "we've been Taylored", but upon closer investigation, we
determined LPT had died and we had, in fact, been Fergusoned
```

------- *Comment #1 From dif 2007-06-14 09:23 [reply]* -------

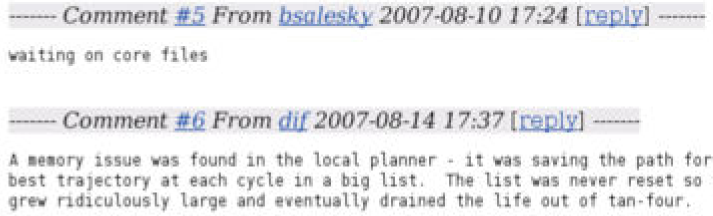
```
We only get two minutes of local planner logs, so this one is going to be hard
to track down.  I'll grab the db files and check out what scenario it was in -
do you recall what the situation was during which it bailed?
```

------- *Comment #2 From bsalesky 2007-06-22 17:10 [reply]* -------

```
only seen problem once, will watch it.
```

I checked out the code and ran Prevent, but it did not point out any obvious crash bugs.

The second bug was due to a memory consumption issue that could not have been found by a vanilla static analysis. The memory was not technically leaked. This incident does, however, point to a potential program annotation that could be used to prevent this kind of bug. That is, that global or heap data items should be annotated with the program scope in which they are allowed to grow. If any global variable is found which is added to but never emptied, the analysis could raise a warning if it is not annotated as being permitted to do so.



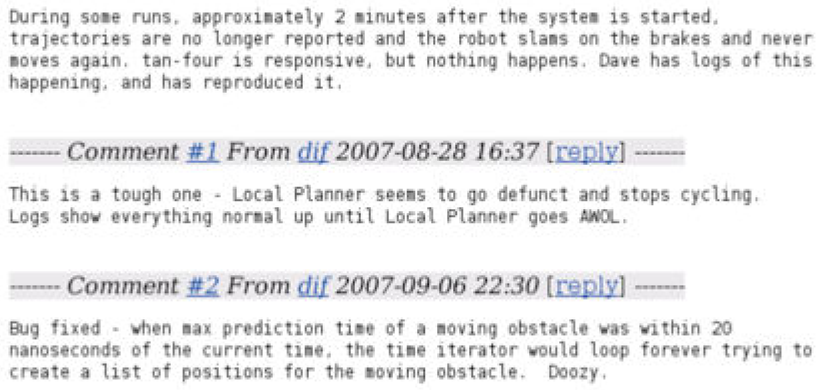
------- *Comment #5 From bsalesky 2007-08-10 17:24 [reply]* -------

```
waiting on core files
```

------- *Comment #6 From dif 2007-08-14 17:37 [reply]* -------

```
A memory issue was found in the local planner - it was saving the path for the
best trajectory at each cycle in a big list.  The list was never reset so it
grew ridiculously large and eventually drained the life out of tan-four.
```

The third problem was due to an infinite loop. I checked out the code for the revision identified as used in the test, but Prevent did not identify any infinite loops. This is especially odd since the main code of all tasks are infinite loops. They only exit when the task receives a signal, which may be why Prevent did not identify them.



```
During some runs, approximately 2 minutes after the system is started,
trajectories are no longer reported and the robot slams on the brakes and never
moves again. tan-four is responsive, but nothing happens. Dave has logs of this
happening, and has reproduced it.
```

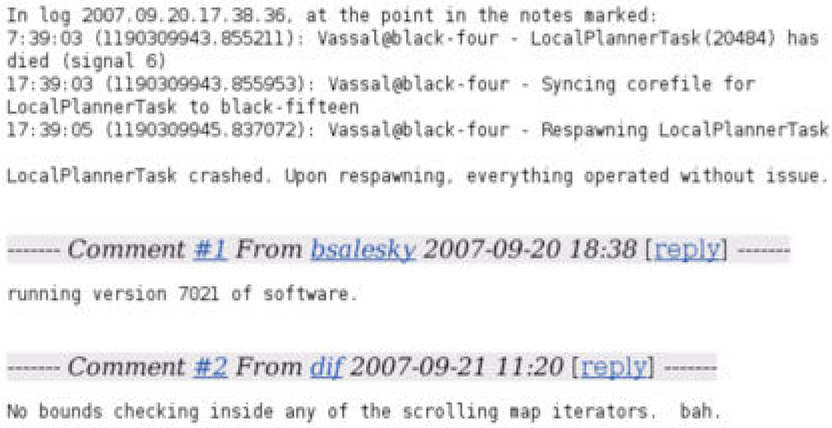
------- *Comment #1 From dif 2007-08-28 16:37 [reply]* -------

```
This is a tough one - Local Planner seems to go defunct and stops cycling.
Logs show everything normal up until Local Planner goes AWOL.
```

------- *Comment #2 From dif 2007-09-06 22:30 [reply]* -------

```
Bug fixed - when max prediction time of a moving obstacle was within 20
nanoseconds of the current time, the time iterator would loop forever trying to
create a list of positions for the moving obstacle.  Doozy.
```

The fourth problem was a bounds check in an iterator class written to iterate over the cells of an occupancy grid data structure, used to represent obstacles in the environment that the vehicle should avoid. This grid data structure is heavily used in time-critical code and is highly optimized for performance. For that reason, there are no bounds checks in the iterator accessor.



```
In log 2007.09.20.17.38.36, at the point in the notes marked:
7:39:03 (1190309943.855211): Vassal@black-four - LocalPlannerTask(20484) has
died (signal 6)
17:39:03 (1190309943.855953): Vassal@black-four - Syncing corefile for
LocalPlannerTask to black-fifteen
17:39:05 (1190309945.837072): Vassal@black-four - Respawning LocalPlannerTask

LocalPlannerTask crashed. Upon respawning, everything operated without issue.
```

------- *Comment #1 From bsalesky 2007-09-20 18:38* [reply] -------

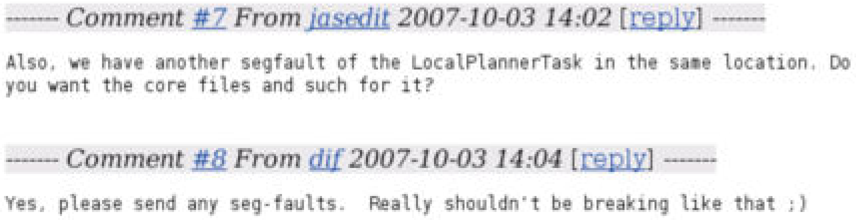
```
running version 7021 of software.
```

------- *Comment #2 From dif 2007-09-21 11:20* [reply] -------

```
No bounds checking inside any of the scrolling map iterators.  bah.
```

Prevent did not find this problem.

The fifth and final issue that I attempted to find with Prevent was with code that assumed that the length of a vector returned to it from a function would always be non-zero. Due to bad input from another part of this system, this was not always true. Prevent did not identify the faulty code in this case.



------- *Comment #7 From jasedit 2007-10-03 14:02* [reply] -------

```
Also, we have another segfault of the LocalPlannerTask in the same location. Do
you want the core files and such for it?
```

------- *Comment #8 From dif 2007-10-03 14:04* [reply] -------

```
Yes, please send any seg-faults.  Really shouldn't be breaking like that ;)
```

It could be argued that it is too much to ask Prevent to identify known crash bugs lurking in a huge code base full of not only messy code, but code similar in style and that appears to take similar risks. This experience does show that the project manager's opinion of static analysis tools was reasonable - he did not trust them to solve many problems. I believe that static analysis tools at least push quality higher, even if they don't guarantee perfection. The possibility of writing custom analyses to address specific types of errors that you know are endemic to your own software, due to the nature of the problem, or of your employees, or of legacy factors, however, is very interesting. In the next section we look at the potential role of custom static analyses to help in debugging a distributed real-time system such as the Tartan Racing code.

Repeatability

In the first of the Local Planner bugs examined above, the developer responsible claimed that there were not enough logs to track down the problem. In fact, even had he had the logs, they may well not have contained enough information to isolate the offending code. Tartan Racing recorded one sort of log - the message passed between tasks. In a typical single-threaded program, the input passed to the program is enough to reproduce its behavior exactly by running it on the same input, as long as its behavior does not depend on the time, or other quantities which are not recorded. If a program has this property we will say it is *repeatable*. However, in a threaded program, or in a system of communicating processes where messages may arrive in slightly different orders from run to run, guaranteeing repeatability is a challenge and may have a cost.

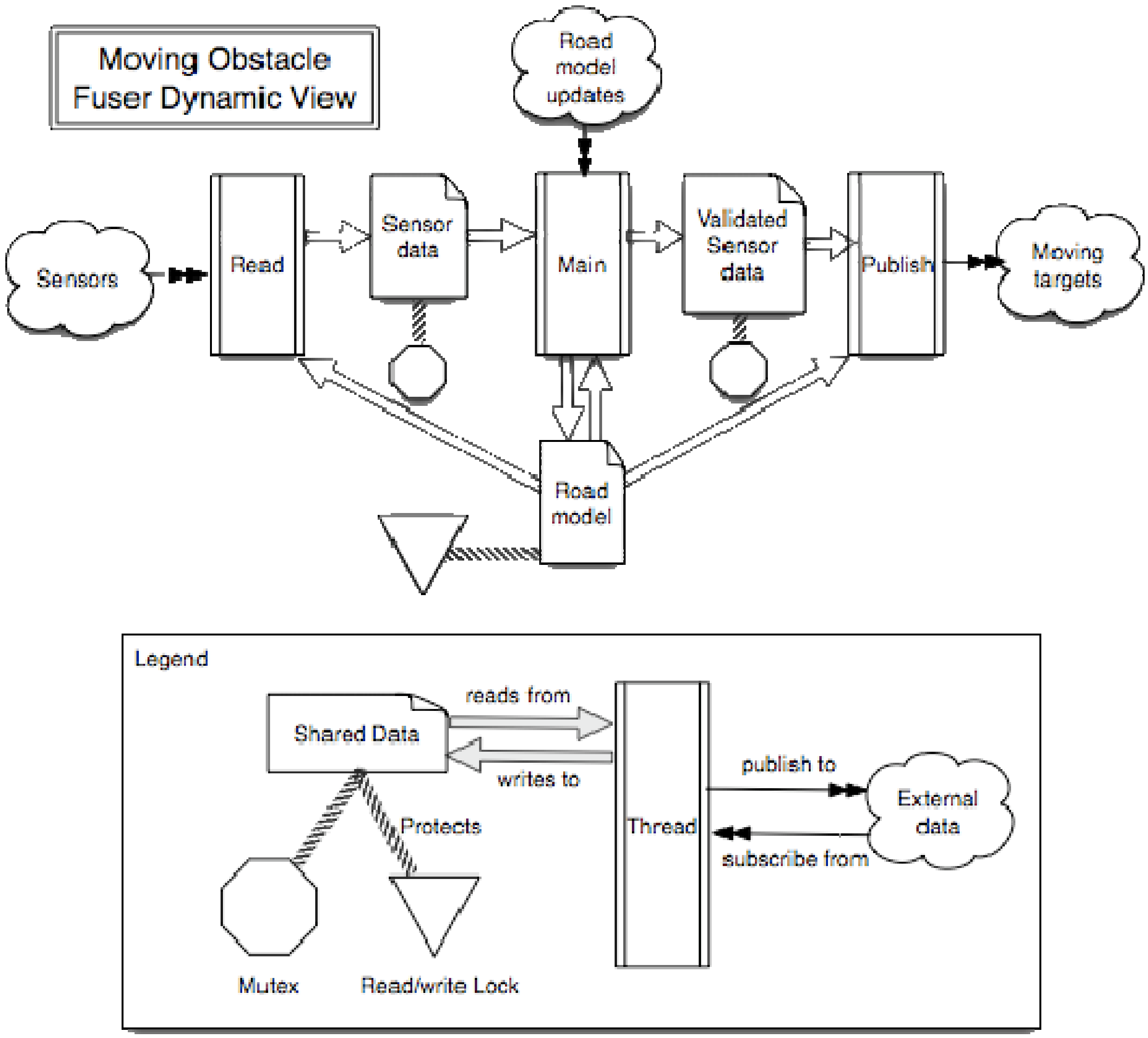
McLean and Fujimoto[1] identified four categories of sources of non-determinism.

- Message delivery - order of, timing of, and losses of
- External inputs such as sensors or operator inputs - these are the quantities normally logged by Boss
- Operating system calls, e.g. gettimeofday()
- Hardware interrupts, affecting thread scheduling

All of these variables must be logged. The results of thread-related calls, shown for example in the following, must be logged by recording in which order threads acquired locks.

```
pthread_mutex_lock(&readReadyMutex_);
receivedMovingTargetSetListReadThread_.push_front(r);
pthread_mutex_unlock(&readReadyMutex_);
```

Threaded applications in the Tartan Racing code base were not too complicated. They were typically perception algorithms that had to be threaded to deal with the tremendous amount of data flowing in from the numerous sensors on the robot. One thread would deal with reading the data into an internal data structure, another with running some perception algorithm on the data, and another with publishing the results out to other tasks. The following figure shows an example.



There are a few data structures shared by the threads, some protected by mutexes, and some with read/write locks. Logging which threads acquired which locks in which order for later playback should be simple enough, but there is a hidden problem - *if there is a race condition in the program, then the logs are not valid and playback is impossible, invalidating the entire exercise*.

Therefore, a certain degree of correctness in the code must be guaranteed before we can use logs to find errors. It should be possible to use static analysis techniques to guarantee that there are no race conditions. Risks of aliasing shared data must be mitigated, which to approach with static analysis would require a wholesale adoption of code annotation techniques in the spirit of the Fluid project in the ISRI. Applying these techniques to C++ would be a useful undertaking.

Conclusion

We examined the performance of Coverity's Prevent static analysis tool on the Tartan Racing code base. Prevent successfully analyzed the code, delivering just a few certain or almost-certain bugs. Prevent was then used to analyze code with known bugs to see if it could have found them before testing, and was unable to identify the cause of the bugs. We found reason to believe that improving the logging of internal process activity would be useful in tracking down the causes of bugs once encountered in a program run, and that static analysis on race conditions in threaded programs would be necessary to make this work.

References

[1] McLean, T. and Fujimoto, R. "Repeatability in real-time distributed simulation executions", PADS 2000.