Abstract Interpretation in Industry – Experience and Lessons Learned

Daniel Kästner[1], Reinhard Wilhelm[2], and Christian Ferdinand[1]

[1] AbsInt GmbH, Science Park 1, 66123 Saarbrücken, Germany
kaestner@absint.com
[2] Saarland University, Stuhlsatzenhausweg 69, 66123 Saarbrücken, Germany

Abstract. In this article we will give an overview of the development and commercialization of two industry-strength Abstract Interpretation-based static analyzers, aiT WCET Analyzer and Astrée. We focus on development steps, adaptations to meet industry requirements and discuss criteria for a successful transfer of formal verification methods to industrial usage.

Keywords: abstract interpretation, WCET analysis, runtime error analysis, functional safety, cybersecurity.

1 Introduction

Abstract interpretation is a formal method for sound semantics-based static program analysis [8]. It supports formal correctness proofs: it can be proved that an analysis will terminate and that it is sound in the sense that it computes an over-approximation of the concrete program semantics. Abstract interpretation-based static analyzers provide full control and data coverage and allow conclusions to be drawn that are valid for all program runs with all inputs.

As of today, abstract interpretation-based static analyzers are most widely used to determine non-functional software quality properties [23, 22]. On the one hand that includes source code properties, such as compliance to coding guidelines, compliance to software architectural requirements, as well as absence of runtime errors and data races [34]. On the other hand also low-level code properties are covered, such as absence of stack overflows and violation of timing constraints [24, 25].

Violations of non-functional software quality requirements often either directly represent safety hazards and cybersecurity vulnerabilities in safety- or security-relevant code, or they can indirectly trigger them. Corresponding verification obligations can be found in all current safety and security norms, such as DO-178C [48], IEC-61508 [15], ISO-26262 [17], and EN-50128 [6].

Many formal verification tools, including abstract interpretation-based static analyzers, originate from academic research projects. However, the transition from academia into industry is far from straightforward. In this article we will give an overview of our experience in development and commercialization of two

industry-strength sound analyzers, aiT WCET analyzer and Astrée. We will discuss the lessons learned, and present recommendations to improve dissemination and acceptance in industrial practice.

2 Sound Worst-Case Execution Time Analysis

Time-critical embedded systems have deadlines derived from the physical environment. They need assurance that their execution time does not exceed these deadlines. Essential input to a response-time analysis are the safe upper bounds of all execution times of tasks to be executed on the same execution platform. These are commonly called *Worst-case Execution times, WCET*. The WCET-analysis problem had a solution for architectures with constant execution times for instructions, so-called *Timing Schemata* [54]. These described how WCETs could be computed by structural induction over programs. However, in the 1990s industry started using microprocessors employing performance-enhancing architectural components and features such as caches, pipelines, and speculation. These made methods based on timing schemata obsolete. The execution-time of an instruction now depended on the execution state in which the instruction were executed. The variability of execution times grew with several architectural parameters, e.g. the cache-miss penalty and the costs for pipeline stalls and for control-flow mis-predictions.

2.1 Our View of and our Solution to the WCET-Analysis Problem

We developed the following view of the WCET-analysis problem for architectures with state-dependent execution times: Any architectural effect that lets an instruction execute longer than its fastest execution time is a *Timing Accident*. Some of such timing accidents are cache misses, pipeline stalls, bus-access conflicts, and branch mis-predictions. Each such timing accident has to be paid for, in terms of execution-time cycles, by an associated *Timing Penalty*. The size of a timing penalty can be constant, but may also depend on the execution state. We consider the property that an instruction in the program will not cause a particular timing accident as a safety property. The occurrence of a timing accident thus violates a corresponding safety property.

The essence of our WCET-analysis method then consists in the attempt to verify for each instruction in the program as many safety properties as possible, namely that some of the potential timing accidents will never happen. The proof of such safety properties reduces the worst-case execution-time bound for the instruction by the penalties for the excluded timing accidents. This so-called *Microarchitectural Analysis*, embedded within a complex tool architecture, is the central innovation that made our WCET analysis work and scale. We use Abstract Interpretation to compute certain invariants at each program point, namely an upper approximation of the set of execution states that are possible when execution reaches this program point and then derive safety properties, that certain timing accidents will not happen, from these invariants.

2.2 The Development of our WCET-Analysis Technique

We started with a *classifying cache analysis* [1, 12], an analysis that attempts to classify memory accesses in programs as either always hitting or always missing the caches, i.e. instruction and data caches. Our *Must* analysis, used to identify cache hits, computes an under-approximation of the set of cache states that may occur when execution reaches a program point. Our *May* analysis determines an over-approximation of this set of cache states. Both can be represented by compact, efficiently updatable *abstract cache states*. At the start of the development, the caches we, and everybody else, considered used LRU replacement. This made our life easy, but application to real-life processors difficult since the hardware logic for implementing LRU replacement is expensive, and therefore LRU replacement is rarely used in real-life processors.

Involved in the European project Daedalus with Airbus we were confronted with two processors using very different cache-replacement strategies. The first processor, flying the Airbus A340 plane, was a Motorola Coldfire processor which used a cheap emulation of a random-replacement cache. The second projected to fly the A380 plane was a Motorola PowerPC 755. It used a Pseudo-LRU replacement strategy. We noticed that our cache analysis for the Coldfire processor could only track the last loads into the cache, and that our cache analysis for the PowerPC 755 could only track 4 out of the 8 ways in each cache set. This inspired us to very fruitful research about *Timing Predictability* [60] and in particular to the first formal notion of timing predictability, namely that for caches [50].

Next Stephan Thesing developed our pipeline analysis [39]. Unfortunately, pipelines in real-life processors do not admit compact abstract pipeline states. Therefore, expensive powerset domains are used. The pipeline analysis turned out to be the most expensive part of the WCET analysis. A basic block could easily generate a million pipeline states and correspondingly many transitions for analysis. There was a tempting idea to follow only local worst-case transitions and ignore all others. However, real-life processors exhibit *Timing Anomalies* [51]. These entail that a local non-worst-case may contribute to a global worst case.

In the Daedalus project, Airbus also asked for a modeling of their system controller. So far, all WCET research had concentrated on processors. However, a system controller contributes heavily to overall system timing and therefore needs an accurate model and precise analysis [59].

The Micro-architectural analysis was applied to basic blocks, i.e. maximally long straight-line code sequences that can only be entered at the beginning and only be left at the end. The control flow, which had been extracted from the binary executable [57], was translated into an Integer Linear Program (ILP) [58]. The solution of this ILP presented a longest path through the program and the associated execution time. This approach, termed *Implicit Path Enumeration Technique (IPET)*, had been adopted from [40].

At EMSOFT 2001 we presented our breakthrough paper [11]. In summary, a generic tool architecture has emerged which consists of the following stages:

Decoding: The instruction decoder identifies the machine instructions and reconstructs the call and control-flow graph.

Value analysis: Value analysis aims at statically determining enclosing intervals for the contents of the registers and memory cells at each program point and for each execution context. The results of the value analysis are used to predict the addresses of data accesses, the targets of computed calls and branches, and to find infeasible paths.

Micro-architectural analysis: The execution of a program is statically simulated by feeding instruction sequences from the control-flow graph to a micro-architectural timing model which is centered around the cache and pipeline architecture. It computes the system state changes induced by the instruction sequence at cycle granularity and keeps track of the elapsing clock cycles.

Path analysis: Based on the results of the combined cache/pipeline analysis the worst-case path of the analyzed code is computed with respect to the execution timing. The execution time of the computed worst-case path is the worst-case execution time for the task.

We had shown that our sound WCET-analysis method not only solved the single-core WCET-analysis problem, but was even more accurate than the unsound, measurement-based method Airbus had previously used. This meant that their worst-case execution times they had presented in certification had been reliable. Consequently we collaborated with Airbus to satisfy their needs for a sound, industrially viable WCET analysis.

2.3 Improvements

Although the results of our our analysis were already quite accurate, overestimating the ever observed worst-case execution times by roughly 25%, Airbus wanted more accurate results. Also the integration into industrial development processes needed consideration and some effort.

Increasing Precision Programs are known to spend most of their time in (recursive) procedures and in loops. The IPET approach using worst-case execution times of basic blocks as input was theoretically pleasing, but lost too much accuracy at the border between basic block and between loop iterations. Controlled loop unrolling increased the accuracy by the necessary extent. However, until today we confuse the competition by using the IPET approach in our explanations.

Often, the software developers knew what they were doing, i.e., they knew properties of their software that influenced execution time, but which were not explicit in the software. Our tool offered to be instructed by adding annotations to the software. Some annotations were even absolutely necessary, like loop and recursion bounds if those could not be automatically derived by our *Value Analysis*, essentially an interval analysis [9], modified to work on binary programs. We will later see that annotations could be automatically inserted if the WCET-analysis tool had been integrated with a model-based design tool.

Integration with Model-Based Design and Schedulability Tools Much of the safety-critical embedded software is developed using *Model-based Design (MBD)* tools. These automatically generate code from models specified by the software developer. When our WCET tool aiT is integrated with such a MBD tool, model information can be automatically inserted as annotations. Also approximate timing information can be provided on the model level to the developer by back annotation during the development process.

The determined WCETs are typically input into a schedulability analysis. Consequently, aiT has been integrated with several such tools.

2.4 Tool Qualification

Whenever the output of a tool is either part of a safety-critical system to be certified or the tool output is used to eliminate or reduce any development or verification effort for such a system, that tool needs to qualified [22]. Safety norms like DO-178C and ISO 26262 impose binding regulations for tool qualification; they mandate to demonstrate that the tool works correctly in the operational context of its users and/or that the tool is developed in accordance to a safety standard. To address this, a Qualification Support Kit has been developed, which consists of several parts.

The Tool Operational Requirements (TOR) document lists the tool functions and technical features which are stated as low-level requirements to the tool behavior under normal operating conditions. Additionally, the TOR describes the tool operational context and conditions in which the tool computes valid results. A second document (Tool Operational Verification and Validation Cases and Procedures, TOVVCP) defines a set of test cases demonstrating the correct functioning of all specified requirements from the TOR. Test case definitions include the overall test setup as well as a detailed structural and functional description of each test case. The test part contains an extensible set of test cases with a scripting system to automatically execute them and generate reports about the results. These tests also include model validation tests, in fact, a significant part of the development effort for aiT is to validate the abstract hardware model; [25] gives an overview.

In addition, the QSK provides a set of documents that give details about the AbsInt tool development and verification processes and demonstrate their suitability for safety-critical software.

2.5 Impact in Industry and Academia

A painful insight was that hardly any two WCET customers of AbsInt used the same hardware configuration in his systems. The costs for an instantiation of our WCET-analysis technology for a new processor can take quite an effort, making the resulting tool by necessity quite expensive. Still, aiT has been successfully employed in industry and is available for a variety of microprocessors ranging from simple processors like ARM7 to complex superscalar processors with timing anomalies and domino effects like Freescale MPC755, or MPC7448,

and multi-core processors like Infineon AURIX TC27x. Our development of a sound method that actually solved a real problem of real industry was considered a major success story for the often disputed formal-methods domain. AbsInt became the favorite partner for the industrialization of academic prototypes. First, Patrick Cousot and his team offered their prototype of Astrée, which in cooperation with some of the developers has been largely extended by AbsInt – more about this in Sec. 3. Then, we entered a cooperation with Xavier Leroy on the result of his much acclaimed research project, CompCert, the first formally verified optimizing C compiler [30, 29]. The CompCert front-end and back-end compilation passes, and their compositions, are all formally proved to be free of miscompilation errors. The property that is formally verified, using machine-assisted mathematical proofs, is *semantic preservation* between the input code and output code of every pass. Hence, the executable code CompCert produces is proved to behave exactly as specified by the formal semantics of the source C program. Both Astrée and CompCert are now available as AbsInt products.

2.6 Application to Non-Timing-Predictable Architectures

Multi-core processors with shared resources pose a severe problem for sound and precise WCET analysis. To interconnect the several cores, buses, meshes, crossbars, and also dynamically routed communication structures are used. In that case, the interference delays due to conflicting, simultaneous accesses to shared resources (e.g. main memory) can cause significant imprecision. Multi-core processors which can be configured in a timing-predictable way to avoid or bound inter-core interferences are amenable to static WCET analysis [27, 63, 64]. Examples are the Infineon AURIX TC275 [16], or the Freescale MPC 5777.

The Freescale P4080 [13] is one example of a multi-core platform where the interference delays have a huge impact on the memory access latencies and cannot be satisfactorily predicted by purely static techniques. In addition, no public documentation of the interconnect is available. Nowotsch et al. [46] measured maximal write latencies of 39 cycles when only one core was active, and maximal write latencies of 1007 cycles when all eight cores were running. This is more than 25 times longer than the observed single-core worst case. Like measuring task execution on one core with interference generators running on all other cores, statically computed WCET bounds will significantly overestimate the timing delays of the system in the intended final configuration.

In some cases, robust partitioning [64] can be achieved with approaches approaches like [53] or [46]. For systems which do not implement such rigorous software architectures or where the information needed to develop a static timing model is not available, hybrid WCET approaches are the only solution.

For hybrid WCET analysis, the same generic tool architecture as described in Sec. 2.2 can be used, as done in the tool TimeWeaver [37]. It performs Abstract Interpretation-based context-sensitive path and value analysis analysis, but replaces the Microarchitectural Analysis stage by non-intrusive real-time instruction-level tracing to provide worst-case execution time estimates. The trace information covers interference effects, e.g., by accesses to shared resources

from different cores, without being distorted by probe effects since no instrumentation code is needed. The computed estimates are upper bounds with respect to the given input traces, i.e., TimeWeaver derives an overall upper timing bound from the execution time observed in the given traces. This approach is compliant to the recommendations of CAST-32a and AMC 20-193 [7, 10].

2.7 Spin-off: Worst-case Stack Usage Analysis

In embedded systems, the run-time stack (often just called "the stack") typically is the only dynamically allocated memory area. It is used during program execution to keep track of the currently active procedures and facilitate the evaluation of expressions. Each active procedure is represented by an activation record, also called stack frame or procedure frame, which holds all the state information needed for execution.

Precisely determining the maximum stack usage before deploying the system is important for economical reasons and for system safety. Overestimating the maximum stack usage means wasting memory resources. Underestimation leads to stack overflows: memory cells from the stacks of different tasks or other memory areas are overwritten. This can cause crashes due to memory protection violations and can trigger arbitrary erroneous program behavior, if return addresses or other parts of the execution state are modified. In consequence stack overflows are typically hard to diagnose and hard to reproduce, but they are a potential cause of catastrophic failure. The accidents caused by the unintended acceleration of the 2005 Toyota Camry illustrate the potential consequences of stack overflows: the expert witness' report commissioned by the Oklahoma court in 2013 identifies a stack overflow as probable failure cause [3, 61].

The generic tool architecture of Sec. 2.2 can be easily adapted to perform an analysis of the worst-case stack usage, by exchanging the Microarchitectural analysis step with a dedicated value analysis for the stack pointer register(s) [24]. In 2001, the resulting tool, called StackAnalyzer, was released, which was the first commercial tool to safely prove the absence of stack overflows in safety-critical systems, and since then has been widely adopted in industry.

3 Sound Runtime Error Analysis

The purpose of the Astrée analyzer is to detect source-level runtime errors due to undefined or unspecified behaviors of C programs. Examples are faulty pointer manipulations, numerical errors such as arithmetic overflows and division by zero, data races, and synchronization errors in concurrent software. Such errors can cause software crashes, invalidate separation mechanisms in mixed-criticality software, and are a frequent cause of errors in concurrent and multi-core applications. At the same time, these defects also constitute security vulnerabilities, and have been at the root of a multitude of cybersecurity attacks, in particular buffer overflows, dangling pointers, or race conditions [31].

3.1 The Origins

Astrée stands for Analyseur statique de logiciels temps-réel embarqués (real-time embedded software static analyzer). The development of Astrée started from scratch in Nov. 2001 at the Laboratoire d'Informatique of the École Normale Supérieure (LIENS), initially supported by the ASTRÉE project, the Centre National de la Recherche Scientifique, the École Normale Supérieure and, since September 2007, by INRIA (Paris—Rocquencourt).

First industrial applications of Astrée appeared two years after starting the project. Astrée has achieved the following unprecedented results on the static analysis of synchronous, time-triggered, real-time, safety critical, embedded software written or automatically generated in the C programming language:

- In Nov. 2003, Astrée was able to prove completely automatically the absence of any RTE in the primary flight control software of the Airbus A340 fly-by-wire system.
- From Jan. 2004 on, Astrée was extended to analyze the electric flight control codes then in development and test for the A380 series.
- In April 2008, Astrée was able to prove completely automatically the absence of any RTE in a C version of the automatic docking software of the Jules Vernes Automated Transfer Vehicle (ATV) enabling ESA to transport payloads to the International Space Station [4].

In Dec. 2009, AbsInt started the commercialization of Astrée in cooperation with LIENS, in particular Patrick Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival.

3.2 Further Development

From a technical perspective, the ensuing development activities can be grouped into several categories:

Usability The original version of Astrée was a command-line tool, however, to facilitate commercial use, a graphical user interface was developed. The purpose is not merely to make the tool more intuitive to use, but – even more importantly – to help users understand the results. Astrée targets corner cases of the C semantics which requires a good understanding of the language, and it shows defects due to behavior unexpected by the programmer. To facilitate understanding the unexpected behavior, we have developed a large set of graphical and interactive exploration views. To give some examples, all parents in the call stack, relevant loop iterations or conditional statements that lead to the alarm can be accessed by mouse click, tool tips show the values of values, the call graph can be interactively explored, etc. [28]. In all of this, there is one crucial requirement: all views and graphs have to be efficiently computable and suitable for large-scale software consisting of millions of lines of code [20].

Further usability enhancements were the integration of a preprocessor into Astrée (the original version read preprocessed C code), automated preprocessor

configuration based on JSON compilation files, Windows support, and the ability to classify and comment findings from the GUI.

Apart from easy usability, an important requirement of contemporary development processes is the ability to integrate a tool in a CD/CI (continuous development / continuous integration) platform. To support this, Astrée can be started from the command line with full functionality, the configuration is given as an XML file which can be automatically created, results can be exported in machine-readable formats (xml, csv, html) that support post-processing. Furthermore, there is a large number of plugins and tool couplings which have been developed, e.g., to model-based development tools like Matlab/Simulink/TargetLink [26, 38], as well as CI tools and IDEs such as Jenkins, Eclipse, and Keil μVision.

Formal Requirements The primary use-case of Astrée is to find defects in safety-critical or security-relevant software, hence the same tool qualification requirements apply as described in Sec. 2.3. So, the development of a Qualification Support Kit for Astrée was a mandatory; its structure is similar to the aiT QSK as described above.

Another constraint is that in certain safety processes, no code modifications are allowed which cannot be traced to functional software requirements. Also, in the case of model-based software development, where the code is automatically generated, it is infeasible to modify the source code to interact with a static analyzer.

Astrée provides numerous analysis directives that allow users to interact with the tool, e.g., to pass certain preconditions such as constraints on input value ranges or volatile variable ranges to the analyzer. Alarms can be classified (e.g., as true defect or false alarms) via source code comments or analysis directives. Finally Astrée's domains have been specifically developed to support fine-grained precision tuning to eliminate false alarms. One example is the trace partitioning domain, a generic framework that allows the partitioning of traces based on the history of the control flow [52]. By inserting analysis directives into the code, users can influence the partitioning strategy of the analyzer for limited parts of the code.

To also support use cases where code modifications are infeasible, a formal language AAL has been developed [36] which provides a robust way to locate analyzer directives in the abstract syntax tree without modifying the source code. It is also possible to automatically generate such annotations from the build environment or an interface specification.

New Capabilities

Interleaving Semantics and Integration Analysis While the first versions of Astrée targeted sequential code, most of today's industry applications are multi-threaded. In such software systems, it is highly desirable to be able to do runtime

error analysis at the integration verification stage, i.e., to analyze the entire software stack in order to capture the interactions between all components of the system, determine their effect on data and control flow and detect runtime errors triggered by them.

To support this, Antoine Miné has developed a low-level concurrent semantics [42] which provides a scalable sound abstraction covering all possible thread interleavings. The interleaving semantics enables Astrée, in addition to the classes of runtime errors found in sequential programs, to report data races and lock/unlock problems, i.e., inconsistent synchronization. The set of shared variables does not need to be specified by the user: Astrée assumes that every global variable can be shared, and discovers which ones are effectively shared, and on which ones there is a data race. To implement its interleaving semantics, Astrée provides primitives which expose OS functionality to the analyzer, such as mutex un-/locks, interrupt dis-/enabling, thread creation, etc. Since Astrée is aware of all locks held for every program point in each concurrent thread, Astrée can also report all potential deadlocks. Astrée also supports several stages of concurrent execution so that initialization tasks can be separated from periodic/acyclic tasks. Each thread can be associated to one or several concurrent execution stages.

Using the Astrée concurrency primitives, abstract OS libraries have been developed, which currently support the OSEK/AUTOSAR and ARINC 653 norms [2, 43]. A particularity of OSEK/AUTOSAR is that system resources, including tasks, mutexes and spin-locks, are not created dynamically at program startup; instead they are hardcoded in the system: a specific tool reads a configuration file in OIL (OSEK Implementation Language) or ARXML (AutosaR XML) format describing these resources and generates a specialized version of the system to be linked against the application. A dedicated ARXML converter has been developed for Astrée which automatically generates the appropriate data structures and access functions for the Astrée analysis, and enables a fully automatic integration analysis of AUTOSAR projects [20].

Code Guideline Checking Coding guidelines aim at improving code quality and can be considered a prerequisite for developing safety- or security-relevant software. In particular, obeying coding guidelines is strongly recommended by all current safety standards. Their purpose is to reduce the risk of programming errors by enforcing low complexity, enforcing usage of a language subset, using well-trusted design principles, etc. According to ISO 26262, the language subset to be enforced should exclude, e.g., ambiguously defined language constructs, language constructs that could result in unhandled runtime errors, and language constructs known to be error-prone. Since the Astrée architecture is well suited for sound and precise code guideline checking, over the years, the analyzer has been extended to support all major coding guidelines, such as MISRA C/C++ [44, 45, 41], SEI CERT C/C++ [55], CWE [56], etc.

Cybersecurity Vulnerability Scanning Many security attacks can be traced back to behaviors undefined or unspecified according to the C semantics. By applying

sound static runtime error analyzers, a high degree of security can be achieved for safety-critical software since the absence of such defects can be proven. In addition, security hyperproperties require additional analyses to be performed which, by nature, have a high complexity. To support this, Astrée has been extended by a generic abstract domain for taint analysis that can be freely instantiated by the users [33]. It augments Astrée's process-interleaving interprocedural code analysis by carrying and computing taint information at the byte level. Any number of taint hues can be tracked by Astrée, and their combinations will be soundly abstracted. Tainted input is specified through directives attached to program locations. Such directives can precisely describe which variables, and which part of those variables is to be tainted, with the given taint hues, each time this program location is reached. Any assignment is interpreted as propagating the join of all taint hues from its right-hand side to the targets of its left-hand side. In addition, specific directives may be introduced to explicitly modify the taint hues of some variable parts. This is particularly useful to model cleansing function effects or to emulate changes of security levels in the code. The result of the analysis with tainting can be explored in the Astrée GUI, or explicitly dumped using dedicated directives. Finally, the taint sink directives may be used to declare that some parts of some variables must be considered as taint sinks for a given set of taint hues. When a tainted value is assigned to a taint sink, then Astrée will emit a dedicated alarm, and remove the sinked hues, so that only the first occurrence has to be examined to fix potential issues with the security data flow.

The main intended use of taint analysis in Astrée is to expose potential vulnerabilities with respect to security policies or resilience mechanisms. Thanks to the intrinsic soundness of the approach, no tainting can be forgotten, and that without any bound on the number of iterations of loops, size of data or length of the call stack. Based on its taint analysis, Astrée provides an automatic detection of Spectre-PHT vulnerabilities [32].

Data and Control Flow All current safety norms require determining the data and control flow in the source code and making sure that it is compliant to the intended control and data flow as defined in the software architecture. To meet this requirement, Astrée has been extended by a data and control flow analysis module, which tracks accesses to global, static, and local variables. The soundness of the analysis ensures that all potential targets of data and function pointers are discovered. Data and control flow reports show the number of read and write accesses for every global, static, and out-of-frame local variable, lists the location of each access and shows the function from which the access is made. All variables are classified as being thread-local, effectively shared between different threads, or subject to a data race.

To further support integration verification, a recent extension of Astrée provides a generic concept for specifying software components, enabling the analyzer to lift the data and control flow analysis to report data and control flow interactions between software components. This is complemented by an automatic taint analysis that efficiently tracks the flow of values between components, and

automatically reports undesired data flow and undesired control dependencies. The combination of augmented data and control analysis and the taint analysis for software components provides a sound interference analysis [35].

C++ To respond to the increasing interest in C++ even in the domain of safety-critical software, since 2020 Astrée also provides a dedicated analysis mode for C++ and mixed C/C++. It uses the same analysis technology as Astrée's semantic C code analysis and has similar capabilities. At the same time it is also subject to the same restrictions. The analyzer is designed to meet the characteristics of safety-critical embedded software. Typical properties of such software include a static execution model that uses a fixed number of threads, no or limited usage of dynamic memory allocation and dynamic data structures. Astrée provides an abstract standard template library, that models the behavior of STL containers in an abstract way suitable for analysis with Astrée. Astrée does not attempt to analyze the control flow of exceptions; it only reports if an exception could be raised.

Precision and Efficiency Constant development effort is required to work at precision and scalability of the analyzer. Over the years, various additional abstract domains have been developed to avoid false alarms on common embedded software elements. Examples are domains for finite integer sets, gauges [62, 21], domains for precise analysis of interpolation functions, finite state machines, etc. Astrée's state machine domain heuristically detects state variables and disambiguates them by state partitioning in the relevant program scope [14]. In consequence the analyzer becomes aware of the exact transitions of the state machine and the false alarms due to control flow over-approximation can be avoided. Over the past years, the size of embedded software has grown significantly; typical automotive AUTOSAR projects span 5-50 million lines of (preprocessed) code. One prerequisite to enable an efficient analysis of such large-scale projects is an efficient strategy to heuristically control the context-sensitivity of the analyzer and distinguish critical call chains where full flow- and context-sensitivity is needed from less critical ones where building a summary context is enough [20].

4 The User Perspective

Whereas from an academic perspective, software verification can be fun and is a topic of great merit, this is not necessarily a view shared by every software developer working in the field. In fact, the ISO 26262 norm puts en emphasis on the need to embrace functional safety in the company organization and establish a safety culture [18]. Verification activities should not be – as they often are – perceived as a burden that drains on development cost, delays delivery and does not provide an added value to the end product. Introducing new verification steps should not be perceived as admitting a mistake. The capability of defect

prevention, the efficiency in defect detection, and the degree of automation is crucial for user acceptance.

Advanced program analysis requires significant technical insights, including knowledge about the programming language semantics, microprocessor design, and system configuration. Without the necessary understanding, program analysis tools are hard to use. On the other hand, it is necessary for tools to expose the information users need to understand the results as intuitively as possible.

Finally, users expect tools to solve real problems, e.g., the worst-case execution time on a particular microcontroller in the configuration given, or the occurrence of runtime errors in the tasks as they are deployed in the real system. When providing partial solutions to a problem, it is necessary to explain how to use them to help dealing with the full problem.

5 The Role of Safety Norms

Functional safety and security are aspects of dependability, in addition to reliability and availability. *Functional safety* is usually defined as the absence of unreasonable risk to life and property caused by malfunctioning behavior of the system. Correspondingly, cybersecurity can be defined as absence of unreasonable risk caused by malicious misusage of the system. Functional safety norms aim at formalizing the minimal obligations for developers of safety-critical systems to make sure that unreasonable safety risks are avoided. In addition, advances in system development and verification since the publication date of a given norm have to be taken into account. In other words, safety norms define the minimal requirements to develop safety-relevant software with due diligence. Safety standards typically are domain-specific; examples DO-178B/DO-178C [47, 48] (aerospace), ISO 26262 [17] (automotive), CENELEC EN 50128/EN 50657 [6, 5] (railway), IEC 61508 [15] (general electrical and/or electronic systems), IEC 62304 (medical products), etc.

The DO-178C [48] has been published with supplements focusing on technical advances since release of the predecessor norm DO-178B, in particular the DO-333 (Formal Methods Supplement to DO-178C and DO-278A) [49], that addresses the use of formal methods to complement or replace dynamic testing. It distinguishes three categories of formal analyses: deductive methods such as theorem proving, model checking, and abstract interpretation. The computation of worst-case execution time bounds and the maximal stack usage are listed as reference applications of abstract interpretation. However, the standard does not mandate the use of formal methods.

Table 7 and Table 10 of ISO 26262 Part 6 [19] give a list of recommended methods for verification of software unit design and implementation, and integration verification, respectively. They contain separate entries for formal verification, control flow analysis, data flow analysis, static code analysis, and static analysis by abstract interpretation. Static analysis in general is *highly recommended* for all criticality levels (ASILs), Abstract Interpretation is *recommended*

for all ASILs. The current versions of EN 50128 and IEC 62304 lack an explicit reference to Abstract Interpretation.

Since for industrial system development, functional safety norms are defining what is considered to be (minimal) state of the art, the availability of mature development and verification techniques should be reflected in them. To create the necessary awareness, an exchange between software and safety communities is essential.

6 Conclusion

The focus of this article is to describe the application of Abstract Interpretation to two different real-life problems: to compute sound worst-case execution time bounds, and to perform sound runtime error analysis for C/C++ programs. We have summarized the development history of aiT WCET Analyzer and Astrée, discussed design choices, and illustrated the exigencies imposed by commercial users and industrial processes. We also addressed derived research and applications to other topics, in particular hybrid WCET analysis and worst-case stack usage analysis. In summary, the tools discussed in this article provide a formal methods-based ecosystem for verifying resource usage in embedded software projects. The three main causes of software-induced memory corruption in safety-critical systems are runtime errors, stack overflows, and miscompilation. The absence of runtime errors and stack overflows can be proven by abstract interpretation-based static analyzers. With the formally proven compiler CompCert, miscompilation can be ruled out, hence all main sources of software-induced memory corruption are addressed. Industrial application of mathematically rigorous verification methods strongly depends on their representation in industrial safety norms; the corresponding methods and tools have to become better known to the safety community and their advantages compared to legacy methods better explained.

7 Acknowledgment

Many people contributed to aiT and Astrée and their success. We want to thank them all.

References

1. Alt, M., Ferdinand, C., Martin, F., Wilhelm, R.: Cache behavior prediction by abstract interpretation. In: Cousot, R., Schmidt, D.A. (eds.) Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1145, pp. 52–66. Springer (1996). https://doi.org/10.1007/3-540-61739-6, https://doi.org/10.1007/3-540-61739-6
2. AUTOSAR: AUTOSAR (AUTomotive Open System ARchitecture). http://www.autosar.org

3. Barr, M.: Bookout v. Toyota, 2005 Camry software Analysis by Michael Barr. http://www.safetyresearch.net/Library/BarrSlides_FINAL_SCRUBBED.pdf (2013)
4. Bouissou, O., Conquet, E., Cousot, P., Cousot, R., Feret, J., Ghorbal, K., Goubault, E., Lesens, D., Mauborgne, L., Miné, A., Putot, S., Rival, X., Turin, M.: Space software validation using abstract interpretation. Proc. 13thData Systems in Aerospace (DASIA 2009) (May 2009)
5. BS EN 50657: Railway applications – Rolling stock applications – Software on Board Rolling Stock (2017)
6. CENELEC EN 50128: Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems (2011)
7. Certification Authorities Software Team (CAST): Position Paper CAST-32A Multi-core Processors (November 2016)
8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of POPL'77. pp. 238–252. ACM Press (1977), http://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml [retrieved: Sep. 2017].
9. Cousot, P., Cousot, R.: Static determination of dynamic properties of generalized type unions. In: Wortman, D.B. (ed.) Proceedings of an ACM Conference on Language Design for Reliable Software (LDRS), Raleigh, North Carolina, USA, March 28-30, 1977. pp. 77–94. ACM (1977). https://doi.org/10.1145/800022.808314, https://doi.org/10.1145/800022.808314
10. EASA: AMC-20 – amendment 23 – AMC 20-193 use of multi-core processors (2022)
11. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: Proceedings of EMSOFT 2001, First Workshop on Embedded Software. LNCS, vol. 2211, pp. 469–485. Springer (2001)
12. Ferdinand, C., Wilhelm, R.: Efficient and precise cache behavior prediction for real-time systems. Real-Time Systems **17**(2-3), 131–181 (1999)
13. Freescale Inc.: QorIQ^{TM} P4080 Communications Processor Product Brief (September 2008), rev. 1
14. Giet, J., Mauborgne, L., Kästner, D., Ferdinand, C.: Towards zero alarms in sound static analysis of finite state machines. In: Romanovsky, A., Troubitsyna, E., Bitsch, F. (eds.) Computer Safety, Reliability, and Security. pp. 3–18. Springer International Publishing, Cham (2019)
15. IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems (2010)
16. Infineon Technologies AG: AURIX^{TM} TC27x D-Step User's Manual (2014)
17. ISO 26262: Road vehicles – Functional safety (2018)
18. ISO 26262: Road vehicles – Functional safety – Part 2: Management of functional safety (2018)
19. ISO 26262: Road vehicles – Functional safety – Part 6: Product development at the software level (2018)
20. Kaestner, D., Wilhelm, S., Mallon, C., Schank, S., Ferdinand, C., Mauborgne, L.: Automatic sound static analysis for integration verification of autosar software. In: WCX SAE World Congress Experience. SAE International (apr 2023). https://doi.org/https://doi.org/10.4271/2023-01-0591, https://doi.org/10.4271/2023-01-0591
21. Karos, T.: The Gauge Domain in Astrée. Master's thesis, Saarland University (October 2015)

22. Kästner, D.: Applying Abstract Interpretation to Demonstrate Functional Safety. In: Boulanger, J.L. (ed.) Formal Methods Applied to Industrial Complex Systems. ISTE/Wiley, London, UK (2014)
23. Kästner, D., Ferdinand, C.: Efficient Verification of Non-Functional Safety Properties by Abstract Interpretation: Timing, Stack Consumption, and Absence of Runtime Errors. In: Proceedings of the 29th International System Safety Conference ISSC2011. Las Vegas (2011)
24. Kästner, D., Ferdinand, C.: Proving the Absence of Stack Overflows. In: SAFECOMP '14: Proceedings of the 33th International Conference on Computer Safety, Reliability and Security. LNCS, vol. 8666, pp. 202–213. Springer (September 2014)
25. Kästner, D., Pister, M., Gebhard, G., Schlickling, M., Ferdinand, C.: Confidence in Timing. Safecomp 2013 Workshop: Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR) (September 2013)
26. Kästner, D., Rustemeier, C., Kiffmeier, U., Fleischer, D., Nenova, S., Heckmann, R., Schlickling, M., Ferdinand, C.: Model-Driven Code Generation and Analysis. In: SAE World Congress 2014. SAE International (2014). https://doi.org/\url {http://dx.doi.org/10.4271/2014-01-0217}
27. Kästner, D., Schlickling, M., Pister, M., Cullmann, C., Gebhard, G., Heckmann, R., Ferdinand, C.: Meeting Real-Time Requirements with Multi-Core Processors. Safecomp 2012 Workshop: Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR) (September 2012)
28. Kästner, D., Wilhelm, S., Nenova, S., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Astrée: Proving the Absence of Runtime Errors. Embedded Real Time Software and Systems Congress ERTS [2] (2010)
29. Kästner, D., Barrho, J., Wünsche, U., Schlickling, M., Schommer, B., Schmidt, M., Ferdinand, C., Leroy, X., Blazy, S.: CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In: ERTS2 2018 - Embedded Real Time Software and Systems. 3AF, SEE, SIE, Toulouse, France (Jan 2018), https://hal.inria.fr/hal-01643290, archived in the HAL-INRIA open archive, https://hal.inria.fr/hal-01643290/file/ERTS_2018_paper_59.pdf
30. Kästner, D., Leroy, X., Blazy, S., Schommer, B., Schmidt, M., Ferdinand, C.: Closing the gap – the formally verified optimizing compiler CompCert. In: SSS'17: Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium. pp. 163–180. CreateSpace (2017)
31. Kästner, D., Mauborgne, L., Ferdinand, C.: Detecting Safety- and Security-Relevant Programming Defects by Sound Static Analysis. In: Rainer Falk, Steve Chan, J.C.B. (ed.) The Second International Conference on Cyber-Technologies and Cyber-Systems (CYBER 2017). IARIA Conferences, vol. 2, pp. 26–31. IARIA XPS Press (2017)
32. Kästner, D., Mauborgne, L., Ferdinand, C.: Detecting Spectre Vulnerabilities by Sound Static Analysis. In: Anne Coull, Steve Chan, R.F. (ed.) The Fourth International Conference on Cyber-Technologies and Cyber-Systems (CYBER 2019). IARIA Conferences, vol. 4, pp. 29–37. IARIA XPS Press (2019), http://www.thinkmind.org/download.php?articleid=cyber_2019_3_10_80050
33. Kästner, D., Mauborgne, L., Grafe, N., Ferdinand, C.: Advanced Sound Static Analysis to Detect Safety- and Security-Relevant Programming Defects. In: Rainer Falk, Steve Chan, J.C.B. (ed.) 8th International Journal on Advances in Security. vol. 1 & 2, pp. 149–159. IARIA (2018), https://www.iariajournals.org/security/

34. Kästner, D., Mauborgne, L., Wilhelm, S., Ferdinand, C.: High-Precision Sound Analysis to Find Safety and Cybersecurity Defects. In: 10th European Congress on Embedded Real Time Software and Systems (ERTS 2020). Toulouse, France (Jan 2020), https://hal.archives-ouvertes.fr/hal-02479217
35. Kästner, D., Mauborgne, L., Wilhelm, S., Mallon, C., Ferdinand, C.: Static Data and Control Coupling Analysis. In: 11th Embedded Real Time Systems European Congress (ERTS2022). Toulouse, France (Jun 2022), https://hal.archives-ouvertes.fr/hal-03694546
36. Kästner, D., Pohland, J.: Program Analysis on Evolving Software. In: Roy, M. (ed.) CARS 2015 - Critical Automotive applications: Robustness & Safety. Paris, France (Sep 2015), https://hal.archives-ouvertes.fr/hal-01192985
37. Kästner, D., Hümbert, C., Gebhard, G., Pister, M., Wegener, S., Ferdinand, C.: Taming Timing – Combining Static Analysis With Non-intrusive Tracing to Compute WCET Bounds on Multicore Processors. Embedded World Congress (2021)
38. Kästner, D., Salvi, S., Bienmüller, T., Ferdinand, C.: Exploiting synergies between static analysis and model-based testing (09 2015). https://doi.org/10.1109/EDCC.2015.20
39. Langenbach, M., Thesing, S., Heckmann, R.: Pipeline modeling for timing analysis. In: Proceedings of the 9th International Static Analysis Symposium SAS 2002. LNCS, vol. 2477, pp. 294–309. Springer (2002)
40. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: Proceedings of the 32nd ACM/IEEE Design Automation Conference. pp. 456–461 (Jun 1995)
41. Limited, M.: MISRA C++:2008 Guidelines for the use of the C++ language in critical systems (June 2008)
42. Miné, A.: Static analysis of run-time errors in embedded real-time parallel C programs. Logical Methods in Computer Science (LMCS) **8**(26), 63 (Mar 2012)
43. Miné, A., Delmas, D.: Towards an Industrial Use of Sound Static Analysis for the Verification of Concurrent Embedded Avionics Software. In: Proc. of the 15th International Conference on Embedded Software (EMSOFT'15). pp. 65–74. IEEE CS Press (Oct 2015)
44. MISRA (Motor Industry Software Reliability Association) Working Group: MISRA-C:2012 Guidelines for the use of the C language in critical systems. MISRA Limited (Mar 2013)
45. MISRA (Motor Industry Software Reliability Association) Working Group: MISRA-C:2023 Guidelines for the use of the C language in critical systems. MISRA Limited (Apr 2023)
46. Nowotsch, J., Paulitsch, M., Bühler, D., Theiling, H., Wegener, S., Schmidt, M.: Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In: ECRTS'14: Proceedings of the 26th Euromicro Conference on Real-Time Systems (July 2014)
47. Radio Technical Commission for Aeronautics: RTCA DO-178B. Software Considerations in Airborne Systems and Equipment Certification (1992)
48. Radio Technical Commission for Aeronautics: RTCA DO-178C. Software Considerations in Airborne Systems and Equipment Certification (2011)
49. Radio Technical Commission for Aeronautics: RTCA DO-333. Formal Methods Supplement to DO-178C and DO-278A (2011)
50. Reineke, J., Grund, D., Berg, C., Wilhelm, R.: Timing predictability of cache replacement policies. Real-Time Systems **37**(2), 99–122 (2007)

51. Reineke, J., Wachter, B., Thesing, S., Wilhelm, R., Polian, I., Eisinger, J., Becker, B.: A Definition and Classification of Timing Anomalies. In: Mueller, F. (ed.) International Workshop on Worst-Case Execution Time Analysis (WCET) (July 2006)
52. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. **29**(5), 26 (2007). https://doi.org/10.1145/1275497.1275501, https://doi.org/10.1145/1275497.1275501
53. Schranzhofer, A., Chen, J.J., Thiele, L.: Timing predictability on multi-processor systems with shared resources. In: Workshop on Reconciling Performance with Predictability (RePP), 2010 (October 2009)
54. Shaw, A.C.: Reasoning about time in higher-level language software. IEEE Trans. Software Eng. **15**(7), 875–889 (1989). https://doi.org/10.1109/32.29487, http://dx.doi.org/10.1109/32.29487
55. Software Engineering Institute SEI – CERT Division: SEI CERT C Coding Standard – Rules for Developing Safe, Reliable, and Secure Systems. Carnegie Mellon University (2016)
56. The MITRE Corporation: CWE – Common Weakness Enumeration, https://cwe.mitre.org [retrieved: Sep. 2017].
57. Theiling, H.: Extracting safe and precise control flow from binaries. In: Proceedings of the 7th Conference on Real-Time Computing Systems and Applications. Cheju Island, South Korea (2000)
58. Theiling, H.: ILP-based interprocedural path analysis. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) Proceedings of EMSOFT 2002, Second International Conference on Embedded Software. LNCS, vol. 2491, pp. 349–363. Springer (2002)
59. Thesing, S.: Modeling a system controller for timing analysis. In: Min, S.L., Yi, W. (eds.) Proceedings of the 6th ACM & IEEE International conference on Embedded software, EMSOFT 2006, October 22-25, 2006, Seoul, Korea. pp. 292–300. ACM (2006). https://doi.org/10.1145/1176887.1176929, http://doi.acm.org/10.1145/1176887.1176929
60. Thiele, L., Wilhelm, R.: Design for timing predictability. Real-Time Systems **28**(2-3), 157–177 (2004). https://doi.org/10.1023/B:TIME.0000045316.66276.6e, http://dx.doi.org/10.1023/B:TIME.0000045316.66276.6e
61. Transcript of Morning Trial Proceedings had on the 14th day of October, 2013 Before the Honorable Patricia G. Parrish, District Judge, Case No. CJ-2008-7969. http://www.safetyresearch.net/Library/Bookout_v_Toyota_Barr_REDACTED.pdf (October 2013)
62. Venet, A.: The gauge domain: Scalable analysis of linear inequality invariants (07 2012). https://doi.org/10.1007/978-3-642-31424-7_15
63. Wegener, S.: Towards Multicore WCET Analysis. In: Reineke, J. (ed.) 17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017). OpenAccess Series in Informatics (OASIcs), vol. 57, pp. 1–12. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017). https://doi.org/10.4230/OASIcs.WCET.2017.7, http://drops.dagstuhl.de/opus/volltexte/2017/7311
64. Wilhelm, R., Reineke, J., Wegener, S.: Keeping up with real time. In: Durak, U., Becker, J., Hartmann, S., Voros, N.S. (eds.) Advances in Aeronautical Informatics, Technologies Towards Flight 4.0., pp. 121–133. Springer (2018). https://doi.org/10.1007/978-3-319-75058-3_9, https://doi.org/10.1007/978-3-319-75058-3_9