2017-01-0054

Finding all Potential Run-Time Errors and Data Races in Automotive Software

Daniel Kästner, Antoine Miné, Stephan Wilhelm, Xavier Rival, André Schmidt, Jérôme Feret, Patrick Cousot, Christian Ferdinand

Absint Angewandte Informatik GmbH, University Pierre and Marie Curie, Daimler AG, INRIA/ENS, New York University

Abstract

Safety-critical embedded software has to satisfy stringent quality requirements. All contemporary safety standards require evidence that no data races and no critical run-time errors occur, such as invalid pointer accesses, buffer overflows, or arithmetic overflows. 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. The static analyzer Astrée has been extended to soundly and automatically analyze concurrent software. This novel extension employs a scalable abstraction which covers all possible thread interleavings, and reports all potential run-time errors, data races, deadlocks, and lock/unlock problems. When the analyzer does not report any alarm, the program is proven free from those classes of errors. Dedicated support for ARINC 653 and OSEK/AUTOSAR enables a fully automatic OS-aware analysis. In this article we give an overview of the key concepts of the concurrency analysis and report on experimental results obtained on concurrent automotive software. The experiments confirm that the novel analysis can be successfully applied to real automotive software projects.

1. Introduction

A failure of a safety-critical system may cause high costs or even endanger human beings. With the unbroken trend towards growing software size in embedded systems more and more safety-critical functionality is implemented in software. Preventing software-induced system failures becomes an increasingly important task. One particularly dangerous class of errors are run-time errors which include 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.

Contemporary safety norms – including DO-178B, DO-178C, IEC-61508, ISO-26262, and EN-50128 – require to identify potential hazards and to demonstrate that the software does not violate the relevant safety goals. In all of them demonstrating the absence of run-time errors is a verification goal which is mostly formulated indirectly by addressing run-time errors (e.g., division by zero, invalid pointer accesses, arithmetic overflows) in general, and additionally corruption of content, synchronization mechanisms, and freedom of interference in concurrent execution [1].

Abstract interpretation is a formal methodology for static program analysis [2]. It supports formal soundness proofs (it can be proven that no error is missed) and scales to real-life industry applications. 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. Such conclusions may be that no timing or space constraints are violated, or that run-time errors or data races are absent: the absence of these errors can be guaranteed [3]. Nowadays, abstract interpretation-based static analyzers that can detect stack overflows and violations of timing constraints [4] and that can prove the absence of run-time errors [5], are widely used in industry. From a methodological point of view, abstract interpretation-based static analyses can be seen as equivalent to testing with full data and control coverage. They do not require access to the physical target hardware, can be easily integrated in continuous verification frameworks and model-based development environments [6], and they allow developers to detect run-time errors as well as timing and space bugs in early product stages.

In the past established semantics-based static analysis tools could not handle concurrent programs with the same level of soundness, coverage, and automation as sequential programs. Typically they did not cover all potential process interleavings, required extensive user interaction, had limited support for concurrency primitives and failed to detect all potential concurrency-specific hazards such as data races [7].

The focus of this article is an extension of the static analyzer Astrée to soundly and automatically analyze concurrent software. The extension covers all possible thread interleavings, and soundly reports all run-time errors, data races, and invalid usage of OS services such as lock/unlock problems. When the analyzer does not report any alarm, the program is proven free from those classes of errors. The novel mechanism enables a fully automatic OS-aware analysis of ARINC 653, OSEK, and AUTOSAR applications, and can also be used for the analysis of POSIX threads. We give an overview of the key concepts of the concurrency analysis and report on experimental results obtained on concurrent automotive software. The experiments confirm that the novel analysis can be successfully applied to real automotive software projects.

2. Automotive Software Characteristics

During the past years automotive software has evolved to be among the most complex embedded applications ever developed. High-end cars feature complex networks of ECUs whose software size is in the range of several millions of lines of code.

To master the growing complexity various standards have been developed that regulate different aspects of system development. OSEK [8] and AUTOSAR [9] provide a standardized software architecture with standardized interfaces – a key requirement for inter-operability between different vendors and different components. However, this genericity comes at the cost of increased software size and complexity. The predominant programming language for control applications is C, which is traditionally written by hand, but more and more constitutes a high-level assembly code for model-based software development environments. Safety-critical automotive C code is typically developed according to the MISRA-C coding guidelines [10, 11] which define a C subset and rules of usage to minimize the risk of programming errors. The standard ISO-26262 [12] defines the minimal requirements to achieve and demonstrate functional safety throughout the system and software life cycle.

So on the one hand there is an established and to a certain degree harmonized infrastructure in place, on the other hand entirely new challenges rise at a fast pace. Among them are increasing demands for car-to-car and car-to-infrastructure communication, connectivity, hybrid and electric driving, and – last but not least –the trend to highly automated driving [13]. More and more advanced driver assistance systems are being developed which are triggering a paradigm shift in automotive system design: the transition from fail-safe to fail-operational behavior. If automatic systems take over critical driving functionality they cannot just shut down in case of failures, they have to continue operating until the car is in a safe state [14].

A consequence for software development is that it is and will remain crucial to detect errors as early as possible, as reliably as possible, and with the lowest possible human effort. The contribution of this article is to provide automatic means to demonstrate the absence of run-time errors and data races which belong to a particularly dangerous class of bugs.

3. Static Run-Time Error Analysis

In the following we will concentrate on the static analyzer Astrée [15] which signals all potential runtime errors and further critical program defects [16]. It is sound, i.e., if no errors are signaled, this means there are no errors from the class of errors under investigation – the absence of errors has been proved. It reports program defects caused by unspecified and undefined behaviors according to the C norm (ISO/IEC 9899:1999 (E)) [17], program defects caused by invalid concurrent behavior, violations of user-specified programming guidelines, and computes program properties relevant for functional safety. Users are notified about:

- integer/floating-point division by zero
- out-of-bounds array indexing
- erroneous pointer manipulation and dereferencing (null, uninitialized, and dangling pointers)
- data races (read/write or write/write concurrent accesses by two threads to the same memory location without proper mutex locking)

- inconsistent locking (lock/unlock problems, deadlocks)
- invalid calls to operating system services (e.g. calls to the OSEK service `TerminateTask`() on a task with unreleased resources)
- integer and floating-point arithmetic overflows
- read accesses to uninitialized variables
- code unreachable under all circumstances
- violations of optional user-defined assertions to prove additional runtime properties, e.g., to guarantee that output variables are within the expected value ranges
- violations of coding rules (MISRA C:2004/2012) and code metric thresholds. The supported code metrics include the statically computable HIS metrics (HIS 2008), e.g., comment density, and cyclomatic complexity.
- non-terminating loops

Floating-point computations are handled precisely and safely by taking all potential rounding errors into account. Furthermore Astrée computes data and control flow reports containing a detailed listing of accesses to global and static variables sorted by functions, variables, and processes and containing a summary of caller/called relationships between functions. The analyzer can also report each potentially shared variable, the list of processes accessing it, and the types of the accesses (read, write, read/write).

The C99 standard does not fully specify data type sizes, endianness nor alignment which can vary with different targets or compilers. Astrée is informed about these target settings by a dedicated configuration file and takes the specified properties into account.

Workflow

In the following we will use the term alarm to denote a notification about a potential run-time error. While Astrée finds all potential run-time errors, it may err on the safe side and produce false alarms. For industrial use producing the fewest possible number of false alarms is an important goal. Only with zero alarms the absence of run-time errors is automatically proven. The design of the analyzer aims at reaching the zero false alarm objective, which was accomplished for the first time on large industrial applications at the end of November 2003. For keeping the initial number of false alarms low, a high analysis precision is mandatory. To achieve high precision Astrée provides a variety of predefined abstract domains, including the following ones:

- The interval domain approximates variable values by intervals.
- The octagon domain [18] covers relations of the form $x \pm y \leq c$ for variables x and y and constants c.
- Floating-point computations are precisely modelled while keeping track of possible rounding errors.
- The memory domain empowers Astrée to exactly analyze pointer arithmetic and union manipulations. It also supports a type-safe analysis of absolute memory addresses.
- The clock domain has been specifically developed for synchronous control programs and supports relating variable values to the system clock [19].
- With the filter domain [20] digital filters can be precisely approximated.

Any remaining alarm has to be manually checked by the developers – and this manual effort should be as low as possible. Astrée explicitly supports investigating alarms in order to understand the reasons for them to occur. Alarm contexts can be interactively explored, the computed value ranges of variables can be displayed for each different context, the call graph is visualized (cf. Figure 2), etc.

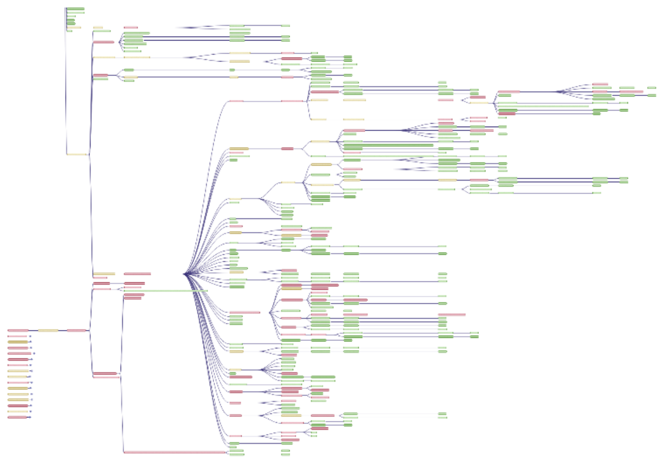
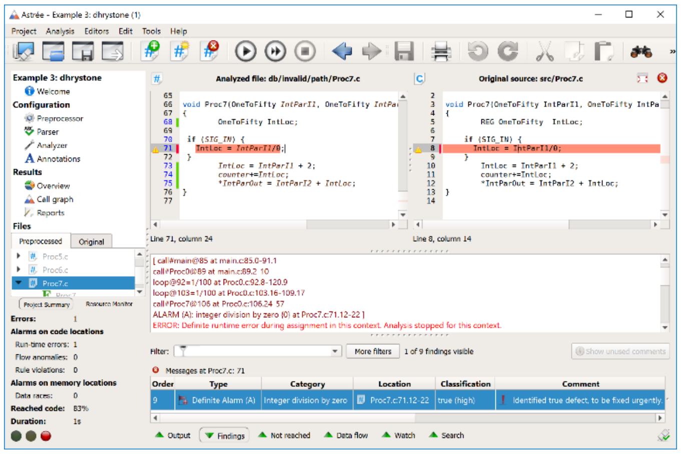


Figure 1. Call Graph Visualization (zoomed out).

If there is a true error it has to be fixed. A false alarm can possibly be eliminated by a suitable parameterization of Astrée: If the error cannot occur due to certain preconditions which are not known to Astrée, they can be made available to Astrée via dedicated directives. These annotations make the side conditions explicit which have to be satisfied for a correct program execution. If the false alarm is caused by insufficient analysis precision, steering directives are available that allow users to locally tune the analysis precision to eliminate the false alarm.

As an example the `__ASTREE_unroll` directive can be used to enforce disambiguating every iteration of one specific loop. The key feature is that Astrée is fully parametric with respect to the abstract domains. Abstract domains can be parameterized to tune the precision of the analysis for individual program constructs or program points [21]. This means that in one analysis run important program parts can be analysed very precisely while less relevant parts can be analysed very quickly – without compromising system safety.



```
void Proc7(OneToFifty IntParI1, OneToFifty IntPa
{
        OneToFifty IntLoc;

 if (SIG_IN) {
   IntLoc = IntParI1/0;
 }
        IntLoc = IntParI1 + 2;
        counter+=IntLoc;
        *IntParOut = IntParI2 + IntLoc;
}
```

Line 71, column 24

```
void Proc7(OneToFifty IntParI1, OneToFifty IntPa
{
        REG OneToFifty  IntLoc;

    if (SIG_IN) {
      IntLoc = IntParI1/0;
    }
        IntLoc = IntParI1 + 2;
        counter+=IntLoc;
        *IntParOut = IntParI2 + IntLoc;
}
```

Line 8, column 14

```
[ call#main@85 at main.c:85.0-91.1
call#Proc0@89 at main.c:89.2 10
loop@92=1/100 at Proc0.c:92.8-120.9
loop@103=1/100 at Proc0.c:103.16-109.17
call#Proc7@106 at Proc0.c:106.24 57
ALARM (A): integer division by zero {0} at Proc7.c:71.12-22 ]
ERROR: Definite runtime error during assignment in this context. Analysis stopped for this context.
```

Errors:	1
Alarms on code locations	
Run-time errors:	1
Flow anomalies:	0
Rule violations:	0
Alarms on memory locations	
Data races:	0
Reached code:	83%
Duration:	1s

1 of 9 findings visible

Messages at Proc7.c: 71

Order	Type	Category	Location	Classification	Comment
9	Definite Alarm (A)	Integer division by zero	Proc7.c:71.12-22	true (high)	Identified true defect, to be fixed urgently.

Figure 2. Astrée alarm classification.

All directives can be specified in a formal language AAL [22] and stored in a dedicated file. An AAL annotation consists of an Astrée directive and a path specifying the program point to insert the directive at. The path is specified in a robust way by exploiting the program's syntactical structure without relying on line number information. E.g., the annotation

```
__ASTREE_annotation(( main {+1 loop}
  insert before: __ASTREE_unroll((3)) ));
```

inserts the directive `__ASTREE_unroll((3))` immediately before the first loop in function main. The AAL language is a prerequisite for supporting model-based code generators. It makes it possible to separate the annotations from the source code, so that when the code is regenerated, all previously generated annotations from structurally unchanged code parts are still valid, even if the line numbers change. In cases where there are structural changes of the code, Astrée provides a mechanism to detect whether annotations are still placed at the intended location [23].

In the case where some alarms cannot be eliminated by increasing the analyzer precision they can be classified and commented by using the `__ASTREE_comment` directive. This can be done in a convenient way from the Astrée findings overview: comment directives and AAL annotation are generated automatically. The available classifications are *uncommented, true, true (high), true (low), true (not a defect), false* or *undecided* (cf. Figure 3).

Handling Concurrency

Whereas previous Astrée versions have been limited to sequential C software, Astrée has been extended by a novel low-level concurrent semantics [24] which provides a scalable abstraction covering all possible thread interleavings. The interleaving semantics enables Astrée, in addition to the classes of run-time errors found in sequential programs, to report data races, i.e., read/write or write/write concurrent accesses by two threads to the same memory location without proper mutex locking 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. After a data race, the analysis continues by considering the values stemming from all interleavings. In addition to the range of each variable at each program point, Astrée reports the set of effectively shared variables, together with the set of threads accessing these variables, the kinds of operations performed (reads or writes), and their range of values.

On sequential programs, Astrée uses a fully flow-sensitive and context-sensitive analysis. However, concurrent programs feature a far more complex control structure than sequential ones, which makes it unpractical to consider a fully flow-sensitive analysis. There is a combinatorial explosion of the number of interleaved execution paths and it would be too costly to distinguish the value of a variable at each combination of thread control locations. To efficiently cover all potential execution interleavings Astrée analyzes separately each thread and collects abstract versions of the effects they have on the shared memory. Threads are reanalyzed iteratively, taking into account such global effects, until stabilization, at which point a sound over-approximation of all possible behaviors has been found. This method is nearly as efficient as a sequential program analysis and still is highly precise as it maintains flow-sensitivity at the intra-thread level.

Thread priorities are exploited to reduce the amount of spurious interleavings considered in the abstraction and to achieve a more precise analysis. A dedicated task priority domain also supports dynamic priorities, e.g., according to the Priority Ceiling Protocol used in OSEK systems. Astrée includes a built-in notion of mutual exclusion locks, on top of which actual synchronization mechanisms offered by operating systems can be modeled (such as POSIX mutexes or semaphores [25]); program-enforced mutual exclusion is also exploited by Astrée to reduce spurious interleavings. When these features are insufficient to match the concurrency semantics of the analyzed program, Astrée reverts to unrestricted preemption, which ensures a sound analysis coverage for all concurrency models, including execution on multi-core processors. In particular, Astrée is not limited to collaborative threads nor discrete sets of preemption points.

4. Modelling Operating Systems

Programs to be analyzed are seldom run in isolation; they interact with an environment. In order to soundly report all run-time errors, Astrée must take the effect of the environment into account. In the simplest case the software runs directly on the hardware, in which case the environment is limited to a set of volatile variables, i.e., program variables that can be modified by the environment concurrently, and for which a range can be provided to Astrée by formal directives. More often, the program is run on top of an operating system, which it can access through function calls to a system library. When analyzing a program using a library, one possible solution is to include the source code of the library with the program. This is not always convenient (if the library is complex), nor possible, if the library source is not available, or not fully written in C, or ultimately relies on kernel services (e.g., for system libraries). An alternative is to provide a stub implementation, i.e., to write, for each library function, a specification of its possible effect on the program. Astrée provides stub libraries for the ARINC 653 standard, the OSEK/AUTOSAR standards [8, 9], and for POSIX threads. More details on ARINC 653 and Real-Time POSIX are available in [26], in the following we are focusing on OSEK/AUTOSAR.

An OSEK/AUTOSAR program consists of a set of tasks, a set of interrupts (also called ISRs), a set of timers (also called alarms), and schedule tables (a data-driven mechanism to activate tasks). Task scheduling and synchronization is achieved through explicit task activation and chaining, the use of priorities, orders to disable and enable interrupts, the use of resource objects (that act as locks), and events (that act as signals). We provide an OSEK/AUTOSAR library that handles these mechanisms by mapping them to Astrée low-level concurrency objects: tasks, ISRs, alarms and schedule tables are mapped to Astrée threads; resources are mapped to Astrée mutexes; events are mapped to Astrée signals; moreover, Astrée natively supports the relevant notions of priorities and offers built-in primitives to achieve chaining, starting, and stopping. The standard proposes several conformance classes, with support for increasingly complex features (such as extended tasks, fully preemptive scheduling, multiple task activation, etc.). The model proposed in Astrée supports the most general class, which guarantees that all programs can be soundly analyzed.

A particularity of OSEK is that system resources, including tasks, are not created dynamically at program startup; instead they are hardcoded in the system: a specific tool reads a configuration file in OIL format describing these resources and generates a dedicated version of the system to be linked against the application. Astrée supports a similar workflow. In the preprocessor stage it can read OIL files and outputs a C file containing a table of the declared resources, with their attributes (task priority, alarm periodicity, etc.). The OIL file also assigns actions to be executed when an OSEK alarm expires, such as activating a given task or event, or calling a call-back. The preprocessor thus generates specific C functions to handle the actions associated to OSEK alarms. A fixed set of application-independent stubs, comprising 3 Klines of C with Astrée directives, implements the 31 OSEK entry points. The fixed stub also contains a main analysis entry point that creates Astrée threads and mutexes according to the generated tables and enters parallel execution mode. Finally, it contains synthetic entry-points for Astrée threads handling OSEK alarms, whose purpose is to call, at non-deterministic intervals, the functions generated by the preprocessor to implement the actions associated to OSEK alarms.

Combining the C sources of the OSEK application, the fixed OSEK stub provided with Astrée, and the C file automatically generated from the OIL file, we get a stand-alone application, without any undefined symbol, that can be analyzed with Astrée and models faithfully the execution of the application in an OSEK environment. This workflow enables a high level of automation with minimal configuration when analyzing OSEK applications.

5. Practical Experience

In this section we will summarize practical experience with two real-life automotive projects. The projects have been selected to cover a wide range of typical automotive use cases: modular analysis vs. analysis of fully integrated ECU, hand code vs. automatically generated code, OSEK system vs. AUTOSAR system. In both cases early development versions have been used still exhibiting known issues to be able to ascertain that they are indeed detected by the analyzer. We did not insert Astrée directives to eliminate false alarms; all results are obtained with default settings. Furthermore we also did not insert bug fixes, i.e., the results correspond to those obtained in an initial analysis run. All experiments were run on a PC with an Intel Core i7-6700K (4.00GHz) processor with 64GB RAM under openSUSE Leap 42.1. Below we will shortly describe the applications, then give an overview of the analysis results and discuss the most important findings.

5.1 Project 1: Brake Control Unit

The first real-life application is a brake control unit for trucks. The analysis project does not cover the full ECU software, it consists of three application components: two control components and one logic component. One of the control components has been manually written, the other two components have been automatically generated by dSPACE TargetLink. The project consists of two tasks comprising 177.608 lines of preprocessed C code (without blank lines and without comments). The project is configured by an .OIL file automatically processed by Astrée. Astrée detects four data races, i.e. shared variables concurrently accessed by both tasks without proper synchronization. In total there are 1041 global variables among which 14 are shared between the two tasks. In addition, Astrée reports 776 code location with potential run-time errors, two data flow anomalies (non-terminating loops in the two task functions). The alarms about potential run-time errors are distributed among the following categories[1]:

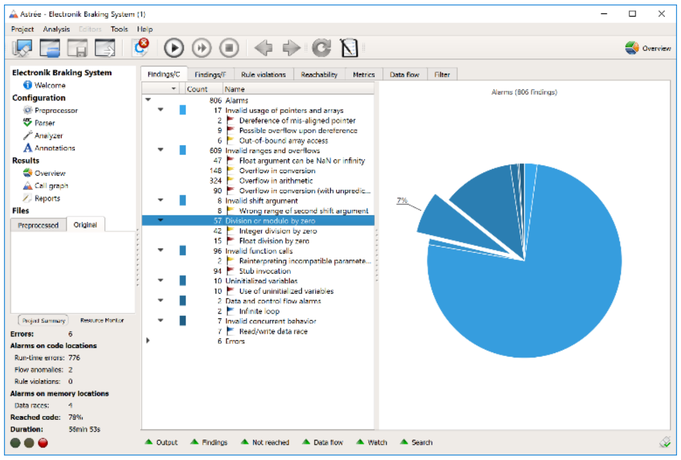
[1] Note that the number of alarms about run-time errors is 797 since there are some locations with multiple alarms. Examples are out-of-

Table 1: Alarm distribution for Brake Control Unit

Alarm Category	Number of Alarms
Invalid usage of pointers and arrays Overflow upon dereference Out-of-bound array access Dereference of mis-aligned pointer	17
Division of modulo by zero Integer division by zero Floating-point division by zero	57
Invalid ranges and overflows	609
Invalid shift argument	8
Uninitialized variables	10
Invalid function calls Stub invocation Incompatible function parameter types	96

The analysis takes 56 min with full precision and activated task priority domain, and consumes 725 MB RAM. It reaches 78% of the code which is expected since some inputs needed to trigger the remaining code have been considered invariant. 94 alarms from the sub-category *Stub invocation* are due to calls to functions whose definition is not contained in the sources under analysis. Since they implement reads from the environment they can be safely handled by stubs automatically generated by Astrée which assume that all potential values of the corresponding data type can be returned.

Astrée reports an alarm of the category *Invalid concurrent behavior* for each variable access contributing to a data race. The number of such alarms can be high, but they help users to distinguish between correct accesses and accesses contributing to a race. In the project there are 6 such alarms which all belong to the sub-category read/write data race. All of them were confirmed to be justified, there were no false alarms about data races. Figure 3 shows a screenshot of the alarm overview of Astrée which includes charts of the distribution of alarms per C function and per alarm category.



```
Count  Name
  806  Alarms
   17  Invalid usage of pointers and arrays
    2    Dereference of mis-aligned pointer
    9    Possible overflow upon dereference
    6    Out-of-bound array access
  609  Invalid ranges and overflows
   47    Float argument can be NaN or infinity
  148    Overflow in conversion
  324    Overflow in arithmetic
   90    Overflow in conversion (with unpredic...
    8  Invalid shift argument
    8    Wrong range of second shift argument
   57  Division or modulo by zero
   42    Integer division by zero
   15    Float division by zero
   96  Invalid function calls
    2    Reinterpreting incompatible paramete...
   94    Stub invocation
   10  Uninitialized variables
   10    Use of uninitialized variables
    2  Data and control flow alarms
    2    Infinite loop
    7  Invalid concurrent behavior
    7    Read/write data race
    6  Errors
```

```
Errors:            6
Alarms on code locations
  Run-time errors: 776
  Flow anomalies:  2
  Rule violations: 0
Alarms on memory locations
  Data races:      4
Reached code:      78%
Duration:          56min 53s
```

Figure 3: Astrée alarm overview for Brake Control Unit

---

bounds array accesses which typically cause two alarms, one about the invalid index value, one about the overflow upon dereference.

The sophistication of Astrée's task interleaving analysis can be illustrated in the following case where Astrée can prove that there is no data race. To improve readability we use artificial function and variable names.

`M` is an array of short integers whose size is 16 bit on the target platform. It is read and written in task `T2`, and written in task `T1`. All read and writes are protected by critical sections, except for the reads occurring in a function `fr` which is called in task `T2`. If `M` was of type `int`, then there would be a data race, but it is an array, and it turns out that in different contexts different parts of the array are written or read. Function `fr` in task `T2` only reads the array at positions 16 to 43, and 68 to 79. Since it is an array of type `short int[]`, that corresponds to the offsets 32 to 87, and 136 to 159.

The only functions writing to `M` and which are called in task `T1` are `f8` and `f16`. The argument of these calls is always a constant. For `f8`, it can be 244, 245, 246 or 247. For `f16` it can only be 248.

Function `f8(x)` writes to `((char*) M)[ord[x]]`, where the offset to access `M` is contained in another array `ord`, so `M` is written at offset `ord[x]` (note that it is the offset, and not the position, because `M` is cast to a pointer to char before dereferencing). Since Astrée precisely tracks the values of the arrays involved, it knows that the only possible value for `ord[244]` is 1, `ord[245]` is 22, `ord[246]` is 23, and `ord[247]` is 0. None of these offsets is in [32,87] or [136,159].

Function `f16(x)` writes to `M[ord[x]]`, so `M` is written at position `ord[x]`, and `ord[248]` is 2, which is not in the set of positions [16,43] or [68,79].

In consequence no write in task `T1` is at an offset that is read without protection in task `T2`, so there is no data race. Important features to show this include Astrée's flow-sensitive and field-sensitive analysis, its byte-level memory model, and the precise tracking of numeric pointer offsets.

5.2 Tank Control Unit

The second project is an automotive tank control unit developed in AUTOSAR 3.2. The code under analysis includes the entire ECU code, including the full AUTOSAR stack. It performs advanced functions like filling level detection, fuel temperature measurement, valve control, etc. The application code has mostly been generated by TargetLink but also contains manually written components. The non-AUTOSAR basic software, e.g., the Complex Device Drivers, also have been manually written. The software under analysis consists of 2.854.057 lines of preprocessed C code (without blank lines and without comments). There are 11 tasks, 13 ISRs, 2 counters and 9 alarms. Again the project is configured by an .OIL file automatically processed by Astrée.

In the following, we will present the results of three analysis configurations: In the first one, we do not take task priorities into account, but rather assume any possible preemption scenario. In the second one we replace the Dem and NVM AUTOSAR components by stub implementations which over-approximate their potential impact on the global program state. In the third one we take task priorities into account and also exploit the fact that the execution is staged, i.e. that there are separate stages where different sets of tasks are active (cf. below).

5.2.1 Priority-Insensitive Analysis

In this project there 2179 global/static variables among which 1329 are shared global variables. Astrée reports 1657 code locations with potential run-time errors, 12 data flow anomalies (11 non-terminating loops in task functions and one busy waiting loop), and 1142 data races. The alarms about potential run-time errors are distributed among the following categories:

Table 2: Alarm distribution for Tank Control Unit (priority-insensitive analysis)

Alarm Category	Number of Alarms
Invalid usage of pointers and arrays Use of dangling pointer Overflow upon dereference Out-of-bound array access Dereference of mis-aligned pointer Dereference of null or invalid pointer Pointer to null or invalid function Invalid pointer comparison Arithmetics on invalid pointers	1349
Division of modulo by zero Integer division by zero	14
Invalid ranges and overflows	836
Invalid shift argument	9
Uninitialized variables	35
Invalid function calls Stub invocation Incompatible function parameter types Incompatible function return type Function call with wrong number of arguments Recursive function call	65

The analysis takes 17 hours, 5min, and consumes 48.9 GB RAM. It reaches 78% of the code. The 29 alarms from the sub-category Stub invocation are mostly calls to assembly functions whose effect does not have to be considered.

As described above Astrée reports an alarm of the category *Invalid concurrent behavior* for each variable access contributing to a data race. In total for the 1142 variables subject to data races, there are 12.731 such alarms, 8839 alarms from the sub-category *Read/write data race*, and 3886 *Write/write data race* alarms. In addition there are 6 alarms from the sub-category *Invalid usage of OS service*. In one case, the end of a task was reached before encountering `TerminateTask` or `ChainTask`, three cases are related to inconsistent resource usage, the two others are caused by stub functions conservatively returning too large value ranges.

5.2.2 Priority-Insensitive Analysis with AUTOSAR Stubs

Astrée provides detailed timing statistics keeping track of how much time is spent in analyzing each C-function. The timing information for the analysis configuration of Sec. 5.2.1 shows that considerable time is spent in the `Dem` and `NvM` AUTOSAR components. Therefore we extended the AUTOSAR OS stub library outlined in Section 4 by stub versions of the `Dem` and `NvM` components which conservatively over-approximate their effect on the global program state.

With this modification the analysis time is reduced by ~70% to 5h 14 min, the maximal memory consumption to 18.4 GB. The analysis reaches 75% of the code and the results essentially correspond to those of the original configuration without alarms and data races reported for the `Dem` and `NvM` components.

Table 3: Alarm distribution for Tank Control Unit (priority-insensitive analysis with AUTOSAR stubs)

Alarm Category	Number of Alarms
Invalid usage of pointers and arrays Overflow upon dereference Out-of-bound array access Dereference of mis-aligned pointer Dereference of null or invalid pointer Pointer to null or invalid function Arithmetics on invalid pointers	922
Division of modulo by zero Integer division by zero	14
Invalid ranges and overflows	698
Invalid shift argument	8
Uninitialized variables	20
Invalid function calls Stub invocation Incompatible function parameter types Function call with wrong number of arguments Recursive function call	56

In total there are 1301 code locations with potential run-time errors, and again 12 data flow anomalies (11 non-terminating loops in task functions and one busy waiting loop). The analyzer reports 1983 global variables, among which 1184 variables are detected as shared and 1044 are reported to be subject to data races.

There are two main reasons for the lower number of alarms compared to the results of Sec. 5.2.1: First, the source files containing the implementation of the `Dem` and `NvM` components have been removed from the analysis since the stubs are used instead of the full implementation. In consequence the alarms reported from those files in the original configuration of Sec. 5.2.1 are missing in the new configuration. Second, some other source files also contain functionality normally called from the `Dem/NvM` components which is not reachable in the new configuration and, in consequence, does not trigger alarms. Both reasons accounts for most of the differences in the findings.

The key observation is that for the remainder of the application replacing the `Dem/NvM` components by stubs only has negligible influence on the findings reported. Therefore unless the `Dem/NvM` components themselves are in the focus of the analysis using the stubbed version leads to comparable results, but with significantly reduced analysis time and memory consumption. This contributes to faster turnaround times and higher analysis efficiency.

5.2.3 Priority-Sensitive Analysis with AUTOSAR Stubs

Not taking into account task priorities provides a sound result but causes spurious preemption scenarios which can lead to false alarms. Therefore in the third configuration we activate Astrée's priority domain such that task priorities and their dynamic changes according to the *Priority Ceiling Protocol* are taken into account.

Furthermore, the application is structured in a way that EEPROM data are set up once by the highest-priority task and all other tasks only issue read accesses to them. The scheduling makes sure that this initialization task is activated in the startup phase and is not interrupted by other tasks. In the default configuration from Sec. 5.2.1 Astrée conservatively assumes that all tasks run at the same time, therefore accesses from the initialization tasks interfere with the accesses from other tasks which results in many spurious data races.

Such situations can be handled by Astrée by assigning tasks to separate execution stages such that at each execution stage different sets of tasks (or ISRs) are active. Task preemptions between tasks from different execution stages cannot occur. In the first phase only one process runs which performs basic initializations and ends with a call to `StartOS`. After that the parallel phase starts with the activation of the interrupt service routines (ISRs): first a dedicated initialization task runs in parallel with the ISRs. It ends with a call to `Rte_Start` which activates all tasks and alarms. In the next stage a second initialization task runs at maximal priority until it ends, again in parallel with the ISRs. In the third stage all remaining periodic and asynchronous tasks, ISRs and alarms are executed in parallel.

The analysis takes 16 hours and 24 minutes, and reaches 75% of the code with a maximal memory consumption of 42.7GB. Compared to the configuration of Sec. 5.2.2 the number of data races is reduced by more than 80% from 1184 to 215, mainly since there are no more spurious interferences between the initialization tasks and the periodic and asynchronous tasks.

Astrée reports 1208 code locations with potential run-time errors, and the same 12 data flow anomalies as in the previous configurations. The alarms about potential run-time errors are distributed among the following categories:

Table 4: Alarm distribution for Tank Control Unit (priority-sensitive analysis with AUTOSAR stubs)

Alarm Category	Number of Alarms
Invalid usage of pointers and arrays Overflow upon dereference Out-of-bound array access Dereference of mis-aligned pointer Dereference of null or invalid pointer Pointer to null or invalid function Arithmetics on invalid pointers	872
Division of modulo by zero Integer division by zero	13
Invalid ranges and overflows	639
Invalid shift argument	8
Uninitialized variables	20
Invalid function calls Stub invocation Incompatible function parameter types Function call with wrong number of arguments Recursive function call	50

The overall reduction of alarms is a consequence of the lower number of data races, since task interferences due to data races often lead to run-time errors.

Conclusion

All current safety norms require safety-critical software to be free of run-time errors and data races. The rising predominance of concurrent software architectures puts classic validation methods, such as testing or code reviews to their limits, because they can hardly cope with the non-deterministic nature of concurrent programs, the huge number of interleavings, and the difficulty to cover all potential execution scenarios. We have given an overview of Astrée, a well-tried static analysis verification tool based on abstract interpretation, and its recent extension to the sound analysis of concurrent C programs which efficiently covers all possible task interleavings and reports all potential run-time errors, deadlocks, and data races. We have shown how Astrée can support programs for various operating systems and concurrency libraries including the OSEK/AUTOSAR standards which are widely used in the automotive domain. Our experimental results show that Astrée is able to report run-time errors and data races in realistic OS configurations with high precision and feasible analysis time. In summary Astrée can be successfully applied to real-life automotive industry projects and can efficiently produce precise results.

References

1. AbsInt GmbH. *Safety Manual for aiT, Astrée, StackAnalyzer*, 2015.
2. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977.
3. D. Kästner. Applying Abstract Interpretation to Demonstrate Functional Safety. In J.-L. Boulanger, editor, *Formal Methods Applied to Industrial Complex Systems*. ISTE/Wiley, London, UK, 2014.
4. Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, Guillaume Borios, and Reinhold 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.
5. 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.
6. D. Kästner, C. Rustemeier, U. Kiffmeier, D. Fleischer, S. Nenova, R. Heckmann, M. Schlickling, and C. Ferdinand. Model-Driven Code Generation and Analysis. In *SAE World Congress 2014*. SAE International, 2014.
7. A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. *Embedded Real Time Software and Systems Congress ERTS*^{2}.
8. OSEK/VDX Operating System. Version 2.2.3, 2005.
9. AUTOSAR (AUTomotive Open System ARchitecture). http://-www.autosar.org.
10. MISRA-C:2004 Guidelines for the use of the C language in critical systems, Oct. 2004.
11. MISRA-C:2012 Guidelines for the use of the C language in critical systems, Mar. 2013.
12. ISO 26262. Road vehicles – Functional safety, 2011.
13. GI/SafeTRANS/VDA. Automotive Roadmap Embedded Systems – Eingebettete Systeme in der Automobilindustrie. Roadmap 2015 – 2030, September 2015.
14. R. Debouk, B. Czerny, and J. d'Ambrosio. Safety Strategy for Autonomous Systems. *International Systems Safety Society Conference*, August 2011.
15. 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 *Proc. of PLDI'03*, pages 196–207. ACM Press, June 7–14 2003.

16. A. Mine, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. *Embedded Real Time Software and Systems Congress ERTS*^{2}.
17. JTC1/SC22. Programming languages – C, 16 Dec. 1999.
18. A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
19. Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of Static Analyzers: A Comparison with ASTRÉE. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007*, pages 3–20. IEEE Computer Society, 2007.
20. Jérôme Feret. Static analysis of digital filters. In *Proc. of ESOP'04*, volume 2986 of *LNCS*, pages 33–48. Springer, 2004.
21. 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.
22. AbsInt. *The Static Analyzer – User Documentation for AAL Annotations*, 2015.
23. Daniel Kästner and Jan Pohland. Program Analysis on Evolving Software. In Matthieu Roy, editor, *CARS 2015 - Critical Automotive applications: Robustness & Safety*, Paris, France, September 2015.
24. A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science (LMCS)*, 8(26):63, Mar. 2012.
25. IEEE Computer Society and The Open Group. Portable operating system interface (POSIX) – Application program interface (API) amendment 2: Threads extension (C language). Technical report, ANSI/IEEE Std. 1003.1c-1995, 1995.
26. A. Miné and D. Delmas. 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)*, pages 65–74. IEEE CS Press, Oct. 2015.