Affordable Safe & Secure Mobility Evolution

Advanced Sequential Static Analysis Methodology

Deliverable D 2.3

Deliverable Information			
Nature		**Dissemination Level**	public
Project	ASSUME	**Project Number**	14014
Deliverable ID	2.3	**Date**	01.08.18
Status	draft	**Version**	V1.0
Contact Person	Gideon Neumann	**Organisation**	Assystem
Phone	+4922183099-0	**E-Mail**	gneumann@assystem.com

Author Table

Name	Company	Email
Reinhold Heckmann	AbsInt	heckmann@absint.com
Anton Paule	FZI	paule@fzi.de
Björn Lisper	MDH	bjorn.lisper@mdh.se
Gideon Neumann	Assystem	gneumann@assystemtechnologies.com
Bernard Schmidt	Bosch	bernard.schmidt@de.bosch.com
Carsten Sinz	KIT	carsten.sinz@kit.edu

Change and Revision History

Version	Date	Reason for Change	Affected pages
V0.1	01.08.2018	Set up initial Version	all
V0.1	09.08.2018	Added subsection about SWEET	Section 3.4
V0.1	22.08.2018	Added subsections on aiT and Astrée	Sections 3.1 and 3.2
V0.1	23.08.2018	Added FZI subsections	Section 3.3
V0.2	18.12.2018	Added subsection: MQAnalyzer	Section 3.5 (3.6)
V0.2	21.12.2018	Added subsection: KIT	Section 3.5
V0.2	21.12.2018	Added introduction and Chapter 2	Section 2
V1.0			

Table of Contents

List of Figures

1. Executive Summary

In ASSUME the methods and tools for Single Core Application were developed and finally result in the Serial Static Analysis Toolkit (SSAT)
This deliverable give an impression how the SSAT works and should correspond to the character of a manual for the individual tools.

2. Methodology, Flow and Handling

In work package "Scalable Zero Defect Analysis for Single Core" a tool chain was developed for solving the challenges of sequential static analysis for real industrial software applications. The following pages give a current overview of the developed toolkit.

The advanced Sequential Static Analysis Toolkit (SSAT) is a flexible solution that addresses all sequential use cases. To get the best results we propose to use the full SSAT flow. To keep the flexibility it is also possible to use only parts of the overall tooling. For example if a user is first interested to do a Timing Analysis he/she should not be forced to spend the effort and computational time on the Source Code Verification as well and it is possible to do the source code verification at a later step in the development process.
In this chapter a brief overview of the methodology for the general SSAT is given together with an explanation of its work flow and tool handling. More details about each individual module and its main functionality can be found in Chapter 3.

The overall structure of the SSAT is shown in Figure 1. The graphical representation of the SSAT show how each module will interact with the other modules and how the overall flow could look like.

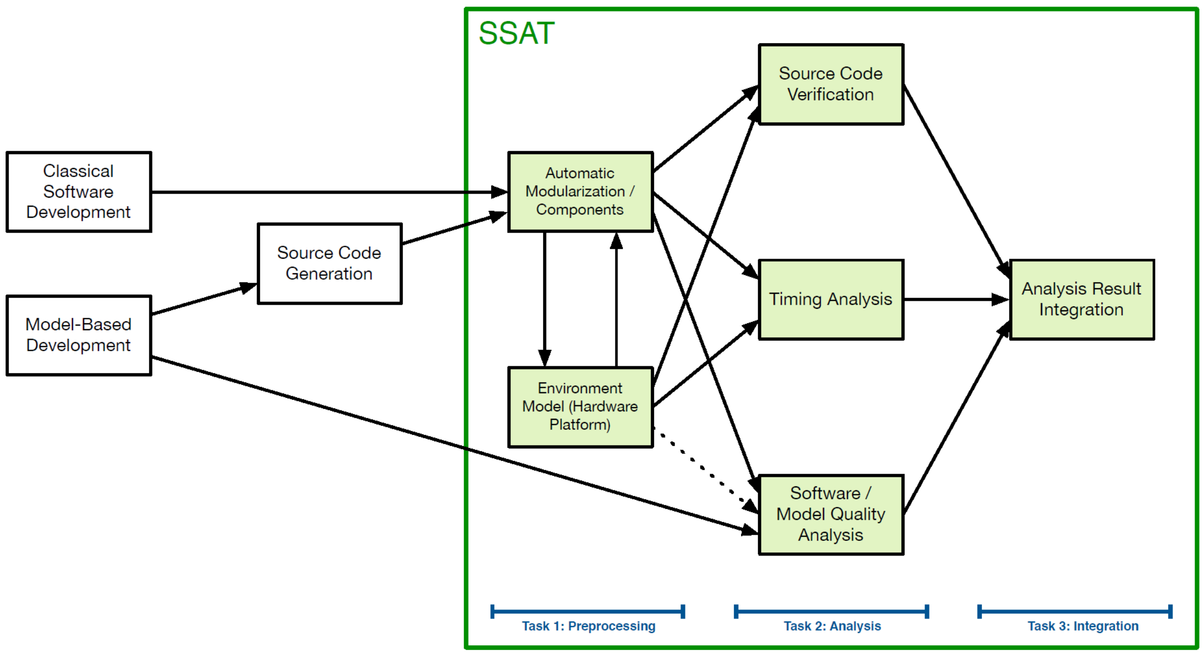

SSAT
Classical Software Development
Model-Based Development
Source Code Generation
Automatic Modularization / Components
Environment Model (Hardware Platform)
Source Code Verification
Timing Analysis
Software / Model Quality Analysis
Analysis Result Integration
Task 1: Preprocessing
Task 2: Analysis
Task 3: Integration

Figure 1 SSAT Structure

2.1. Environment Model

Static analysis tools struggle with checking embedded systems software for defects and runtime errors, because the analyzed source code, often written in C, is heavily tailored to a specific

hardware platform and execution environment. Correct functionality of the embedded system design requires co-verification techniques that are able to address not only the individual hardware (HW) and software (SW) components but also their complex interactions. Generic analysis tools check source code that can be run on a multitude of existing hardware platforms and configurations. Detailed information about the system environment is abstracted and discarded, resulting in an over approximation of analysis results and thus an increase in false warnings (false positives).
Any formal analysis requires a formal model of the design under verification. In case of software, this model often comprises a nontrivial environment that the code runs in (libraries, other programs, and hardware components). As programs often interact with the environment and rely on platform specific properties, substantial manual effort is required to model these parts of the environment. Neglected consideration of hardware properties causes ignorance of interrupts, timers, and I/O-registers. In critical embedded systems, interfaces are often modelled as "volatile" variables and the interface specification typically as constraints on these variables. Modern "intelligent" HW components go beyond simple Port I/O and thus work directly on shared-memory, perform direct memory access (DMA), all asynchronously from the main processor. System side effects, caused by embedded assembly instructions, direct access to system memory and specific I/O-registers via Memory Mapped I/O (MMIO), make it impossible to truly verify the SW component without taking the HW component into account, because incorrect programming of the HW component can have severe consequences, such as memory zones being erased. Abstraction of the HW behavior and the interaction with the SW component often results in an overwhelming amount of false alarms, weakening the overall acceptance of static analysis tools during software development.
The specification of system invariants, hardware constraints and behavior is therefore a necessary but daunting task that has to be performed manually, using configuration files and annotations in the source code. A deep understanding of the underlying hardware architecture and system constraints is required to write invariants for embedded systems software but this specific domain knowledge is often hidden in numerous hardware abstraction layers. One cannot assume that a developer writing the application code neither has a detailed understanding of the whole system architecture, nor is able to write invariants as code annotations using an annotation/assertion language that is specific to an analysis tool. Environment models, which capture hardware properties and constraints, can alleviate the tedious task of analysis setup and invariant specification by utilizing domain knowledge about the system architecture to automatically generate annotations that can be used by different analysis tools.
A key to building an effective co-verification toolchain is to translate the HW and SW models to a common representation with common semantics (co-verification model), enabling specification of system-level-properties across HW/SW boundaries. One approach to achieve co-verification is to model HW properties within the SW domain and capture temporal correlations between SW and HW events using special-purpose primitives to model HW/SW interactions.

Even though building an environment model, which represents the whole hardware platform and all its components, can be a challenging task, which has to be performed by domain experts, the benefits of automatically generating co-verification models for checking run-time errors of hardware-dependent software outweigh the initial modelling effort. The environment model is an abstract representation of domain knowledge about the hardware architecture and serves as the primary source for generating tool specific source code annotations that can be used by static analysis back ends.

2.2. Source Code Verification

Examples of runtime errors in C-Programs include Buffer-Overflows (i.e. data is written beyond the boundary of legal memory which causes different parts of the memory to be filled with wrong data) or Division-By-Zero (depending on the hardware the result may get saturated or an interrupt could be triggered). Such runtime errors are troublesome since they may lead to very costly program failures (e.g. the explosion of the Ariane-5 rocket). The main focus of source code verification is to statically (i.e. without execution) prove the absence of runtime errors in code. For the embedded systems studied in ASSUME the language of choice usually is C.
Sound tools based on techniques such as Abstract Interpretation (Astrée) or (Bounded) Model Checking (LLBMC/QPR) will discover such issues. Hence, being sound guarantees (with respect to the assumptions used, e.g. on hardware-behavior) that no errors are missed by an analysis. The drawback is that since the analysis must be computable, some over-approximations are needed to make the process terminate. These over-approximations lead to false alarms being reported which must eventually be dismissed by a trained human.

The benefit of Abstract Interpretation is that it scales to very large programs (hundreds of thousands or even million lines of code) but due to over-approximations (e.g. widenings) the results are dominated by false alarms. As a remedy, tools like Astrée allow the user to selectively increase the precision of the analysis in order to get rid of false alarms (in subsequent analyses). More (older and more recent) references on Astrée can be found in [1,2].

Bounded Model Checking (BMC) on the contrary, is based on the idea to exhaustively explore all states of a program up to a certain bound k – if no error is found then the program is safe up to this bound. Hence, the technique is under-approximating the system behavior – however, if the bound k is sufficient to guarantee the whole state-space is explored then the system can also be proven safe. Since, embedded programs are inherently bounded (e.g. due to timing requirements), the technique is very effective in verifying these systems. The most successful technique for BMC is to efficiently encode the system behavior up to depth k as a formula (either in propositional calculus or more generally as an SMT-formula). In addition, the error-locations are encoded as a second formula and if the conjunction of the two is satisfiable, then the error-location is reachable and an erroneous program path (counterexample) can be reconstructed from the satisfying assignment. The low-level bounded model checker LLBMC [3] uses the LLVM intermediate representation as a frontend to be able to parse and verify C/C++ programs. The benefit of BMC is its precision – bit-manipulating and floating-point operations can be analyzed in a bit-precise manner. The drawback of this precision is that the technique is less likely to scale to very large programs than Abstract Interpretation.

[1] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Varieties of Static Analyzers: A Comparison with Astrée, invited paper. First IEEE & IFIP International Symposium on ``Theoretical Aspects of Software Engineering'', TASE'07, Shanghai, China, 6—8 June 2007, pp. 3—17.

[2] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, C. Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress, Jan 2016, Toulouse, France.

[3] Florian Merz, Stephan Falke, and Carsten Sinz: LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In Proceedings of the 4th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE '12), Philadelphia, USA. ©Springer-Verlag

2.3. Automatic Modularization

Automatic modularization is necessary for the integration verification of very large software applications written in C. The goal of the automatic modularization is to provide software modules that the verification tool can handle. Each module should be in such a form, that the verification tool can use its strengths and be able to deliver high precision results. Every cut in the software and missing function or variable content introduces new spurious warnings so that the quality of the results is decreasing.

To prevent or reduce these sources of imprecision, an intelligent modularization is needed. To keep individual modules from getting too large or even ending up containing the full software careful optimizations are needed to limit the modules' complexity.
This includes features like recognizing the full original scheduling of the complete software and to construct a new module specific scheduling based on this original scheduling and to remove unneeded functions.
In addition to minimizing modules' complexity the automatic modularization should still allow sound verification. Hence, missing information resulting from any "cutting" needs to be handled in a conservative way by the verification tools.

2.4. Timing Analysis

Hard real-time systems need to satisfy stringent timing constraints, which are derived from the systems they control. In general, upper bounds on the execution times are needed to show the satisfaction of these constraints. To obtain such upper bounds in an automatic way, a timing analysis has to be performed. The two main criteria for evaluating a method or tool for timing analysis are safety – does it produce upper bounds or merely estimates? – and precision – are the bounds or estimates close to the exact values?

Usually, a real-time system consists of a number of tasks or processes that realize the required functionality. For the temporal verification of such systems, it is necessary to obtain worst-case execution time information for the individual, sequential tasks (so-called code-level timing analysis) and to integrate them in a temporal analysis of the entire system taking communication, interference, pre-emption and scheduling overheads into account (system-level timing analysis) [4].

Code-level timing analysis refers to the analysis of "un-preempted" execution times of individual sequential tasks. A task may be a unit of scheduling by an operating system, a subroutine, or some other software unit, which is usually available as a fully-linked executable.

A task typically shows a certain variation of execution times depending on the input data or different behavior of the environment. The longest possible execution time is called the worst-case execution time (WCET). However, in most cases the state space is too large to exhaustively explore all possible executions and thereby determine the exact WCET. Dynamic timing analysis is a method to estimate the WCET of a task by measuring its end-to-end execution time for a subset of the possible executions (the test cases). This determines the maximal observed execution time, which will, in general, underestimate the WCET and so is not safe for hard real-time systems (see Figure 2).

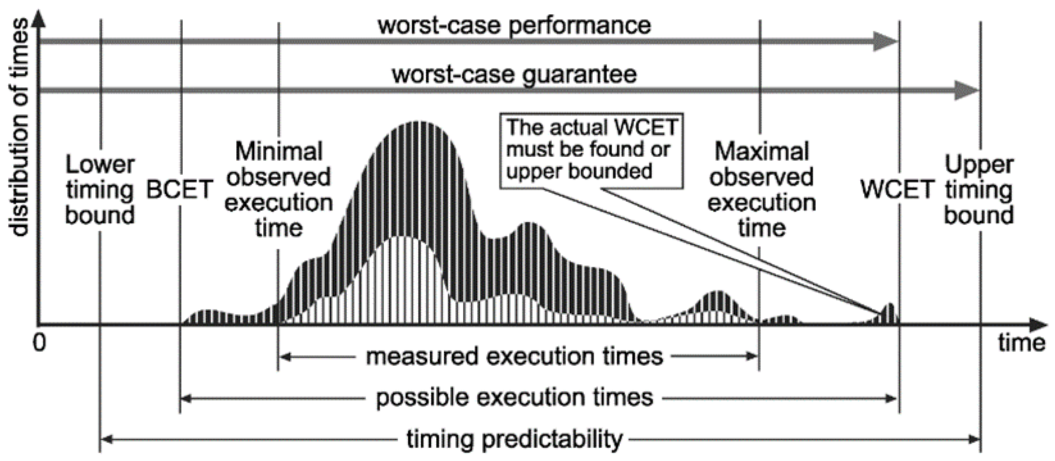


Figure 2: Measured and predicted execution times of tasks (from [5])

Newer measurement-based approaches make more detailed measurements of the execution time of different parts of the task and combine them to give better estimates of the WCET for the whole task. Still, these methods are rarely guaranteed to give safe upper bounds on the execution time. Such bounds can be computed only by methods that consider all possible execution times, that is, all possible executions of the task. These methods use static program analysis together with abstraction of code properties to make timing analysis of the task feasible. Abstraction loses information, so the computed WCET bound usually overestimates the exact WCET. The WCET bound represents the worst-case guarantee the method or tool can give. How much is lost depends both on the methods used for timing analysis and on overall system properties, such as the hardware architecture and characteristics of the software. These system properties can be subsumed under the notion of timing predictability.

Traditionally, static worst-case execution-time estimation can be divided into two related sub-problems. The first problem relates to the program control-flow, determining which parts of the program are executed in which order and which parts are mutually exclusive. This can require extensive value tracking to help determine loop bounds and find points where the program execution depends on input data of the code. Using this information, the worst-case execution-time estimation tool can then reason about the code structure within the application which is commonly performed either on the structure itself, or through implicit-path enumeration [5].

The second problem is finding proper execution-time bounds for each of the basic blocks of the program flow (i.e. groups of operations that form a single node in the control-flow graph of the application). Here the execution time depends mostly on architectural features of both the processor that executes the program as well as the underlying memory hierarchy. For simple architectures, it might be possible to just count the number of operations within the block, but doing so will result in inaccuracies for more complex architectures. For example, processor features such as out-of-order instruction execution and cached memory accesses can insert uncertainty in the execution time. In some cases, the program will execute quickly while in others it will be delayed significantly, depending on the context of the block within the program (or the program within its environment).

Combining the execution-time information for each program part with that of the application structure provides the critical path of the overall application, which in turn results in the worst-case execution-time of the entire task. A possible way to this is by integer linear programming. The analyzer sets up a linear program with integer variables corresponding to the execution counts of the basic blocks. The target function multiplies these variables with the corresponding basic-block execution times. The constraints of the linear program are derived from the control structure, the loop bounds, and other flow properties. Constraints derived from the control structure are for instance those that assert that the sum of the execution counts of the incoming edges of a block equals the sum of the execution counts of the outgoing edges.

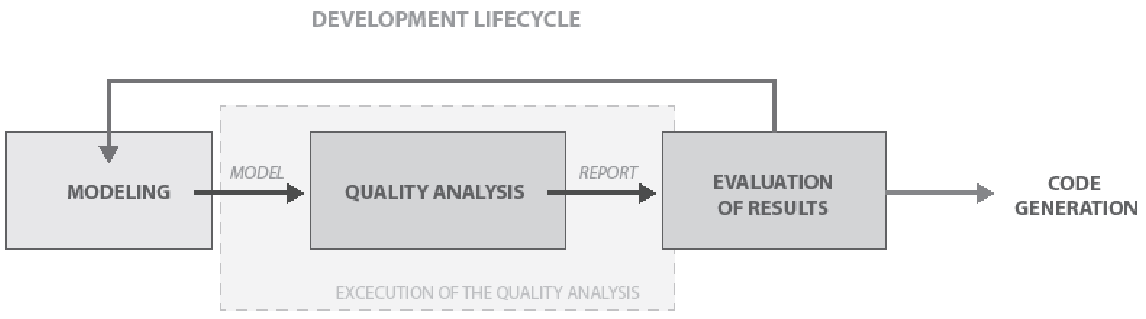
[4] D. Kästner, C. Ferdinand, R. Heckmann, M. Jersak, P. Gliwa. An Integrated Timing Analysis Methodology for Real-Time Systems. Embedded World Congress 2011, Nürnberg, 2011.

[5] Wilhelm, Reinhard, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, et al. 2008. The worst-case execution-time problem – Overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) 36:7.3.

2.5. Software/Model Quality Analysis

The integration of this analysis takes place in the higher-level of the ASSAT tool design rather than on functional source code. Analyzing the quality of the software model is a continuous quality-increasing process to accompany the software developer by improving software quality and detect errors as early as possible in the software development lifecycle.

Overview of the continuous application of the quality analysis process on software model level:



2.6. Analysis Result Integration

The SSAT contains a number of different tools that produce different outputs. Depending on the used flow it is helpful to get only one overall result report instead of checking several analysis from the same tool or from different tools. Different phase of integration are introduced:

- Loose Coupling of Components
- Tight Coupling of Components and Complex Interactions

- Arbitrary Combinations of Components

The loose coupling of components is realized by using the results from one tool as input for the next tool. It happened by original code modification (e.g. portioning of large SW), providing additional information for a correct configuration of the next tool or by providing additional information that can be used as inputs (e.g. Data Ranges). In this case no results need to be integrated.
The tight coupling allow an intensive exchange of information between similar tools or the same tool. This can be used to transfer helpful extracted information from one analysis to the next. This information help to reduce the computational effort and increase the precision of the analysis. Intermediate results from the same tool from different verification runs can be reviewed. An overall analysis result integration is helpful.
The last phase allows to combine tools and results on meaningful paths based on provided use cases and is a combination of the first two phases.
To be able to use the results from different tools and to be able to combine them into an integrated result some additional points need to be fulfilled:

- The results based on the same sources.
- The source code specific tool configurations needs to be comparable (e.g. assumed HW environment, compiler behavior)

The ASSUME Static Code analysis tool Common Configuration Format" (ASC3F) - developed in WP2 – allows to provide a universal configuration and report format for different static analysis tools. The configuration format facilitates the specification of analysis tasks without introducing tool specific configuration files and setup methodologies. The ASC3F also enables the specification of machine-independent configurations, e.g. hiding differences in the concrete paths of source code files. Utilizing a modular configuration approach, reuse of existing analysis configurations is supported as well. Besides the common configuration of static analysis tools, the integration of analysis reports, generated by the analysis tools, is a significant aspect that needs to be considered in the integration of analysis results. The ASC3F format addresses this aspect by establishing a common hierarchy of check categories with accompanying check semantics, making results, obtainable from different tools, directly comparable. The format is designed to be extensible, allowing the addition of new language features and configuration options in the future. **Fehler! Verweisquelle konnte nicht gefunden werden.** shows an overview of the ASC3F components and their dependencies.

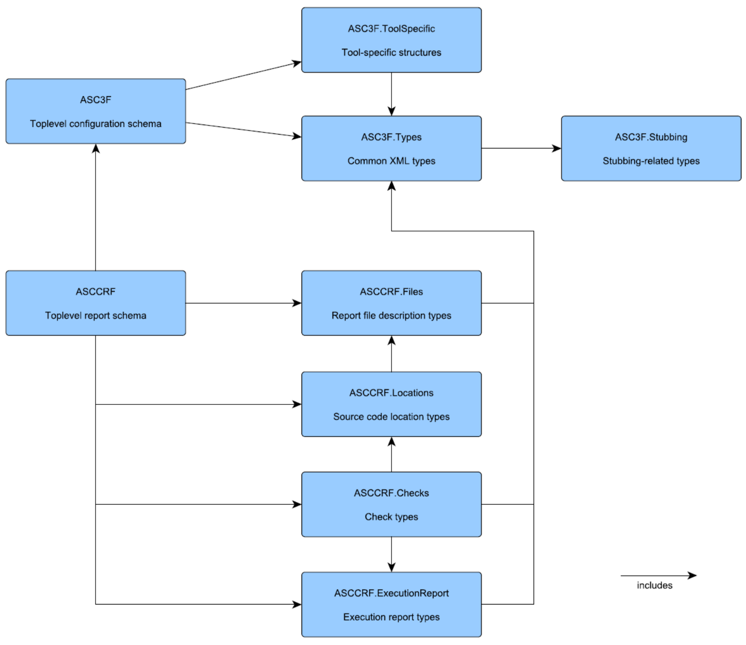


Figure 3: Module structure of the ASC3F format

In WP3, FZI, KIT, and MDH demonstrate the coupling of different tools using the ASC3F format, OSLC and the Linked Data approach. Every tool that allows the setup of analysis project using a configuration file can easily adapt this method. Only an adapter that translates the ASC3F format to the tool specific configuration file needs to be added once. In addition, generated reports are consumed by the adapter and transformed into a common representation using the ASC3F format.

3. Overview of the individual tools

3.1. AbsInt: aiT

3.1.1. Tool Features and Benefit

The aiT tool (https://www.absint.com/ait/) determines safe and precise upper bounds for the worst-case execution times (WCETs) of tasks in real-time systems. Here, a task means a sequentially executed piece of code (no threads, no parallelism, no waiting for external events, and assuming no interference from the outside). aiT operates on binary executables for selected target architectures. In the ASSUME project, support for the MPPA2 architecture of Kalray was added so that aiT can be used in the avionics use cases.

aiT employs static program analysis by means of abstract interpretation. The analysis covers all possible program runs with all possible inputs without actually executing the program. Therefore, it does not require access to physical hardware, nor any code instrumentation or complex test setup. aiT can be seamlessly integrated into the development process and continuous verification frameworks.

AbsInt already offers a plugin for automatic integration of aiT in Jenkins, an open-source automation server for continuous integration and continuous delivery. Using this plugin, developers can automatically analyze the worst-case execution times of their Jenkins builds, automatically mark a build as failed depending on the analysis results according to self-defined criteria such as violated expectations or specific errors, view a compact summary of the analysis results and failed items in the Jenkins build output, access detailed analysis results via the Jenkins web interface, and archive report files directly to the Jenkins workspace.

3.1.2. Input and Output Formats

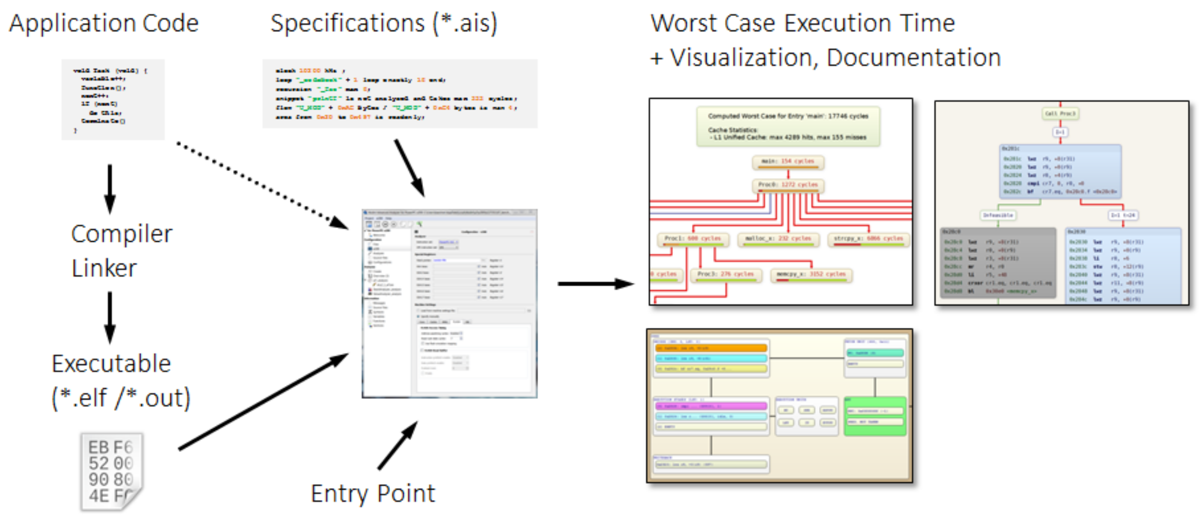
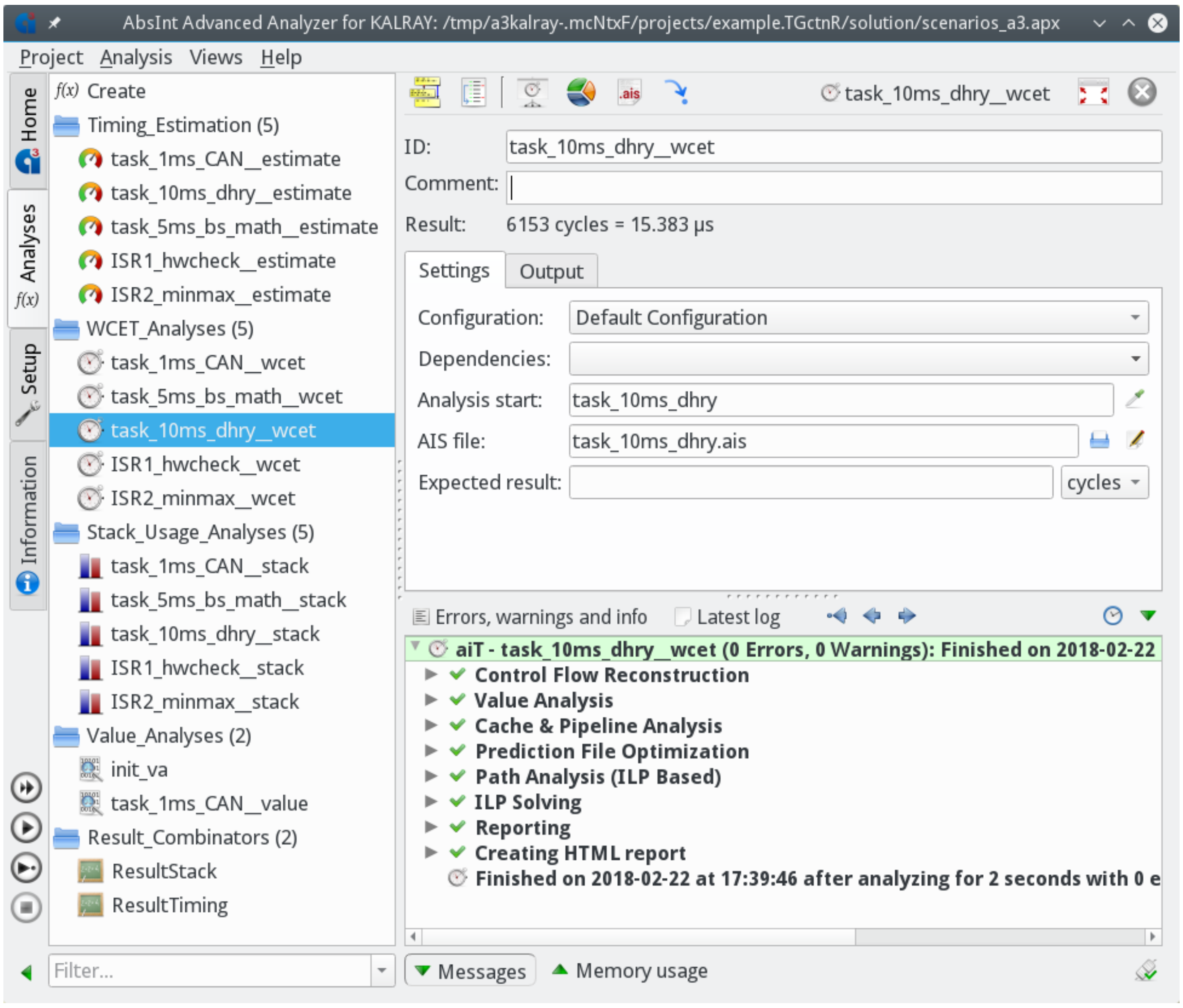


Figure 4: Input and output of aiT

The main input of aiT is a statically linked binary executable containing the task(s) to be analyzed. Secondary input is given by annotations that provide additional information about the analyzed program, e.g. targets of computed calls, loop bounds, and restrictions on the range of variables. aiT tries to compute such information by itself, but sometimes is not able to obtain sufficiently precise useful results.

Annotations may be given in separate annotation files or as specific comments in the C source code. As aiT analyzes binary executables, the presence of C source code is not required, but if it is available, it is read by aiT to watch out for embedded annotations and to be able to refer to C source code in its output.

aiT also requires information about the hardware configuration. Such information can be specified by options in the graphical user interface (GUI), textual descriptions in an annotation file, or specification of the contents of configuration registers (details depend on the target architecture). Names of executable and annotation files are entered in the GUI. From this information, a project file can be formed.



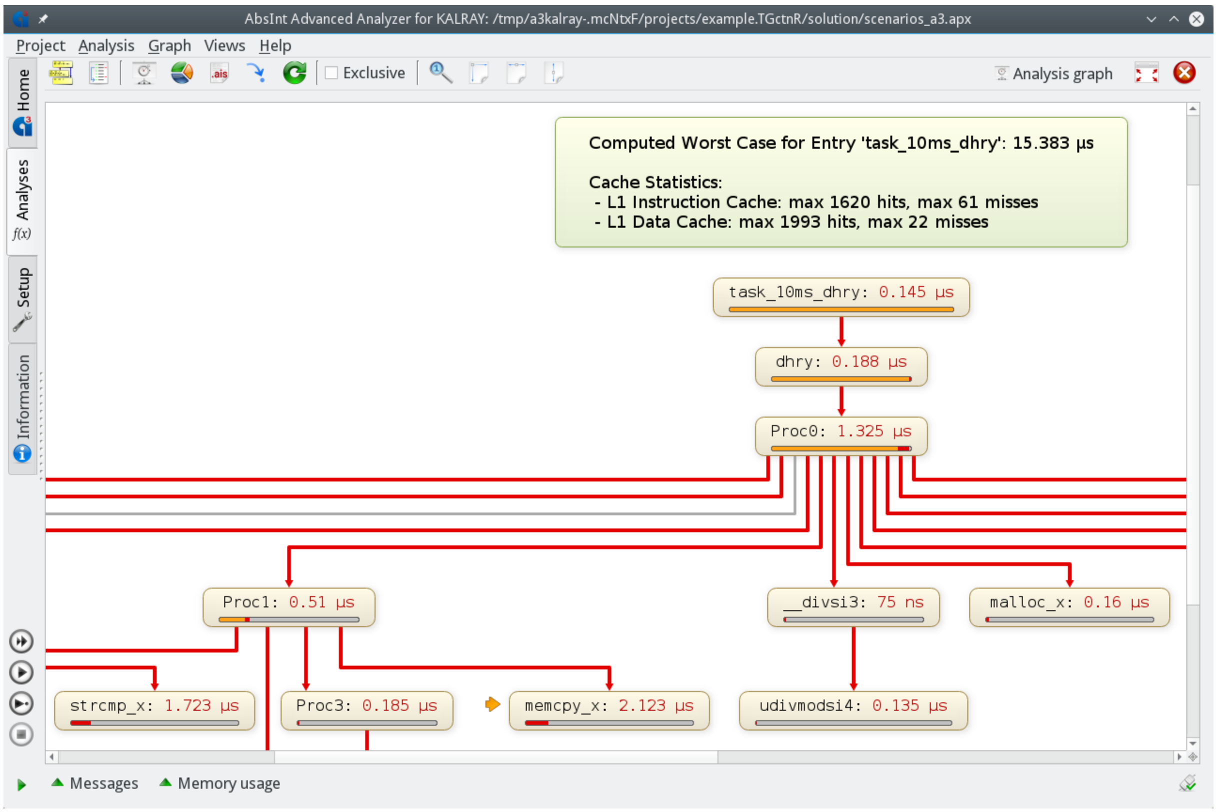
ID: task_10ms_dhry__wcet
Comment:
Result: 6153 cycles = 15.383 µs

Configuration: Default Configuration
Dependencies:
Analysis start: task_10ms_dhry
AIS file: task_10ms_dhry.ais
Expected result: cycles

```
aiT - task_10ms_dhry__wcet (0 Errors, 0 Warnings): Finished on 2018-02-22
  Control Flow Reconstruction
  Value Analysis
  Cache & Pipeline Analysis
  Prediction File Optimization
  Path Analysis (ILP Based)
  ILP Solving
  Reporting
  Creating HTML report
  Finished on 2018-02-22 at 17:39:46 after analyzing for 2 seconds with 0 e
```

Figure 5: GUI after running an aiT analysis

aiT is part of the AbsInt analyzer framework a³, which comes with a graphical user interface (GUI) – see Figure 5. Thus, aiT can be started for interactive work by starting the GUI, loading a project file, and starting a WCET analysis. aiT can also be started in batch mode without user interaction.

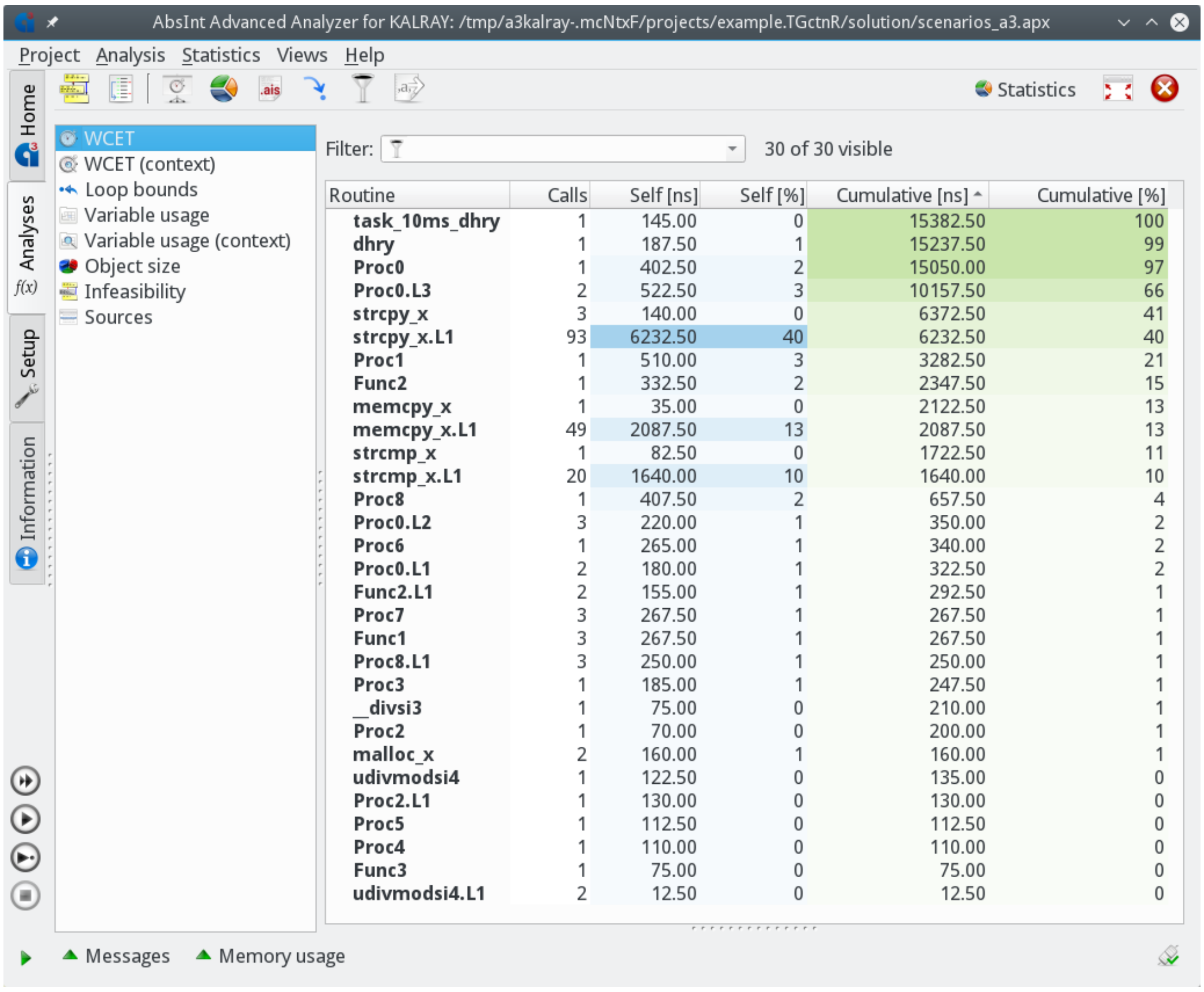


Computed Worst Case for Entry 'task_10ms_dhry': 15.383 µs

Cache Statistics:
- L1 Instruction Cache: max 1620 hits, max 61 misses
- L1 Data Cache: max 1993 hits, max 22 misses

Figure 6: Fragment of control-flow graph with overall WCET and routine WCETs

aiT produces safe over-approximations of the overall worst-case execution time (WCET), the WCETs for routines and basic blocks, worst-case execution numbers for routines and basic blocks, and the worst-case path. This information is given in a textual report file for human inspection and an XML report file that may be read by other applications. aiT produces also combined call graphs and control-flow graphs showing the structure of the analyzed program with analysis results attached to the structure elements (see Figure 6). A comprehensive statistics view gives an overview of the direct vs. cumulative execution times, i.e., the time consumed by each function itself vs. the time consumed by itself and all its callees (see Figure 7). The statistics view also gives detailed information about the variable usage at the binary level, e.g., how often a global variable was accessed (categorized by read and write accesses) per function and in total.



Routine	Calls	Self [ns]	Self [%]	Cumulative [ns]	Cumulative [%]
task_10ms_dhry	1	145.00	0	15382.50	100
dhry	1	187.50	1	15237.50	99
Proc0	1	402.50	2	15050.00	97
Proc0.L3	2	522.50	3	10157.50	66
strcpy_x	3	140.00	0	6372.50	41
strcpy_x.L1	93	6232.50	40	6232.50	40
Proc1	1	510.00	3	3282.50	21
Func2	1	332.50	2	2347.50	15
memcpy_x	1	35.00	0	2122.50	13
memcpy_x.L1	49	2087.50	13	2087.50	13
strcmp_x	1	82.50	0	1722.50	11
strcmp_x.L1	20	1640.00	10	1640.00	10
Proc8	1	407.50	2	657.50	4
Proc0.L2	3	220.00	1	350.00	2
Proc6	1	265.00	1	340.00	2
Proc0.L1	2	180.00	1	322.50	2
Func2.L1	2	155.00	1	292.50	1
Proc7	3	267.50	1	267.50	1
Func1	3	267.50	1	267.50	1
Proc8.L1	3	250.00	1	250.00	1
Proc3	1	185.00	1	247.50	1
__divsi3	1	75.00	0	210.00	1
Proc2	1	70.00	0	200.00	1
malloc_x	2	160.00	1	160.00	1
udivmodsi4	1	122.50	0	135.00	0
Proc2.L1	1	130.00	0	130.00	0
Proc5	1	112.50	0	112.50	0
Proc4	1	110.00	0	110.00	0
Func3	1	75.00	0	75.00	0
udivmodsi4.L1	2	12.50	0	12.50	0

Figure 7: Statistics view showing direct and cumulative WCETs of routines

3.1.3. Tool Constraints

There are aiT WCET analyzers for various different target processors. In ASSUME, the aiT family was extended by aiT for Kalray, which can only analyze fully linked binary executables for Kalray MPPA2 (Bostan).

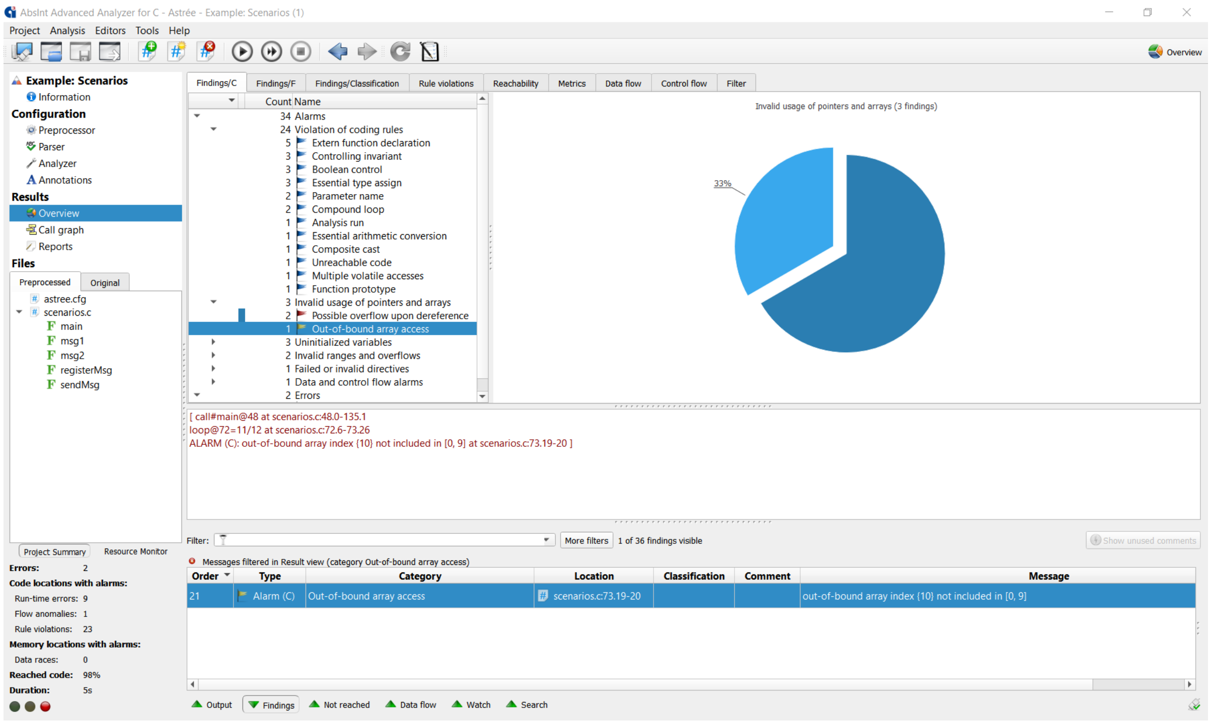
There are Windows and Linux versions of aiT. The Windows version requires 64-bit Windows 7 SP1 or newer, and the Linux version 64-bit CentOS/RHEL 6 or compatible. For proper functioning, there should be at least 4 GB of RAM (16 GB is recommended) and 4 GB of disk space. The Linux version requires the libxcb-* family of libraries to be installed.

aiT can be obtained from AbsInt Angewandte Informatik GmbH (support@absint.com). AbsInt offers commercial licenses, including training, support, and maintenance.

3.2. AbsInt: Astrée

3.2.1. Tool Features and Benefit

Astrée (https://www.absint.com/astree/) is a sound static program analyser that has been developed by ENS and is licensed by AbsInt for industrialization. Addition of new features is done in cooperation with ENS and Sorbonne University (formerly known as UPMC). Astrée was designed to prove the absence of runtime errors and further critical program defects, including array index out of bounds, invalid pointer dereferences, int/float division by 0, arithmetic overflows and wrap-arounds, floating point overflows and invalid operations (Inf and NaN), and uninitialized variables. In floating-point computations, all possible rounding errors, and their cumulative effects, are taken into account. The tool is based on abstract interpretation, a provably correct formal method, and does not require the program under analysis to be instrumented, executed, or stimulated by test cases. The tool can be used on handwritten code, automatically generated code, or any combination thereof. It can be integrated into continuous verification frameworks such as Jenkins (a Jenkins plugin exists).



```
Count Name
   34 Alarms
   24 Violation of coding rules
    5 Extern function declaration
    3 Controlling invariant
    3 Boolean control
    3 Essential type assign
    2 Parameter name
    2 Compound loop
    1 Analysis run
    1 Essential arithmetic conversion
    1 Composite cast
    1 Unreachable code
    1 Multiple volatile accesses
    1 Function prototype
    3 Invalid usage of pointers and arrays
    2 Possible overflow upon dereference
    1 Out-of-bound array access
    3 Uninitialized variables
    2 Invalid ranges and overflows
    1 Failed or invalid directives
    1 Data and control flow alarms
    2 Errors
```

```
[ call#main@48 at scenarios.c:48.0-135.1
loop@72=11/12 at scenarios.c:72.6-73.26
ALARM (C): out-of-bound array index {10} not included in [0, 9] at scenarios.c:73.19-20 ]
```

Filter: 1 of 36 findings visible

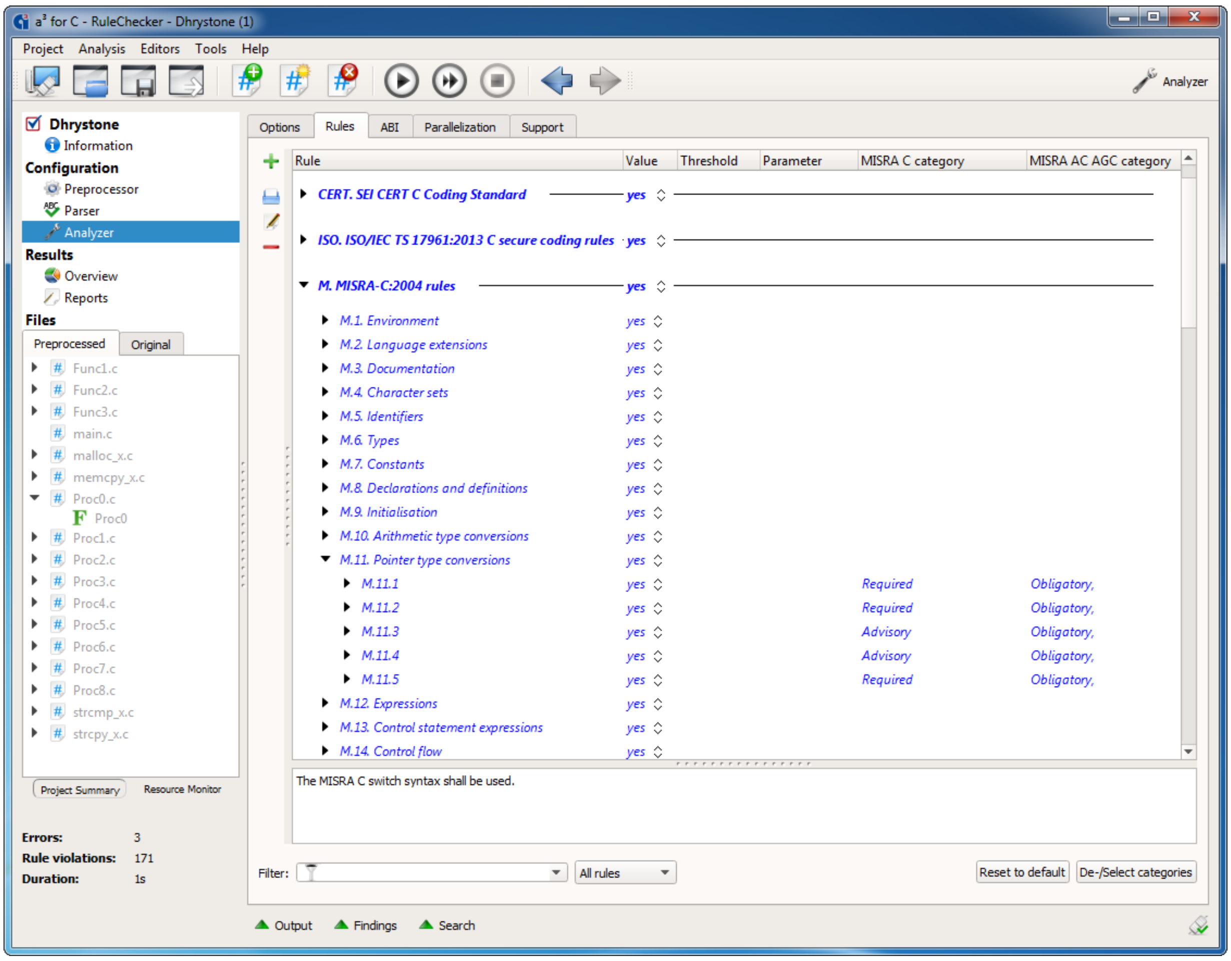
Messages filtered in Result view (category Out-of-bound array access)

Order	Type	Category	Location	Classification	Comment	Message
21	Alarm (C)	Out-of-bound array access	scenarios.c:73.19-20			out-of-bound array index {10} not included in [0, 9]

Errors:	2
Code locations with alarms:	
Run-time errors:	9
Flow anomalies:	1
Rule violations:	23
Memory locations with alarms:	
Data races:	0
Reached code:	98%
Duration:	5s

Figure 8: Overview on Astrée analysis results

Astrée contains the seamlessly integrated RuleChecker that checks code for compliance with MISRA, CWE, ISO/IEC, and SEI CERT C coding rules. The tool can also check for various code metrics such as comment density or cyclomatic complexity. Custom extensions for user-defined coding guidelines are available on request. Using RuleChecker in conjunction with the sound semantic analyses offered by Astrée guarantees zero false negatives and minimizes false positives on semantical rules. No standalone MISRA checker can offer this, and no testing environment can match the full data and path coverage provided by the static analysis.



Rule	Value	Threshold	Parameter	MISRA C category	MISRA AC AGC category
CERT. SEI CERT C Coding Standard	yes				
ISO. ISO/IEC TS 17961:2013 C secure coding rules	yes				
M. MISRA-C:2004 rules	yes				
M.1. Environment	yes				
M.2. Language extensions	yes				
M.3. Documentation	yes				
M.4. Character sets	yes				
M.5. Identifiers	yes				
M.6. Types	yes				
M.7. Constants	yes				
M.8. Declarations and definitions	yes				
M.9. Initialisation	yes				
M.10. Arithmetic type conversions	yes				
M.11. Pointer type conversions	yes				
M.11.1	yes			Required	Obligatory,
M.11.2	yes			Required	Obligatory,
M.11.3	yes			Advisory	Obligatory,
M.11.4	yes			Advisory	Obligatory,
M.11.5	yes			Required	Obligatory,
M.12. Expressions	yes				
M.13. Control statement expressions	yes				
M.14. Control flow	yes				

The MISRA C switch syntax shall be used.

Errors:	3
Rule violations:	171
Duration:	1s

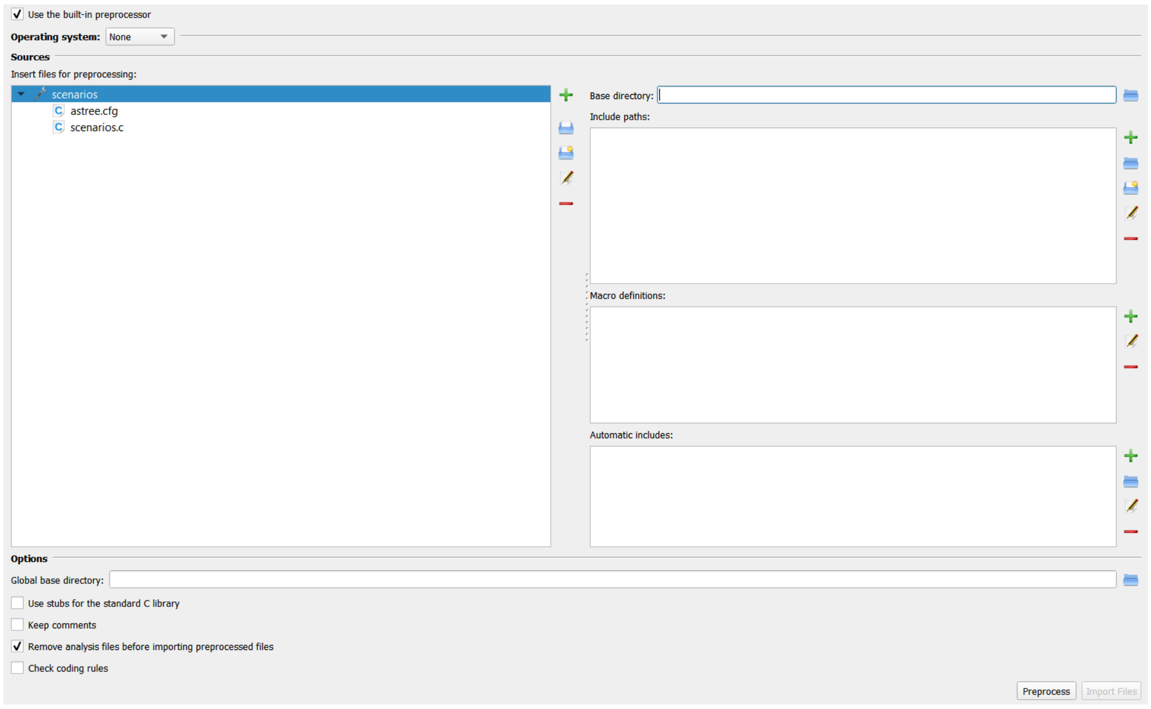
Figure 9: RuleChecker configuration view

In the ASSUME project, the Astrée tool has been evaluated by the project partners, in particular Bosch and Daimler. The evaluation showed the need for various extensions of the tool. First, Astrée had to be extended to support some C extensions occurring in the partners' code. Then the main goal was to improve the user interface, to increase the precision of the analysis, and to reduce the effort to examine (possibly false) alarms. To this end, the repeated reporting of errors with the same cause has been avoided, e.g. only the first access is reported now in case of repeated accesses to a non-initialized variable. Another goal was to reduce the analysis time and the memory consumption of the analyzer, in particular when analyzing very large code bases. The results are impressive, e.g. for some example, the analysis time dropped from 10 days to 3 days, for another example from 26 hours to 2.5 hours, and for a third from 2 hours to 38 minutes.

3.2.2. Input and Output Formats

Input

1. Astrée works on preprocessed C code. If desired, a built-in preprocessor can be used to obtain preprocessed code. The code is then parsed and translated into an intermediate representation on which the runtime error analysis is performed.



✓ Use the built-in preprocessor

Operating system: None

Sources

Insert files for preprocessing:

- scenarios
  - astree.cfg
  - scenarios.c

Base directory:

Include paths:

Macro definitions:

Automatic includes:

Options

Global base directory:

☐ Use stubs for the standard C library

☐ Keep comments

✓ Remove analysis files before importing preprocessed files

☐ Check coding rules

Figure 10: Preprocessor configuration view

2. For each analysis, Astrée needs an entry point – typically a function of particular interest, or simply main. Astrée will then analyze all portions of the code that can be reached by non-interrupted sequential program execution from that entry point.

3. It is possible to provide an analysis wrapper – e.g. to model reactive system behavior – in a dedicated C file associated with the analysis.

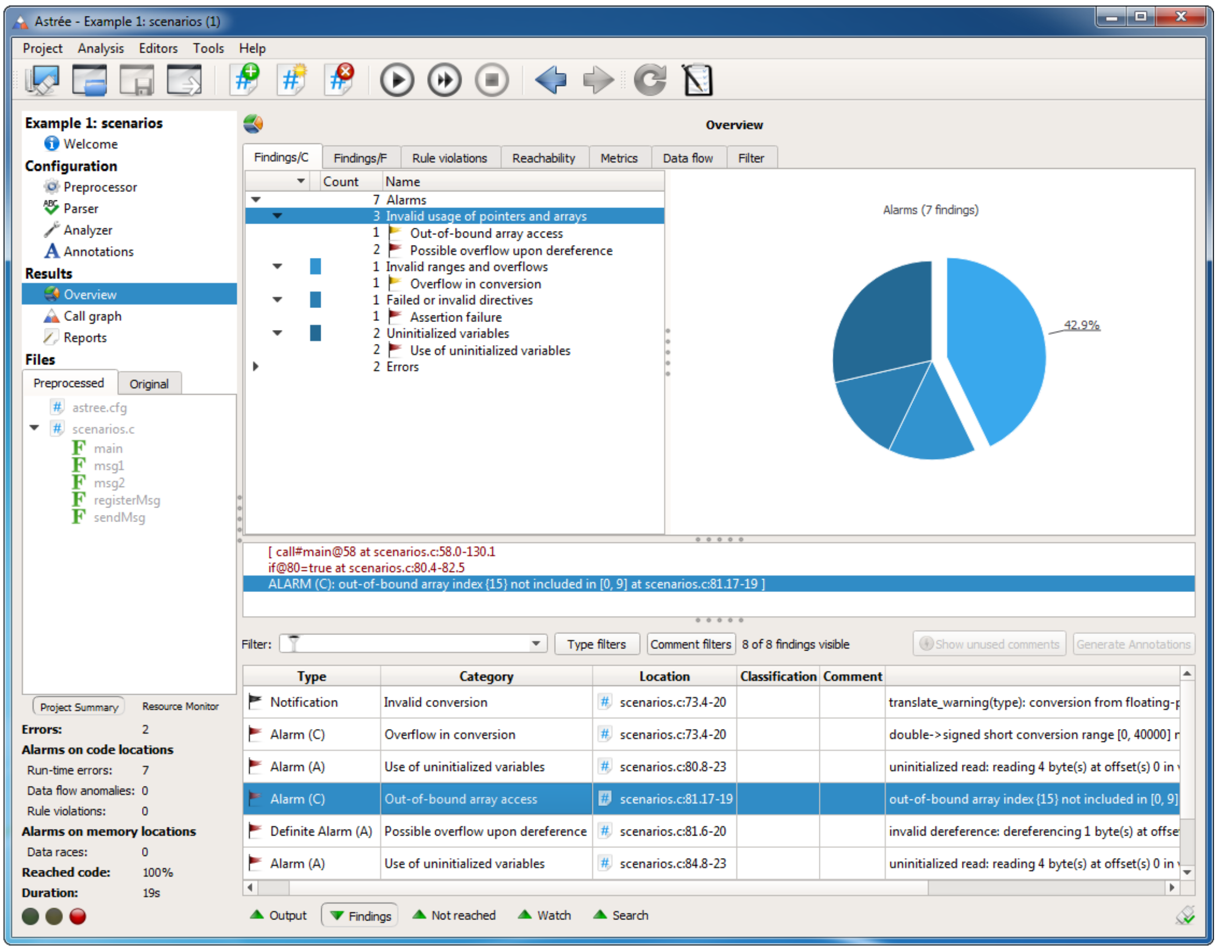
4. Astrée can also be configured with different ABI (application binary interface) settings.

5. Lastly, Astrée accepts formal analysis directives that provide external information to the analyzer, e.g., about the environment, or to steer the analysis precision. The directives are specified in the dedicated, human-readable Astrée Annotation Language (AAL) so that the source code does not have to be modified. The locations to which the directives refer are specified over the program structure and are robust with respect to line numbers.

<u>**Output**</u>

The most important result of the analysis is a list of alarms, i.e. of potential runtime errors. Each error is reported together with its type and the source code location where it occurs. If Astrée can prove that an alarm will always occur in a specific context, it is classified as a definite runtime error. In addition, various kinds of statistics are compiled. Interactive tables, graphs and charts let you quickly see which code areas are most prone to which kinds of errors.

Report files can be generated for documentation and certification purposes. The entire analysis project can be saved as well, including all files, settings, annotations and comments. The analyzer also provides coverage information showing unanalyzed code statements. In absence of definite runtime errors, code reported as unanalyzed is definitely unreachable.



Count	Name
7	Alarms
3	Invalid usage of pointers and arrays
1	Out-of-bound array access
2	Possible overflow upon dereference
1	Invalid ranges and overflows
1	Overflow in conversion
1	Failed or invalid directives
1	Assertion failure
2	Uninitialized variables
2	Use of uninitialized variables
2	Errors

```
[ call#main@58 at scenarios.c:58.0-130.1
if@80=true at scenarios.c:80.4-82.5
ALARM (C): out-of-bound array index {15} not included in [0, 9] at scenarios.c:81.17-19 ]
```

Filter: | Type filters | Comment filters | 8 of 8 findings visible | Show unused comments | Generate Annotations

Type	Category	Location	Classification	Comment	
Notification	Invalid conversion	scenarios.c:73.4-20			translate_warning(type): conversion from floating-p
Alarm (C)	Overflow in conversion	scenarios.c:73.4-20			double->signed short conversion range [0, 40000] r
Alarm (A)	Use of uninitialized variables	scenarios.c:80.8-23			uninitialized read: reading 4 byte(s) at offset(s) 0 in
Alarm (C)	Out-of-bound array access	scenarios.c:81.17-19			out-of-bound array index {15} not included in [0, 9]
Definite Alarm (A)	Possible overflow upon dereference	scenarios.c:81.6-20			invalid dereference: dereferencing 1 byte(s) at offse
Alarm (A)	Use of uninitialized variables	scenarios.c:84.8-23			uninitialized read: reading 4 byte(s) at offset(s) 0 in

Errors:	2
Alarms on code locations	
Run-time errors:	7
Data flow anomalies:	0
Rule violations:	0
Alarms on memory locations	
Data races:	0
Reached code:	100%
Duration:	19s

Figure 11: Overview on reported code defects

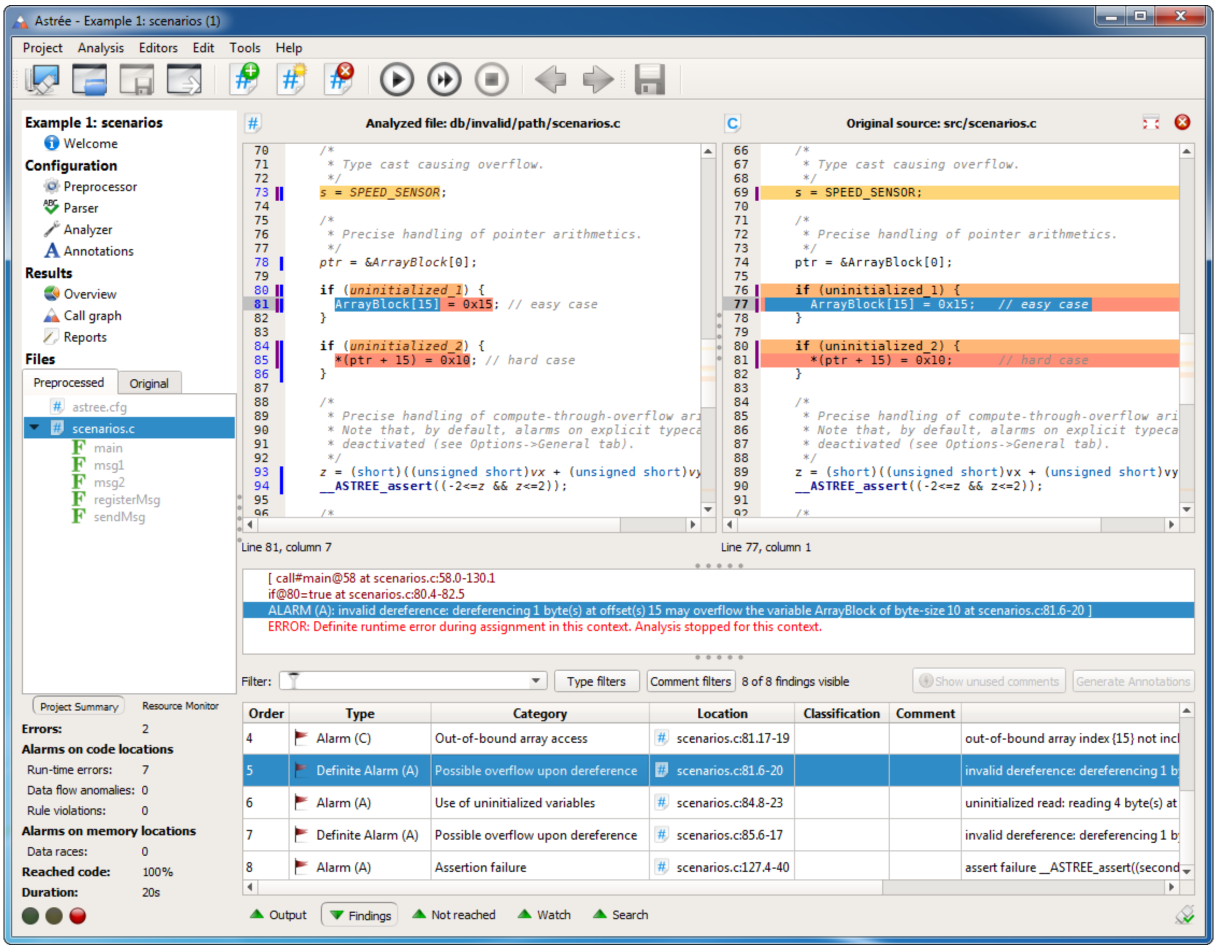
Lastly, Astrée can be used to check for functional program properties by a static assertion mechanism. If Astrée does not report the assertion to be violated, the asserted C expression has been proven correct.

Astrée will always stop with an error if indispensable data is missing or if source files cannot be correctly parsed and translated.

Handling the alarms

Each reported alarm indicates a potential runtime error, which can be interactively explored, commented on, or fixed right away in the built-in C source code editor. Possible false alarms can be marked as such using AAL annotations so that they no longer occur on subsequent analysis runs. Alternatively, you can tweak the analysis settings or increase the analysis precision for selected code parts. After that, you can run the analysis once again and examine the improved results.

These steps are repeated as needed until all alarms have been dealt with and no errors are reported anymore. At that point, the absence of errors in the code has been formally proven.



Errors: 2
Alarms on code locations
Run-time errors: 7
Data flow anomalies: 0
Rule violations: 0
Alarms on memory locations
Data races: 0
Reached code: 100%
Duration: 20s

Analyzed file: db/invalid/path/scenarios.c

```
    /*
     * Type cast causing overflow.
     */
    s = SPEED_SENSOR;

    /*
     * Precise handling of pointer arithmetics.
     */
    ptr = &ArrayBlock[0];

    if (uninitialized_1) {
      ArrayBlock[15] = 0x15; // easy case
    }

    if (uninitialized_2) {
      *(ptr + 15) = 0x10; // hard case
    }

    /*
     * Precise handling of compute-through-overflow ari
     * Note that, by default, alarms on explicit typeca
     * deactivated (see Options->General tab).
     */
    z = (short)((unsigned short)vx + (unsigned short)vy
    __ASTREE_assert((-2<=z && z<=2));

    /*
```

Line 81, column 7

Original source: src/scenarios.c

```
    /*
     * Type cast causing overflow.
     */
    s = SPEED_SENSOR;

    /*
     * Precise handling of pointer arithmetics.
     */
    ptr = &ArrayBlock[0];

    if (uninitialized_1) {
      ArrayBlock[15] = 0x15;   // easy case
    }

    if (uninitialized_2) {
      *(ptr + 15) = 0x10;      // hard case
    }

    /*
     * Precise handling of compute-through-overflow ari
     * Note that, by default, alarms on explicit typeca
     * deactivated (see Options->General tab).
     */
    z = (short)((unsigned short)vx + (unsigned short)vy
    __ASTREE_assert((-2<=z && z<=2));

    /*
```

Line 77, column 1

```
[ call#main@58 at scenarios.c:58.0-130.1
if@80=true at scenarios.c:80.4-82.5
ALARM (A): invalid dereference: dereferencing 1 byte(s) at offset(s) 15 may overflow the variable ArrayBlock of byte-size 10 at scenarios.c:81.6-20 ]
ERROR: Definite runtime error during assignment in this context. Analysis stopped for this context.
```

Filter: Type filters | Comment filters | 8 of 8 findings visible | Show unused comments | Generate Annotations

Order	Type	Category	Location	Classification	Comment	
4	Alarm (C)	Out-of-bound array access	scenarios.c:81.17-19			out-of-bound array index {15} not incl
5	Definite Alarm (A)	Possible overflow upon dereference	scenarios.c:81.6-20			invalid dereference: dereferencing 1 b
6	Alarm (A)	Use of uninitialized variables	scenarios.c:84.8-23			uninitialized read: reading 4 byte(s) at
7	Definite Alarm (A)	Possible overflow upon dereference	scenarios.c:85.6-17			invalid dereference: dereferencing 1 b
8	Alarm (A)	Assertion failure	scenarios.c:127.4-40			assert failure __ASTREE_assert((second

Figure 12: Alarm investigation using editor views

3.2.3. Tool Constraints

There are Windows and Linux versions of Astrée. The Windows version requires 64-bit Windows 7 SP1 or newer, and the Linux version 64-bit CentOS/RHEL 6 or compatible. There should be at least 4 GB of RAM (16 GB is recommended) and 4 GB of disk space.

Astrée actually consists of two parts:

1. The Astrée client, for setting up an analysis and viewing the results. The client offers both a GUI and a batch mode for easy automation and integration.
2. The analysis server, which carries out the actual analysis (or several analyses as separate processes).

Both parts may run together on the same machine. In production, however, the server typically runs on a powerful remote host, while clients are run by the individual developers and managers on their PCs or other devices.

Astrée can be obtained from AbsInt Angewandte Informatik GmbH (support@absint.com). AbsInt offers commercial licenses, including training, support, and maintenance. The licensing models are very flexible, ranging from single analysis servers with limited client connections to department or company licenses. The license file determines how many clients may connect to the server at the same time, and how many analysis processes can run there in parallel.

3.3. FZI: C-SAPP

3.3.1. Tool features and benefit

C-SAPP (C Static Analysis Preprocessor) is a research prototype, developed at FZI, to support analyses of Hardware-dependent Software (HdS) [1]. The central idea is to augment the input to static program analysis tools (e.g., Astrée) by auxiliary information in a way that specific properties of the underlying hardware (HW) platform are considered during runtime-defect analysis of the software (SW). The C-SAPP tool can be thought of as a sophisticated preprocessor for the hardware-dependent software code under analysis. Its main tasks include the identification of hardware specific code sections (e.g., inline assembler, MMIO, interrupts) and various transformation steps, augmenting those sections with in-source directives (assertions, assumptions) that can be processed by static analysis tools such as Astrée.

In cross-boundary HW/SW defect analysis a formal model of the design under verification – a common representation of HW and SW with formal semantics – is of paramount importance for co-verification across HW/SW boundaries where the intricate nature of HW/SW interaction constitutes an acute challenge. In critical embedded systems, HW/SW interfaces are often modelled as "volatile" variables and constraints on these variables. Modern "intelligent" HW components go beyond simple Port I/O and thus work directly on shared-memory, perform direct memory access (DMA), all asynchronously from the main processor. System side effects, caused by embedded assembly instructions, direct access to system memory and specific I/O-registers via Memory Mapped I/O (MMIO), make it impossible to truly verify the SW component without considering HW properties.

The C-SAPP tool addresses the co-verification challenge by a model-driven approach where HW properties and their interface are specified using a SW/HW Interface Model (SHIM) followed by a generation and annotation step that augments the source code under analysis and makes HW side-effects explicit during program analysis. The overall tool flow is shown in Figure 13. The main benefit of the tool lies in the support for design automation of safety critical, embedded systems in the context of program analysis. Specifications of hardware components (e.g., timers, DMA controllers, I2C controllers, etc.) and whole platforms can be reused when developing new SW applications. A concise model of the HW/SW interface and hardware properties (SHIM model) facilitates the static analysis of embedded software across multiple projects because the co-verification model can be generated using the C-SAPP tool. A SHIM model of a HW component has to be created once and can be reused in multiple scenarios to generate source code and annotations for various static analysis back ends (e.g., Astrée, QPR, Goblint, etc.).

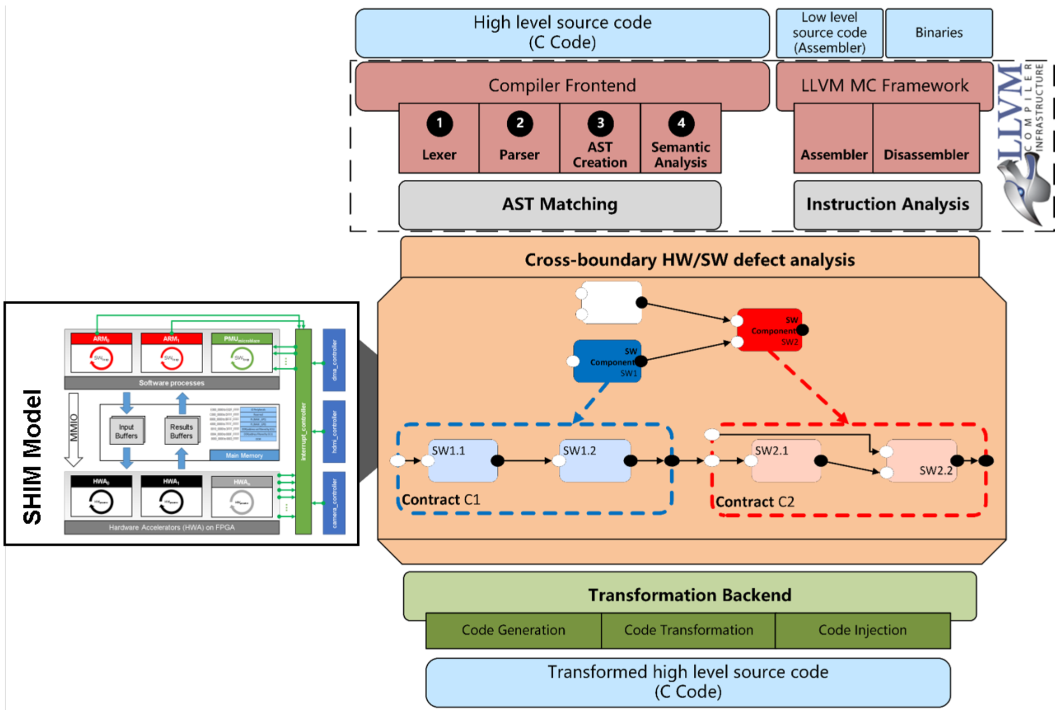


Figure 13: C Static Analysis Preprocessor (C-SAPP)

3.3.2. Input and Output formats

The C-SAPP tool is based on the LLVM/Clang LibTooling framework and is run from the command line with a small set of command-line flags.

The primary input of the tool is C code. A prerequisite for running the C-SAPP tool is the existence of a “Clang Compilation Database” [2]. The C-SAPP tool parses the source code and thus needs full information on how to parse a translation unit (C file) with all compiler flags, include paths and other options. The “Clang Compilation Database” stores this additional information in a JSON file that can be generated using build systems such as CMake.

The secondary input is the SHIM model that describes the hardware platform, the software is executed on. The SHIM model can be very detailed but also very abstract, modelling only specific properties of HW/SW interaction that are of interest during program analysis. All HW/SW interactions are specified in two steps (Figure 14).

The SW model is a list that contains all routines to access a specific HW component (SW-HW transaction) and is only mandatory if the source code for the device drivers is not included in the source code under analysis. Direct writes to HW registers using MMIO are SW-HW transactions and C-SAPP will identify all potential interactions via shared registers.

The HW model specifies the HW structure with all its registers and memories using IP-XACT or SystemRDL. The HW model can be created manually from specifications where all registers are described in detail (e.g., Xilinx VDMA Figure 15) or generated by design automation tools that generate IP-XACT files for the Intellectual Property (IP) components. Interrupt signals in the HW model are considered as HW-SW transactions and the corresponding Interrupt Service Routine (ISR) is part of the SW model.

From the SHIM model, a skeleton implementation of the hardware component is generated as C code, modelling the interaction of SW and HW via their shared register interface. The SHIM model is an Ecore model and can be created in a separate modelling environment based on Eclipse.

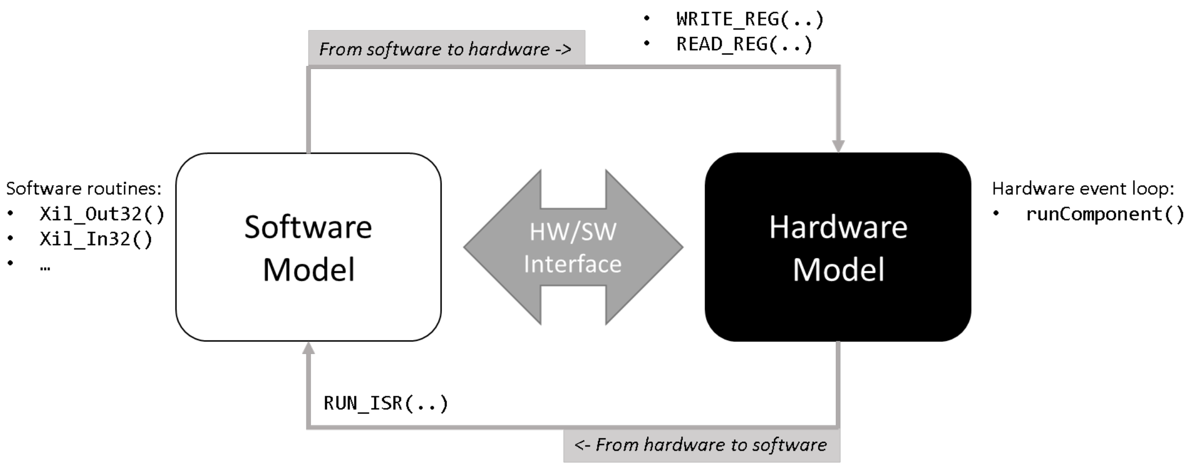


Figure 14: HW/SW interaction

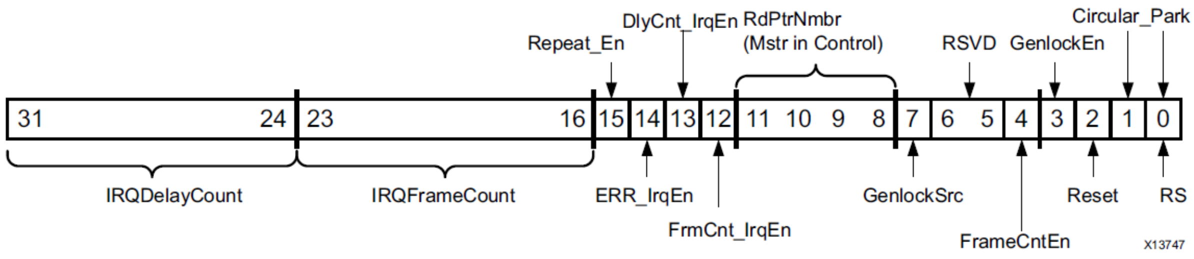


Figure 15: MM2S VDMA Control Register specification [3]

3.3.3. Tool constraints

The C-SAPP tool is an early research prototype that is far away from an “All-in-One Application” for HW/SW cross-boundary program analysis. Currently, the tool can only generate the structure for the HW/SW interface and a skeleton implementation of hardware interaction with software. Behavior models for the SW-HW transactions need to be added manually as a set of transformation rules (C code templates) during the code generation steps.
The overall tool flow is still immature and fragmented because of the missing integration of all steps into a common environment and thus ongoing work focuses on the integration of all modelling and code generation task into an Eclipse toolchain.
Even though the C-SAPP tool is a standalone tool, only the generation of Astrée specific code annotations has been tested so far. Other back ends such as QPR or Goblint can be easily supported in the future by adjusting the transformation rules in the C-SAPP “Transformation Backend”.

3.4. Mälardalen University/SWEET

3.4.1. Tool features and benefit

SWEET (SWEdish Execution Time tool) is a static WCET analysis research prototype tool. Its main use is to calculate program flow constraints ("flow facts"), like loop iteration bounds and infeasible path constraints, The main use of SWEET is to calculate flow facts automatically, and export these to some tool that can use this information to perform a safe and precise WCET estimation. SWEET can also calculate coarse BCET/WCET estimates by itself using simple timing models. The estimates are not safe in general, but can be used to give the developer early feedback about expected timing bounds by a source code-level analysis. In addition SWEET can perform a number of other analyses, such as a conventional value analysis using abstract interpretation, data flow analysis (reaching definitions), and static backwards program slicing.

3.4.2. Input and Output formats

SWEET analyses an intermediate format called ALF [1,2]. This format is native to SWEET, and is designed to represent primarily code on fairly low level (like C, or binaries). Analysis of other formats than ALF is done through a translation to ALF. Analysis results are presented on ALF level, but can be mapped back to the original code provided that the translator provides the necessary information about the mapping from original code to ALF. Several translators to ALF exist: the most prominent, "AlfBackend", translates C to ALF using the clang/LLVM compiler framework. AlfBackend also computes the information necessary to map analysis results back to the original C code. Besides AlfBackend, experimental translators from PowerPC and NEC V850 binaries to ALF exist. The structure of ALF is shown in the figure below.

SWEET is run from the command line. There is a rich set of options to direct the analysis: these are set through command-line flags, or through provided files. Analysis results are also communicated via written files.

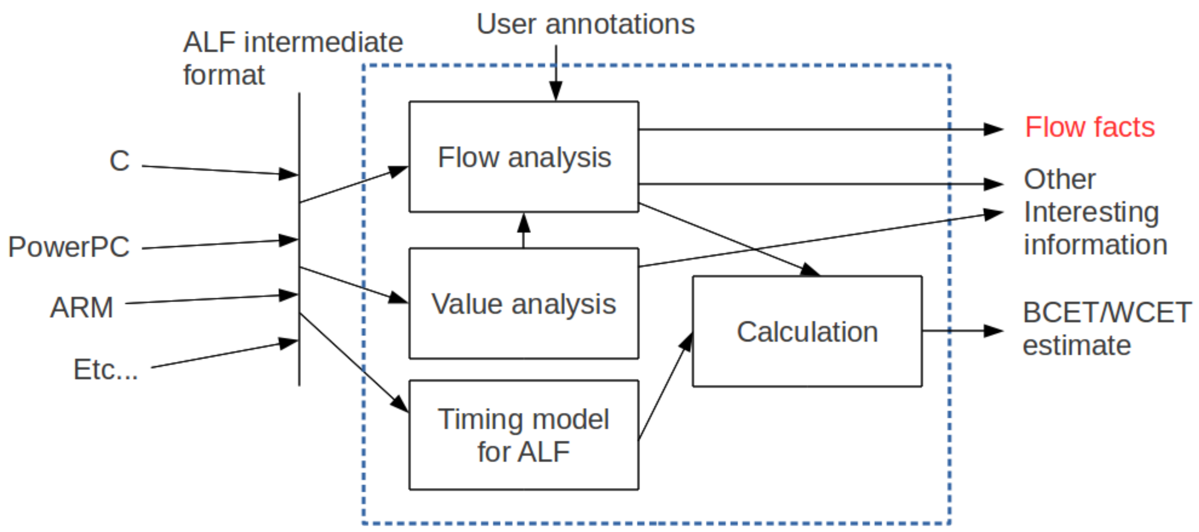
SWEET has a simple native format to specify intervals constraining the values of program variables in different program points. This information can be used by SWEET to calculate tighter flow facts. Variables can also be marked as volatile.

SWEET has an expressive native format for the generated flow facts. This includes context information in the form of call strings, as the calculated flow facts can be context-sensitive. SWEET can also export flow facts for C code to the AIS format for flow facts that is used by aiT from AbsInt. In addition, SWEET has simple formats to export the results of its value analysis, and program slices.

Further information about the various formats of SWEET can be found at http://www.mrtc.mdh.se/projects/wcet/sweet/manual/html/.

3.4.3. Tool constraints

There are some limitations on which codes SWEET can handle. Recursion, and dynamic memory handling, is not supported. The AlfBackend C to ALF translator also has some limitations – for instance, it does not handle polyadic functions (like printf). As AlfBackend is build within the LLVM framework it uses the clang C parser, and is thus also subject to any limitations in clang.
More information about SWEET, including an extensive user manual, can be found at http://www.mrtc.mdh.se/projects/wcet/sweet/. The tool is open source under an allowing BSD style license. Access to SWEET and AlfBackend can be requested by sending a mail to the mailing list wcet@list.mdh.se: you will then get read access to the svn repository where the tools live. Instructions for how to build the tools on various platforms can be found in the user manual.


User annotations
ALF intermediate format
C
PowerPC
ARM
Etc...
Flow analysis
Value analysis
Timing model for ALF
Calculation
Flow facts
Other Interesting information
BCET/WCET estimate

3.5. KIT – LLBMC / QPR-Verify

3.5.1. Tool features and benefit

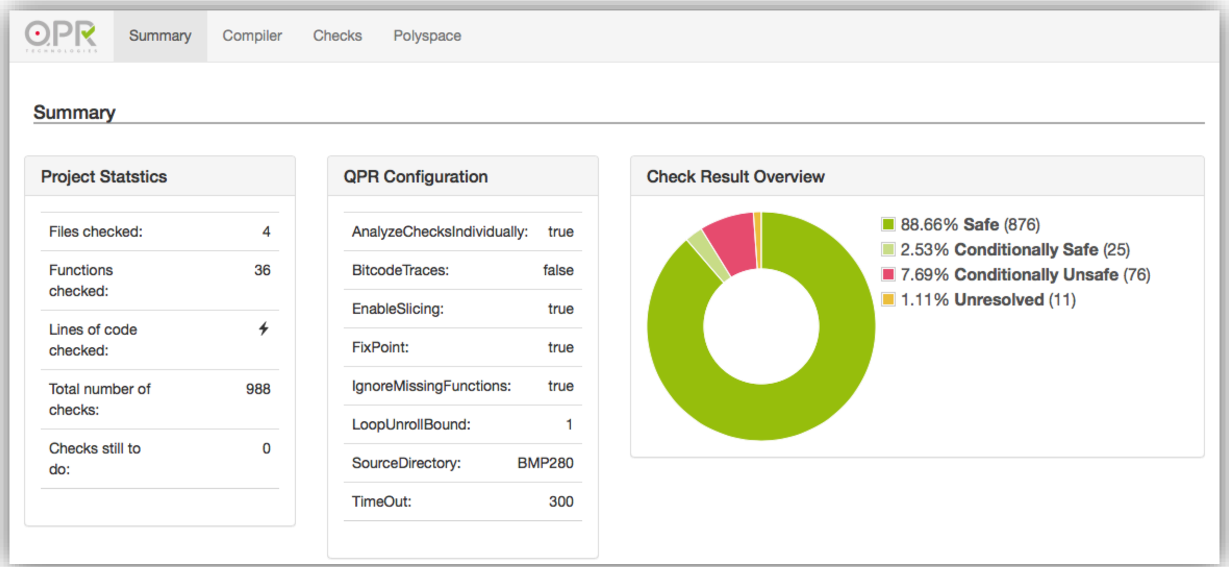
QPR-Verify is a tool for analyzing software written in the programming language C for runtime errors such as integer overflows, division by zero, or other kind of undefined behavior. It employs a technique called bounded model checking, which is realized in QPR-Verify's core solver LLBMC. Bounded model checking, by encoding program properties on the bit-level, achieves extremely high precision. Moreover, it is able to produce detailed traces for program errors. On the other hand, its scalability to large programs is limited.

The kind of errors that QPR-Verify can detect encompass:

- Arithmetic overflows (for both signed and unsigned integer variables)
- Integer division by zero
- Overflows in type casts
- Undefined shift operations (negative arguments, too large arguments, overflows)
- Array index out of bounds
- Uninitialized local variables
- Contract violations (with respect to a user-provided data range specification)
- Failed user-provided assertions

QPR-Verify can provide

- a summary of an analysis run, indicating program locations that are safe or unsafe
- a list of compiler warnings for the analyzed program
- a list of checks, which QPR-Verify performed, each with details about the kind of check, the result, and possibly further annotations about assumptions that the tool made
- a trace view for each detected error



Summary

Project Statstics

Files checked:	4
Functions checked:	36
Lines of code checked:	⚡
Total number of checks:	988
Checks still to do:	0

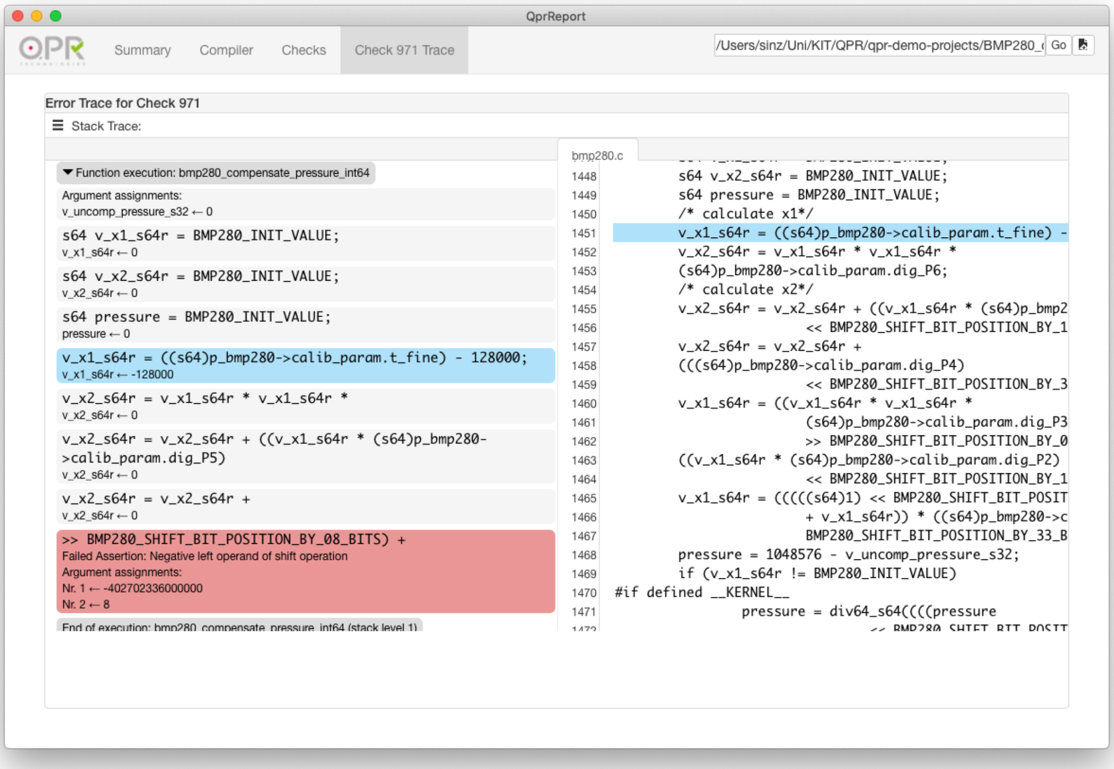
QPR Configuration

AnalyzeChecksIndividually:	true
BitcodeTraces:	false
EnableSlicing:	true
FixPoint:	true
IgnoreMissingFunctions:	true
LoopUnrollBound:	1
SourceDirectory:	BMP280
TimeOut:	300

Check Result Overview

- 🟩 88.66% **Safe** (876)
- 🟩 2.53% **Conditionally Safe** (25)
- 🟥 7.69% **Conditionally Unsafe** (76)
- 🟨 1.11% **Unresolved** (11)

The trace view shows a step-by-step execution of the program together with variable value changes up to the error location.



Error Trace for Check 971

Stack Trace:

```
▼ Function execution: bmp280_compensate_pressure_int64
Argument assignments:
v_uncomp_pressure_s32 ← 0
s64 v_x1_s64r = BMP280_INIT_VALUE;
v_x1_s64r ← 0
s64 v_x2_s64r = BMP280_INIT_VALUE;
v_x2_s64r ← 0
s64 pressure = BMP280_INIT_VALUE;
pressure ← 0
v_x1_s64r = ((s64)p_bmp280->calib_param.t_fine) - 128000;
v_x1_s64r ← -128000
v_x2_s64r = v_x1_s64r * v_x1_s64r *
v_x2_s64r ← 0
v_x2_s64r = v_x2_s64r + ((v_x1_s64r * (s64)p_bmp280-
>calib_param.dig_P5)
v_x2_s64r ← 0
v_x2_s64r = v_x2_s64r +
v_x2_s64r ← 0
>> BMP280_SHIFT_BIT_POSITION_BY_08_BITS) +
Failed Assertion: Negative left operand of shift operation
Argument assignments:
Nr. 1 ← -402702336000000
Nr. 2 ← 8
End of execution: bmp280_compensate_pressure_int64 (stack level 1)
```

bmp280.c

```
        s64 v_x2_s64r = BMP280_INIT_VALUE;
        s64 pressure = BMP280_INIT_VALUE;
        /* calculate x1*/
        v_x1_s64r = ((s64)p_bmp280->calib_param.t_fine) -
        v_x2_s64r = v_x1_s64r * v_x1_s64r *
        (s64)p_bmp280->calib_param.dig_P6;
        /* calculate x2*/
        v_x2_s64r = v_x2_s64r + ((v_x1_s64r * (s64)p_bmp2
                        << BMP280_SHIFT_BIT_POSITION_BY_1
        v_x2_s64r = v_x2_s64r +
        (((s64)p_bmp280->calib_param.dig_P4)
                        << BMP280_SHIFT_BIT_POSITION_BY_3
        v_x1_s64r = ((v_x1_s64r * v_x1_s64r *
                        (s64)p_bmp280->calib_param.dig_P3
                        >> BMP280_SHIFT_BIT_POSITION_BY_0
        ((v_x1_s64r * (s64)p_bmp280->calib_param.dig_P2)
                        << BMP280_SHIFT_BIT_POSITION_BY_1
        v_x1_s64r = (((((s64)1) << BMP280_SHIFT_BIT_POSIT
                        + v_x1_s64r)) * ((s64)p_bmp280->c
                        BMP280_SHIFT_BIT_POSITION_BY_33_B
        pressure = 1048576 - v_uncomp_pressure_s32;
        if (v_x1_s64r != BMP280_INIT_VALUE)
#if defined __KERNEL__
                pressure = div64_s64((((pressure
```

Due to the employed technology of bounded model checking, QPR-Verify can proof correctness of program constructs. However, the analysis is only up to a maximal user-provided loop bound. Errors that occur only, if more iterations are needed, will not be reported (the tool notifies the user, however, if this case occurs).

To increase scalability of the model checking approach, QPR-Verify has two operation modes: a global analysis (which is less scalable) and a local analysis mode, where only a fraction of the program is analyzed. These fragments consist of a sub-graph of the call-graph, i.e., they cover more than a purely intraprocedural analysis.

3.5.2. Input and Output formats

The tool is operated from the command line, and possesses an additional graphical viewer for analysis results. The command line interface gives access to the following commands:

```
Usage:
  qpr <command> <command-specific-options>

Commands:
  add-checks                      Add checks to a job
  add-compiler-option                Add compiler options to compile units
  add-source-files                 Add source files to the project
  analyze-check-in-context            Analyze a check in a specific context
  analyze-globally                 Analyze all unproven checks globally
  build-compile-commands              Quickly assemble a set of compile command
  check-all-functions-locally          Check all functions locally
  check-function-locally             Check a function locally
  check-function-locally-to-fixpoint Check a function locally (to fixpoint)
  clean-project                   Clean files from the project directory
  clear-collections                 Clear resource collections (e.g. checks, locations, ...)
  clone-job                      Clone a job for further analysis refinement and set all checks to status 'todo'
  compile                        Compile and link together all files in the compilation DB
```

configure-all-jobs	Set an option for all jobs that were not yet executed
configure-job	Set an option in a job specific configuration
configure-project	Configure analysis and project.
copy-source-files	Copy source and header file contents
copy-static-files	Copy static files into the project directory
create-job	Create a new job
find-source-files	Find all source files in a given directory
help	Print a help text about available commands
import-drs	Import a DRS file and assert the DRS assumptions.
import-kw-inject	Import a kwinject.out file
import-polyspace-project	Import a polyspace project
integrate-jobs	Integrate results from multiple jobs into one
list-callers	List all callers of a given function
list-functions	List all functions containing checks
merge-job	Merge a job's result back into the main list of checks
murphy	Analyse code for properties to be checked
optimize-bitcode	Runs a set of optimization passes on the bitcode.
print	Print a resource as xml
print-ast	Print the abstract syntax tree
print-bitcode-info	Print info about bitcode relevant for analysis
print-call-graph	Print the call graph
print-slice	Print the slice of the bitcode relevant for a given check
process	Run a command in a child process
remap-compile-commands	Remap paths in compile commands
run	Run a local analysis on all functions
run-job	Run a model checker job
run-script	Run a qpr script file
version	Print out the version information

Input to the `qpr` command is a set of C source code and header files, specified in a compilation database together with compiler definitions and possibly macros. Additionally, different aspects of an analysis run can be configured using the command `configure-project`.

The output of the tool consists of a set of XML files, which contain the analysis result. These file can be viewed with the QPR-Report GUI tool.

Experimental support for the ASSUME Static Analysis Exchange Format is also integrated.

3.5.3. Tool constraints

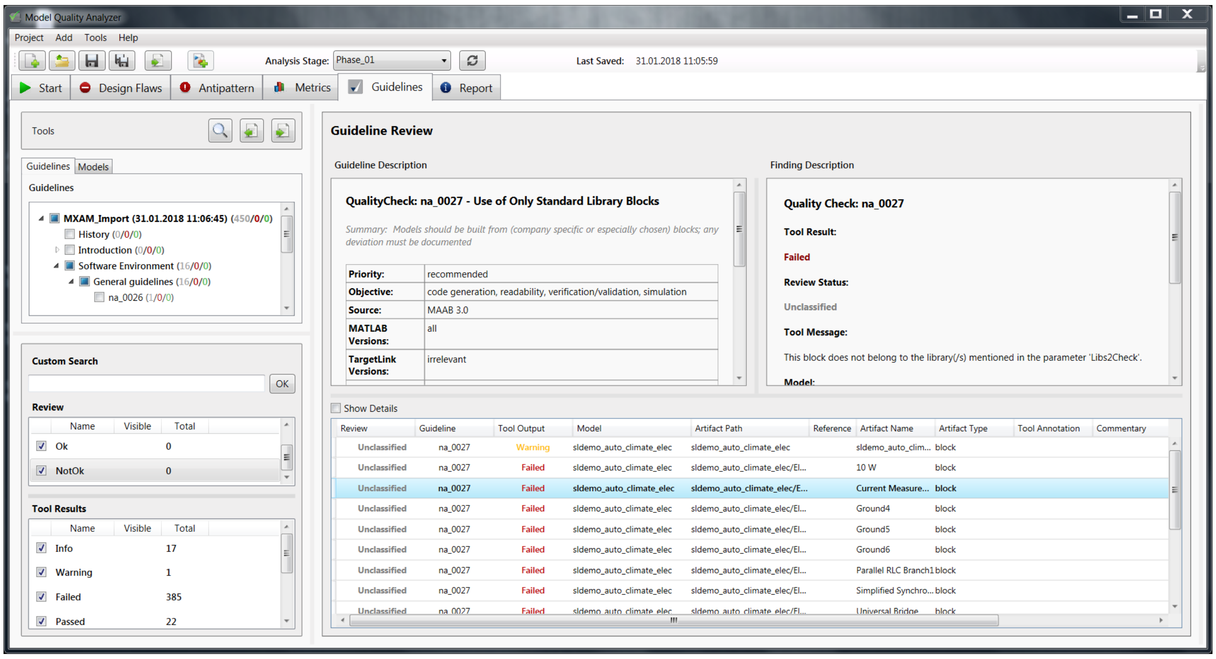
- Scalability, although considerably improved during the run of the ASSUME project, is still limited to programs with a size in the order of 100.000 lines of source code.
- Seldom occurring program constructs might not be supported.
- The source code to be analyzed must be compilable with the `clang` compiler.

3.6. Assystem: MQAnalyzer

3.6.1. Tool features and benefit

The MQAnalyzer is a prototypic implementation of the methodology developed by Assystem. It serves as a comprehensive aggregating component with smart mapping functionality for other software quality analysis tools and also supports certain analyses by itself. Aggregating, merging and mapping of various analysis results, as well as assisted reviewing and automated exporting of merged and revised analysis data is seen as its core functionality. The implementation is designed to assist software developers during the whole software development life cycle where model-based development is in use.

Purpose of this aspect of the methodology is to gather all relevant information from different sources and propagate them through the aggregation and review process of issue assessment and classification. Therefore, the tool's aggregation interfaces are designed to be extendable for all relevant Model Quality Analysis tools on the market. The exported results are standardized and comparable and allow a flexible integration into quality assurance processes.



Analysis Stage: Phase_01

Last Saved: 31.01.2018 11:05:59

Guidelines

- MXAM_Import (31.01.2018 11:06:45) (450/0/0)
  - History (0/0/0)
  - Introduction (0/0/0)
  - Software Environment (16/0/0)
    - General guidelines (16/0/0)
      - na_0026 (1/0/0)

Custom Search

Review

	Name	Visible	Total
✓	Ok		0
✓	NotOk		0

Tool Results

	Name	Visible	Total
✓	Info		17
✓	Warning		1
✓	Failed		385
✓	Passed		22

Guideline Review

Guideline Description

QualityCheck: na_0027 - Use of Only Standard Library Blocks

Summary: Models should be built from (company specific or especially chosen) blocks; any deviation must be documented

Priority:	recommended
Objective:	code generation, readability, verification/validation, simulation
Source:	MAAB 3.0
MATLAB Versions:	all
TargetLink Versions:	irrelevant

Finding Description

Quality Check: na_0027

Tool Result:

Failed

Review Status:

Unclassified

Tool Message:

This block does not belong to the library(/s) mentioned in the parameter 'Libs2Check'.

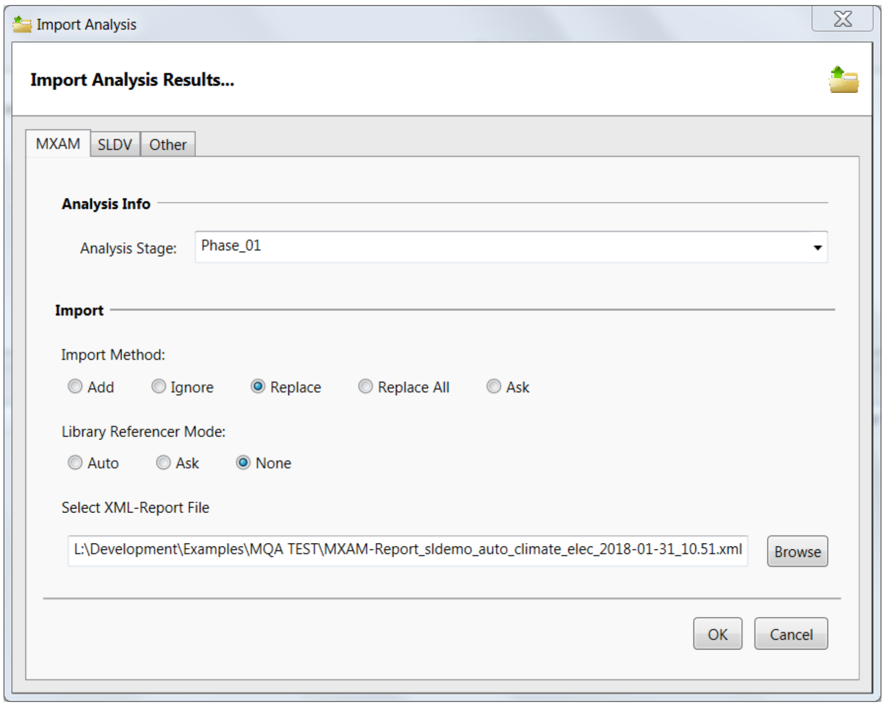
Model:

Review	Guideline	Tool Output	Model	Artifact Path	Reference	Artifact Name	Artifact Type	Tool Annotation	Commentary
Unclassified	na_0027	Warning	sldemo_auto_climate_elec	sldemo_auto_climate_elec		sldemo_auto_clim...	block		
Unclassified	na_0027	Failed	sldemo_auto_climate_elec	sldemo_auto_climate_elec/El...		10 W	block		
Unclassified	na_0027	Failed	sldemo_auto_climate_elec	sldemo_auto_climate_elec/E...		Current Measure...	block		
Unclassified	na_0027	Failed	sldemo_auto_climate_elec	sldemo_auto_climate_elec/El...		Ground4	block		
Unclassified	na_0027	Failed	sldemo_auto_climate_elec	sldemo_auto_climate_elec/El...		Ground5	block		
Unclassified	na_0027	Failed	sldemo_auto_climate_elec	sldemo_auto_climate_elec/El...		Ground6	block		
Unclassified	na_0027	Failed	sldemo_auto_climate_elec	sldemo_auto_climate_elec/El...		Parallel RLC Branch1	block		
Unclassified	na_0027	Failed	sldemo_auto_climate_elec	sldemo_auto_climate_elec/El...		Simplified Synchro...	block		
Unclassified	na_0027	Failed	sldemo_auto_climate_elec	sldemo_auto_climate_elec/El...		Universal Bridge	block		

Figure 16: Assisted review of analysis results in the MQAnalyzer (Example: Guidelines)

The prototype was designed and developed in the context of ASSUME. So far, interfaces to the MES tools MXAM and MXRAY as well as several tools by MathWorks have been integrated. The process of data aggregation, comprehensive review and classification and automated export can already be conducted.

3.6.2. Input and Output formats



Analysis Stage: Phase_01

Import Method: Add Ignore [✓ Replace] Replace All Ask

Library Referencer Mode: Auto Ask [✓ None]

Select XML-Report File: L:\Development\Examples\MQA TEST\MXAM-Report_sldemo_auto_climate_elec_2018-01-31_10.51.xml

Figure 17: Importing the results of a guideline analysis by MXAM

As mentioned above, the interfaces for the output of the MES tools, as well as a direct interface to Matlab/Simulink, have been implemented. The interface to Matlab not only provides a way to import the results of the MathWorks analysis tools, it can also be used to integrate custom checks and scripts implemented in Matlab to be used in the model analysis process.

After concluding the assisted review process, the MQAnalyzer will provide an automatically generated report containing all found flaws and defects. The report serves two purposes at the same time. On one side, it provides an overview about the found results supported by graphics and charts. On the other side, it gives a list of all discovered issues and flaws to assist the developer when addressing the issues. The report also contains mechanisms to assist the tracking of the contained issues.

3.6.3. Tool constraints

ASSUME partners are free to receive the prototype in binary form. It cannot be downloaded, as it is currently not commercially available.

Some of the functionality, especially concerning the extension with respect to other tools and special analysis methods are still under development and only available in a preliminary form.
For more information about the prototypic implementation and the available services provided, please contact softwarequality@assystemtechnologies.com.

References

[1] Wolfgang Ecker, Wolfang Müller, Rainer Dörmer. “Hardware-dependent Software.” Springer Netherlands, 2009

[2] https://clang.llvm.org/docs/JSONCompilationDatabase.html

[3] https://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v6_3/pg020_axi_vdma.pdf

[4] Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg. ALF - A Language for WCET Flow Analysis. In *Proc. 9th International Workshop on Worst-Case Execution Time Analysis (WCET09)*, July 2009. http://www.es.mdh.se/publications/1420-

[5] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. *ALF (ARTIST2 Language for Flow Analysis) Specification*. Technical report, MDH, Oct. 2011. http://www.es.mdh.se/publications/1138-

[6] Florian Merz, Stephan Falke, Carsten Sinz. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. *Proc. of the Intl. Conf. on Verified Software: Tools, Theories, Experiments (VSTTE)*, Springer, 2012.