Technische Universität München

Functionally Safe Software in Embedded Systems

Martin Becker <becker@rcs.ei.tum.de>
Technische Universität München
Institute for Real-Time Computer Systems (RCS)

Munich, 2014 August 28

What does it encompass?
How to reach safety?
How can I be sure?

This presentation only deals with *static* systems,
i.e., not self-adaptive systems.

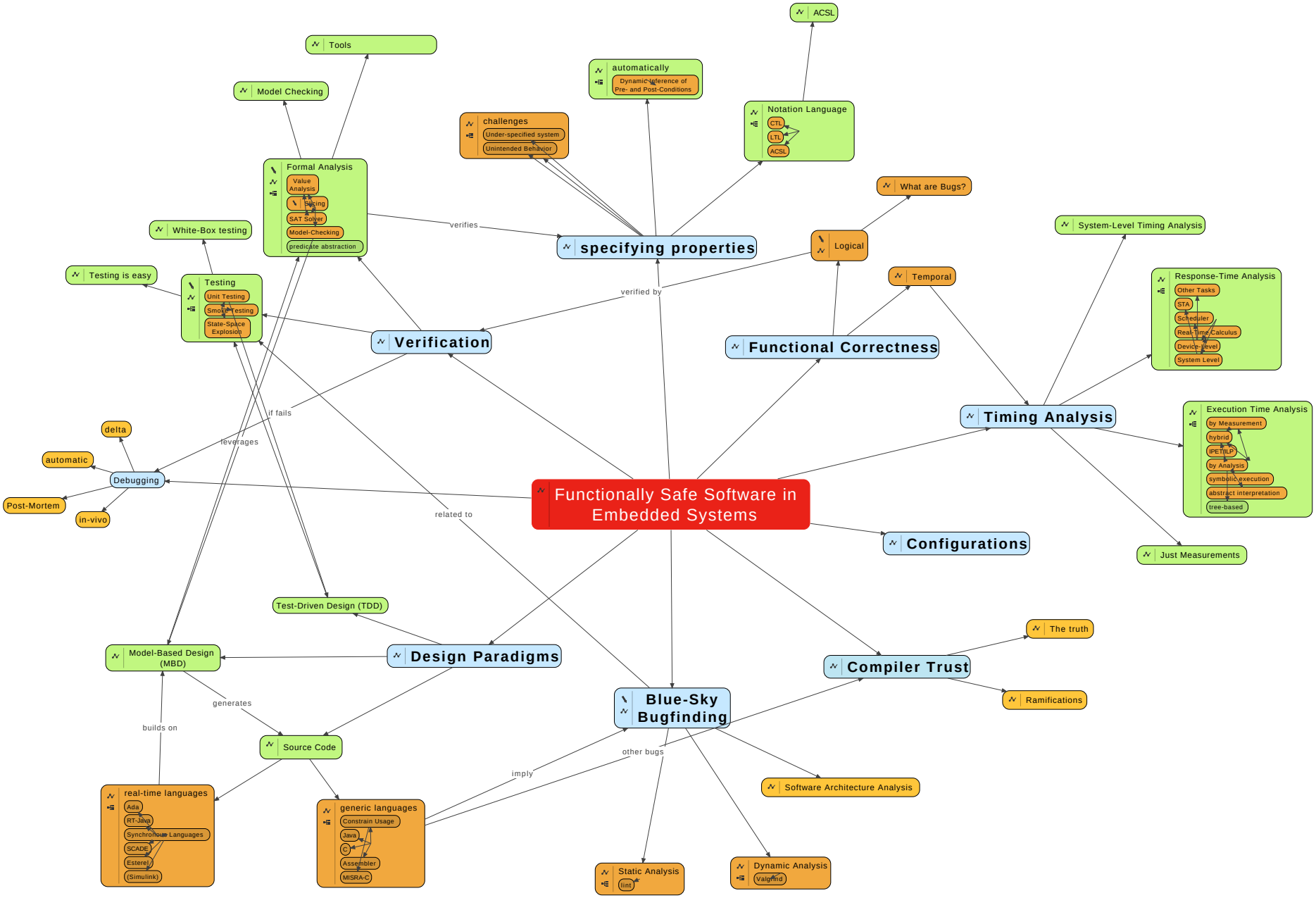
pdf | RCS.pdf

Disclaimer

This material was designed for an interactive talk. It reflects my personal view on the problem of "functionally safe software" in this very moment. Though it is backed up by meaningful scientific references, some parts might be simplified or incorrect in certain circumstances.

Last but not least, please do not mind the low-grade layout of the slides, it was created with the great software tool VUE (www.vue.tufts.nl), which the author might not have fully understood when creating these slides.

Martin Becker
2014 August 28th


Functionally Safe Software in Embedded Systems
specifying properties
Verification
Functional Correctness
Timing Analysis
Configurations
Design Paradigms
Blue-Sky Bugfinding
Compiler Trust
Debugging
Testing
Formal Analysis
Model Checking
Tools
ACSL
automatically
Dynamic Inference of Pre- and Post-Conditions
challenges
Under-specified system
Unintended Behavior
Notation Language
CTL
LTL
ACSL
What are Bugs?
Logical
Temporal
System-Level Timing Analysis
Response-Time Analysis
Other Tasks
STA
Scheduler
Real-Time Calculus
Device-Level
System Level
Execution Time Analysis
by Measurement
hybrid
IPET/ILP
by Analysis
symbolic execution
abstract interpretation
tree-based
Just Measurements
Value Analysis
Pacing
SAT Solver
Model-Checking
predicate abstraction
White-Box testing
Testing is easy
Unit Testing
Smoke Testing
State-Space Explosion
verifies
verified by
if fails
leverages
related to
delta
automatic
Post-Mortem
in-vivo
Test-Driven Design (TDD)
Model-Based Design (MBD)
generates
builds on
Source Code
real-time languages
Ada
RT-Java
Synchronous Languages
SCADE
Esterel
(Simulink)
generic languages
Constrain Usage
Java
C
Assembler
MISRA-C
imply
other bugs
Static Analysis
lint
Dynamic Analysis
Valgrind
Software Architecture Analysis
The truth
Ramifications

Functional Correctness

Functional Safety: "Something bad never happens"

Most critical: What does "bad" mean?

- a *specified* bad situation never occurs
- an *unspecified* bad situation never occurs -> hm..

What we need:
- requirements, as complete as they can get
- software, that does what the programmer intended
- software, that is not ambiguous
- fault-free hardware

CONFIDENCE in the resulting implementation!

Logical Correctness

- is what most people mean
- e.g., do not want that "there are errors"

Hypothesis:
In a real-time system, even a logically perfect program may be incorrect.

WHY?

What are Bugs?

- an "infection" is an error in state
- a "defect" is an error in code, that *might* lead to an infection
- a "failure" is an error in the result, caused by a chain of infections

```
// compute n-th order moment of data series
double get_moment(float*x, unsigned int datalen, int n) {
    double part, sum=0.;
    for (unsigned int k=0; k<datalen; k++) {
        part = fabs(pow(x[k],n)); // defect: fabs() is wrong
        sum += part;              // only sometimes infected
    }
    return sqrt(sum);             // only sometimes failure
}
```

Trivia:

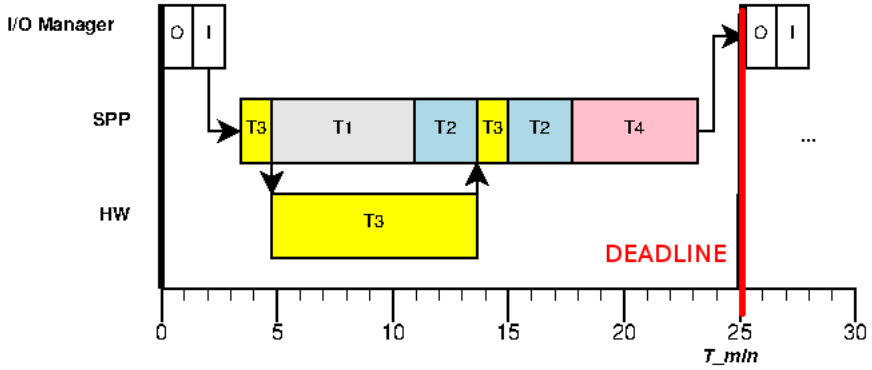
- infections may decease!
- failures can be "dormant"
- no adverse effect on the system
- not even detected

When people talk about "bugs", they mean "failure",
but that is only the tip of the iceberg.

Temporal Correctness

A perfect logic implementation can be incorrect, if it happens at the wrong time:

- too early
- too late (deadline violation)
- irregular
- ...



These are "real-time" criteria.
- soft real-time: occasional mis-timing is tolerated (e.g., your computer takes 5 seconds more to boot)
- hard real-time: even a single mis-timing can lead to catastrophic consequences (e.g., crash an aircraft)

"Real-Time Systems", Jane W.S. Liu, Prentice Hall, 2000.

Design Methods

How to "program" a system?

- classic source code
- model-based design (MBD)
- model-driven design (MDD)

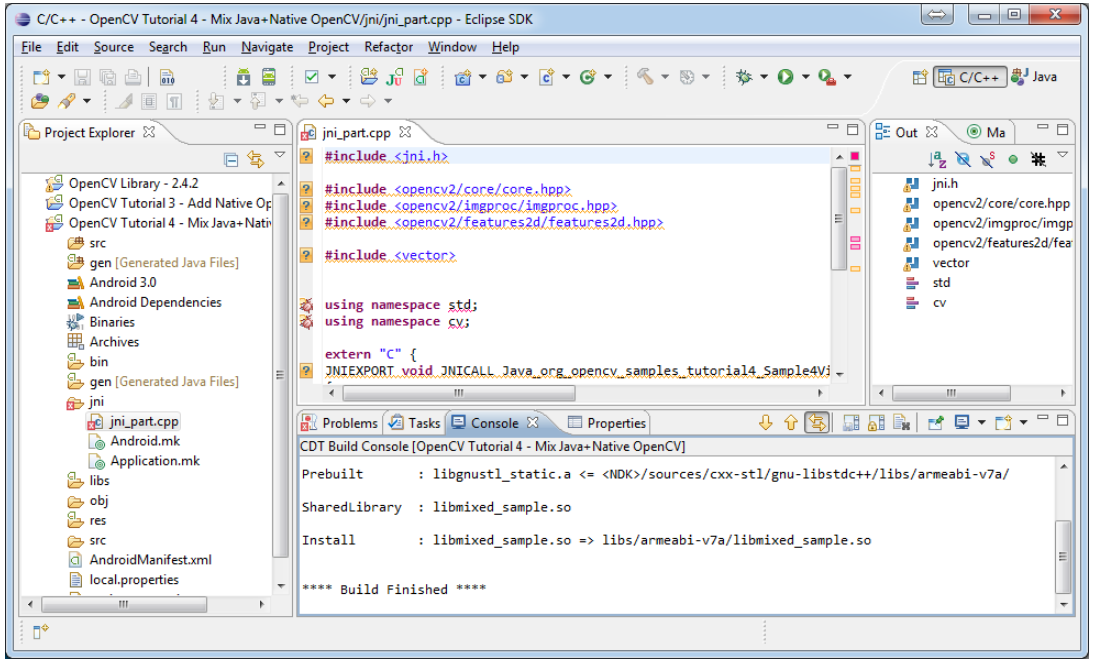
In reality, we need all of them:
- low-level software is mostly source code (e.g., drivers)
- high-level software is too complex to be written by hand
- specify it as a graphical model, and then generate code
- e.g., controller design in Mathworks' Simulink

Source Code

Writing a program = writing source code
- currently most wide-spread view among programmers
- considered the "true way of viewing a program"

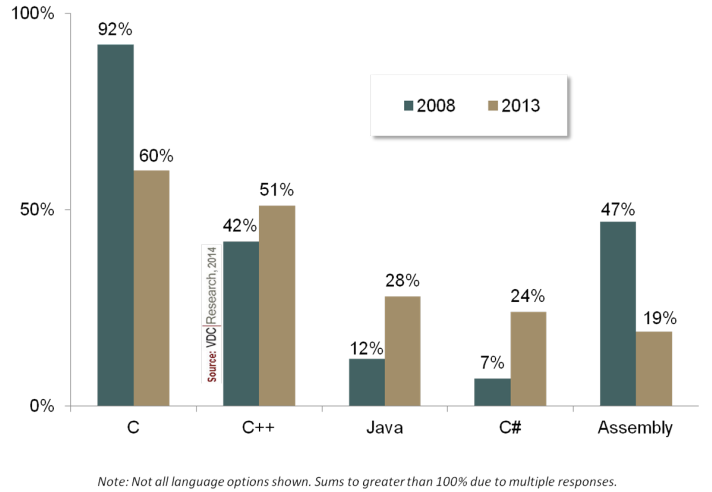
Source code has a lot of problems:
- hard to document & understand understand
- easy to miss errors

Good things:
- well-known
- mature paradigm



Generic Programming Languages

Traditional languages for safety-critical systems:
- C++ (53%), C++ (52%), Java (22%), Assembly (21%)
- whereas the high-level languages are on the rise



[survey of VDC research, 2008-2013]

Why this is bad:
- not made for safety-critical systems
- often undefined semantics (C Std: "...Anything at all can happen; the Standard imposes no requirements. [...]").
- example: overflow. int i = INT_MAX + 1;
- the linux kernel contains such bugs!

Workaround:
- only use a "safe subset"
- proof: coding standards such as "MISRA-C "in automotive domain

http://www.misra.org.uk

Real-time Programming Languages

Languages *made for* safety-critical systems
- strict semantics
- temporal specifications (delays, inherent parallelism etc.)
- functional+temporal verification is easier than in generic languages (we can run proofs)
- Examples: synchronous languages (Esterel, Lustre, Signal), time-triggered languages (HTL), RT-Java, Ada

Disadvantages:
- sometimes not intuitive or convenient
- not wide-spread

"The Esterel Synchronous Programming Language: Design, Semantics, Implementation", G. Berry, G. Gonthier, 1992.

Example:

```
module ABRO:
input A, B, R;
output O;
loop
   [ await A || await B ];
   emit O
each R
end module
```

Model-Based Design (MBD)

- graphical models are the center of development, not source code
- graphical blocks have defined semantics:
- e.g., UML, Simulink blocks, etc.

- generate code from the model
- code generators can be certified -> assume no errors
- one model can generate code for multiple platforms
- generate: C code, ADA code, VHDL code, ...
- translate model to a "safe" subset of these languages

- model is easier to analyze than source code
- faster and tighter results

Paper:

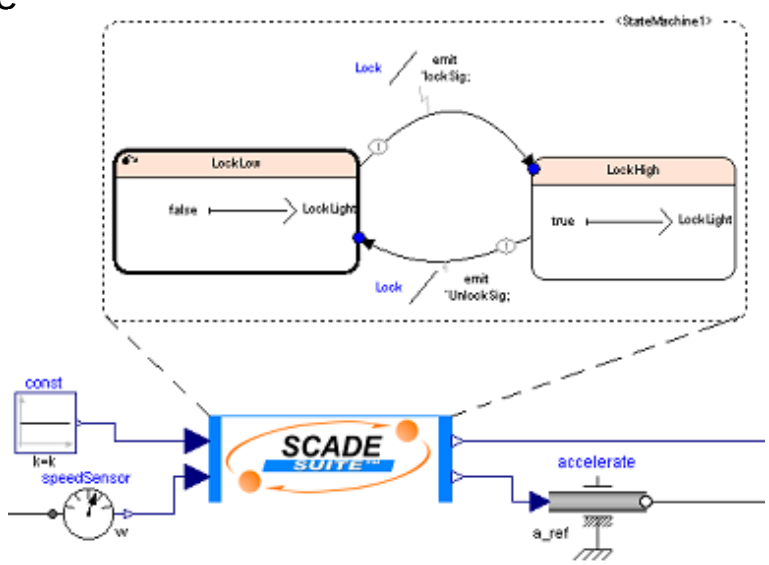
Model-based design of embedded control software for hybrid vehicles, IEEE SIES, 2011

Avionics:

["SCADE SUITE", Esterel Technologies, www.esterel-technologies.com]

Automotive:

Simulink, Vector, UML, ...



Verification

Evaluate, whether a system-under-test is compliant to a given requirement.
- "When the user presses this button, the program must show the preferences dialog."

Some are easy to check, others not:
- "The embedded control system be operational within 200ms after a cold reset."

Broadly, two approaches:
- Testing: bringing the system in the specfied situation, and "see what happens" for a specific input
- Analysis: constructing some sort of proof based on known system properties, showing that it is compliant

Testing

Idea:
- provide inputs and observe outputs to a black-box (BB)
- requires "coverage", i.e., that all possible control flows are activated in the black box

Forms:
- unit testing: check tiny parts of software against its spec.
- smoke testing: randomize inputs with the goal of crashing ("smoking") the application
- ...

Disadvantages:
- naive testing -> state-space explosion! (requires clustering into equivalence classes at least)
- finding infections or even defects requires impractically high instrumentation

White-Box testing

- run tests by looking at internal structure of system
- "smart" test vectors are generated
- requires source code or model

Advantages:
- can find defects, not only failures
- finds more infections than BB testing
- can guarantee coverage

Disadvantages:
- tester needs to understand the technique
- harder to implement than BB testing
- state space explosion

["Different approaches to white-box testing technique for finding errors", M.E. Khan, Journal on SW Engineering and its Applications, July 2011]

Predominance of Verification by Testing

Testing is widely used, despite of its weaknesses. Reasons:
- there is great tool support (>>100 tools, e.g. JUnit, CPPUTest, ...), even quantifying coverage (>"99.99%")
- result is easy to understand
- easy verification of known bad cases

Automotive industry: "Testing is the only truth" (afaik)

Fun Fact:

Automotive industry has 1000x more "flying hours" than the entire Boeing 737 fleet has made since 1968.

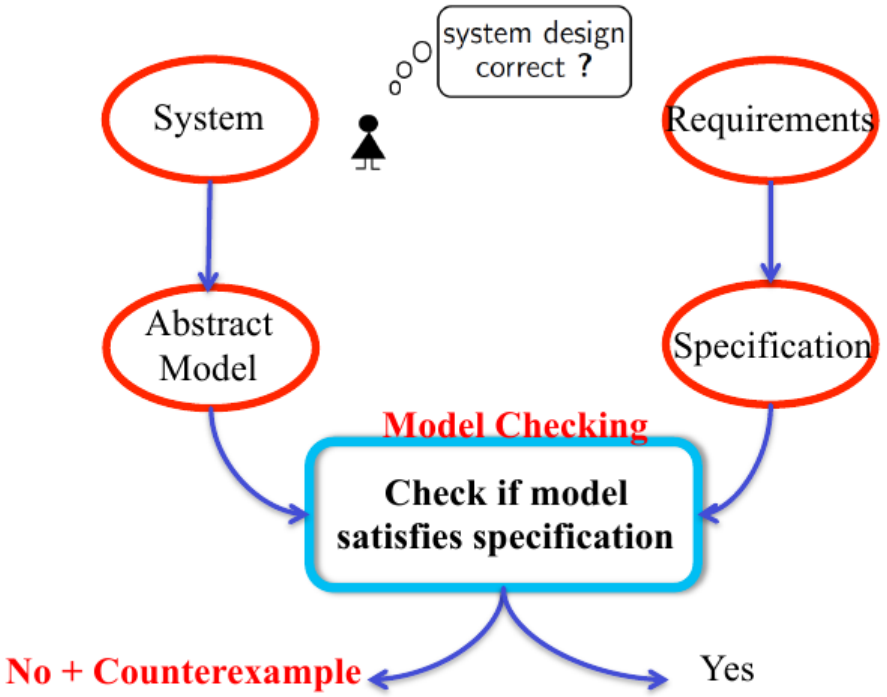
...but they keep on testing, whereas aeronautic industry is formally analyzing.

["Real-Time in the Prime Time", Buttle, ETAS 2012.]

Formal Analysis

- find proof of a specified property by analysis and reasoning
- requires a lot of effort and tools:
- requires SW as a model (!)
- graph theory
- value analysis
- slicing
- predicate abstraction
- approaches:
- *model checking*
- SAT solving
- statistical model checking

general approach of Model Checking:

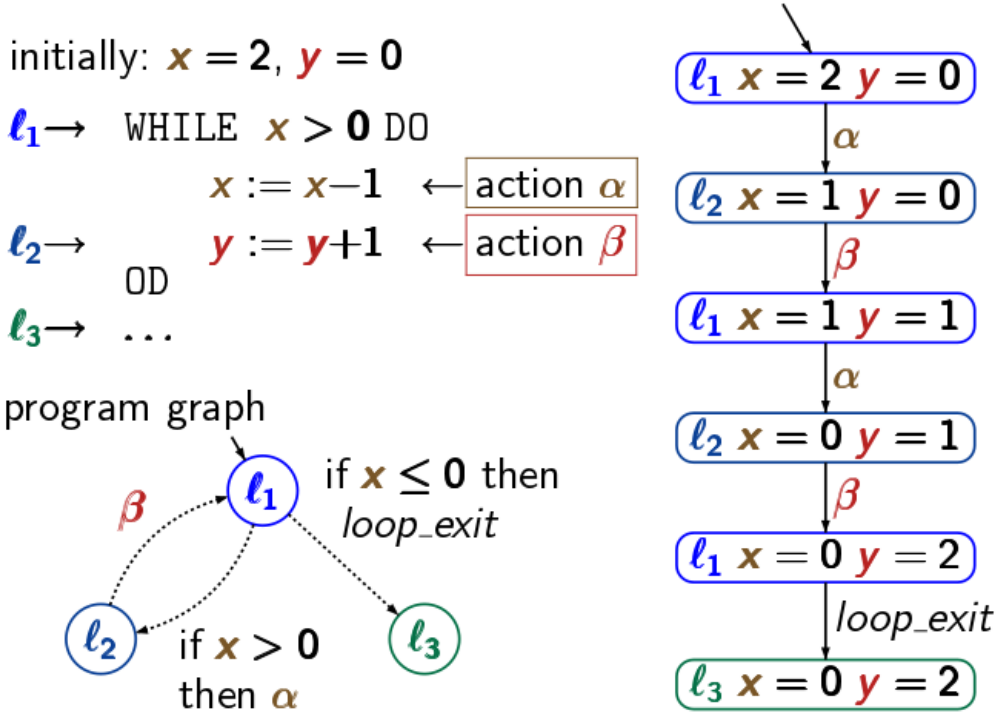


["Embedded Systems and Software Validation", A. Roychoudhury, Elsevier, 2009]
["Principles of Model Checking", C. Baier, J.P. Katoen, MIT Press, 2008]

Model Checking

- requires that you have a model of your system

Option 1: turn your code into a model:



["LLBMC: Bounded Model Checking of C and C++ Programs using a Compiler IR*", F. Merz et. al., VSTTE 2012]

["Frama-C", P. Cuoq et. al., Springer, 2012]

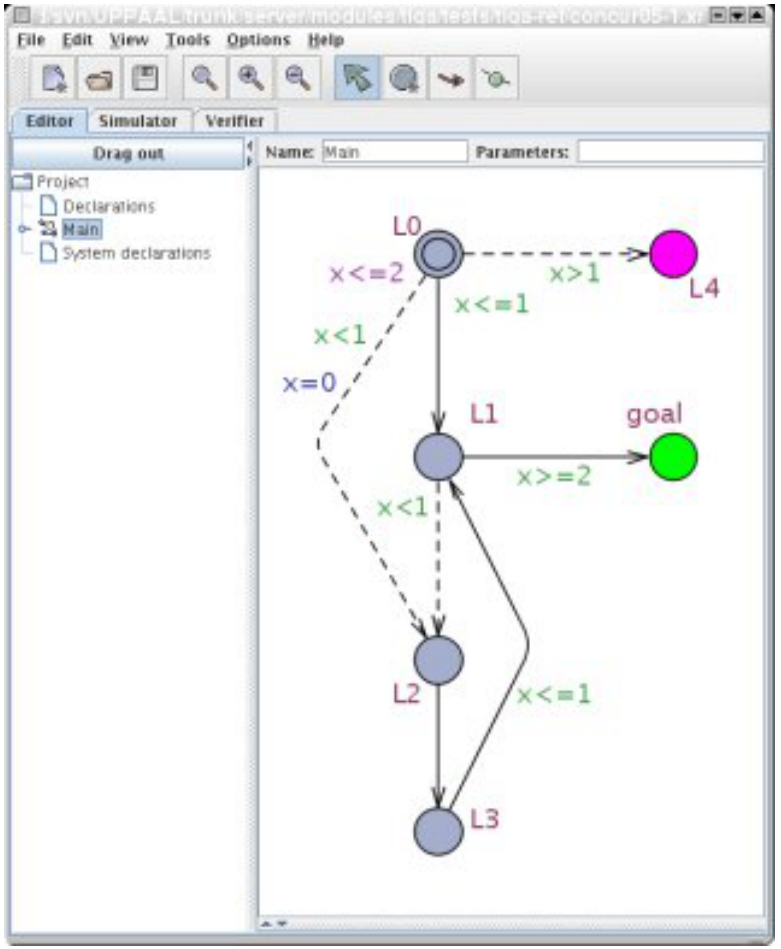
["vUML: a tool for verifying UML models", J. Lilius et. al., IEEE ASE, 2012]

Option 2: Use a programming method, which directly supports model checking (languages like SCADE, StateCharts)

Important Tools

- SPIN, www.spinroot.com
- model checking on automata, LTL
- UPPAAL, www.uppaal.org
- model checking on automata, CTL
- Frama-C, www.frama-c.com
- model checking on C code via predicate logic
- sal-smc, www.sal.csl.sri.com
- model checking on BDD via boolean formulas

- ... many more...



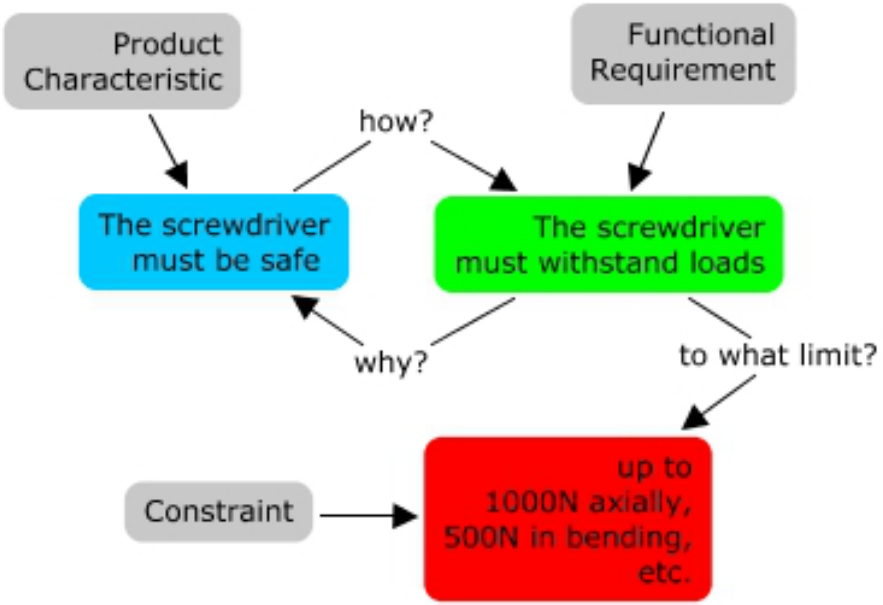
Specifying Functional Properties

- specification = requirement
- we can only verify what we specified

Two main challenges:

1. How to specify a property (technically)?

2. Where to get the property from in the first place?



Property Specification Languages

How to formally capture functional requirements/properties:
- linear temporal logic (LTL)
- "when A, then after 2 time units, B is true" -> f:=A=>xxB
- property templates exist, to help writing them
- computational tree logic (CTL, CTL*)
- different model of time (specify paralle, unknown future)
- predicate logic
- existential and universal quantifiers, formulas where variables can be quantified
- Hoare-Style

Which one to use, and how to write it down exactly, depends on which model checker is employed. They are not equally expressive.

- SPIN =>LTL, Omega-regular
- UPPAAL => automata (CTL core)
- Frama-C => predicate logic, ACSL

Predicate Logic (ACSL) by example

```
/*@
  requires  IsValidRange(a, n);

  assigns  \nothing;

  behavior some:
    assumes  \exists integer i; 0 <= i < n && a[i] == val;
    ensures  0 <= \result < n;
    ensures  a[\result] == val;
    ensures  \forall integer i; 0 <= i < \result ==> a[i] != val;

  behavior none:
    assumes  \forall integer i; 0 <= i < n ==> a[i] != val;
    ensures  \result == n;

  complete behaviors;
  disjoint behaviors;
*/
size_type find(const value_type* a, size_type n, value_type val);
```

The tool "Frama-C" can verify, whether the implementation of find() is compliant to the specification.

["ACSL by Example", Burghardt et. al., Fraunhofer Whitepaper, Version 9.3.0, 2013]
www.frama-c.com

automatically infering properties

Dynamic invariant detection (DAIKON):
- run program with test vectors
- each function call is traced, incl. argument values
- Pre- and post-conditions are derived from the traces
- e.g.: this function must never be called with x<0: sqrt(x)

"The Daikon system for dynamic detection of likely invariants", M. Ernst et. al., Science of Computer Programming, 2007.

Incremental extension of existing properties:
- code change -> property update
- observe code changes and infer new properties automatically

"Property Differencing for Incremental Checking", Yang et. al, ICSE2014, Hyderabad, India.

Challenges in defining properties

- we can only verify what we specified (as a requirement)
- where to get the properties from?
- dynamic inference (see earlier)
- experience of the designer
- failures from past projects (Apollo 11 priority inversion)
- ...

This is a synthesis problem, which requires system thinking.
Naive approaches:
- simulation/virtual prototyping ("run and see what happens")
- iterations in the design phase

Other problems:
- unintended behavior resulting from other requirements

"Coping with unintended behavior of users and products: ontological modelling of product functionality and use," v.d. Vegte et. al., ASME, 2004.
"Definition and Review of Virtual Prototyping", G. Wang, J. Comput. Inf. Sci. Eng, 2003.

Timing Analysis

- functional correctness = error-free logic + right timing
- other terms: reaction time, performance, latency
- indirectly related: processor load, update rate, ...

How long does it take, to execute a piece of code?

- on a given target
- considering all possible inputs and program states
- with or without interfering processes
- always in the *worst case*

Subproblems:

- worst-case execution time (WCET) analysis
- finds the longest execution path , considering processor (architecture, caches, pipelines, ...)
- worst-case response time (WCRT) analysis
- requires WCET next to other information
- accounts for scheduling policy, background processes etc.
- System-level time analysis
- requires WCRT
- computes end-to-end latency in a system (buses, sensors, etc.)

Just Measurements

from www.mrtc.mdh.se (Mälardalen Real-Time Research Center)

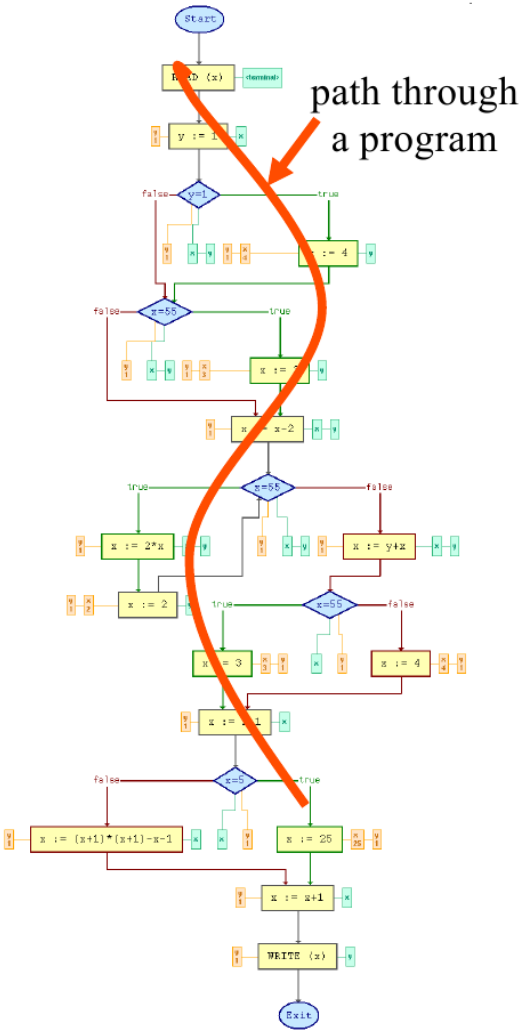
- often people just measure execution times
- problems:
- measurement instrumentation skews the execution time
- cannot guarantee that worst case is encountered, unless all possible paths are triggered (state-space explosion!)

- high-watermarking occasionally employed in in-service systems, to get closer to WCET over the years

Worst-Case Execution Time Analysis

Goal: get a safe upper bound for WCET.
- boils down to find the longest possible execution path in control flow
- not easy, as it depends on processor (cache state!)
- some paths might even be unreachable

Popular Methods:
- *IPET/ILP:* formulate an ILP problem to find longest path. Inefficient.
- *abstract interpretation:* symbolic execution of the program, with reasoning on lattice structures. complex, but efficient.
- *tree-based:* bottom-up summation. Fast but crude overestimation, limited to simple processors.
- *hybrid* with measurements: generate test vectors that trigger long executions. Not safe.

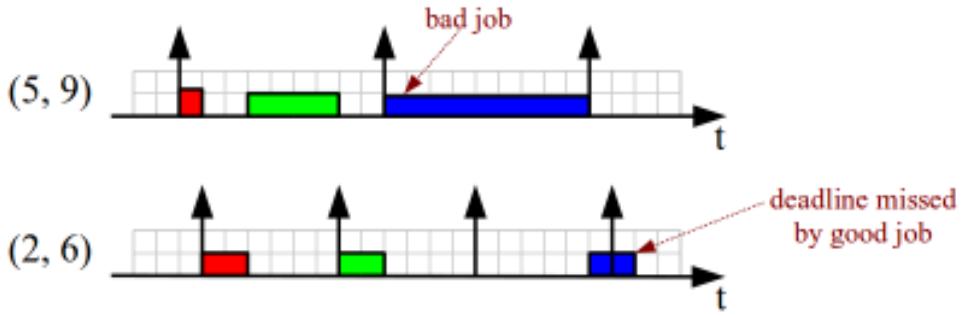


["The WCET problem - overview of methods and survey of tools", R. Wilhelm, ACM TECS, 2008.]

Response-Time Analysis (WCRT)

- WCET assumes no interruptions, no other tasks
- very often task does not run alone on processor (think OS)
- the time to finish computation gets longer -> Worst-Case reaction time (WCRT)

Another way of looking at the same problem
- Schedulability Analysis: check if all tasks finish before their deadline
- techniques differ depending on policy and req. accuracy

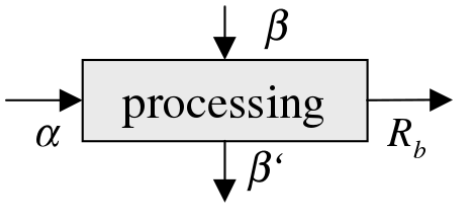


["Handbook of Real-Time and Embedded Systems", Insup Lee et. al., CRC Press, 2007]
["Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", Liu, Layland, ACM, 1973]

System-Level Timing Analysis

- WCRT only works for single-computer systems, i.e., no buses, no other computeres to "talk to"
- today's embedded systems are networked computers
- we need to find the timing of the *networked* system

Approaches:
- Symbolic Timing Analysis (SymTA)
- Real-Time Calculus (RTC)
- computations are represented as "event streams", travelling through the network
- special algebra propagates worst-case bounds



locally performed WCRT analysis, propagated through network

["SymTa/S Symbolic Timing Analysis for Systems", A. Hamann et al, 2004]
["Real-time calculus for scheduling hard real-time systems", L. Thiele, S. Chakraborty, M. Naedele, ISCAS, 2000, Geneva]

Blue-Sky Bugfinding

Means looking for defects and infections, which did not result in failure so far, and possibly never will.

- testing is impractical (except smoke testing), as it requires complete instrumentation

Approaches:
- dynamic analysis:
- you run your program
- (only) the executed trace is checked for common error patterns, e.g., use of uninitialized variables
- static analysis:
- source/object code analysis to find common error patterns, e.g., writing beyond bounds of array

From experience: They are mostly complementary. Use both.

Dynamic Analysis

The analyzer *runs* your program in a sandbox.
- it interceps system calls: memory allocation, jumps, reads and writes to memory, ...
- it can only find infections in the executed trace
- can also find "hard bugs"
- some analyzers even point to the defect (source code line!)

Example: Valgrind

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv) {
    char*mybuf = (char*)malloc(20);
    int option;

    if (argc > 2) {
        option = atoi(argv[1]);
    }
    printf("Option 1 is: %d\n", option);
    if (option > 3) {
        strncpy(mybuf, argv[1], 20);
        printf("Option 2 is %s\n", mybuf);
    }
    return 0;
}
```

```
becker@rcs-x220-mb:/tmp$ valgrind --leak-check=yes ./a.out
...
==22327== Use of uninitialised value of size 8
==22327== at 0x4E7229B: _itoa_word (_itoa.c:195)
==22327== by 0x4E74276: vfprintf (vfprintf.c:1622)
==22327== by 0x4E7D549: printf (printf.c:35)
==22327== by 0x400638: main (test.c:19)
...
==22327== Conditional jump or move depends on uninitialised
value(s)
==22327== by 0x400638: main (test.c:19)
...
==22327== Conditional jump or move depends on uninitialised
value(s)
==22327== at 0x40063D: main (test.c:20)
...
==22327== 20 bytes in 1 blocks are definitely lost in loss record 1
of 1
==22327== at 0x4C28BED: malloc (vg_replace_malloc.c:263)
==22327== by 0x400604: main (test.c:13)
==22327== ERROR SUMMARY: 6 errors from 6 contexts (suppressed: 4
from 4)
```

http://www.valgrind.org

Static Analysis

Dynamic checkers can only find infections in that parts of the code which run, and only with the given valuations.

Static checkers:
- semantic analysis of the source or object code, not run
- can report similar things as dynamic checkers
- can only find "less tricky" bugs
- there is a large number of such tools: splint, cppcheck, ...

Example: CPPcheck

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv) {
    char*mybuf = (char*)malloc(20);
    int option;

    if (argc > 2) {
        option = atoi(argv[1]);
    }
    printf("Option 1 is: %d\n", option);
    if (option > 3) {
        strncpy(mybuf, argv[1], 20);
        printf("Option 2 is %s\n", mybuf);
    }
    return 0;
}
```

```
becker@rcs-x220-mb:/tmp$ cppcheck --enable=all test.c
Checking test.c...
[test.c:24]: (error) Memory leak: mybuf
[test.c:22]: (error) Dangerous usage of 'mybuf' (strncpy
doesn't always 0-terminate )
Checking usage of global functions..
```

Analyzers look at different things! Using them is easy and fast, so use multiple of them.

https://cppcheck.sourceforge.net

Software Architecture Analysis

Recent approaches try to find bugs by mining data on the software repository, or looking at the structure of the software:
- they point to modules that are historically error-prone
- they estimate where yet unknown errors may sit

All still very new, but interesting to follow...

"Design Rule Spaces: A new form of architecture insight", Xiao et. al., ICSE2014, Hyderabad, India.

"An automated Approach to Detect Violations with high confidence in incremental code using a learning system", Radhika et. al., ICSE2014, Hyderabad, India.

Compiler Trust

Almost always programmers do trust the compiler.
- they assume it preserves semantics of the language
- analog for code generators (which technically are compilers)

Implications:
- when model-checking or static analysis attests error-freeness, then this means the compiled version must also be error-free
- when a infection or failure occurs, it's the programmer's fault
- we hunt for defects in the source code

How good are compilers really? Good enough to be trusted?

The truth about compilers

Pitfall #1: undefined behavior (C std)
- some compilers assume, user does avoid undefined behavior
- leads to scary bugs:

```
char *buf = ...;
char *buf_end = ...;
unsigned int len = ...;
if (buf + len >= buf_end)
    return;   /* len too large */
if (buf + len < buf)
    return;   /* overflow, buf+len wrapped around */
/* write to buf[0..len-1] */
```

gcc compiler →

```
char *buf = ...;
char *buf_end = ...;
unsigned int len = ...;
if (buf + len >= buf_end)
    return;   /* len too large */

// Stack overflow goes here...
```

Whilst this is annoying, it's not a technically a bug (RTFM).
But compilers *do* have bugs, which introduce new defects
- nearly all known compilers have/had bugs
- more optimization = more bugs
- between 2008 and 2011, a group of UUtah discovered > 325 previously unknown compiler bugs. Among them: gcc, LLVM.

Approach: model-checking of the compiler, see "compcert" by INRIA

"Formal Verification of a realistic compiler", X. Leroy, ACM, 2011.
"Finding and Understanding Bugs in C Compilers"., Yang et. al., PLDI, 2011.
"Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior", Wang et. al., SOSP, 2013.

Ramifications

- the "chain" of reasoning might be broken
- verification of high-level models may be meaningless
- systems always carry bugs (nothing new)

Proposal

- fundamentally, every system has a non-zero probability of failure, i.e., reliability < 100%
- hardware: MTBF
- software: remaining bugs, dormant failure
- makes no sense to constrain ourselves to hard real-time
- we have to allow for failure, but it needs to be controlled *in average*
- we need redundancy

Configurations

Software that has "options" can be valid only under certain combinations of them. Each combination is a "configuration".
- there is a dependency tree (cmp: .config of the Linux kernel) between configuration options
- e.g., if you chose "multi-threaded I/O locking", you need "threading"

Novel approaches can find obvious misconfigurations:
- identify configurations, where syntax or grammar breaks

```
1 #ifndef Y
2 void foo() { ... }
3 #endif
4
5 #ifdef X
6 void bar() { foo(); }
7 #endif
```

But *if* it does compile,
may there be problems left?

Unless *all* configurations are verified,
yes. But usually only "common" ones are .

"Mining Configuration Constraints", S. Nahdi et. al., ICSE2014, Hyderabad, India.

Contact

If you do have any questions or corrections, please do not hesitate to contact me:

Martin Becker

Institute for Real-Time Computer Systems
Technische Universität München
Arcisstraße 21
80290 München
GERMANY
mailto: becker@rcs.ei.tum.de

I'm a PhD student, working at various topics in the real-time domain. For my recent work, visit

https://www.rcs.ei.tum.de/persons/becker