Towards an Industrial Use of Sound Static Analysis for the Verification of Concurrent Embedded Avionics Software*

Antoine Miné
École Normale Supérieure
45, rue d'Ulm
F-75230 Paris Cedes 05, France
mine@di.ens.fr

David Delmas
Airbus Operations S.A.S.
316, route de Bayonne
31060 Toulouse Cedex 9, France
david.delmas@airbus.com

ABSTRACT

Formal methods, and in particular sound static analyses, have been recognized by Certification Authorities as reliable methods to certify embedded avionics software. For sequential C software, industrial static analyzers, such as Astrée, already exist and are deployed. This is not the case for concurrent C software. This article discusses the requirements for sound static analysis of concurrent embedded software at Airbus and presents AstréeA, an extension of Astrée with the potential to address these requirements: it is scalable and reports soundly all run-time errors with few false positives. We illustrate this potential on a variety of case studies targeting different avionics software components, including large ARINC 653 and POSIX threads applications, and a small part of an operating system. While the experiments on some case studies were conducted in an academic setting, others were conducted in an industrial setting by engineers, hinting at the maturity of our approach.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Validation, Assertion checkers*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning About Programs—*Assertions, Invariants, Mechanical verification*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

Keywords

Static analysis, abstract interpretation, embedded software, concurrent software

*This work is supported by the INRIA project "Abstraction" common to CNRS and ENS in France, and by the project ANR-11-INSE-014 from the French *ANR*.

General Terms

Experimentation, Reliability, Verification

1. INTRODUCTION

The safety of embedded critical software, such as those found in avionics, automotive, space, medical, and power industries is crucial, as the slightest software error can have dramatic consequences. Their verification and validation process is well specified in domain-specific international standards (e.g., [1] for avionics systems). While testing remains a key method, its shortcomings are well-known, and there is a strong movement towards formal methods to complement or replace them. Such methods provide strong, mathematical guarantees about systems behaviors. In particular, semantic-based static analysis can discover at compile-time properties of the dynamic behaviors of programs by analysis of the source code; it is automated, sound (full control and data coverage), and can be made precise and efficient by employing powerful abstractions (as advocated by abstract interpretation [8]), making it an attractive method in an industrial context, where the cost of deploying new methods must be taken into account. Nowadays, commercial static analysis tools are deployed in embedded industries. One example is the Astrée static analyzer [4], which detects all the run-time errors in embedded C code. Astrée is however limited to sequential codes and is not sound for concurrent codes, which constitute an increasing share of critical embedded software. Concurrent software is also a prime target for formal verification because testing methods scale poorly with the combinatorial explosion of concurrent executions.

This article discusses AstréeA, a recent extension of Astrée to analyze soundly, efficiently, and precisely concurrent codes, and its application to verify avionics code from Airbus: Sec. 2 discusses the place of formal methods in avionics certification and its implementation at Airbus, Sec. 3 presents the challenges of certifying concurrent avionics software, Sec. 4 presents the technology behind AstréeA, Sec. 5 presents on some case studies how AstréeA can address these challenges, Sec. 6 discusses related work, and Sec. 7 concludes. The foundations and use of Astrée were covered, from both academic and industrial perspectives, in a number of publications [5, 4]. The theoretical foundations underlying AstréeA were covered in [17, 18]. This article discusses the effective use of AstréeA. It presents novel case studies, introducing for the first time studies performed by industrial end-users, and builds a case for the widespread adoption of sound static analysis to verify concurrent embedded software. It brings an industrial perspective to AstréeA.

2. SEMANTIC VERIFICATION AT AIRBUS

2.1 Industrial context

Avionics software running on on-board computers is a critical component of the systems of civil aircraft. They are thus subject to certification by Certification Authorities, and developed according to stringent rules imposed by the applicable DO-178/ED-12 international standards.

Among the many processes described in DO-178, verification processes are responsible for more than half of the overall costs of avionics software development. Considering the steady increase in size and complexity of this kind of software, classical V&V processes, based on massive testing campaigns and complementary intellectual reviews and analyses, no longer scale up within reasonable costs.

Some formal verification techniques have been shown to scale up to real-size industrial software. For a decade, Airbus has therefore been introducing such techniques into their own verification processes [20], in order to replace or complement legacy methods. Significant effort is currently being invested into updating Airbus avionics software development and verification processes to take maximum advantage from formal methods, i.e., improve industrial efficiency while maintaining the safety and availability of avionics systems.

2.2 The requirement for soundness

Revision B of DO-178 states that software verification process objectives are met through a combination of reviews, analyses, and testing. It mentions formal methods as an alternative method, but does not provide any guidance, due to inadequate maturity at the time the document was written and limited applicability to airborne software. Therefore, DO-178B compliant avionics software verification processes cannot rely on formal techniques.

This issue was addressed in revision C [1] of DO-178, applicable to new software developments as of 2014. It introduces a technical supplement, DO-333, providing guidance on the use of formal techniques to meet DO-178C objectives. The supplement introduces standard categories of formal analysis techniques: deductive methods, model-checking, and abstract interpretation. Abstract interpretation, in particular, is presented as a method to construct semantic-based analysis algorithms for the automatic, static, and sound determination of dynamic properties of infinite-state programs. It emphasizes soundness as the key criterion for an analysis to be considered compliant: the applicant is required to provide justifications that the method never asserts that a property is true when it may not be true.

DO-333 acknowledges that objectives of reviews and analyses of *source code* can be achieved using formal methods, provided a formal semantics is well-defined at source code level. Such objectives include: compliance with the software architecture (correct data and control flows), accuracy and consistency (stack and memory usage, floating-point arithmetic, resource contention and limitations, worst-case execution time, exception handling, use of uninitialized variables, and data corruption due to task or interrupt conflicts). It also acknowledges that some verification objectives traditionally addressed by testing *executable object code* can be achieved using formal techniques. Such objectives include robustness to complex incoming data, freedom from arithmetic overflows, data corruption, inadequate numerical resolution, incorrect responses to missing or corrupted input data, incorrect handling of exceptions, arithmetic faults, violations of array limits. Such analyses should be performed either on executable object code, or at source code level, provided that property preservation can be demonstrated between source and executable code. We refer the reader to [6] for more information on DO-178C.

2.3 Static analysis at Airbus

Several formal techniques are currently being used operationally as part of the verification processes of avionics software. In order to verify functional properties, program proof techniques have been successfully introduced to replace unit testing on some software subsets. They are used for certification credit [12] on small sequential C codes. In contrast, the verification of non-functional properties requires automatic analyses that scale up to very large programs. As a consequence, static analysis based on abstract interpretation [8] is currently used industrially for certification credit on many avionics software products developed at Airbus, to compute safe upper-bounds of stack consumption or worst-case execution time [19], or to verify data and control flows [10].

Among major non-functional properties of interest is freedom from run-time errors, i.e., integer or floating-point overflow or division by zero, array out of bounds, invalid pointer dereference, etc. Many commercial bug finders allow for the automatic detection of possible run-time errors. While useful in a lightweight debugging approach to help detect systematic errors, such tools cannot be used for verification purposes. Indeed they implement unsound analysis methods, based on (implicit and incorrect) simplifying assumptions, and thus do not attempt to provide any assurance that code free from warnings will actually run without errors.

In contrast, semantic based static analyzers allow for automatic proofs of absence of run-time errors on sequential and synchronous software written in the C language. For instance, Astrée is being used industrially at Airbus on safety-critical synchronous control/command programs [11] prior to certification. These programs are large (up to 650 000 lines of C), and perform intensive floating-point computations. They are certified with the highest DO-178 *Development Assurance Level* (DAL A). Airbus plans to claim certification credit from the use of Astrée in the near future, in order to alleviate intellectual reviews and analyses of source code. To this aim, Astrée will have to undergo a dedicated qualification process, as defined by the DO-178 standard.

In addition, extensive experiments [3] are being conducted with the CompCert formally verified compiler to allow optimizing compilation of DAL A software, while guaranteeing semantic preservation between source and executable code. This guarantee will enlarge the scope of sound source code analyzers to also achieve verification objectives that traditionally require a machine code level analysis. In particular, it will enable Airbus to use Astrée to alleviate robustness testing, but also remove part of the local robustness code (proved to be unnecessary), with obvious benefits for worst-case execution time and structural coverage analysis.

3. CONCURRENT AVIONICS SOFTWARE

As of today, the scope of sound formal verification has been mostly limited to sequential and synchronous software. This encompasses the most safety-critical systems of Airbus aircraft, e.g. flight control systems. Such systems are designed for dedicated specialized computers running avionics

software on simple, deterministic architectures, in order to ease verification. For instance, Airbus fly-by-wire control-command software is bare metal synchronous programs.

In contrast, for less safety-critical aircraft functions, the required level of automation, HMI comfort, and system interoperability and configurability tends to increase from one aircraft generation to the next, resulting in an exponential increase of the complexity of the underlying embedded computers, networks and software. Moreover, considering the steady pressure on cost and weight reduction, such avionics functions tend to be integrated into generic computers, as shown by the current trend for Integrated Modular Avionics [23]. As a consequence, the development of such complex functions leads to more sophisticated designs, involving synchronizations between concurrent tasks, shared resource management, etc. They are implemented in asynchronous software, composed of a set of concurrent threads interacting in a shared memory. Traditional verification techniques, based on reviews, analyses and tests, are especially inefficient on asynchronous programs, because of the huge number of thread interleavings to be considered. Nonetheless, despite the limited impact of potential failures of related systems on aircraft safety, software correctness is of paramount importance for efficient aircraft operation and maintenance, and thus on aircraft availability and profitability. Formal techniques, e.g. static analysis, would thus be especially useful for scalable verification of asynchronous software.

AstréeA is the first example of such a sound static analyzer. It is an extension of Astrée aiming at proving the absence of run-time error in asynchronous multithreaded C software. The current underlying model matches that of asynchronous applications developed at Airbus. Such applications run on a single mono-core processor, on top of commodity real-time operating systems, implementing a preemptive, priority-based, real-time scheduling policy, e.g. the ARINC 653 [2] standard or *POSIX threads real-time* [14].

Application software is analyzed for a given specification of the operating system, and is thus sound for all operating systems meeting this specification. In practice, such specifications are formalized by means of a library of stubs for the API functions of the operating systems, which are written in C with dedicated AstréeA primitives. Some characteristics thereof may ease static analysis: for instance, all threads are created in an initialization phase (no dynamic thread creation), and dynamic memory allocation and recursion are forbidden by coding standards. However, some other characteristics are challenging. These are large data-intensive programs, from a few hundred thousand to a few million lines of C source code, composed of many nested loops processing string buffers as well as rich data structures based on pointers, e.g. statically allocated linked lists. Threads communicate through shared memory and standard synchronization primitives offered by the operating system (such as POSIX mutexes). Due to stringent (though usually not hard real-time) timing constraints, such software may rely on real-time scheduling to implement (implicit) critical sections, so as to save on synchronization primitives with significant worst-case execution time.

A more recent research direction (Sec. 5.3) is the verification of actual operating systems (or fragments of thereof).

4. THE ASTRÉEA STATIC ANALYZER

4.1 Abstract Interpretation

In the general sense, the safety verification problem consists in computing the set of program states reachable during all possible executions and proving that it does not include any unsafe state. Here, a program state denotes the current contents of the memory (a map from variables to values) and program position (PC, call stack). Program executions and unsafe states are defined based on the language standards, which must be translated from an informal English description to an unambiguous mathematical one.[1]

Industrial programs typically feature a large state space, which cannot be enumerated in practice. The core idea of abstract interpretation [8] is that it is often sufficient to reason at a more abstract, simpler level. Instead of considering sets of memory states (so called concrete elements), we can, for instance, consider an *interval abstraction* that only remembers the upper and lower bounds of each variable, and forgets the exact values reachable within these bounds. An *abstract element* then represents the set of memory states that satisfy these bounds, i.e., a property on states, but with a more compact representation (two numbers per variable instead of an unbounded set of memory maps). We will compute the reachable states entirely in this *abstract domain*. Naturally, some concrete properties cannot be represented in the abstract and must be approximated. The *soundness principle* formally states that any property computed in the abstract must also be true of all the concrete executions. For safety verification, this means that the computed abstract states must *over-approximate* the concrete ones. The over-approximation may, in certain cases, add spurious unsafe states that do not exist in the concrete semantics, i.e., we get a *false alarm*, which we want to avoid.

In practice, the reachable states of a program are defined by composing in a generic way atomic semantic operations from a small alphabet corresponding to language constructs (assignments, tests, etc.). It is thus sufficient to provide a sound abstract version for each atomic operation and combine them to compute the abstract semantics of any program. Abstract operators often induce a loss of precision that accumulates over the abstract computation, so that we may not be able to infer the most precise property expressible in the abstract domain (e.g., the interval domain may not find the tightest bounds). To reduce the false alarm rate, it is necessary to resort to more powerful (but more costly) abstractions. Many general-purpose abstractions are already available (for instance, we may replace intervals with polyhedra and infer linear relationships), and novel ones can be designed to specifically remove classes of false alarms.

4.2 Design Principles for Astrée and AstréeA

The AstréeA analyzer is based on the same design principles as the Astrée analyzer which it extends: both are specialized modular analyzers by syntax-directed interpretation of programs in a collection of abstract domains. We recall here briefly these common principles (an in-depth presentation can be found in [4]), while Sec. 4.3 is devoted to the extension to concurrency implemented in AstréeA.

[1]More precisely, ambiguity in the standard can be rigorously formalized using a non-deterministic semantics, which defines the set of all possible executions. This allows verifying, in one analysis, the correctness with respect to several interpretations of the standard. Non-determinism is also useful to model interactions with an unknown environment.

4.2.1 Concrete semantics

Astrée analyzes a large subset of C, including integers, floats, pointers, structured data-types, loops, gotos. It excludes notably: dynamic memory allocation, recursion, and long jumps. The concrete semantics is based on the C standard and the floating-point arithmetic standard. The analyzer reports all run-time errors, including: arithmetic overflows, invalid operations, invalid dereferences, assertion failures. The C semantics is notable for being under-defined: it leaves a lot of room for implementation choices and maps errors to undefined behaviors which have a truly random (possibly catastrophic) outcome. To fit more closely programmers' expectations, Astrée allows specializing the analysis by describing the actual implementation, among reasonable choices, for unspecified or undefined behaviors. For instance, by default Astrée reports signed integer overflows and continues the analysis assuming the wrap-around result.[2] The semantics of floats includes a sound model of rounding errors, as well as infinities and not-a-number. Astrée's pointer and memory semantics is also very lax and low-level, allowing unrestricted pointer arithmetic, casts, union types, and even type-punning.

4.2.2 Syntax-directed interpretation

Astrée functions literally as an interpreter, except that instead of propagating a single environment, it propagates an abstraction of a set of environments. Astrée's iterator traverses the syntax tree of the program, starting from the entry of the main function. Complex control structures are handled by induction on their sub-components and, ultimately, the iterator calls the abstract domain only on atomic statements, such as assignments and tests:

– For tests `if (`c`)` s_1 `else` s_2, both branches s_1 and s_2 are interpreted, after filtering the current abstract environment by, respectively, the condition c and its negation $\neg c$, and then the outputs of the branches are merged, using an abstract version of the union of state sets (e.g., the interval hull in the interval domain).

– Loops present the main difficulty for automated verification: they generate a large, possibly infinite number of executions of unbounded length that cannot be all explored explicitly, and must thus be approximated. In abstract interpretation, a loop `while (`c`)` s is handled by iteration: starting from the abstract environment before the loop, we accumulate the iterated effect of the loop body s filtered by the loop condition c until reaching a fixpoint. A special operator, the *widening* $\triangledown$, is used to accelerate the iteration and reach a fixpoint in a finite, generally small, number of steps. For instance, the standard interval widening enlarges unstable bounds to the type's extreme values in one step, where they cannot grow further. Upon stabilization, the abstract environment represents an inductive loop invariant. The analysis continues after the loop using the invariant filtered by the exit condition $\neg c$.

– Functions are analyzed by interpreting the body of the function at every call site. The result is a fully flow-sensitive and context-sensitive analysis, and is hence very precise. Such precision can come at a cost as full sensitivity does not scale up for programs with complex control flow or recursive functions; however, for embedded software, which employs neither, it achieves a sweet spot between precision and scalability.

[2] Considering that erroneous executions halt the program would greatly simplify the analysis, but is not acceptable for a safety analysis. In case the overflow was intentional, it may cause the analysis to miss a more serious error after wrap-around.

4.2.3 Collections of domains

Astrée does not use a single monolithic abstract representation of memory states, but rather a collection of interacting abstract domains, which provides improved scalability and flexibility to the analysis design. Astrée uses a stack of domains, corresponding to different levels of complexity in C expressions. Each domain handles specific C constructions, abstracts a specific aspect of the memory, and uses this information to dynamically translate expressions using simpler constructions for lower domains to handle. A memory domain decomposes each variable into a collection of scalar cells (integers, pointers, floats), and resolves dereferences, structures, arrays, and union accesses. A pointer domain maintains the base variables targeted by each pointer cell, and translates pointer arithmetic into integer arithmetic on the offset. The numeric domains are left with abstracting numeric cells and need only handle expressions containing integer and float values. The numeric abstraction is actually composed of a large set of domains, as software performs a huge variety of computations that lead to different flavors of numeric properties that no single domain can maintain efficiently nor conveniently.

Information about the abstract state is distributed among these domains, which collaborate to achieve the analysis of every single C statement. The domains communicate by transferring information expressed in a set of common properties (such as variable bounds), on a per-need basis to ensure efficiency.

4.2.4 Specialization

There is no universal abstraction able to analyze adequately all programs. Astrée is based on a specialization principle: it contains abstractions specifically adapted to analyze very well a given kind of programs, embedded avionics control-command software, while it is sound but possibly imprecise or not scalable on other software. It was designed by starting from a generic scalable interval analyzer and analyzing a selected code in the family of interest. Initial analyses suffered from many false alarms. They were removed by a manual process consisting in inspecting the origin of each false alarm, determining which property was missed by the analyzer, and either designing a new abstract domain, if the property was not expressible in Astrée, or strengthening the abstract operators and domain communication, if it was. This resulted in an analyzer with no alarm on the code of interest, which remains efficient (by parsimony, more complex and costly abstractions are added only when necessary). Abstract domains are not specific to a single program but general enough to handle a programming pattern. We observed experimentally [5] that a specialized analyzer was able to handle a family of similar programs, requiring only slight adjustments of analysis parameters (such as widening aggressiveness) which can be performed by industrial end-users [11].

The complete report of this experiment is described in [5]. We give here only two examples of additional abstractions. Firstly, the precise analysis of loops required the addition of relational domains. As general polyhedra are

not scalable, we used instead the octagon domain [16], an expressive enough domain. A reasonable tradeoff between precision and scalability was further achieved by applying the domain parsimoniously, to selected variable packs. Secondly, control-command software features digital filter computations, which require quadratic invariant relations to be precisely analyzed. Feret thus designed a specific ellipsoid domain [13] for this task. The octagon domain is of general use and was employed in AstréeA, while ellipsoids are specific to control-command software and not reused in AstréeA.

4.3 Extension to concurrent programs

AstréeA extends Astrée to handle concurrent embedded C programs. It reuses its iterator and abstractions, and adds new ones, as discussed here.

4.3.1 Concrete semantics

AstréeA supports a generic concurrency model. Program executions have two phases: firstly, an initialization phase, able to execute sequential code and create threads, and a second phase, where threads execute concurrently but thread creation is no longer allowed. This matches precisely the ARINC 653 semantics [2] as well as current practice in avionics software. AstréeA analyzes both phases for run-time errors; in addition, it collects the set of threads created during the sequential phase and checks that no thread is created during the concurrent phase. The set of threads is thus not fixed beforehand but discovered by the analyzer. In the second phase, the semantic of the program is an interleaving of atomic execution steps (such as assignments) from each thread. AstréeA assumes a mono-core real-time execution model: only the unblocked thread of highest priority can run. This matches current avionics practice and permits a more precise analysis than considering true parallel execution (such an extension will be considered in the future). The execution model is fully preemptive: a high priority thread can enter a wait state, e.g. waiting for a resource to be available, let a lower-priority thread run for a while, and preempt it at any point of its execution when the resource becomes available. As a consequence, a large number of interleavings of concrete executions are possible. Threads execute in a shared memory: it is the analysis' responsibility to detect which global variables are actually shared and their possible values. AstréeA supports a small set of flexible but low-level primitives on top of which C stub code simulating high-level concurrency libraries can be written; for instance, it supports non-nesting mutual exclusion locks, on top of which more complex locks are constructed (Sec. 5). AstréeA reports data-races for accesses not protected by locks.

4.3.2 Thread-modular interpretation

Iterating over all possible thread interleavings is not feasible. AstréeA thus employs a thread-modular analysis approach, inspired from *rely-guarantee reasoning* [15]. It is sound with respect to all interleavings, scalable, and allows reusing the abstractions developed for Astrée. This analysis is composed of a sequence of iterations. In the first one, each thread is analyzed as an independent sequential program, ignoring the effect of other threads (which is unsound), but collecting the effect it has on the global variables. Starting from the second step, each thread is reanalyzed, but now taking into account the interferences computed in the last step, which discovers new behaviors, and possibly new interferences. The analyses are iterated until the interferences stabilize, at which point we have explored a superset of all possible program behaviors and reported all the possible errors. The increasing sequence of interferences is stabilized efficiently by using a widening. When several threads execute the same code, it is only necessary to analyze them once per iteration; this enables the efficient analysis of programs creating an arbitrary (possibly unbounded) number of instances of some threads. The theoretical foundation of this method is presented in [17, 18].

4.3.3 Interference abstraction

Thread-modular analyses introduce the notion of interference. As for program states, interference sets are not computed exactly but rather over-approximated at some abstract level. Interferences require abstract domains of a different nature because they represent *state transitions*. Initially, AstréeA used a simple interference abstraction, which only remembers, for each variable and thread, an interval over-approximating the values stored by that thread into that variable. This technique is very scalable and sufficient in many cases. It was refined by adding a measure of flow-sensitivity and relationality. Firstly, AstréeA takes mutual exclusion into account by partitioning according to the locks taken and thread priorities, thus removing spurious interferences. This can be further improved by using relational domains (such as octagons [16]) to track *lock invariants*, i.e., relations between variables that are maintained by locks: they may be temporarily invalidated inside locks, but are restored before releasing the lock, hence, the violation remains invisible as long as all accesses are correctly protected (which is checked by AstréeA). As second example, AstréeA includes a domain tracking which variables are incremented. This is an example of specialization: the domain was added specifically to analyze code with clocks that are sampled and integrated, as shown in Fig. 3. The correctness of the program depends on the fact that, between two successive reads of a clock by a thread, the clock can only be incremented by other threads and never decremented. Additionally, these abstractions can be proved to be sound with respect to widespread weakly consistent memory models (such as *Total Store Ordering*, used notably for multi-core intel x86).

4.3.4 Additional abstractions

In addition to being concurrent, the codes we consider in our study are more general than the control-command codes considered by Astrée. Handling them precisely and efficiently required some specialization of the abstractions.

Firstly, we need to analyze software making an extensive use of large data-structures, such as nested arrays and structures. We have thus enriched the abstraction with a notion of *dynamic array folding*: a contiguous sequence of array elements can be represented with a single abstract element. Arrays start unfolded, and are folded dynamically when encountering an imprecise pointer targeting many array elements. This improves the time and memory efficiency without jeopardizing precision. We also developed a new numeric domain for offsets able to represent succinctly complex access patterns (e.g., $\{ 4 \times i + 100 \times j \mid i \in [0, 10], j \in [0, 10] \}$ is an access to a part of a matrix flattened, as is common in C, into a uni-dimensional array). Similar dynamic abstractions are employed to merge dynamically several concrete variables into an abstract one, and we also defined a pointer

Sec.	Size	Added	Select.	Time	Mem.	Ctx.
5.1.1	2.1 M	5.2 K	99.94%	24 h	27 GB	A
5.1.2	1.9 M	2.4 K	99.56%	154 h	18 GB	I
5.1.2	2.2 M	2.3 K	99.52%	160 h	23 GB	I
5.2.1	31.8 K	2.2 K	97.28%	50 mn	0.6 GB	I
5.2.2	33.1 K	1.2 K	97.18%	35 h	2.5 GB	I
5.3	—	1.5 K	94.5%	22 mn	0.9 GB	A

Figure 1: Summary of case studies with the original size (in lines), the lines of code added for the analysis, the selectivity, the time and memory consumption, and the analysis context (academic or industrial setting).

widening to accelerate loops accessing linked lists.

Secondly, the case studies contain more complex control than usually found in control-command software, including deeply-nested functions and loops. A drawback of interpretation by induction on the syntax is that, to analyze nested loops, the inner loop must be completely reanalyzed for each iteration of the outer loop. We solved this problem by caching loop invariants, and reusing them to bootstrap subsequent analyses of the same loops, reducing the number of iterations needed to find a new invariant. The cache also accelerates the iterations required to stabilize interferences.

5. CASE STUDIES AND EXPERIMENTS

We have applied AstréeA to the analysis of various industrial concurrent cockpit avionics software. The case studies are described in the following sections, and the results are summarized in Fig. 1. Additionally to the size of the use cases, we indicate the number of lines we had to add or modify to perform the analysis (which gives an idea of how much work is required to prepare a new analysis), and the selectivity (a measure of precision defined as the percentage of alarm-free lines). Some information is omitted due to non-disclosure agreements. Some analyses were performed by researchers in an academic setting, and others by engineers in an industrial setting. A preliminary version of the first case study (Sec. 5.1.1) has been presented in [17]; it is presented here updated and in more details. The other cases studies are new.

5.1 Analysis of ARINC 653 Applications

5.1.1 Primary case study (DAL C)

Our first analysis target is a large embedded avionics code, featuring 15 threads and 2.1 M lines of C (after preprocessing and removing redundant declarations). This DAL C[3] application monitors and aggregates a large number of data coming from ports and displays synthetic summaries in an interactive way in the cockpit. It contains 100 K lines of hand-written C code performing a variety of tasks, including: parsing binary messages, formatting strings, managing and sorting arrays and lists, as well as 2 M lines of automatically generated code, in particular reactive synchronous logic *à la* SCADE running in threads concurrently with other tasks and featuring boolean, integer, and float computations. We analyze almost completely the original application; to simplify, we omitted the custom error handler, which eventually halts the application and thus does not add errors.

[3]DAL (*Development Assurance Level*) C denotes a software whose failure has a major incidence on the aircraft; DAL A are the most critical applications; DAL E are the least ones.

ARINC 653 model.

Astrée and AstréeA are whole-program analyzers, that take as input programs without undefined symbols. This is not a problem for Astrée as it focuses on synchronous programs that run on bare metal, and are inherently self-contained. On the contrary, the programs analyzed by AstréeA interact with an operating system through function calls. For instance, the applications considered in this section run on ARINC 653, a specification for embedded avionics real-time operating systems [2]. We analyze the application without the actual system implementation, but with a hand-crafted model of its specification. This has several benefits: the analysis is sound with respect to any system implementation obeying the specification; the analyzed code does not exhibit hard-to-analyze low-level features encountered in system implementations; and we can enrich the specification with assertions to check that the application obeys API contracts.

Internally, AstréeA supports several kinds of objects and a set of primitives to manipulate them, including for instance:

– Threads,[4] which must be registered during a sequential initialization phase with a directive `__ASTREE_create_process(i,p,f)`. They are assigned by the program an integer identifier `i`, an integer priority `p`, and an entry point `f`. Other primitives, taking `i` as argument, can change the thread state (e.g., stopping, pausing, yielding, etc.).

– Mutexes, which are also denoted by integers. AstréeA assigns a mutex to every 32-bit integer `i`, and so, mutexes do not need to be registered before being used. For instance, the primitive `__ASTREE_lock_mutex(i)` will simply lock the mutex identified by `i`.

ARINC 653 objects are, however, more complex. They must be created during initialization and have a rich set of API functions as well as properties (such as a name). The model thus consists of an abstract implementation of each API function written in C enriched with built-in primitives, so that the combination of the analyzed application and the model is a stand-alone program with no undefined symbol. Figure 2 gives, as example, a simplified version of our stub for semaphore locking, and illustrates certain interesting points: we validate arguments to report API violations (`__ASTREE_error`); we distinguish between locking with and without a timeout (`__ASTREE_lock` and `__ASTREE_yield` include a non-deterministic wait allowing rescheduling lower-priority threads[5]); we model both the case where a timeout occurs without the semaphore being locked and a successful locking and select between them with a non-determinism choice (`__ASTREE_rand`) to ensure that the analysis soundly consider both cases. Identifiers for ARINC 653 and AstréeA objects (such as `SEMAPHORE_ID`) are allocated at creation time and all their properties are maintained in plain C arrays. The built-in support for non-determinism makes it very easy to model soundly an unknown environment; for instance, we model reading a message from a port connected to another application, which is not analyzed, as reading

[4]Execution units are actually called *processes* in ARINC 653, but behave like POSIX threads as they execute in a shared memory. We call them *thread* here for consistency.

[5]Note that our system is fully preemptive: the current thread can return from `__ASTREE_yield` by interrupting a lower priority thread at any point of its execution.

```
void WAIT_SEMAPHORE(SEMAPHORE_ID_TYPE SEMAPHORE_ID,
                    SYSTEM_TIME_TYPE TIMEOUT,
                    RETURN_CODE_TYPE * RETURN_CODE) {
  *RETURN_CODE = NO_ERROR;
  if (SEMAPHORE_ID<0
      || SEMAPHORE_ID>=NB_SEMAPHORE) {
    __ASTREE_error("invalid semaphore");
    *RETURN_CODE = INVALID_PARAM;
  }
  else if (TIMEOUT>0) {
    if (TIMEOUT==INFINITE_SYSTEM_TIME_VALUE
        || __ASTREE_rand())
      __ASTREE_lock_mutex(SEMAPHORE_ID);
    }
    else {
      __ASTREE_yield();
      *RETURN_CODE = TIMED_OUT;
    }
  }
  else {
    if (__ASTREE_rand()) *RETURN_CODE = NOT_AVAILABLE;
    else __ASTREE_lock_mutex(SEMAPHORE_ID);
  }
}
```

Figure 2: Stub implementation for locking a semaphore in ARINC 653.

non-deterministic values. We implemented 66 ARINC 653 system calls in a 3.9 K lines model. Note that the correction of the analysis depends on the correction of the stubs: they must include all the behaviors of the actual OS implementation. However, the quantity of code to be trusted is small (1 stub line for 500 lines of application), stubs can be reused from one analysis of an ARINC 653 application to the other, and are easily understandable by engineers, which improves our confidence in their correctness.

Analysis results and refinement.

Currently, our analysis exhibits 1095 alarms, i.e., a 99.94% selectivity. The selectivity is higher (99.98%) on automatically generated parts than in manual parts (99.2%), which is expected as the latter is more complex and less regular (although much smaller). Early experiments using Astrée's state abstractions and a non-relational flow-insensitive interference abstraction reported more than 12000 alarms [17], and this number was decreased by specialization in AstréeA. Similarly to our experiments with Astrée this included the design of new abstractions. In some cases, however, AstréeA already contained adequate domains and it was sufficient to configure the analyzer to use them on specific variables and program parts (by default, costly domains are only used parsimoniously for scalability reasons). An example is given in Fig 3: the three variables `Clock` (a monotonic clock), `Prev` (its previously sampled value), and `Dst` (a time accumulator) must be related in the octagon domain. This information is communicated to the analyzer through the insertion of a directive, `__ASTREE_octagon`.

Additional directives are used to gain precision by unrolling loops, handling arrays in a field-sensitive way, or enabling path-sensitivity. We added 2302 directives, most of which (2183) appear in manual code; directives in automatically generated code appear in macros that are massively duplicated by macro-expansion. Ultimately, adapted heuristics can be incorporated to control the precision and achieve

```
#define CLOCK(Dst,Enable,Clock) {          \
  static unsigned Prev;                    \
  __ASTREE_octagon(Dst,Prev,Clock);        \
  if (Enable) {                            \
    Dst += Clock - Prev;                   \
    Prev = Clock;                          \
  }                                        \
  else Prev = Clock;                       \
}
```

Figure 3: Time accumulator. The precise analysis requires proving that other threads only increment `Clock`, and tracking relations with the octagon domain (hence the `__ASTREE_octagon` directive).

fully automated analyses of unmodified source code but, for fast prototyping, the ability to manually insert syntactic directives is convenient. Unlike designing a new domain, which is a research activity, directive insertion can be performed by industrial end-users, to adapt the analysis to new software.

5.1.2 Additional case study (DAL C)

Our second application of AstréeA is the analysis of an application featuring 11 threads, and implementing similar functions as the first one, but for a different aircraft. As a consequence, several interfaces and functionalities differ significantly. It has, however, a similar overall structure and sets of threads, and about 20% of the hand-written source code is common. The automatically generated code is different, though its structure is very similar. Two major (non-consecutive) versions of this programs were analyzed. The first one is composed of 1.9 M lines, among which 155 K are hand-crafted. The second one is composed of 2.1 M lines, among which 160 K are hand-crafted.

This case study was conducted in an industrial environment. The first version was analyzed by one avionics software engineer experienced in static analysis with Astrée and AstréeA, while the second was analyzed by a software engineer experienced with Astrée, but with no prior exposure to AstréeA. This case study benefited from the specialization work performed for the primary case study. The ARINC 653 model was reused with minor adaptation (about 8%) and 7 new API functions were modelled. Much of the analysis refinement effort consisted in adapting AstréeA directives from the first case study to the similar (but different) application software: 2178 directives were used for the first version, with limited adaptation for the second one (about 5%).

Currently, our analyses exhibit 8573 alarms, i.e., a 99.56% selectivity, for the first version, and 10735 alarms, i.e., a 99.52% selectivity, for the second. The selectivity is again higher for automatically generated code (99.79% for both versions) than for hand-written code (96.88% for the first version, and 95.97% for the second). Analyzing these applications is 6 times slower and slightly less precise than our primary case study (Sec. 5.1.1), for a similar code size. Indeed, our first case study benefited from a larger code-specific specialization effort, which improved not only precision but also efficiency, by carefully selecting useful abstractions only.

5.2 Analysis of POSIX Applications

AstréeA was extended to analyze a family of embedded avionics applications developed at Airbus and running under POSIX systems. The POSIX standard is more general and

more complex than ARINC 653. However, Airbus applications rely only on a subset of POSIX similar to ARINC 653. This subset includes POSIX extensions such as Threads, Thread Execution Scheduling, Realtime, Message Passing, Semaphores, Timeouts and Timers. The applications use a variety of POSIX objects such as processes, threads, mutexes, condition variables, message queues, and semaphores.

They also use sophisticated objects not directly related to parallelism, including regular files, named pipes, shared memories, and environment variables. Peculiarities of the semantics of some of the associated primitives make analysis in the large especially challenging. For instance, the `shmat` function returns a valid pointer upon successful completion, or `-1` in case of failure. To avoid false alarms, we use partitioning techniques which allow the analyzer to distinguish, by path sensitivity, between normal and error cases. To facilitate the analysis by non expert engineers in an industrial context, we chose to restrict the scope from complete POSIX applications to so-called subsets of them. These subsets exist independently from the needs of static analysis, as part of the (piece-wise) development strategy at Airbus, and are subject to software integration testing: an application is typically decomposed into 5 to 10 subsets.

5.2.1 Analysis of a DAL E application

This case study was conducted in by an avionics software engineer experienced in static analysis, with extensive support from AstréeA developers. It aimed at analyzing a subset of an avionics application developed at Airbus with low safety-criticality, composed of 300 K lines of hand-crafted C code. This application implements embedded system failure monitoring and correlation to optimize aircraft maintenance on ground. It is composed of 70 threads, some of which have the same priority, and sometimes the same entry point. It is a complex program performing intensive string processing, and traversing large arrays of structures by means of nested loops and pointer arithmetics. The case study focused on a subset of this application composed of 7 threads and 32 K lines of C. 790 lines of stubs were developed to abstract away the threads (or parts of threads) excluded from the analysis. Such stubs are non-deterministic C programs using AstréeA directives. This work requires a precise knowledge of the design of the software, and can be inspired by existing simulations developed for integration testing purposes.

POSIX model.

As for ARINC 653, a library of stubs was developed to model the POSIX primitives used by the program to be analyzed. 1200 lines of C code and AstréeA directives were written to model 45 API functions, among which 31 are related to multi-threading. These primitives handle threads, mutexes, condition variables, message queues, and semaphores, but also time, string management and I/Os.

Analysis results and refinement.

The first static analyses yielded about 1300 alarms, while not covering all accessible code. Therefore the model of the unanalyzed threads was refined, e.g., adding missing initialization for correctness, and tuning the sets of values that may be written for precision. AstréeA provides useful indicators for data and control coverage, such as the set of unanalyzed control points and invariants reduced to singletons (i.e., variables deemed stuck to their initial values).

```
if (b==0) {          | if (b==1) {
  access(&X);        |   access(&X);
  b=1;               |   b=0;
}                    | }
```

Figure 4: Mutual exclusions between threads with arbitrary priorities

The analyzed program was annotated with 197 AstréeA directives. The subsequent analyses cover all the reachable control points, while yielding 865 alarms.

Remaining alarms.

Most of the remaining alarms are not related to parallelism but are caused, e.g., by complex string processing and pointer arithmetic. However, a concurrency-related source of imprecision is the use, in some cases, of *ad hoc* boolean flags playing the role of mutexes to implement critical sections. This non standard choice is motivated by performance or system configuration constraints. For instance, the algorithm of Fig. 4 synchronizes the access to `X` between two threads with arbitrary priorities. The first (resp. second) thread accesses `X` only if the boolean flag `b` is set to 0 (resp. 1), and then resets `b` to 1 (resp. 0). Atomic access is guaranteed by the size of data and the use of the `volatile` qualifier for the shared variables `b` and `X`. This synchronization mechanism is incorrect when considering weakly consistent memories, but it is correct for a real-time scheduler on a single mono-core processor, provided that complementary verification activities are conducted to ensure that the compiler does not suppress nor reorder volatile accesses.

5.2.2 Analysis of DAL C middleware

The next use case was conducted in an industrial environment, by an intern with no prior exposure to static analysis. This use case considers a subset of a complex avionics software platform. The full platform features 11 privileged multithreaded POSIX processes, running on top of an embedded real-time operating system offering POSIX services to applications. The platform is composed of 400 K lines of C, more than 80% of which are hand-written. It implements a variety of communication protocols, as well as human-machine interface functions.

The case study addresses the process in charge of interactive cockpit displays. It is composed of 33 K lines of C and 4 threads. The process reads a set of constant binary configuration files at start up, which must be taken into account to prove the correctness of the application. Hence, we model precisely a part of the file system service, including the constant file data. The library of stubs of POSIX primitives from case study 5.2.1 was also extended with 25 new system calls used by this process, including: named pipes (used to read inputs) and shared memories (to communicate with other privileged processes of the platform). Significant adaptations of the existing primitives were also necessary, as the underlying operating system implements a different version of POSIX, with different implementation choices. Altogether, 70 API functions were modelled in a POSIX stub library comprising 1000 lines of C with AstréeA directives. The model of the environment of the analyzed process (including a model of the non-analyzed processes) is only a few tens of lines long and models asynchronous

writes into pipes and shared memories. The program itself was annotated by means of 228 AstréeA directives. Current analyses exhibit 932 alarms, i.e., a 94.5% selectivity. Ongoing experiments target other processes of the same avionics platform.

5.3 Analysis of an OS fragment

Our last case study is an on-going project to analyze a part of an embedded operating system with AstréeA. We focus on the component providing ARINC 653 entry points to applications. We analyze almost all of the component, including the underlying implementation of preemptive multithreading through a priority-driven scheduler and the various communication objects, timers, error handling mechanisms, but excluding machine-language context switching.

Stubbing and modeling.

The ARINC 653 implementation communicates to applications as they perform system calls. To achieve a standalone analysis of the implementation that takes into account all its execution contexts, we have written a 1.3 Kloc *analysis driver*. Firstly, it takes care of calling ARINC 653 initialization routines (this is normally performed as part of OS bootstraping, which we do not analyze as it is in assembly). Secondly, it provides abstract configuration tables that declare an arbitrary number (up to the system limit) of ARINC 653 objects of arbitrary name and properties. Normally, the OS is compiled with a fixed set of *partitions* (conceptually similar to POSIX processes), each composed of a fixed set of threads, described by a table fixed at compilation. Instead of a fixed table, our analysis considers the set of all valid tables. Thirdly, it models an arbitrary application execution by issuing all possible sequences of ARINC 653 system calls with all possible argument values (within the range authorized by the specification). We use non-determinism extensively to achieve, in a single analysis, a sound coverage of a large number of execution environments. The result is an analysis that is sound for any multi-partition multithread application respecting the API contract. Note that, when analyzing ARINC 653 applications (Sec. 5.1), we explicitly checked API contracts by inserting into the OS stubs assertions that are verified by the analyzer. Thus, separate analyses of the OS and the applications ensure the safety of the whole system. An additional 150 line stub provides a C implementation for assembly functions. Most notably, threads and context switching are implemented in the OS by, respectively, allocating a stack for each thread and switching stacks, which cannot be expressed in C. In our model, we associate instead an AstréeA thread to each stack and model stack switching as a non-deterministic wait allowing arbitrary threads to run. Handling concurrency is critical to model soundly all the cases where several threads and partitions access concurrently the ARINC 653 implementation.

Results.

An early experiment conducted in an academic setting achieved a 94.3% selectivity rate within 22 mn computation time and 900 MB memory consumption. The false alarms come from data-structures and program patterns never encountered by AstréeA before, and for which it lacks dedicated abstract domains. We stress the fact that a better precision can be achieved, in the future, by specializing the analysis to this new class of software. This experiment validates the fact that sound static analyses checking significant parts of embedded operating systems are possible, and that AstréeA provides a promising architecture to achieve it.

6. RELATED WORK

The problem of verifying concurrent systems has been considered by formal methods for decades. Many works have been inspired by Jones' rely–guarantee reasoning [15], primarily to design deductive methods, although AstréeA's static analysis applies similar principles to abstract interpretation instead. Deductive methods put a heavy burden on the user by requiring manual code annotations and, especially, providing, for each thread, a model of its environment (i.e., of the other threads). Moreover, deductive tools for concurrent programs are not as mature as for sequential programs. Sequential deductive methods are used at Airbus to replace unit-testing, as interfaces must be developed for such tests anyway. Model checking can also handle concurrent programs, but suffers from state (in explicit state methods) or path (in SAT-based methods) explosition problems, that have only been partially addressed by partial-order reduction methods. Hence, popular software model checkers often explore only a part of software behaviors (in bounded or context bounded model checking), which is useful to find errors but cannot serve for verification. An application to the model-checking of ARINC 653 software is reported in [21]. Static analysis by abstract interpretation allows, on the other hand, designing sound and automated tools, making it well suited for industrial use, provided that scalability and precision objectives can be reached. Apart from the work on Astrée and AstréeA discussed at length in this article, thread-modular static analysis has been considered in other works. The Goblint analyzer [22] focuses only on detecting data-races. The POSIX Thread analysis of Carré and Hymans [7] uses non-relational abstractions, which limits its precision. We refer the reader to [9] for an in-depth comparison of various formal methods and static analysis techniques. Dynamic analyzers, such as Valgrind, are also popular tools, but have a different purpose: they can be used for testing and debugging but, as they are not sound, cannot be employed to gain certification credit and replace less cost-effective verification methods.

7. CONCLUSION

The use of formal methods in embedded avionics software is now sanctioned by Certification Authorities, in the DO-178C and DO-333 international standards. They can thus complement or replace some reviews, intellectual analyses and tests required for certification. Formal methods are particularly satisfying as they provide strong guarantees based on rigorous mathematical theories. Their use is however subject to the availability of tool sets that both comply with DO-333 requirements and are cost-effective in an industrial context. Sound static analysis tools present a compelling choice: for instance, the Astrée industrial analyzer is being used at Airbus to help verifying non-functional safety properties of large sequential C codes. For concurrent software, however, no industrial tool exists. We have discussed AstréeA, a prototype extension of Astrée to concurrent C code, presented several case studies analyzing a variety of concurrent avionics software, and reported promising experimental results. A key factor is that 4 out of our 6 studies were suc-

cessfully conducted by engineers in an industrial context, which shows that our tool is mature and cost effective.

Future work.

For static analysis of concurrent software to be fully exploitable for certification in avionics industries, two lines of work need to be conducted.

It is necessary to improve the precision, automation, and generality of current analyzers. Reducing the number of false alarms is critical because, for certification, each alarm must be proved to be spurious by other, more costly means (such as code review). In the limited scope of synchronous fly-by-wire, Astrée was able to achieve the 0 false alarm goal on specific code. For more complex, concurrent, less critical software, a more modest objective is acceptable, i.e.: one alarm for every 500 lines in hand-written code, and one alarm for 10000 lines of automatically generated code (which is almost but not quite reached in our case studies). Furthermore, the results reported here were achieved at the cost of a significant manual parameterization of the analysis. The automated parameterization techniques from Astrée must be adapted to the codes targeted by AstréeA to achieve a full automation. Finally, models of the underlying operating systems are necessary to conduct a sound analysis. Currently, only ARINC 653 and a subset of POSIX Threads are supported. Future work will consider additional models, including larger subsets of POSIX, alternate operating systems, or implementation-specific variants.

The analysis must also be integrated into industrial processes. A first step, which is underway, is to transfer AstréeA to a suitable tool provider company, able to provide support and extensions to industrial end-users. The analyzer then needs to be qualified by the industrial end-user, in the context of avionics software products to be certified. After this step, the source-level analysis of AstréeA will be usable to automate reviews and analyses of source code required by DO-178. To go further and alleviate robustness tests, DO-333 requires that a proof of soundness with respect to the binary is provided. An attractive solution on sequential software is to use a certified C compiler, such as CompCert [3], that ensures that a proof performed on the C source is also valid on the binary. For AstréeA, the proof of equivalence between source and binary must be extended to concurrent programs. Additionally, we must make sure that CompCert and AstréeA have equivalent formal notions of the semantics of C programs. We believe that, in the near future, certification objectives using formal methods can be achieved on concurrent avionics software. We are confident that sound static analysis, and in particular AstréeA, can also benefit other industries employing concurrent embedded software with similarly stringent certification processes.

8. REFERENCES

[1] DO-178C: Software considerations in airborne systems and equipment certification, 2011.

[2] Aeronautical Radio Inc. *ARINC 653. Avionics application software standard interface*, Mar. 2006.

[3] R. Bedin França, D. Favre-Felix, X. Leroy, M. Pantel, and J. Souyris. Towards formally verified optimizing compilation in flight control software. In *PPES 2011*, volume 18 of *OASIcs*, pages 59–68, 2011.

[4] 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*, number 2010-3385, pages 1–38. AIAA, Apr. 2010.

[5] 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 *PLDI'03*, pages 196–207. ACM, June 2003.

[6] D. Brown, H. Delseny, K. Hayhurst, and V. Wiels. Guidance for using formal methods in a certification context. In *ERTS'12*, 2012.

[7] J.-L. Carré and C. Hymans. From single-thread to multithreaded: An efficient static analysis algorithm. Technical Report arXiv:0910.5833v1, EADS, 2009.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, Jan. 1977.

[9] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Varieties of static analyzers: A comparison with Astrée. In *TASE'07*, pages 3–17. IEEE, June 2007.

[10] P. Cuoq, D. Delmas, S. Duprat, and V. M. Lamiel. Fan-C, a Frama-C plug-in for data flow verification. In *ERTS'12*. SIA, 2012.

[11] D. Delmas and J. Souyris. Astrée: from research to industry. In *SAS'07*, volume 4634 of *LNCS*, pages 437–451. Springer, Aug. 2007.

[12] S. Duprat, D. Favre-Félix, and J. Souyris. Formal verification workbench for airbus avionics software. In *ERTS'08*. SIA, 2008.

[13] J. Feret. Static analysis of digital filters. In *ESOP'04*, volume 2986 of *LNCS*, pages 33–48. Springer, 2004.

[14] IEEE Computer Society and The Open Group. POSIX API amendment 2: Threads extension. Technical report, ANSI/IEEE Std. 1003.1c-1995, 1995.

[15] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, Jun. 1981.

[16] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[17] A. Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *ESOP'11*, volume 6602 of *LNCS*, pages 398–418. Springer, 2011.

[18] A. Miné. Relational thread-modular static value analysis by abstract interpretation. In *VMCAI'14*, volume 8318 of *LNCS*, pages 39–58. Springer, 2014.

[19] J. Souyris, E. L. Pavec, G. Himbert, V. Jégu, and G. Borios. Computing the worst case execution time of an avionics program by abstract interpretation. In *WCET*, pages 21–24, 2005.

[20] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. pages 532–546, 2009.

[21] S. Thompson, A. Venet, and G. Brat. Software model checking of ARINC-653 flight code with MCP. In *NASA Formal Methods*, pages 171–181, 2010.

[22] V. Vojdani and V. Vene. Goblint: path-sensitive data race analysis. In *SPLST'07*, 2007.

[23] C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *DASC'07*, pages 1–10. IEEE, Oct. 2007.