Efficient Verification of Non-Functional Safety Properties by Abstract Interpretation: Timing, Stack Consumption, and Absence of Runtime Errors

Daniel Kästner, Christian Ferdinand

AbsInt GmbH, Science Park 1, 66123 Saarbrücken, GERMANY

Keywords: static analysis, software certification, worst-case execution time, stack usage, runtime errors.

Abstract

In automotive, railway, avionics and healthcare industries more and more functionality is implemented by embedded software. A failure of safety-critical software may cause high costs or even endanger human beings. Also for applications which are not highly safety-critical, a software failure may necessitate expensive updates. Contemporary safety standards – including DO-178B, DO-178C, IEC-61508, ISO-26262, and EN-50128 – require to identify potential functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals. For ensuring functional program properties automatic or model-based testing, and formal techniques like model checking become more and more widely used. For non-functional properties identifying a safe end-of-test criterion is a hard problem since failures usually occur in corner cases and full test coverage cannot be achieved. For some non-functional program properties this problem is solved by abstract interpretation-based static analysis techniques which provide full control and data coverage and yield provably correct results. In this article we focus on static analyses of worst-case execution time, stack consumption, and runtime errors, which are increasingly adopted by industry in the validation and certification process for safety-critical software. First we will give an overview of the most important safety standards with a focus on the requirements for non-functional software properties. We then explain the methodology of abstract interpretation based analysis tools and identify criteria for their successful application. The integration of static analyzers in the development process requires interfaces to other development tools, like code generators or scheduling tools. Using them for certification requires an appropriate tool qualification. We will address each of these topics and report on industrial experience.

Introduction

The use of safety-critical embedded software in the automotive, avionics and healthcare industries is increasing rapidly. Failures of such safety-critical embedded systems may create high costs or even endanger human beings. Also for applications which are not highly safety-critical, a software failure may necessitate expensive updates. Therefore, utmost carefulness and state-of-the-art techniques for verifying software safety requirements have to be applied to make sure that an application is working properly.

Classical software validation methods like code review and testing with debugging cannot really guarantee the absence of errors. Formal verification methods provide an alternative, in particular for safety-critical applications. One such method is *abstract interpretation* (ref. 6), which allows to obtain statements that are valid for all program runs with all inputs. Such statements may be absence of violations of timing or space constraints, or absence of runtime errors. Static analysis tools are in industrial use that can detect stack overflows, violation of timing constraints (ref. 25), and can prove the absence of runtime errors (ref. 8).

The advantage of static analysis based techniques is that they enable full control and data coverage, but at the same time can reduce the test effort. For approaches based on test and measurement identifying end-of-test criteria for non-functional program properties like timing, stack size, and runtime errors is an unsolved problem. In consequence the required test effort is high, the tests require access to the physical hardware and the results are not complete. In contrast, static analyses can be run by software developers from their workstation computer, they can be integrated in the development process, e.g., in model-based code generators, and allow developers to detect runtime errors as well as timing and space bugs in early product stages. From a methodological point of view, static analyses can be seen as equivalent to testing with full coverage. For validating non-functional program properties they define the state-of-the-art technology.

In the following we will give an overview of the most important safety standards with a focus on the requirements for non-functional software properties. Then we explain the basic methodology of static analysis and present the underlying concepts of tools from three different application areas: aiT for worst-case execution time analysis, StackAnalyzer for stack usage analysis and Astrée for runtime error analysis. Industrial experience is summarized in Section 6 and Section 7 concludes.

Safety Standards

Safety standards like DO-178B (ref. 24), DO-178C, IEC-61508 (ref. 15), ISO-26262 (ref. 16) and EN-50128 (ref. 5) require to identify functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals. Examples for important non-functional safety-relevant software characteristics are runtime errors, execution time and memory consumption. Depending on the criticality level of the software the absence of safety hazards has to be demonstrated by formal methods or testing with sufficient coverage. In the following, we give a short overview on the assessment of non-functional program properties by the safety standards for avionics, space, automotive and railway systems, for general Electric/Electronic systems, and for medical software products.

DO-178B/DO-178C: Published in 1992, the DO-178B (ref. 24) ("Software Considerations in Airborne Systems and Equipment Certification"), is the primary document by which the certification authorities such as FAA, or EASA approve all commercial software-based aerospace systems. The purpose of D0-178B is "to provide guidelines for the production of software for airborne systems and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements". The software levels defined are Level A (most critical) to Level E (least critical).
The DO-178B emphasizes the importance of software verification. Verification is defined as a technical assessment of the results of both the software development processes and the software verification process. Sec. 6.0 of the DO-178B states that "verification is not simply testing. Testing, in general, cannot show the absence of errors." The standard consequently uses the term "verify" instead of "test" when the software verification process objectives being discussed are typically a combination of reviews, analyses and test. The purpose of the software verification process is to detect and report errors that may have been introduced during the software development processes. Removal of the errors is an activity of the software development processes. The general objectives of the software verification process are to verify that the requirements of the system level, the architecture level, the source code level and the executable object code level are satisfied, and that the means used to satisfy these objectives are technically correct and complete. At the code level the objective is to detect and report errors that may have been introduced during the software coding process. The non-functional safety properties are explicitly mentioned, including stack usage, worst-case execution timing and absence of runtime errors.
The DO-178C, due to be finalized in 2011, will be a revision of DO-178B to bring it up to date with respect to current software development and verification technologies. It specifically focuses on model-based software development, object-oriented software, the use and qualification of software tools and the use of formal methods to complement or replace dynamic testing (theorem proving, model checking, and abstract interpretation).

IEC-61508 Edition 2.0: In 2010 a new revision of the functional safety standard IEC-61508 has been published, called Edition 2.0 (ref. 15). It sets out a generic approach for all safety lifecycle activities for systems comprised of electrical and/or electronic and/or programmable electronic (E/E/PE) elements that are used to perform safety functions. The safety integrity levels are called SIL1 (least critical) to SIL4 (most critical). The non-functional program properties are part of the software safety requirements specification, including invalid, out of range or untimely values, response time, best case and worst case execution time, and overflow and underflow of data storage capacity. The IEC-61508 states that verification includes testing and analysis. In the software verification stage, static analysis techniques are recommended for SIL1 and highly recommended for SIL2-SIL4. Among these techniques, data flow analysis is highly recommended in SIL2-SIL4, static analysis of runtime error behavior recommended in SIL1-SIL3 and highly recommended in SIL4, and static worst-case execution time analysis is recommended in SIL1-4. Among the criteria to be considered for selecting specific techniques is the completeness and repeatability of testing, so where testing is used, completeness has to be demonstrated. The results of abstract interpretation based static analyses are considered a mathematical proof; their reliability is rated maximal (R3).
The IEC-61508 also provides requirements for mixed-criticality systems: "Where the software is to implement safety functions of different safety integrity levels, then *all* of the software shall be treated as belonging to the *highest* safety integrity level, unless adequate *independence* between the safety functions of the different safety integrity levels can be shown in the design. It shall be demonstrated either (1) that independence is achieved by both *in the spatial and temporal domains*, or (2) that any violation of independence is controlled. The justification for independence shall be documented". This has significant consequences for hardware selection and system configuration: it has to be ensured that there are no unpredictable timing-related interferences which might affect real-time functions. Cache-related preemption costs, pipeline effects, and timing anomalies have to be taken into account. For multicore processors it has to be shown that there are no inherent timing interferences between cores –

which are quite common, e.g., due to collisions on shared memory buses between cores, or due to shared cache levels (ref. 7). For achieving temporal independence the standard suggests deterministic scheduling methods. One suggestion is using a cyclic scheduling algorithm which gives each element a defined time slice supported by worst case execution time analysis of each element to demonstrate statically that the timing requirements for each element are met. Other suggestions are using time triggered architectures, or strict priority based scheduling implemented by a real-time executive (ref. 15).

ISO-26262: ISO-26262 (Road vehicles – Functional safety) (ref. 16) is the adaptation of the Functional Safety Standard IEC-61508 for Automotive Electric/Electronic Systems. The current version is a draft which will be published as an international standard in mid-2011, replacing the IEC-61508 as formal legal norm for road vehicles.
It requires functional and non-functional hazards to be identified, and it requires demonstrating that the software does not violate the relevant safety goals. The ISO-26262 defines four Automotive Safety Integrity Levels, ASIL A (lowest) to ASIL D (highest). Section 5.4.8. of Part 6 (ref. 16) lists criteria for selecting suitable modeling and programming languages. They include support for embedded real-time software and runtime error handling. It is highly recommended to exclude language constructs which might result in unhandled runtime errors. When using the C programming language this implies using tools to demonstrate the absence of runtime errors.
A further demand of ISO-26262 is that the timing constraints of time-critical functions have to be covered by the specification of the software safety requirements. Here, especially the response time at the system level has to be considered (cf. Section 3.2). During the development of the software architectural design upper bounds of the required resources for the embedded software have to be given. The resources explicitly mentioned by the standard include execution time and storage space. During unit testing and integration testing bounds on execution time and memory consumption have to be established. Among the verification techniques listed by ISO-26262 are static analysis techniques, which are recommended or highly recommended for all ASIL levels.

CENELEC prEN-50128: The CENELEC EN-50128 currently is under revision. The current draft (ref. 5) provides a set of requirements with which the development, deployment and maintenance of any safety-related software intended for railway control and protection applications shall comply. It addresses five software safety integrity levels – from SIL0 (lowest) to SIL4 (highest) – and identifies and lists appropriate techniques and measures for each level of software safety integrity.
Static analysis based on abstract interpretation, e.g. applied to worst-case execution time analysis and runtime error analysis, belongs to the referenced techniques. It is highly recommended for SIL3/SIL4, recommended for SIL1/SIL2 and should be applied throughout the development process: in the software validation stage, the software integration test, software/hardware integration test and software component test.

Regulations for Medical Software: Standards relevant for medical software are the EN-60601 and IEC-62304. The EN-60601 formulates requirements for the software lifecycle and risk management (ref. 32). The standard IEC-62304 describes a lifecycle for software development with a focus on maintenance and on component-oriented software architectures (ref. 29).
Beyond these standards country-specific requirements have to be respected. In the following we will shortly discuss the American and German regulations. The presentation of the US regulations follows reference 31. Software validation is a requirement of the Quality System regulation (cf. Title 21 Code of Federal Regulations (CFR) Part 820, and 61 Federal Register (FR) 52602, respectively). Validation requirements apply to software used as components in medical devices, to software that is itself a medical device, and to production software. Verification "means confirmation by examination and provision of objective evidence that specified requirements have been fulfilled" (ref. 34). In a software development environment, software verification is confirmation that the output of a particular phase of development meets all of the input requirements for that phase. While software testing is a necessary activity, in most cases software testing by itself is not considered sufficient to establish confidence that the software is fit for its intended use. Additional verification activities are required, including static analysis.
In Europe, the validation of medical software has to follow the EU-directive 2007/47/EC (ref. 30), which, in Germany has been incorporated into national law in 2010 (ref. 33). It states that software in its own right, when specifically intended to be used for medical purpose, has to be considered a medical device. For devices which incorporate software or which are medical software in themselves, the software *must* be validated according to the *state of the art*.

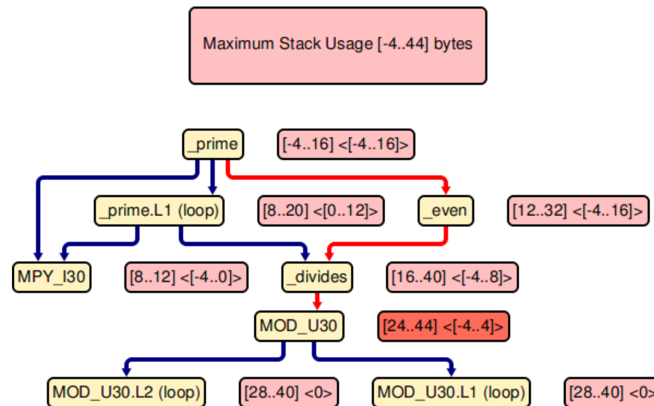
Abstract Interpretation

Static data flow analyses compute invariants for all program points by fixed point iteration over the program structure or the control-flow graph. The theory of abstract interpretation (ref. 6) offers a semantics-based methodology for static program analyses. The concrete semantics is mapped to an abstract semantics by abstraction functions. While most interesting program properties are undecidable in the concrete semantics, the abstract semantics can be chosen for them to be computable. The static analysis is computed with respect to that abstract semantics. Compared to an analysis of the concrete semantics, the analysis result may be less precise but the computation may be significantly faster. By skillful definition of the abstract domains a suitable trade-off between precision and efficiency can be attained.

For program validation there are two essential properties of static analyzers: *soundness* and *safety*. A static analysis is called *sound* if the computed results hold for any possible program execution. Abstract interpretation supports formal correctness proofs: it can be proved that an analysis will terminate and that it is sound, i.e., that it computes an overapproximation of the concrete semantics. In a *safe* static analysis imprecision can occur, but it can be shown that imprecisions will always occur on the safe side.
Let's illustrate this with two application scenarios: In runtime error analysis, soundness means that the analyzer never omits to signal an error that can appear in some execution environment. If no potential error is signaled, definitely no runtime error can occur: there are no false negatives. If a potential error is reported, the analyzer cannot exclude that there is a concrete program execution triggering the error. If there is no such execution, this is a false alarm (false positive). This imprecision is on the safe side: it can never happen that there is a runtime error which is not reported. In WCET analysis, soundness means that the computed WCET bound holds for any possible program execution. Safety means that the only imprecision occurring is overestimation: the WCET will never be underestimated.

3.1 Stack Usage Analysis: A possible cause of catastrophic failure is a stack overflow which might cause the program to behave in a wrong way or to crash altogether. When they occur, stack overflows can be hard to diagnose and hard to reproduce. The problem is that the memory area for the stack usually must be reserved by the programmer. Underestimation of the maximum stack usage leads to stack overflow, while overestimation means wasting memory resources. Measuring the maximum stack usage with a debugger is no solution since one only obtains a result for a single program run with fixed input. Even repeated measurements with various inputs cannot guarantee that the maximum stack usage is ever observed.
The tool StackAnalyzer from AbsInt employs a global AI-based static program analysis to compute safe upper bounds on the maximal stack usage of tasks. The main input of StackAnalyzer is the binary executable. The analysis does not require any code modification and does not rely on debug information. The results are independent from flaws in the debug output and refer to exactly the same code as in the shipped system. First, the control-flow graph (CFG) is reconstructed from the input file, the binary executable. Then a static value analysis computes value ranges for registers and address ranges for instructions accessing memory. By concentrating on the value of the stack pointer during value analysis, StackAnalyzer computes how the stack increases and decreases along the various control-flow paths.
This information can be used to derive the maximum stack usage of the entire task. StackAnalyzer takes the entire application into account and interprocedurally analyzes each call site with its precise stack height. The results of StackAnalyzer are presented as annotations in a combined call graph and control-flow graph (cf. Figure 1). It shows the critical path, i.e., the path on which the maximum stack usage is reached which gives important feedback for optimizing the stack usage of the application under analysis.

Figure 1 — Call graph and control-flow graph with stack analysis results



3.2 WCET Analysis: Worst-Case Execution Time Prediction: Many tasks in safety-critical embedded systems have hard real-time characteristics. Failure to meet deadlines may be as harmful as producing wrong output or failure to work at all. Yet the determination of the Worst-Case Execution Time (WCET) of a task is a difficult problem because of the characteristics of modern software and hardware (ref. 27).

Embedded control software (e.g., in the automotive industries) tends to be large and complex. The software in a single electronic control unit typically has to provide different kinds of functionality. It is usually developed by several people, several groups or even several different providers. Code generator tools are widely used. They usually hide implementation details to the developers and make an understanding of the timing behavior of the code more difficult. The code is typically combined with third party software such as real-time operating systems and/or communication libraries.

Concerning hardware, there is typically a large gap between the cycle times of modern microprocessors and the access times of main memory. Caches and branch target buffers are used to overcome this gap in virtually all performance-oriented processors (including high-performance micro-controllers and DSPs). Pipelines enable acceleration by overlapping the executions of different instructions. Consequently the execution behavior of the instructions cannot be analyzed separately since it depends on the execution history. Cache memories usually work very well, but under some circumstances minimal changes in the program code or program input may lead to dramatic changes in cache behavior. For (hard) realtime systems, this is undesirable and possibly even hazardous. Making the safe yet – for the most part – unrealistic assumption that all memory references lead to cache misses results in the execution time being overestimated by several hundred percent.

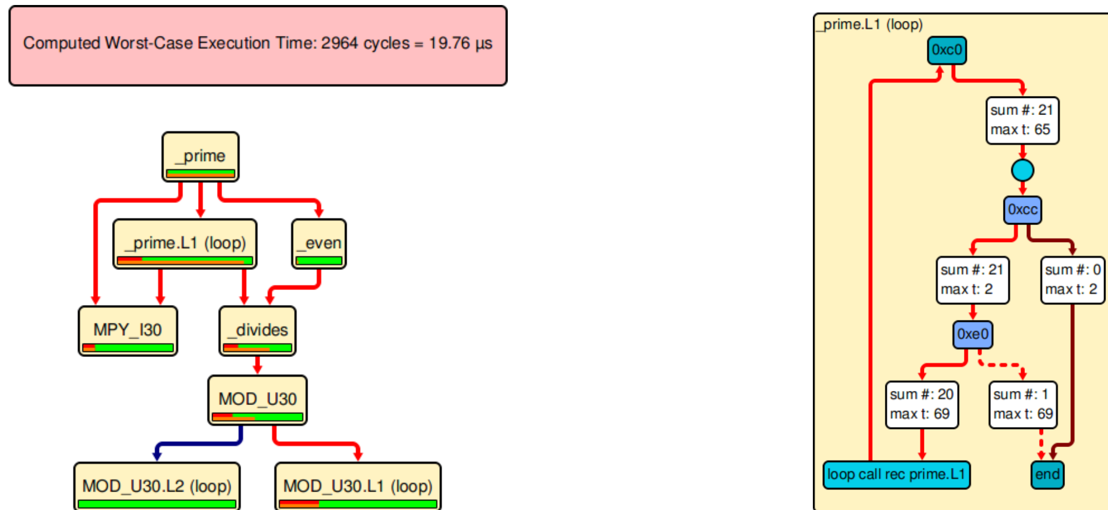
The widely used classical methods of predicting execution times are not generally applicable. Software monitoring and dual-loop benchmarks modify the code, which in turn changes the cache behavior. Hardware simulation, emulation, or direct measurement with logic analyzers can only determine the execution time for some fixed inputs. They cannot be used to infer the execution times for all possible inputs in general.

In contrast, abstract interpretation can be used to efficiently compute a safe approximation for all possible cache and pipeline states that can occur at a program point in any program run with any input. These results can be combined with ILP (Integer Linear Programming) techniques to safely predict the worst-case execution time and a corresponding worst-case execution path. A survey of methods for WCET analysis and of WCET tools is given in (ref. 28).

AbsInt's timing verifier aiT (ref. 11) has been used by NASA as an industry-standard static analysis tool for demonstrating the absence of timing-related software defects in the Toyota Motor Corporation Unintended Acceleration Investigation (ref. 23). It computes a safe upper bound for the WCET of a task, assuming no interference from the outside. Effects of interrupts, IO and timer (co-)processors are not reflected in the predicted runtime and have to be considered separately within system-level timing analysis. The main input of aiT is the binary executable. Like StackAnalyzer the analysis does not require any code modification and does not rely on debug information. The results are independent from flaws in the debug output and refer to exactly the same code as in the shipped system. aiT determines the WCET of a program task in several phases (ref. 12), which makes it possible to use different methods tailored to each subtasks (ref. 26). First, the control-flow graph (CFG) is reconstructed from the input file, the binary executable. Then value analysis computes value ranges for registers and address ranges for instructions accessing memory; a loop bound analysis determines upper bounds for the number of iterations of simple loops. Subsequently, a cache analysis classifies memory references as cache misses or hits (ref. 10) and a

pipeline analysis predicts the behavior of the program on the processor pipeline (ref. 17). Finally the path analysis determines a worst-case execution path of the program (ref. 26).

Figure 2 — Call graph and basic-block graph with WCET results



The results of aiT are reported as annotations in call graphs and control-flow graphs (cf. Fig. 2), and as report files in text format and XML format. The overall WCET bounds/estimations for sequential code pieces can also be communicated to the system-level analyzer SymTA/S (ref. 14), which computes worst-case response times from the sequential WCETs, taking into account interrupts and task preemptions.
aiT is available for a wide range of 16-bit and 32-bit microcontrollers. In general, the availability of safe worst-case execution time bounds depends on the predictability of the execution platform. Especially multicore architectures may exhibit poor predictability because of essentially non-deterministic interferences on shared resources which can cause high variations in execution time. Reference 7 gives a more detailed overview and suggests example configurations for available multicores to support static timing analysis.

3.3 Runtime Error Analysis: Another important goal when developing critical software is to prove that no runtime errors can occur. Examples for runtime errors are floating-point overflows, array bound violations, division by zero, or invalid pointer accesses. A well-known example for the possible effects of runtime errors is the explosion of the Ariane 5 rocket in 1996 (ref. 18).
As detailed above, software testing can be used to detect errors, but not to prove their absence. The success of static analysis is based on the fact that safe overapproximations of program semantics can be computed. For runtime error analysis this means that the analysis result for a statement x will be either “(i) statement x will not cause an error”, or “(ii) statement x may cause an error”. This imprecision allows to compute results in acceptable time, even for large software projects. In the first case, the user can rely on the absence of errors, in the second case either an error has been found, or there was a false alarm.
Each alarm has to be manually investigated to determine whether there is an error which has to be corrected, or whether it was just a false alarm. If all the alarms raised by an analysis have been proved to be false, then the proof of absence of runtime errors is completed. This could be checked manually, but the problem is that such a human analysis is error-prone and time consuming, especially since there might be interdependencies between the false alarms and in some cases deviations from the C standard may be willingly accepted. Therefore the number of alarms should be reduced to zero, since then the absence of runtime errors is automatically proved by the analyzer run. To that end, it is important that the analyzer is precise, i.e., produces only few false alarms. This can only be achieved by a tool that can be “specialized” to a class of properties for a family of programs. Additionally the analyzer must be parametric enough for the user to be able to fine-tune the analysis of any particular program of the family. General software tools not amenable to specialization usually report a large number of false alarms which is the reason why such tools are only used in industry to detect runtime errors, and not to prove their absence.
Additionally the analyzer must provide flexible annotation mechanisms to communicate external knowledge to the analyzer. Only by a combination of high analyzer precision and support for semantic annotations the goal of zero

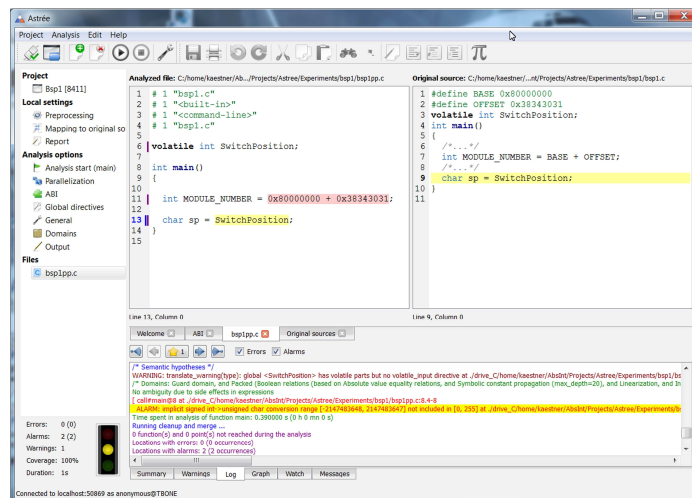
false alarms can be achieved. A prerequisite is that users get enough information to understand the cause of an alarm so that they can either fix the bugs or supply the missing semantic information.

Astrée (Analyseur statique de logiciels temps-réel embarqués) (ref. 1) has been specifically designed to meet these requirements: it produces only a small number of false alarms for control/command programs written in C according to “ISO/IEC 9899:1999 (E)” (C99 standard) (ref. 4). And it provides the user with enough options and directives to help reduce this number significantly. Thus, in contrast to many other static analyzers Astrée cannot only be used to detect runtime errors, but to actually prove their absence.

Astrée (ref. 3) can be adapted to the software project under analysis in order to improve analysis precision. The key feature here is that Astrée is fully parametric with respect to the abstract domains. There is a variety of predefined abstract domains, including the following ones: The interval domain approximates variable values by intervals. The octagon domain (ref. 22) covers relations of the form $x \pm y \leq c$ for variables x and y and constants c. Floating-point computations are precisely modeled while keeping track of possible rounding errors (ref. 21). The memory domain empowers Astrée to exactly analyze pointer arithmetics and union manipulations. The clock domain has been specifically developed for synchronous control programs and supports relating variable values to the system clock. With the filter domain (ref. 13) digital filters can be precisely approximated. Based on inspecting reported false alarms the abstract domains can be stepwise refined for a given program class. It is also possible to incorporate additional abstract domains into Astrée (ref. 2).

In a further step, there are two mechanisms for adapting Astrée to individual programs from a program family. First, abstract domains can be parameterized to tune the precision of the analysis for individual program constructs or program points (ref. 19). This means that in one analysis run important program parts can be analyzed very precisely while less relevant parts can be analyzed very quickly – without compromising system safety. Second there are annotations for making external information available to Astrée in a well-defined and concise way. As current experience shows the parameterization of the programs under analysis rarely has to be changed when the analyzed software evolves over time. So in contrast e.g., to theorem provers the parameterization is very stable.

Figure 3 — Astrée user interface



The alarms reported by Astrée can be interactively investigated (cf. Fig. 3) by checking the alarm cause, investigating variable values, the relevant context, browsing the call stack, etc. The call graph of the application can be interactively explored.

Integration in the Development Process

Static analysis tools are not only applicable at the validation and verification stage but also during the development stage. One advantage of static analysis methods is that no testing on physical hardware is required. Thus the analyses can be called just like a compiler from a workstation computer after the compilation or linking stage of the project. For all tools mentioned in Sec. 3, aiT, StackAnalyzer, and Astrée, there are batch versions facilitating the integration in a general automated build process. This enables developers to instantly assess the effects of program

changes on WCET and stack usage and runtime errors. Defects are detected early, so that late-stage integration problems can be avoided. Furthermore there are tool couplings of aiT and StackAnalyzer with model-based code generators (e.g. Esterel SCADE (ref. 9) or ETAS ASCET) and with system-level scheduling tools (e.g. SymTA/S (ref. 14)). Such couplings enable a seamless integration of static analysis tools in the development process.

Tool Qualification

In the validation stage the goal is to verify that the stack limits or worst-case execution time bounds of the application are not exceeded and that no runtime errors may occur. Both aiT and StackAnalyzer have successfully been qualified as analysis tools according to DO-178B. The qualification process can be automated to a large degree by Qualification Support Kits. Qualification kits are available for aiT and StackAnalyzer, a qualification support kit for Astrée has been announced. The kits consist of a report package and a test package. The report package lists all functional requirements and contains a verification test plan describing one or more test cases to check each functional requirement. The test package contains an extensible set of test cases and a scripting system to automatically execute all test cases and evaluate the results. The generated reports can be submitted to the certification authority as part of the certification package.

Experience

In recent years tools based on static analysis have proved their usability in industrial practice and, in consequence, have increasingly been used by avionics, automotive and healthcare industries. In the following we report some experiences gained with aiT WCET Analyzer, StackAnalyzer and Astrée.

StackAnalyzer results are usually precise for any given program path. Statements about the precision of aiT are hard to obtain since the real WCET is usually unknown for typical real-life applications. For an automotive application running on MPC 555, the results of aiT have been 5-10% above the highest execution times observed in a series of measurements (which may have missed the real WCET). For an avionics application running on MPC 755, Airbus has noted that aiT’s WCET for a task typically is about 25% higher than some measured execution times for the same task, the real but non-calculable WCET being in between (ref. 25). Measurements at AbsInt have indicated overestimations ranging from 0% (cycle-exact prediction) till 10% for a set of small programs running on M32C, TMS320C33, and C166/ST10.

Astrée has been used in several industrial avionics and space projects. One of the examined software projects from the avionics industry comprises 132,000 lines of C code including macros and contains approximately 10,000 global and static variables (ref. 3). The first run of Astrée reported 1200 false alarms; after adapting Astrée the number of false alarms could be reduced to 11. The analysis duration was 1h 50 min on a PC with 2.4 GHz and 1 GB RAM. The report (ref. 8) gives a detailed overview of the analysis process for an Airbus avionics project. The software project consists of 200,000 lines of preprocessed C code, performs many floating-point computations and contains digital filters. The analysis duration for the entire program is approximately 6 hours on a 2.6 GHz PC with 16 GB RAM. At the beginning, the number of false alarms was 467 and could be reduced to zero in the end.

Conclusion

The quality assurance process for safety-critical embedded software is of crucial importance. The cost for system validation grows with increasing criticality level to constitute a large fraction of the overall development cost. The problem is twofold: system safety must be ensured, yet this must be accomplishable with reasonable effort.

Contemporary safety standards require to identify potential functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals. Tools based on abstract interpretation can perform static program analysis of embedded applications. Their results are determined without the need to change the code and hold for all program runs with arbitrary inputs. Especially for non-functional program properties they are highly attractive, since they provide full data and control coverage and can be seamlessly integrated in the development process.

We have presented three exemplary tools in this article: aiT allows to inspect the timing behavior of (time-critical parts of) program tasks. It takes into account the combination of all the different hardware characteristics while still obtaining tight upper bounds for the WCET of a given program in reasonable time. StackAnalyzer calculates safe upper bounds on the maximum stack usage of tasks. Astrée can be used to prove the absence of runtime errors in C programs. It can be specialized to the software under analysis and achieves very high precision. Industrial

synchronous real-time software from the avionics industry could be successfully analyzed by Astrée with zero false alarms.
aiT, StackAnalyzer and Astrée can be used as analysis tools for the certification according to safety standards like DO-178B or ISO 26262. They are used by many industry customers from avionics and automotive industries and have been proved in industrial practice. The tool qualification process can be automatized to a large extend by dedicated Qualification Support Kits.

Acknowledgement

The work presented in this paper has been supported by the European FP6 project INTEREST, the European FP7 projects INTERESTED and PREDATOR, and the ITEA2 project ES_PASS.

References

1. AbsInt Angewandte Informatik GmbH. Astrée web page. http://www.astree.de.
2. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In AIAA Infotech@Aerospace 2010, number AIAA-2010-3385, pages 1–38. American Institue of Aeronautics and Astronautics, April 2010.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03), pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.
4. JTC1/SC22. Programming languages – C, 16 Dec. 1999.
5. CENELEC DRAFT prEN 50128. Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems, 2009.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Proceedings of the 4th ACM Symposium on Principles of Programming Languages, pages 238–252, Los Angeles, California, 1977.
7. C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza (Burguière), J. Reineke, B. Triquet, S.Wegener, and R. Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. Ingénieurs de l'Automobile, 807:26–42, 2010.
8. D. Delmas and J. Souyris. ASTRÉE: from Research to Industry. In Proc. 14th International Static Analysis Symposium (SAS2007), number 4634 in LNCS, pages 437–451, 2007.
9. Esterel Technologies. SCADE Suite. http://www.esterel-technologies.com/products/scade-suite.
10. C. Ferdinand. Cache Behavior Prediction for Real-Time Systems. PhD thesis, Saarland University, 1997.
11. C. Ferdinand and R. Heckmann. Worst-case execution time – a tool provider's perspective. In 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing ISORC 2008, Orlando, Florida, USA, May 2008.
12. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In Proceedings of EMSOFT 2001, First Workshop on Embedded Software, volume 2211 of Lecture Notes in Computer Science, pages 469–485. Springer-Verlag, 2001.
13. J. Feret. Static Analysis of Digital Filters. In European Symposium on Programming (ESOP'04), number 2986 in LNCS, pages 33–48. Springer-Verlag, 2004. Springer-Verlag.
14. R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis – the SymTA/S approach. IEEE Proceedings on Computers and Digital Techniques, 152(2), Mar. 2005.
15. IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.
16. ISO 26262-WD. Road vehicles – Functional safety, 2009.
17. M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In Proceedings of the 9th International Static Analysis Symposium SAS 2002, volume 2477 of Lecture Notes in Computer Science, pages 294–309. Springer-Verlag, 2002.
18. J. Lions et al. ARIANE 5, Flight 501 Failure. Report by the Inquiry, 1996.
19. L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In 14th European Symposium on Programming ESOP'05, number 3444 in LNCS, pages 5–20, 2005.
20. Medizinproduktegesetz in der Fassung der Bekanntmachung vom 7. August 2002 (BGBl. I S. 3146); zuletzt geändert durch Artikel 12 des Gesetzes vom 24. Juli 2010 (BGBl. I S. 983).

21. A. Miné. Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors. In Proc. of the European Symposium on Programming (ESOP'04), volume 2986 of Lecture Notes in Computer Science, pages 3–17. Springer, Barcelona, Spain 2004. http://www.di.ens.fr/˜mine/publi/article-mine-esop04.pdf.
22. A. Miné. The Octagon Abstract Domain. Higher-Order and Symbolic Computation, 19(1):31–100, 2006. http://www.di.ens.fr/˜mine/publi/article-mine-HOSC06.pdf.
23. NASA Engineering and Safety Center. Technical Support to the National Highway Traffic Safety Administration (NHTSA) on the Reported Toyota Motor Corporation (TMC) Unintended Acceleration (UA) Investigation. January 18, 2011.
24. Radio Technical Commission for Aeronautics. RTCA DO-178B. Software Considerations in Airborne Systems and Equipment Certification.
25. J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis, pages 21–24, 2005.
26. H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modeling and program path analysis. In Proceedings of the 19th IEEE Real-Time Systems Symposium, pages 144–153, Madrid, Spain, Dec. 1998.
27. R. Wilhelm. Determining bounds on execution times. In R. Zurawski, editor, Handbook on Embedded Systems, pages 14–1 – 14–23. CRC Press, 2005.
28. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem—overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems, 7(3):1–53, 2008.
29. DIN EN-62304. Medical device software – Software life cycle processes, 2006.
30. Directive 2007/47/EC of the European Parliament and of the Council amending Council Directive 90/385/EEC on the approximation of the laws of the Member States relating to active implantable medical devices, Council Directive 93/42/EEC concerning medical devices and Directive 98/8/EC concerning the placing of biocidal products on the market, 05.09.2007.
31. General Principles of Software Validation; Final Guidance for Industry and FDA Staff. U.S. Department of Health and Human Services, Food and Drug Administration; Center for Devices and Radiological Health, Office of Device Evaluation, Office of In Vitro Diagnostics; Center for Biologics Evaluation and Research, Office of Blood Research and Review, 11.05.2005.
32. IEC 60601-1-4:1996, Medical electrical equipment, Part 1: General requirements for safety, 4. Collateral Standard: Programmable electrical medical systems. International Electrotechnical Commission, 1996.
33. Medizinproduktegesetz in der Fassung der Bekanntmachung vom 7. August 2002 (BGBl. I S. 3146); zuletzt geändert durch Artikel 12 des Gesetzes vom 24. Juli 2010 (BGBl. I S. 983).
34. Title 21 Code of Federal Regulations (21 CFR) Subpart C – Design Controls of the Quality System Regulation.

Biography

Dr. Daniel Kästner, CTO, AbsInt GmbH, Science Park 1, 66123 Saarbrücken, Germany, telephone +49-681-38360-0, facsimile +49-681-38360-20, e-mail: info@absint.com.

Dr. Daniel Kaestner is co-founder and CTO of AbsInt GmbH, a spin-off company from Saarland University (Germany) founded in 1998. He studied Computer Science and Business Economics and received his Ph.D. on code optimization for embedded processors in the year 2000. He has been a lecturer at Saarland University and was a program committee member of numerous international conferences. His current work is focused on program analysis for embedded software and timing analysis for real-time software.

Dr. Christian Ferdinand, CEO, AbsInt GmbH, Science Park 1, 66123 Saarbrücken, Germany, telephone +49-681-38360-0, facsimile +49-681-38360-20, e-mail: info@absint.com.

Dr. Christian Ferdinand is co-founder and CEO of AbsInt GmbH. He studied Computer Science and Electrical Engineering at Saarland University and received his Ph.D on "Cache Behavior Prediction for Real-Time Systems" in 1997. His work is focused on code generation for digital signal processors and on timing analysis for real-time systems.