WWW.BITRON.NET

11th Automotive SPIN Italy Workshop

BITRON
Unità di GRUGLIASCO

Str. del Portone 95 - 10095 Grugliasco (TO) Italy
Tel.: ++39/011/4029111 - Fax: ++39/011/781364
e-mail: grugliasco.info@bitron-ind.com

CSQ UNI EN ISO/TS 9001: 2008

IQNet CERTIFIED MANAGEMENT SYSTEM

CSQ AUTO UNI EN ISO/TS 16949:2002

CSQ eco ISO 14001

BITRON Industrie S.p.A.

info.bitron@bitron-ind.com

Is the Code We Have Verified What We Really Have Embedded?

Toward a more reliable and efficient software verification technology

Did we verify the right code?

- A bug that is only discovered in production is expensive!
- We thus want to catch bugs in the early development phases
- There are several methods to help achieve that:
  - Software walkthrough
  - Unit testing in a simulated environment
  - Model-in-the-loop
  - Formal methods
  - Adherence to coding standards such as MISRA-C

This positively interacts with all other activities, increasing readability, testability, …

Did we verify the right code?

- All software verification methods based on source code share one problem: did we verify the right code or something else?
- Ever had to spend weeks writing “compiler personalities” or “project personalities”?
- We did and we had enough:
  - it is time consuming
  - it is not really our job
  - it is very error-prone
- Bitron is partnering with BUGSENG to transition toward a more reliable and efficient verification technology
  - Today we focus on just one aspect: how do we ensure that the embedded code is the verified code?

Is the Code We Have Verified What We Really Have Embedded?

Roberto Bagnara

bugSeng
http://bugseng.com

& Applied Formal Methods Laboratory
Department of Mathematics and Computer Science
University of Parma, Italy

11th Workshop on Automotive Software & Systems, Milan, November 7th, 2013

Outline


I can remember
the exact instant
when I realized
that a large part
of my life from then
on was going
to be spent
finding mistakes
in my own programs.
Maurice Wilkes, 1949

Bugs Cause Frequent and Expensive Recalls in Automotive

year	manufacturer	model	# recalled	problem
2005	Toyota	Prius	160,000	sudden stall or shut down
2008	Chrysler	Jeep	24,535	automatic transmission
2008	Volkswagen	Passat	4,079	unexpected acceleration
2010	GM	CTS	12,662	front passenger knee airbag
2011	Jaguar	X-type	17,678	cruise control
2012	BMW	7-Series	45,500	automatic transmission
2013	GM	various	26,582	engine braking off
2013	Chrysler	Jeep	409,000	airbags and seat belts
2013	Chrysler	various	224,254	airbags deploy improperly
2013	Honda	Fit Sport	43,782	vehicle stability assist
Nov 4!	Honda	Odyssey	344,000	vehicle stability control

And Worse Things

Okla. jury: Toyota liable in acceleration crash

(By SEAN MURPHY / Associated Press / October 24, 2013)

OKLAHOMA CITY (AP) Toyota Motor Corp. is liable for a 2007 crash that left one woman dead and another seriously injured after a Camry suddenly accelerated, an Oklahoma jury decided Thursday.

The jury awarded $1.5 million in monetary damages to Jean Bookout, the driver of the car who was injured in the crash, and $1.5 million to the family of Barbara Schwarz, 70, who died.

It also decided Toyota acted with "reckless disregard" for the rights of others, a determination that sets up a second phase of the trial on punitive damages that is scheduled to begin Friday.

High-Quality Verification Tools Are Needed

Tools of insufficient quality make things worse

- They provide a false sense of security
- They can decrease the overall quality of the produced software
- They definitely increase development costs

In this talk

- We substantiate the claim that quality software requires quality tools. . .
- . . . focusing on source code static verification tools
- We will do so by showing some key features high-quality verification tools should possess. . .
- . . . along with the consequences of not possessing them

SCSV Tools' Usability and Reliability Requirements

- Different source code static verification (SCSV) tools have different requirements. . .
- . . . but they share most of them

Proper use of an SCSV tool requires

- *Identify and configure the verification task to be conducted*
- Identify and configure the sources that compose the software system to be analyzed
- Identify and configure the precise semantics of such sources
- Identify and configure the way object code is produced and manipulated to build executables
- Configure the reports sought and establish their connection to the running system

MISRA-C

- A software development standard for the C programming language
- Objectives: facilitate code safety, portability and reliability of embedded systems programmed in C
- Developed by MISRA (Motor Industry Software Reliability Association)
- In widespread use in automotive, aerospace, railway, medical devices, ...
- Now in its third edition: 1998, 2004, 2012
- MISRA-C++ (2008) is an analogous standard for C++

USA Case No. CJ-2008-7969

A. But function[s] should [not call] themselves. [...] in the 2004 [MISRA C] standard this rule [...] is No. 16.2. So this is a violation of the MISRA C rule.

Q. [Is] the violation of this rule related to unintended accelerations?

A. Yes.

[...]

A. Toyota also failed to comply with standards [...] Here I'm talking about, for example, the MISRA C guidelines.

Q. And in the review of what Toyota had done did NASA [find] any violation of these codes?

A. Yeah, NASA found a number of violations of MISRA rules.

Q. Did you find violations?

A. Yes. NASA looked at about 35 of the rules [...] they found over 7,000 violations [...] I checked the full set [...] and found more than 80,000 violations in the 2005 Camry.

Example: MISRA-C:YYYY Compliance the Painful Way

Identify and configure the sources that compose the system

Study the build procedure and the used compilers; then figure out:

1. which files are compiled and how
2. the algorithm used to search the header files and how it is influenced by the used compilers' options
3. the predefined macros and how they are influenced by the used compilers' options
4. report everything in the tool configuration so as to ensure that the analyzed code is the right one

Issues

Basically impossible to get right for the ordinary user

Example: MISRA-C:YYYY Compliance the Painful Way

Identify and configure the precise semantics of the sources

Study the build procedure and the used compilers; then figure out:

1. which of the used compiler options influence the semantics of the program
2. (whether and) how such semantics can be captured in the tool configuration
3. report everything in the tool configuration so as to ensure the analyzed translation units and the compiled ones match

Issues

- Very error-prone
- Some tools are simply not configurable enough

Example: MISRA-C:YYYY Compliance the Painful Way

Identify and configure the way object code is manipulated

Study the build procedure and each tool used to manipulate object code (compilers, linkers, librarians, . . .), then figure out:

1. where each object file comes from. . .
2. . . . tracking its creation, insertion into a library, extraction from a library, . . . , without losing the relation with the sources it comes from
3. report everything in the tool configuration so as to ensure the analyzed code and the resulting executables match

Issues

Basically impossible to get right unless the build process is really straightforward

Example: MISRA-C:YYYY Compliance the Painful Way

Obtain reports and establish their connection to the running system

1. Turn the tool output into the kind of report needed
2. Make sure the final report matches the running system

Issues

1. Often done by hand
2. Mission impossible if at least one of the mistakes previously outlined has been committed

Example: MISRA-C:YYYY Compliance the Painful Way

Summarizing

1. This process is so flawed that there are no guarantees about the MISRA-C:YYYY compliance of the running code
2. Still, it cost the user an enormous amount of time, pain and frustration
3. All this is largely due to poor quality of the verification tool

A non-solution: bad for the user, bad for the industry

- The vendor advertizes the product, *per se*, as certified
- Nonsense for all industrial safety standards: qualification of a tool is achieved in the precise context of a specific usage, including the operational environment, the inputs definition, the options used...

Example: MISRA-C:YYYY Compliance the Right Way

Identify and configure the sources that compose the system

Not necessary

How is this done?

- The tool intercepts all invocations to the tool-chain components and is able to interpret all arguments and options passed to them
- The tool embodies an abstract model of each tool-chain component so that, e.g., header files will be searched the same way they are searched by the compiler
- The predefined macros are automatically extracted from the compiler, using the right options

Example: MISRA-C:YYYY Compliance the Right Way

Identify and configure the precise semantics of the sources

Not necessary

How is this done?

- The tool embodies an abstract model of all supported compilers...
- ...which includes all the implementation-defined aspects of the language...
- ...taking into account the compiler options that are used in the actual compilation of each translation-unit

Example: MISRA-C:YYYY Compliance the Right Way

Identify and configure the way object code is manipulated

Not necessary

How is this done?

- All the tool-chain components are intercepted
- Their options and arguments are understood
- Their effects are modeled by the tool

Example: MISRA-C:YYYY Compliance the Right Way

Obtain reports and establish their connection to the running system

Push a button and obtain the compliance matrix

How is this done?

- The wanted compliance matrix format is part of the configuration
- Part of the deviation information is in the configuration: which rules are switched off, which file are exempted from which rules. . . all along with the corresponding justification
- Part of the deviation information is in the code
- Precise information about each violation has been collected by the tool
- The tool assembles all these pieces of evidence into a complete and coherent compliance matrix

Example: MISRA-C:YYYY Compliance the Right Way

Obtain reports and establish their connection to the running system

Nothing to do: it is just a consequence of good design

How is this done?

- The analysis process precisely matches the build process and happens at the same time
- Error-prone human activities have been reduced to the bare minimum
- Compliance matrices are automatically generated
- Such reports may contain a cryptographic hash function of all the sources compiled, the used libraries, the objects and the executables generated; the complete version information of all used tool-chain components; . . .

Testing the Tool-Chain

MISRA-C:2004 rule 1.4 (required)

"The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers."

- Most tools on the market provide no support for this and other rules...
- ...because they have no access to the actually used compilers and linkers!
- Intercepting the tool-chain components allows all sorts of safety checks to be performed upon them
- It also allows debugging the build procedure (e.g., against the wrong/inconsistent use of compiler flags)

Example: MISRA-C:YYYY Compliance the Right Way

Summarizing

1. Full automation of whatever is automatizable
2. Much increased reliability thanks to the direct interaction with the tool-chain
3. So-called personalities can be forgotten along with the consequent money/time losses and frustration
4. Basic analysis of the code (i.e., with default configuration and no deviations) can start in minutes after tool installation
5. Analysis reports are strictly keyed to the running code

A More Efficient and Trustable Process

1. Maximum automation greatly increases the significance of the results

> Everything you do by hand is subject to non-systematic errors

2. It also implies time and effort are spent where they are needed

> Everything you do by hand has to be maintained

3. And translates into robustness in front of retargeting or changes to the tool-chain
4. Fully-automatic reports favor the development of trust all along the supply chain

Full Disclosure

This talk reflects the points of view and the philosophy of the BUGSENG team that designed the ECLAIR software verification platform

The whole methodology is being introduced in the automotive software development and verification process of Bitron

bugseng

no shortcuts,
no compromises,
no excuses:
software verification done right

The End