

Resource Usage on Evolving Software

(Durchgängige Berücksichtigung des Ressourcenverbrauchs durch automatische statische Analyse)

Jörg Herter
AbsInt GmbH
2016

Software Quality

- Embedded control systems must satisfy **high quality** objectives.
 - Software failures can
 - in general: cause high costs, e.g. due to recall campaigns
 - in safety-critical systems: endanger human beings
 - In highly safety-critical systems software test and validation responsible for significant part of development costs ($\geq 50\%$).
 - Challenge: **ensure system safety** at **reasonable costs**.
 - The cost of a software defect is related to the time it is discovered: the **later** a bug is found, the **higher** the costs to fix it.
- ⇒ Verification activities should be performed **continuously** during the development process to detect **critical errors** and **prevent late-stage integration problems**.

Functional Safety

- Demonstration of **functional** correctness
 - Well-defined criteria
 - Automated and/or model-based testing
 - Formal techniques: model checking, theorem proving
- Satisfaction of **non-functional** requirements
 - No crashes due to **runtime errors** (Division by zero, invalid pointer accesses, overflow and rounding errors)
 - Resource usage:
 - **Timing** requirements (e.g. WCET, WCRT)
 - **Memory** requirements (e.g. no stack overflow)
 - **Insufficient**: Tests & Measurements
 - Test end criteria unclear
 - No full coverage possible
 - "Testing, in general, cannot show the absence of errors." [DO-178B]
 - Formal technique: **abstract interpretation** finds the worst case and can provide **guarantees**.

Required by
DO-178B / DO-178C /
ISO-26262, EN-50128,
IEC-61508

Required by
DO-178B / DO-178C /
ISO-26262, EN-50128,
IEC-61508

Non-Functional Software Failures

- Time drift in Patriot missiles in 1991
 - Floating-point rounding error
- Crash of railway switch controller 1995 in Hamburg-Altona
 - Stack overflow
- Explosion of Ariane rocket 1996
 - Arithmetic overflow
- USS Yorktown cruiser propulsion system failure 1997
 - Divide by zero
- Toyota Camry 2004 Unintended Acceleration
 - Stack overflow, memory corruption

Automotive: ISO-26262

Table 1 — Topics to be covered by modelling and coding guidelines

Topics		ASIL			
		A	B	C	D
1a	Enforcement of low complexity	++	++	++	++
1b	Use of language subsets ^b	++	++	++	++

Criticality levels:
A(lowest)
to
D (highest)

- ^b The objectives of method 1b are
- Exclusion of ambiguously defined language constructs which might be interpreted differently by different modellers, programmers, code generators or compilers.
 - Exclusion of language constructs which from experience easily lead to mistakes, for example assignments in conditions or identical naming of local and global variables.
 - Exclusion of language constructs which might result in unhandled run-time errors.

7.4.17 An upper estimation of required resources for the embedded software shall be made, including:

- a) the execution time;
- b) the storage space; and

Excerpt from:
ISO 26262-6 Road vehicles - Functional safety –
Part 6: Product development: Software Level, 2011.

SW Unit Design & Implementation

Table 9 — Methods for the verification of software unit design and implementation

Methods		ASIL			
		A	B	C	D
1a	Walk-through ^a	++	+	o	o
1b	Inspection ^a	+	++	++	++
1c	Semi-formal verification	+	+	++	++
1d	Formal verification	o	o	+	+
1e	Control flow analysis ^{bc}	+	+	++	++
1f	Data flow analysis ^{bc}	+	+	++	++
1g	Static code analysis	+	++	++	++
1h	Semantic code analysis ^d	+	+	+	+
^a In the case of model-based software development the software unit specification design and implementation can be verified at the model level.					
^b Methods 1e and 1f can be applied at the source code level. These methods are applicable both to manual code development and to model-based development.					
^c Methods 1e and 1f can be part of methods 1d, 1g or 1h.					
^d Method 1h is used for mathematical analysis of source code by use of an abstract representation of possible values for the variables. For this it is not necessary to translate and execute the source code.					

Excerpt from:

Final Draft ISO 26262-6 Road vehicles - Functional safety –

Part 6: Product development: Software Level.

Version ISO/FDIS 26262-6:2011(E), 2011.

Static Program Analysis

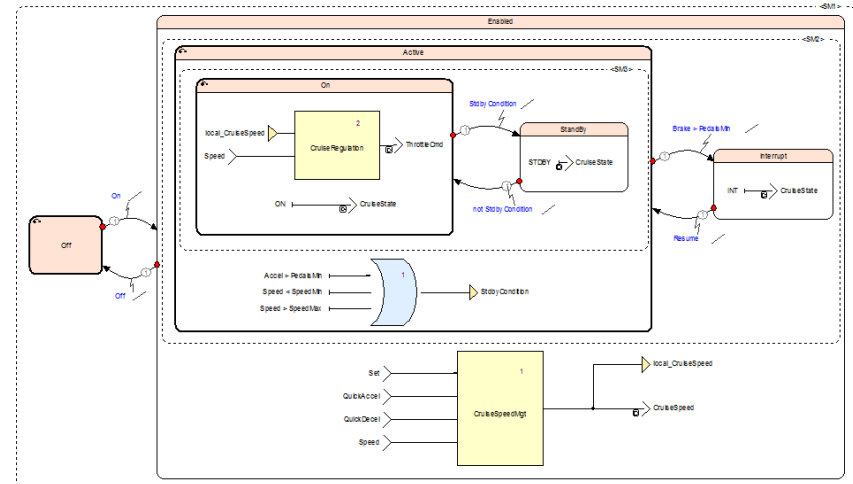
- General Definition: results are only computed from the program structure, **without executing** the program under analysis.
- Analysis scope
 - Binary code: **worst-case execution time, worst-case stack usage**
 - Source code: violations of **coding rules, runtime errors**
- General advantage:
 - No need to perform time-consuming measurements on physical hardware
 - Well-suited for **continuous verification**

Static Program Analysis

- Analysis depth:
 - **Syntax-based**: Style checkers (e.g. MISRA-C)
 - **Unsound** semantics-based: Bug-finders / bug-hunters.
 - Can find some bugs, but cannot guarantee that all bugs are found.
 - E.g. Coverity CMC, Klocwork K7, CodeSonar, Polyspace Bug Finder, ...
 - **Sound** semantics-based / **Abstract Interpretation**-based
 - Can guarantee that all bugs (from the class under analysis) are found.
 - Results valid for every possible program execution with any possible input scenario.
 - Examples: **aiT WCET Analyzer**, **StackAnalyzer**, Polyspace Code Prover, **Astrée**.

Model-based Software Development

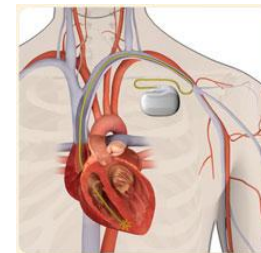
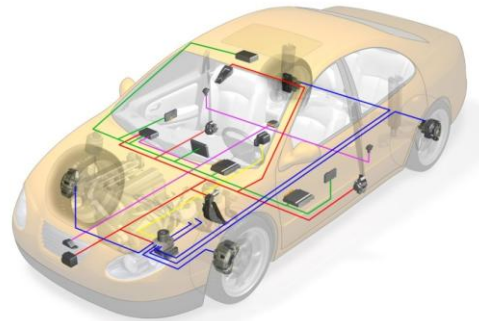
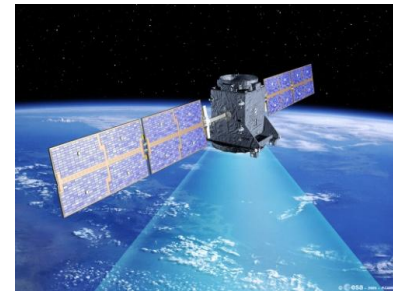
- Application graphically specified by data flow diagrams and/or finite state machines
- Model is software specification and has executable semantics
- Automated & integrated development tools:
 - automatic target code generation (typically C code)
 - automatic simulation / model-based testing
 - formal verification at model level
- Examples: Esterel SCADE, Matlab/Simulink + dSPACE TargetLink
- **BUT**: Higher level of abstraction than with programming in C.
- What about timing, memory consumption, runtime errors, integration issues?



Mastering Resource Usage in Evolving Software

Real-Time Systems

- Controllers in planes, cars, plants, ... are expected to finish their tasks within **reliable time bounds**.
- Hence, it is essential that an upper bound on the execution times of all **runnables/tasks** is known.
- Commonly called the **Worst-Case Execution Time (WCET)**.
- **Schedulability analysis** must be performed.
 - **Worst-Case Response Time (WCRT)**
- **Hard** real-time systems: deadline violations can have **catastrophic** consequences.

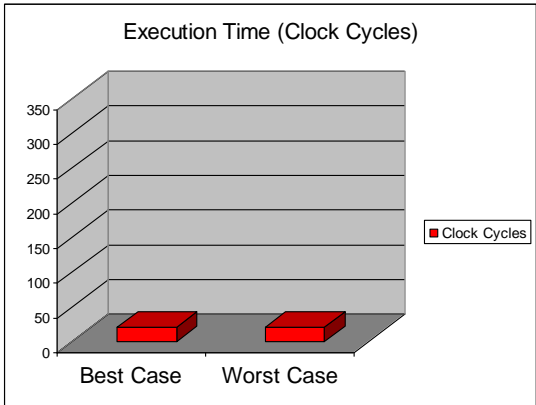


The Timing Problem

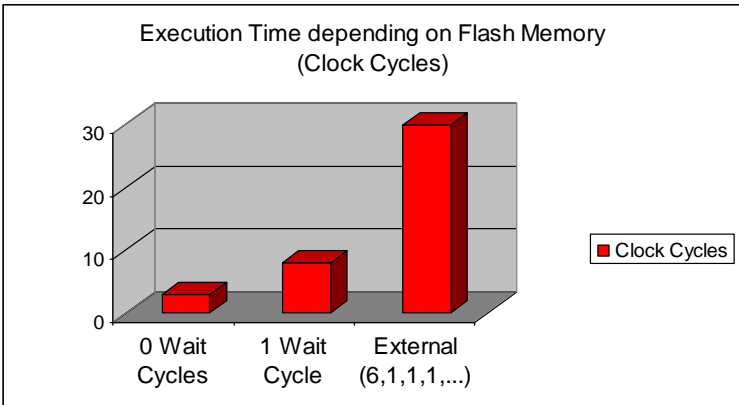
$$x = a + b;$$

```
LOAD    r2, _a
LOAD    r1, _b
ADD     r3, r2, r1
```

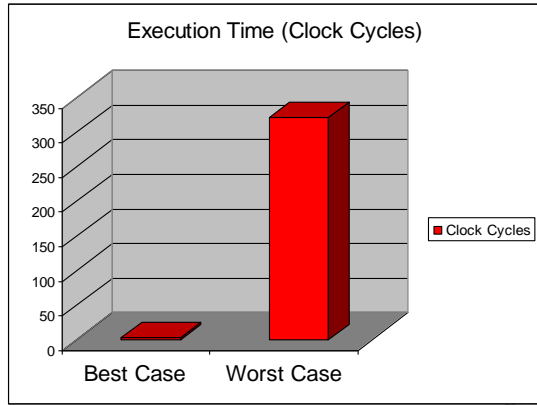
68000 (1990)



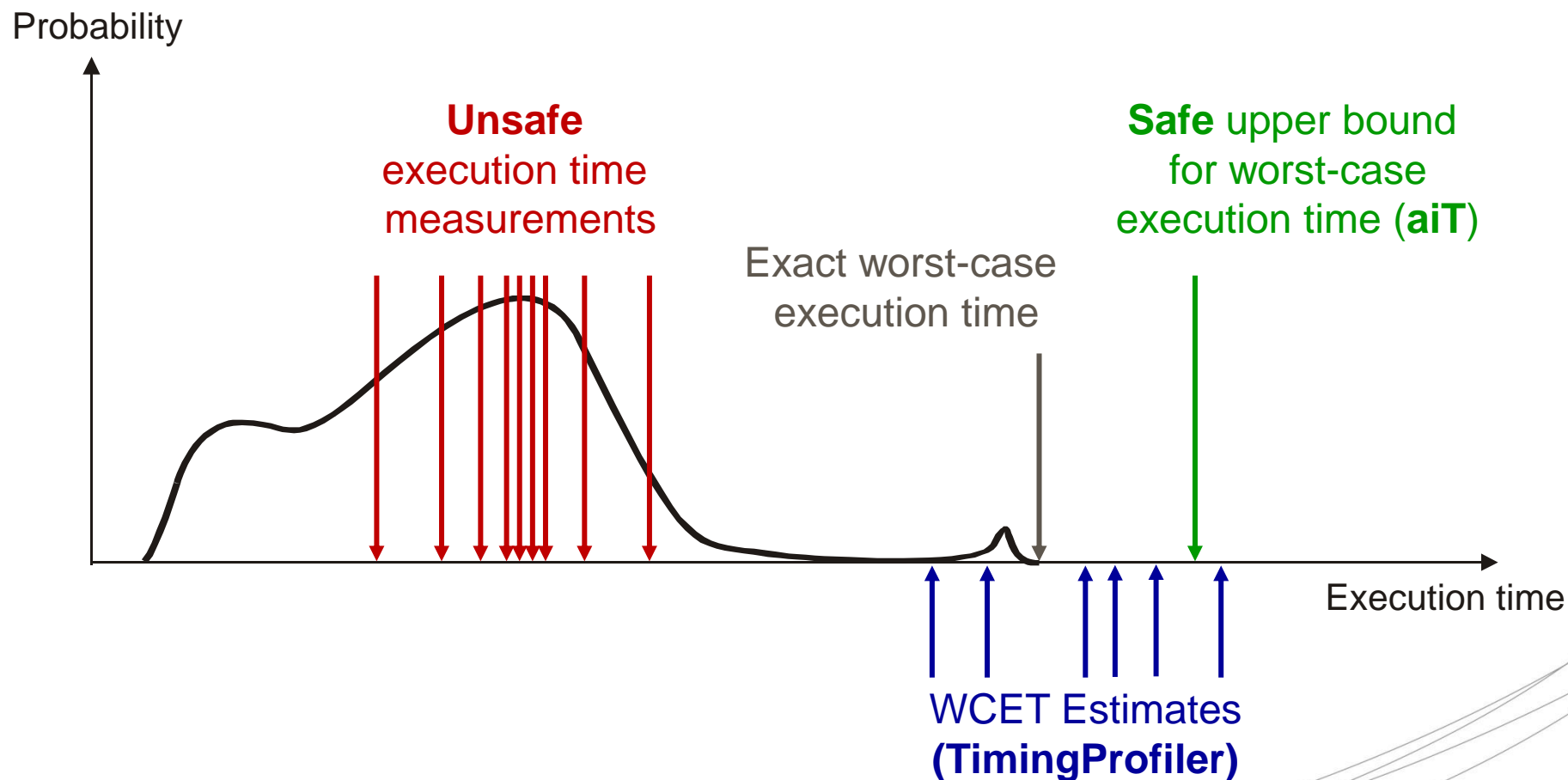
MPC 5xx (2000)



PPC 755 (2001)



The Timing Problem



aiT WCET Analyzer

- Global static program analysis by **Abstract Interpretation** (sound): microarchitecture analysis (caches, pipelines, ...) + **value analysis**
- Integer linear programming for **path analysis**
- Safe and precise bounds on the worst-case execution time (WCET)



Application Code

```
void Task (void) {
  variable++;
  function();
  next++;
  if (next)
    do this;
  terminate()
}
```

Specifications (*.ais)

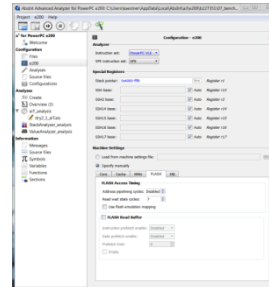
```
clock 10200 kHz ;
loop "_codebook" + 1 loop exactly 16 end;
recursion "fac" max 6;
snippet "printf" is not analysed and takes max 333 cycles;
flow "U_MOD" + 0x4C bytes / "U_MOD" + 0x4 bytes is max 4;
area from 0x20 to 0x497 is readonly;
```

Compiler
Linker

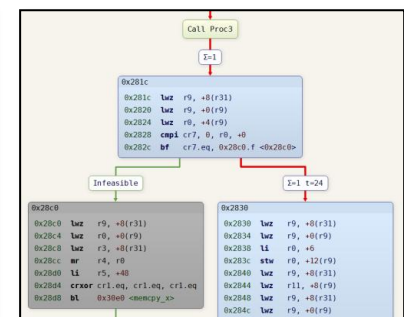
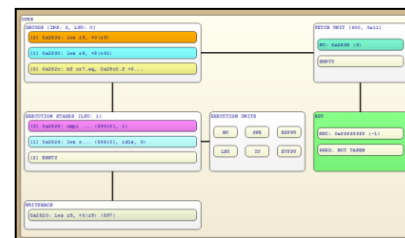
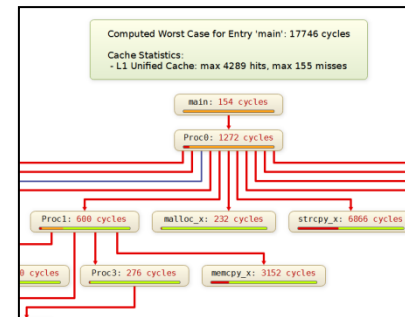
Executable
(*.elf /*.out)

```
EB F6
52 00
90 80
4E F6
```

Entry Point



Worst Case Execution Time
+ Visualization, Documentation



From aiT to TimingProfiler

- aiT is designed to give timing **guarantees**
 - High precision
 - In-depth modelling of microprocessor architecture
 - Extensively validated
 - Requires **exact microcontroller variant** and **detailed configuration** to be known
- In early design stages
 - Precise microcontroller variant & configuration may **not** be fixed yet
 - **Estimations** of the WCET are sufficient
 - Timing information must be available with **low computing effort**
 - User interaction must be **minimal**
 - Solution: **TimingProfiler**

TimingProfiler Characteristics

- Based on a **representative core** implementing the **target instruction set**
 - No in-depth modelling of each specific core + peripherals
- **Defaults** for **hardware** configuration provided
 - No user specification needed
 - Can be adapted by users, if desired
- **Defaults** for **software** characteristics provided
 - Default loop and recursion bounds are provided if exact bound not known
 - Can be refined by users, if desired
- **Computation time** and **memory** requirements **minimized**
 - Pipelines analysis follows local worst-case
 - Much faster than safe analysis, but typically close to safe result

Result Exploration

- **Visualization** of call and control flow graph with **timing information** for critical path, functions, basic blocks.
- Table overview of **direct vs. cumulative execution time** for each function: context-insensitive and context-sensitive
 - *What is maximal contribution of function in all potential execution contexts?*
 - *What is contribution of function when called from A vs. called from B?*
- **Executable browser**: information about variables (names, types, address, size), sections (code/data, access rights, allocated/relocated, ...)
- **Variable usage statistics** based on counting memory accesses on WCET path
- **Memory map explorer**: size and contents of memory areas

Integration in the Development Process

- **Batch mode execution** enables TimingProfiler to be used in continuous integration environments
 - **fully automatic** tool execution
 - **continuous profiling** of timing behavior per night/build/commit/...
- Result and configuration files available in **open XML formats**
- TimingProfiler can be complemented by StackAnalyzer in the same GUI (a³), enabling **continuous timing and stack analysis**.
- Tool coupling to **SymTA/S** enables seamless **system-level** timing analysis from instruction level to interECU end-to-end times.
- Tool coupling to **dSPACE TargetLink** enables continuous timing monitoring during **model-based development**.

The a³ Jenkins Plug-In



- Provides automatic integration of TimingProfiler in the **Continuous Verification Framework Jenkins**
- One-click installation
- Easy to configure
- Fully integrated
- Features:
 - Configure an analyzer run as a Jenkins **build step**.
 - **Automatically mark** build step as **erroneous** depending on analysis results and pedantic levels.
 - **Generate analysis reports** directly in your Jenkins workspace.
 - **Access analysis results** via the Jenkins web interface.

TimingProfiler Benefits

- Provides **continuous feedback about timing** behavior during development
 - Enables **significant reduction of effort** compared to trace and measurement-based timing testing
 - Available even at stages where application is not mature enough for timing measurements
 - Intuitive **graphical exploration** of execution time **distribution**, time-critical **paths**, and **variable usage**
 - Can be **seamlessly integrated** in development process and continuous verification frameworks
- ⇒ **Enables time-conscious software development**
- ⇒ **Avoids late-stage integration problems**

Worst-Case Stack Usage Analysis

The Stack Usage Analysis Problem

- In embedded systems the required stack space has to be reserved for each task at configuration time => **maximal stack usage** has to be **statically known**.
- **Underestimating** the maximal stack usage can cause **stack overflows**. Stack overflows are severe errors:
 - they can cause wrong reactions and program crashes,
 - they are hard to recognize,
 - they are hard to reproduce and fix.
- **Overestimating** the maximal stack usage means **wasting resources** (storage space, energy consumption, weight, money).
- **Testing** is **error-prone** and **expensive**
 - **Typical** stack usage of a task can be very different from **maximum** stack usage.
 - Dynamic testing (e.g. stack pollution checks) typically cannot guarantee that the worst case stack usage has been observed.

Computing the Worst-Case Stack Height

- **StackAnalyzer** computes safe upper bounds of the stack usage of the tasks in a program for all inputs
- Static program analysis based on **Abstract Interpretation**

Application Code

Specifications (*.ais)

Worst Case Stack Usage

+ Visualization, Documentation

```
void Task (void) {
  variable++;
  function();
  next++;
  if (next)
    do this;
  terminate()
}
```

```
recursion "_fac" max 6;
instruction "_main" + 1 computed calls
  "_fact", "_factB", "_factC";
routine "_fib" incarnates max 5;
```

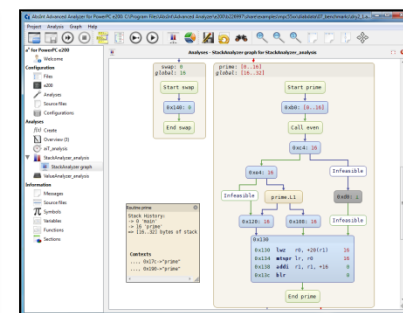
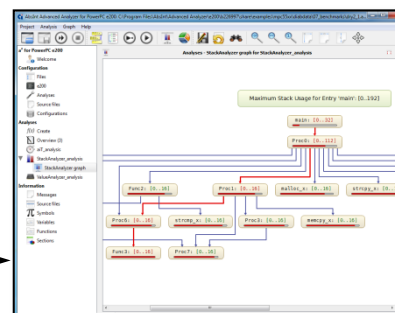
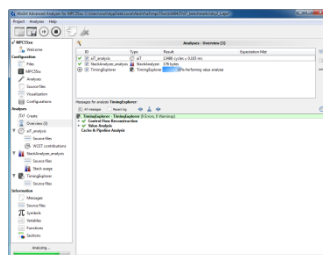
StackAnalyzer

Compiler
Linker

Executable
(*.elf /*.out)

```
EB F6
52 00
90 80
4E F6
```

Entry Point



Information - Sections

Address	End	Size	Name	Readable	Writable	Executable	Allocated	Relocated	Type	File order	Linear address
00000000	00000010	16	.text	R	-	X	-	-	code	0	0x00000000
00000010	00000014	4	.data	R	W	-	A	-	data	1	0x00000010
00000014	00000018	4	.bss	R	W	-	-	-	data	2	0x00000014
00000018	0000001C	4	.rodata	R	-	-	A	-	data	3	0x00000018
0000001C	00000020	6	.comment	R	-	-	-	-	data	4	0x0000001C
00000020	00000024	4	.symtab	R	-	-	-	-	data	5	0x00000020
00000024	00000028	4	.strtab	R	-	-	-	-	data	6	0x00000024
00000028	0000002C	4	.shstrtab	R	-	-	-	-	data	7	0x00000028
0000002C	00000030	6	.debug	R	W	-	-	-	data	8	0x0000002C
00000030	00000034	4	.debug_abbrev	R	-	-	-	-	data	9	0x00000030
00000034	00000038	4	.debug_info	R	-	-	-	-	data	10	0x00000034
00000038	0000003C	4	.debug_line	R	-	-	-	-	data	11	0x00000038
0000003C	00000040	6	.debug_loc	R	-	-	-	-	data	12	0x0000003C
00000040	00000044	4	.debug_ranges	R	-	-	-	-	data	13	0x00000040
00000044	00000048	4	.debug_str	R	-	-	-	-	data	14	0x00000044
00000048	0000004C	4	.debug_sframe	R	-	-	-	-	data	15	0x00000048
0000004C	00000050	6	.debug_unwind	R	-	-	-	-	data	16	0x0000004C
00000050	00000054	4	.debug_zdebug	R	-	-	-	-	data	17	0x00000050
00000054	00000058	4	.debug_gdb_scripts	R	-	-	-	-	data	18	0x00000054
00000058	0000005C	4	.debug_aranges	R	-	-	-	-	data	19	0x00000058
0000005C	00000060	6	.debug_pubnames	R	-	-	-	-	data	20	0x0000005C
00000060	00000064	4	.debug_pubtypes	R	-	-	-	-	data	21	0x00000060
00000064	00000068	4	.debug_types	R	-	-	-	-	data	22	0x00000064
00000068	0000006C	4	.debug_weaknames	R	-	-	-	-	data	23	0x00000068
0000006C	00000070	4	.debug_weaktypes	R	-	-	-	-	data	24	0x0000006C
00000070	00000074	4	.debug_macro	R	-	-	-	-	data	25	0x00000070
00000074	00000078	4	.debug_line_str	R	-	-	-	-	data	26	0x00000074
00000078	0000007C	4	.debug_line_str	R	-	-	-	-	data	27	0x00000078
0000007C	00000080	6	.debug_line_str	R	-	-	-	-	data	28	0x0000007C
00000080	00000084	4	.debug_line_str	R	-	-	-	-	data	29	0x00000080
00000084	00000088	4	.debug_line_str	R	-	-	-	-	data	30	0x00000084
00000088	0000008C	4	.debug_line_str	R	-	-	-	-	data	31	0x00000088
0000008C	00000090	6	.debug_line_str	R	-	-	-	-	data	32	0x0000008C
00000090	00000094	4	.debug_line_str	R	-	-	-	-	data	33	0x00000090
00000094	00000098	4	.debug_line_str	R	-	-	-	-	data	34	0x00000094
00000098	0000009C	4	.debug_line_str	R	-	-	-	-	data	35	0x00000098
0000009C	000000A0	6	.debug_line_str	R	-	-	-	-	data	36	0x0000009C
000000A0	000000A4	4	.debug_line_str	R	-	-	-	-	data	37	0x000000A0
000000A4	000000A8	4	.debug_line_str	R	-	-	-	-	data	38	0x000000A4
000000A8	000000AC	4	.debug_line_str	R	-	-	-	-	data	39	0x000000A8
000000AC	000000B0	6	.debug_line_str	R	-	-	-	-	data	40	0x000000AC
000000B0	000000B4	4	.debug_line_str	R	-	-	-	-	data	41	0x000000B0
000000B4	000000B8	4	.debug_line_str	R	-	-	-	-	data	42	0x000000B4
000000B8	000000BC	4	.debug_line_str	R	-	-	-	-	data	43	0x000000B8
000000BC	000000C0	6	.debug_line_str	R	-	-	-	-	data	44	0x000000BC
000000C0	000000C4	4	.debug_line_str	R	-	-	-	-	data	45	0x000000C0
000000C4	000000C8	4	.debug_line_str	R	-	-	-	-	data	46	0x000000C4
000000C8	000000CC	4	.debug_line_str	R	-	-	-	-	data	47	0x000000C8
000000CC	000000D0	6	.debug_line_str	R	-	-	-	-	data	48	0x000000CC
000000D0	000000D4	4	.debug_line_str	R	-	-	-	-	data	49	0x000000D0
000000D4	000000D8	4	.debug_line_str	R	-	-	-	-	data	50	0x000000D4
000000D8	000000DC	4	.debug_line_str	R	-	-	-	-	data	51	0x000000D8
000000DC	000000E0	6	.debug_line_str	R	-	-	-	-	data	52	0x000000DC
000000E0	000000E4	4	.debug_line_str	R	-	-	-	-	data	53	0x000000E0
000000E4	000000E8	4	.debug_line_str	R	-	-	-	-	data	54	0x000000E4
000000E8	000000EC	4	.debug_line_str	R	-	-	-	-	data	55	0x000000E8
000000EC	000000F0	6	.debug_line_str	R	-	-	-	-	data	56	0x000000EC
000000F0	000000F4	4	.debug_line_str	R	-	-	-	-	data	57	0x000000F0
000000F4	000000F8	4	.debug_line_str	R	-	-	-	-	data	58	0x000000F4
000000F8	000000FC	4	.debug_line_str	R	-	-	-	-	data	59	0x000000F8
000000FC	00000100	6	.debug_line_str	R	-	-	-	-	data	60	0x000000FC

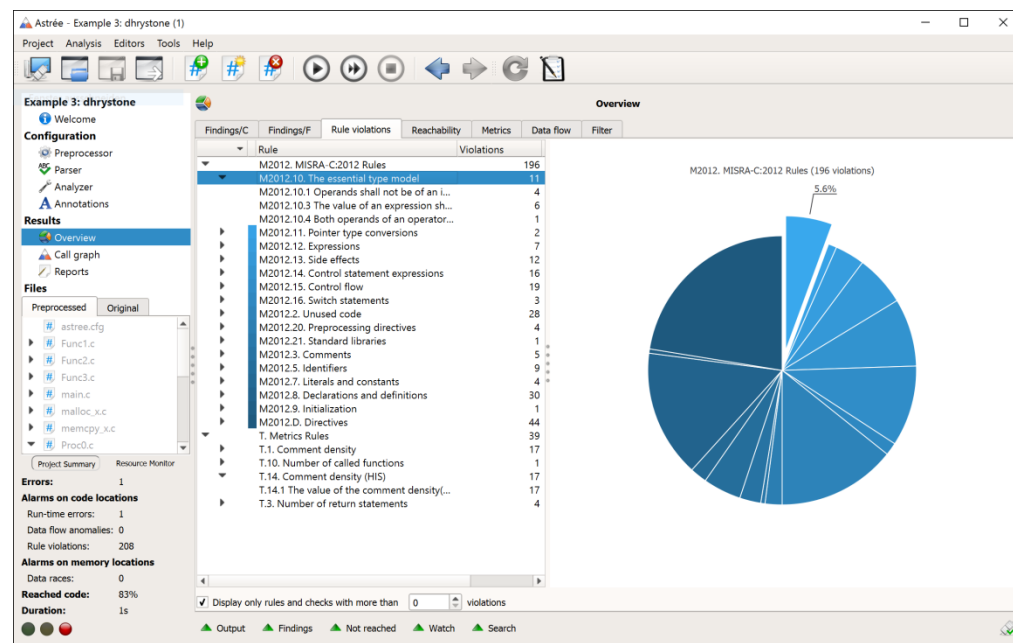
Demonstrating the Absence of Runtime Errors in Evolving Software

The Astrée Analyzer

- Sound static Analyzer based on [Abstract Interpretation](#) designed to [prove the absence of runtime errors and data races](#) in C programs (C99 standard)
 - If Astrée does not report any alarm (potential runtime error / data race) this is a [formal proof](#) that there are [no such errors](#) in the code.
- Reference customer: Airbus flight control software (DO-178B level A).
[No false alarm](#) on [>755.000 LOC](#), [analysis time 6h](#).
- [Automatic tool qualification](#) according to ISO-26262, DO-178B/DO-178C, IEC-61508, IEC-60880, etc. by [Qualification Support Kits](#) (QSK) and [Qualification Software Life Cycle Data reports](#) (QLSCD).
- Support for [model-based code generation](#); tool couplings to dSPACE [TargetLink](#), model link to MATLAB/ SIMULINK available.
- Open formats, full continuous verification support.

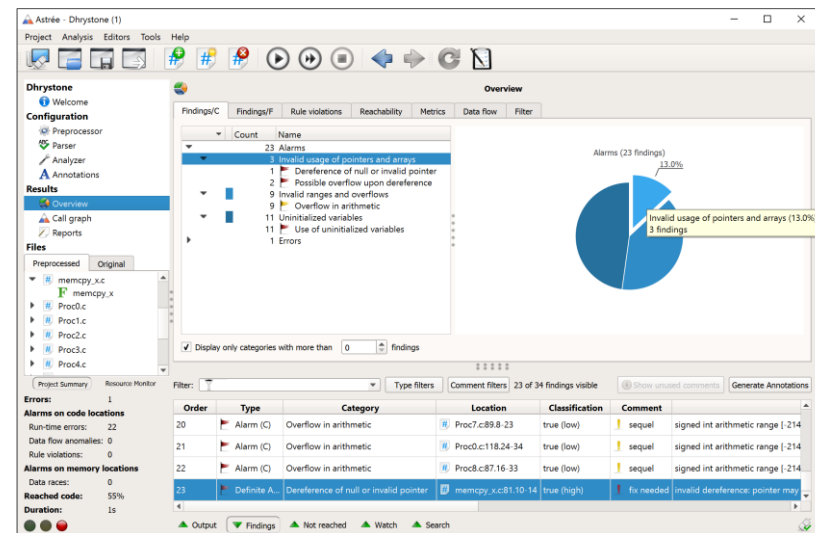
The Astrée Rulechecker

- Check for compliance with **coding rules**
 - Support for MISRA C:2004, MISRA C:2012, CERT, CWE, and ISO/IEC 17961:2013
- Determine **code metrics** and checks for **threshold violations**
 - Includes common **HIS metrics** with compliant default thresholds.
 - E.g.: comment density, cyclomatic complexity, ...
- Extensible** architecture
 - Configurable rules
 - User-defined sets of coding rules can be added (engineering service by AbsInt).



The Astrée Analyzer

- Sound static analysis based on **abstract interpretation** to prove absence of **runtime errors**.
- Astrée detects **all** runtime errors with **few** false alarms:
 - Array index out of bounds
 - Int/float division by 0
 - Invalid pointer dereferences
 - Arithmetic overflows and wrap-arounds
 - Floating point overflows and invalid operations (Inf and NaN)
 - Uninitialized variables
 - Data races, inconsistent locking
 - + Floating-point rounding errors taken into account
 - + User-defined assertions, unreachable code, non-terminating loops

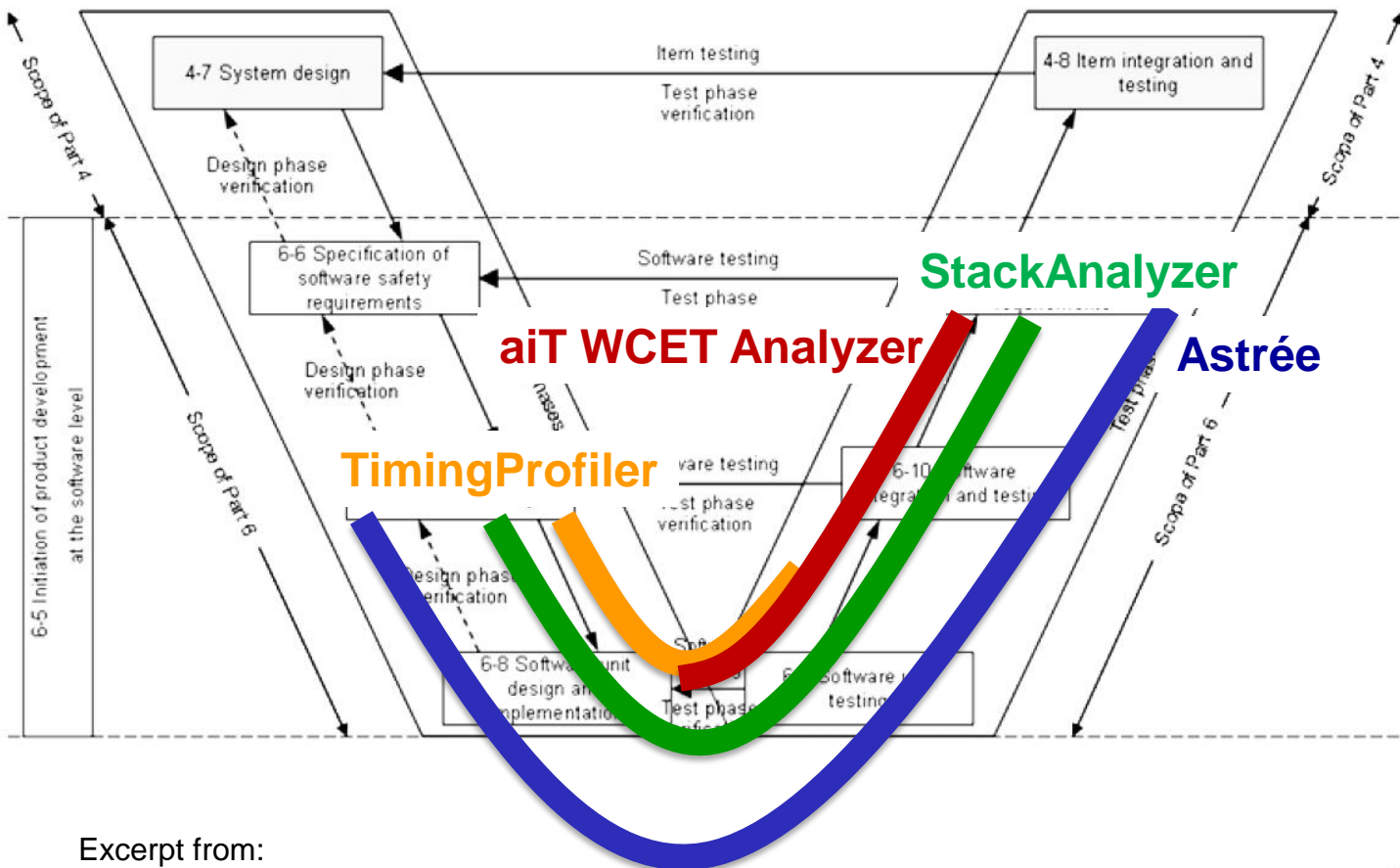


The Astrée Jenkins Plug-In



- Provides automatic integration of Astrée in the **Continuous Verification Framework Jenkins**
- One-click installation
- Easy to configure
- Fully integrated
- Features:
 - Configure an analyzer run as a Jenkins **build step**.
 - Launch **Astrée analysis** as **newly created analysis revision**.
 - **Automatically mark** build step as **erroneous** depending on the categories and statuses of findings.
 - **Generate analysis reports** directly in your Jenkins workspace.
 - **Access analysis results** via the Jenkins web interface.

Development Process



Excerpt from:
Final Draft ISO 26262-6 Road vehicles - Functional safety –
Part 6: Product development: Software Level.
 Version ISO/FDIS 26262-6:2011(E), 2011.

Conclusions

- Static analysis is ideally suited for continuous verification activities throughout the software development stage
- **TimingProfiler** provides **continuous feedback about the timing behavior** without accessing physical hardware and without need to stimulate, test and measure
 - Enables **time-conscious software development**
 - Avoids late-stage integration problems
- **StackAnalyzer** provides **continuous feedback about the stack consumption** without accessing physical hardware and without need to stimulate, test and measure
 - Enables **time-conscious software development**
 - Avoids late-stage integration problems

Conclusions continued

- **Astrée** provides a **history-aware analysis concept** which records the evolution of the analyzer configuration and links it to the software evolution
 - Easily **determine differences** between analysis revisions
 - Easily **adapt analyzer configuration** to new source code versions
 - Analysis results on previous software versions **easily reproducible**



email: info@absint.com
<http://www.absint.com>