Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée

Antoine Miné, Laurent Mauborgne, Xavier Rival, Jerome Feret, Patrick Cousot, Daniel Kästner, Stephan Wilhelm, Christian Ferdinand

Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée

Antoine Miné[2], Laurent Mauborgne[5], Xavier Rival[1,3], Jerome Feret[1,3], Patrick Cousot[4], Daniel Kästner[5], Stephan Wilhelm[5], Christian Ferdinand[5]

[1]École normale supérieure, Paris, France
[2]Sorbonne University, University Pierre and Marie Curie, CNRS, LIP6
[3]INRIA, France
[4]Courant Institute of Mathematical Sciences, NYU, New York
[5]AbsInt GmbH, Saarbrücken, Germany, http://www.absint.com

Abstract

We present an extension of Astrée to concurrent C software. Astrée is a sound static analyzer for run-time errors previously limited to sequential C software. Our extension employs a scalable abstraction which covers all possible thread interleavings, and soundly reports all run-time errors and data races: when the analyzer does not report any alarm, the program is proven free from those classes of errors. We show how this extension is able to support a variety of operating systems (such as POSIX threads, ARINC 653, OSEK/AUTOSAR) and report on experimental results obtained on concurrent software from different domains, including large industrial software.

1 Introduction

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 last years have seen the emergence of semantics-based static analysis tools able to detect run-time errors, such as the Astrée analyzer [6]. However, such tools cannot handle concurrent programs at all, or with the same level of soundness, coverage, and automation as sequential programs: they would not cover all potential process interleavings, or require the user to enter manually the set and range of shared variables, or miss support for concurrency primitives (such as mutexes) or the detection of concurrency-specific hazards (such as data races). We present here an extension of Astrée to analyze soundly and automatically concurrent software.

The article is structured as follows: first, in Sec. 2, we give an overview of sound static analysis and of Astrée. In Sec. 3, we explain the key concepts underlying our interleaving semantics, which makes it possible to analyze concurrent programs in a scalable and sound way, and report all run-time errors and data races. Section 4 discusses our support for several standard operating systems, enabling the automated analysis of software running under these OS. Section 5 discusses our experiments: the analysis of industrial avionic software, as well as preliminary results on ongoing experiments on OSEK software. Section 6 discusses related work. Section 7 concludes.

2 Overview of Astrée

Sound static analysis. Astrée discovers errors by inspecting the source code without running it. It traverses the program control structure and interprets program instructions according to the language semantics to build automatically a model of its executions. To ensure efficiency, the model must be approximated, but we take care to always use over-approximations. Thus, in contrast to most other static analyzers, Astrée makes sure that all possible program executions are taken into account: it achieves a full coverage of the whole control and data space of the program. For this reason, it is sound: whenever no error is reported, we are certain that no error can exist in the actual program executions either. Like all sound static analyzers, Astrée may report false alarms (notifications about potential run-time errors which do not occur in real program executions). An important design goal of the analyzer, reached in 2003, was to achieve zero false alarm on a significant class of sequential software: large industrial avionics control-command software [6].

Language. Astrée has been developed for safety-critical C programs and is based on the C99 standard [17]. It supports all C control structures and C datatypes, provides a stubbed C library and even supports dynamic memory allocation. The only notable limitations are that recursive calls will be detected and reported as an alarm without trying to analyze the recursive invocations; moreover, long jumps are not supported. As a recent extension, Astrée supports concurrency features, such as threads and locks (Sec. 3) and can analyze software running on top of operating systems implementing common standards (e.g., POSIX, ARINC 653 [4], OSEK/AUTOSAR [1], see Sec. 4).

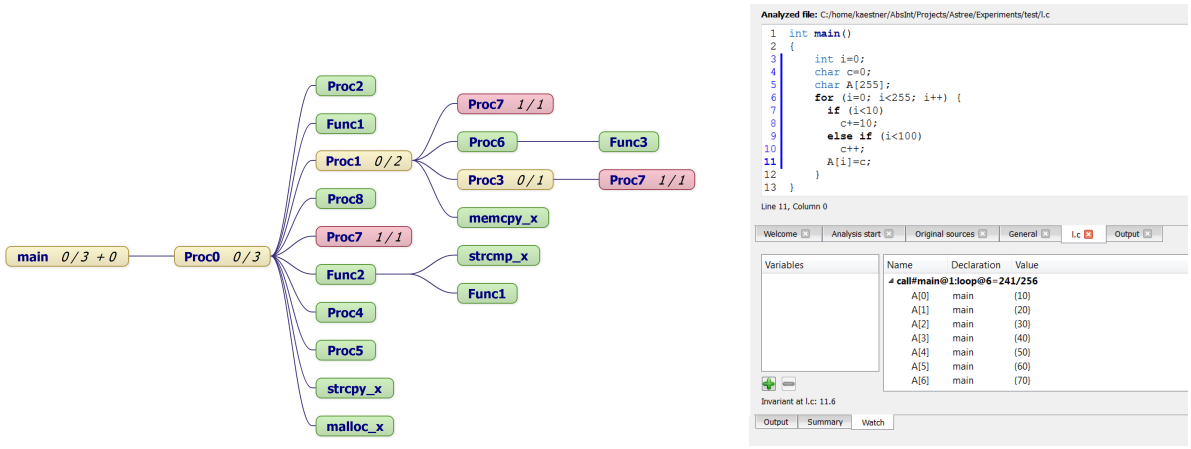
Semantics. The semantics of programs used by Astrée is based on the C99 standard [17]. However, the standard provides a high-level and abstract semantics, leaving many aspects of program behaviors implementation-defined, unspecified, or undefined. Implementation-defined features, such as the bitsize of integers can be configured. Moreover, Astrée employs a low-level memory semantics, which is aware of the bit-level representation of objects, that can be configured as well [21] (cf. also Sec. 3). This gives a semantics to undefined behaviors, such as type punning or wrap-around after signed arithmetic overflow, often used in low-level embedded code, and allows Astrée to analyze such programs correctly and precisely. This low-level semantics also frees Astrée from the reliance on static type information, so that it can handle the common case where an unstructured array of bytes is dynamically reinterpreted as a structure of some type. Astrée can also handle multi-dimensional arrays encoded explicitly in a single array using index arithmetic. Floating-point numbers are modelled faithfully according to the IEEE 754 norm [15], including special numbers (infinities and *NaN*) as well as rounding.

Error checking. Astrée signals all potential runtime errors and further critical program defects. It reports program defects caused by unspecified and undefined behaviors according to the C99 standard [17], program defects caused by invalid concurrent behavior, violations of user-specified programming guidelines, and computes program properties relevant for functional safety. Astrée raises alarms for operations resulting in unpredictable program behaviors, such as invalid array and pointer accesses. Alarms are also raised for invalid operations triggering exceptions, such as divisions by zero or floating-point overflows, and for dangerous operations whose result, although well-defined either in the C99 standard or in Astrée's more refined semantic, may be unexpected, for instance wrap-around after unsigned or signed arithmetic overflow. Astrée does not stop at the first error, but strives to continue the analysis with a reasonable result. This is useful to handle, e.g., programs with intended wrap-around, and prevents a benign error from masking a subsequent, more serious one. Astrée also permits users to specify their own functional properties to be checked with an assertion mechanism (similar to C's assert command), and will report any violation. Finally, Astrée includes a rule checker that supports MISRA C:2004 [26] and MISRA C:2012 [27] and can be extended for customer-specific rule sets.

Abstraction. Astrée is based on abstract interpretation [7]: it uses abstractions to represent and manipulate efficiently over-approximations of program states. One simple example of abstraction used pervasively in Astrée is to consider only the bounds of a numeric variable, forgetting the exact set of possible values within these bounds. However, more complex, but also more costly, abstractions can also be necessary, such as tracking linear relationships between numeric variables (which is useful for the precise analysis of loops). As no single abstraction is sufficient to obtain sufficiently precise results, Astrée is actually built by combining a large set of efficient abstractions (e.g., the octagon domain [22]). Some of them, such as abstractions of digital filters [10], have been developed specifically to analyze control-command software as these constitute an important share of safety-critical embedded software. In addition to numeric properties, Astrée contains abstractions to reason about pointers, pointer arithmetics (abstracting offsets as numeric variables), structures, arrays (in a field-sensitive or field-insensitive way). Finally, to ensure precision, Astrée keeps a precise representation of the control flow, by performing a fully context-sensitive, flow-sensitive (and even partially path-sensitive) interprocedural analysis.

Analysis options allow fine-tuning the analysis precision, either with global parameters or with local directives focusing precision on some program parts and some variables. All Astrée directives, e.g., for specifying range information for inputs or adapting the precision of the analyzer can be specified in the formal language AAL [3] by locating them in the abstract syntax tree without modifying the source code — a prerequisite for analyzing automatically generated code. To deal with evolving software Astrée provides a mechanism to detect whether annotations are still placed at the intended location after structural code changes [19].

Analysis output. In addition to the list and location of alarms, Astrée makes the semantic information computed during the analysis available to the user. For instance, Astrée constructs, based on its analysis of function pointers, a control-flow graph, which can be visualized graphically and interactively explored after the analysis. Furthermore, the range computed for each variable, at each location and for each call context, can be looked-up. This provides additional useful information about the program: it can be used, beyond run-time error checking, to verify design specifications. It is also



(a) (b)

Figure 1: Astrée call graph visualization (a) and variable range (b) visualization.

useful for alarm investigation, to understand the origin of run-time errors and spurious alarms. Astrée also reports unreachable code and non-terminating loops.

3 Concurrent Program Analysis

Astrée has been recently extended [23] to support concurrency-related constructions, with a specific focus on concurrency features for embedded C software. Traditionally (prior to the C11 standard, which includes concurrency into the C language, but even after), concurrency is provided to a C implementation through additional libraries, with varying, incompatible semantics. To solve this issue, Astrée provides universal low-level building blocks for concurrency features, on top of which realistic models of actual concurrency libraries can be programmed. This section focuses on the semantics and analysis of the low-level concurrent semantics, while concurrency library modelling is discussed in Sec. 4.

Threading model. Astrée's low-level concurrency semantics is based on POSIX-style threads [16]. Each thread is a fully preemptable execution unit with independent control and local variables, but shared global memory. A program execution is then an interleaving of thread executions. Thus Astrée threads can be used to model POSIX threads, but also ARINC 653 processes, OSEK/AUTOSAR tasks, interrupts, etc. Depending on the concurrency model, threads may be declared in an external configuration file (such as OSEK tasks) or programmatically (as in POSIX threads). Astrée supports both models, but assumes in the latter case that threads cannot be created arbitrarily during program execution. Instead, program execution is decomposed into two separate phases: an initialization phase that executes arbitrary sequential C code and can create, but not execute, threads; and a second phase where all threads execute concurrently but new threads cannot be created. This limitation matches the current practice (and sometimes the OS limitations) in embedded software. It is exploited to achieve a simpler and more precise analysis. The set of threads created programmatically is discovered during the analysis fully automatically. Additionally, Astrée supports the concept of thread instances, i.e., multiple creations of threads with the same entry point. The thread-modular abstraction used in Astrée, described below, reduces the analysis of the program to that of a single instance of each thread. Hence, Astrée naturally supports unbounded instances of threads, which is useful to analyze parameterized systems, i.e., systems where the number of instances of a thread is an unknown constant.

Shared memory. Following the POSIX thread model, Astrée assumes that all the threads can access all the global variables, i.e., the global variables are implicitly shared. By analyzing the threads, Astrée then infers automatically which variables are actually shared and reports precisely which part of each variable is accessed by each thread and the access mode (read, write or read/write). While it is possible for one thread to access the local variables of another thread (e.g., sharing a pointer to a local variable through a global variable) this is a dangerous practice as the local variables can be deallocated by the time the other thread accesses it. Astrée thus detects and reports such usages as errors. Similarly, while Astrée supports dynamic memory allocation (e.g., with `malloc`), it is an error (reported to the user) for one thread to access the memory allocated by another thread.

Synchronization. Astrée has built-in support for thread synchronization. In particular, Astrée has a notion of mutual exclusion locks, so-called mutexes, with

the property that a given mutex can be locked by at most one thread at a time. Astrée's mutexes are very simple non-recursive variants of POSIX's mutexes: if a thread locks a mutex that is locked by anther thread, it enters a waiting state until the other thread unlocks the mutex; locking again a mutex that is already locked by the same thread, or unlocking a mutex that the thread has not locked, has no effect. More complex locking mechanisms can be programmed on top of such simple mutexes to model the semantics of realistic concurrency libraries (such as recursive mutexes that feature a lock counter, or mutexes that fail when locked again by the same thread). In Astrée, mutexes are identified by 32-bit integers and need not be created *a priory*. It is the responsibility of the OS modelling (Sec. 4) to allocate such integers, either statically (e.g., associate a mutex to each resource in an OSEK program) or programmatically (e.g., use a counter to allocate at run-time unique mutex identifiers when a new mutex is created by a POSIX thread system call).

Astrée tracks which part of each thread is protected by each mutex, and discovers automatically regions that are in mutual exclusion. This information is combined with the inference of shared memory locations, so that Astrée can report all data races (both read/write and write/write data races). In case of a data race, Astrée continues the analysis by considering the possible values steaming from all possible interleavings.

producer	consumer

`for (i=0;i<100000;i++)`
`{`
`  lock(1);`
`  x=x+1;`
`  if (x>100) x=100;`
`  unlock(1);`
`}`
	`for (j=0;j<100000;j++)`
	`{`
	`  lock(1);`
	`  if (x>0) x=x-1;`
	`  unlock(1);`
	`}`

Figure 2: Producer and consumer threads protected by a mutex.

Example. Figure 2 gives an example program composed of one or several instances of a producer thread and one or several instances of a consumer thread, where the resource is abstracted as a counter variable `x`. In this example, Astrée will be able to discover that `x` is shared and that there is no data race, as all the accesses to `x` are correctly protected by mutex 1. Additionally, Astrée reports that `x` is always in the range $[0,100]$, except just after `x=x+1`, where it can be 101. Failure to use a mutex would cause Astrée to report a data race at each access to `x`. It would also cause the range of `x` to grow beyond 101 as several producer instances can now concurrently increase `x` before the test `if (x>100) x=100`.

Astrée does not currently detect deadlocks caused by improperly nesting of mutex locks by concurrent threads. This is not an inherent limitation of our method, but a limitation of the tool, and this detection is planned for future work, by leveraging the automatic detection of which mutexes are locked by each thread at each program point. Additionally, Astrée has a preliminary support for additional synchronization primitives: read/write locks, signals, and barriers, which are currently handled in a sound but sometimes imprecise way, and future work to improve their support is planned.

high priority	low priority

`for (i=0;i<100000;i++)`
`{`
`  if (!islocked(1))`
`  {`
`  x=x+1;`
`  if (x>100) x=100;`
`  }`
`  yield();`
`}`
	`for (j=0;j<100000;j++)`
	`{`
	`  lock(1);`
	`  if (x>0) x=x-1;`
	`  unlock(1);`
	`}`

Figure 3: Priority-based producer-consumer example.

Real-time scheduling. Astrée is sound with respect to all possible interleavings of threads, which would correspond to a fully preemptive and non-deterministic scheduler. However, embedded programs often employ specific real-time schedulers that partially restrict thread interleavings. Notably, each thread is given a priority, and higher priority threads cannot be preempted by lower priority ones, unless they stop explicitly by issuing a blocking system call, such as locking a mutex or waiting for an external event. Astrée takes priority information into account, when available, to detect portions of threads in mutual exclusion due to priority scheduling, and it uses this information to remove spurious thread interactions and data races.

Example. Figure 3 presents a variant of Fig. 2 using priorities. After testing whether the mutex is unlocked, the high priority thread can assume that the low level priority thread is not in its critical section; it can then safely test and modify `x` atomically, without fear of being interrupted by the low priority thread. The effect is thus the same as in the program of Fig. 2. Astrée proves the absence of data race and provides precise bounds for `x`.

Note that, at the end of its critical section, the high priority thread explicitly yields to allow the lower priority thread to run. The semantics of the `yeild` primitive is that of a non-deterministic wait, which is useful to model waiting for an external event or for a delay (as Astrée does not keep track of execution time). As a consequence of this non-determinism, the high priority thread may interrupt the lower priority thread at *any* point during its execution. This highlights the fact that, despite a deterministic, priority-based scheduling, embedded programs often feature a large possible number of thread interleavings. Unlike previous works on embedded real-time applications [11], Astrée is not limited to collaborative threads, nor discrete sets of preemption points, which would not soundly account for all possible executions. Note that, to ensure scalability, Astrée employs possibly imprecise abstractions of thread priorities and real-time scheduling. For instance, threads

with dynamically changing priorities are supported, but considered to be preemptable by all threads at any point (i.e., their exact priority relative to other threads is not tracked), which is sound but imprecise. To improve precision we are currently implementing the priority ceiling protocol which is the standard scheduling scheme in OSEK systems. When unable to use priorities to reduce the interleaving space, Astrée reverts to unrestricted preemption, which ensures a coverage of all concurrency models.

Thread-modular analysis. On sequential programs, Astrée employs a fully flow-sensitive and context-sensitive analysis: an abstraction of the possible memory states is propagated along the program control flow graph, and abstract states are merged at control-flow joins (such as the end of an if-then-else or a loop iteration). Flow-sensitivity, i.e., the ability to distinguish the value of a variable at different control points, is often necessary to achieve a degree of precision sufficient to prove the absence of run-time error. Concurrent programs, however, 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 interleaved execution paths and it would be too costly to distinguish the value of a variable at each combination of thread control locations.

For concurrent programs, Astrée thus employs instead a thread-modular analysis. In a nutshell, each individual thread of the program is analyzed separately, as would be a sequential program. In addition to potential run-time errors, each thread analysis collects the effect it can have on the global memory. The threads are then reanalyzed, but now taking into account the effect from other threads as gathered at the previous analysis. As this new analysis may expose new behaviors of threads, and so, more effects, it triggers a reanalysis of the threads. The analysis thus proceeds in rounds, starting from an empty set of thread interactions, and reanalyzing the threads with an increasing interaction set, until stabilization. A standard abstract interpretation technique, iteration extrapolation with widening, is used to ensure that this process terminates after a finite, small number of iterations (experiences point towards around 6 iterations, independently from the program size and number of threads). A theoretical result [23] states that, after stabilization, the thread-modular analysis has explored an over-approximation of all the possible interleavings; it is thus sound.

Example. Consider again the producer-consumer example from Fig. 2. The first analysis round, considering each thread in isolation, will deduce that, at the end of the producer loop, `x` necessarily equals 100 while, at the end of the consumer loop, `x` necessarily equals 0, which is obviously inconsistent. However, the analysis also deduces that, during its execution, the producer stores a value in $[1;101]$ into `x`, and the consumer does not modify `x` (yet). This information is used at the second analysis round. In particular, now, when the consumer performs `x=x-1`, this is understood as storing into `x` the last value stored into `x` by the consumer minus 1, or storing a value stored by the producer, i.e. $[1;101]$, minus 1. The analysis of mutexes further deduces that the value 101 is not actually visible by the consumer, hence the second case stores a value in $[1,100]-1=[0,99]$ into `x`. At the end of the consumer loop, `x` would thus read either a value in $[0,99]$, when reading the last value stored by the consumer, or a value in $[1,100]$, if a write from the producer was performed since that last write by the consumer. A third analysis round, where the consumer takes into account the values $[0,99]$ stored by the consumer, yields the same set of interferences, hence, the analysis finishes and deduce that, at the end of the program, `x` is in the range $[0,100]$, which is the expected result.

The benefit of this method is threefold. Firstly, it provides a sweet spot between cost and precision: it is nearly as efficient as a sequential program analysis and maintains flow-sensitivity at the intra-thread level. Secondly, each thread analysis is but a sequential program analysis, slightly modified to extract and apply interferences on the shared memory; thus, all the infrastructure present in sequential Astrée could be reused as is. Thirdly, the analysis is parametric independently in the abstraction chosen to abstract the memory and the abstraction chosen to abstract thread interferences. The former exploits all the memory abstractions developed for sequential Astrée. For the later, the above example employs a simple and scalable, non-relational and flow-insensitive abstraction: the range of values stored by a thread into a variable, but recent work [24] has proposed new abstractions that can improve the precision without sacrificing the scalability by adding a small measure of relationality or flow-sensitivity; Astrée is thus able to infer that a thread modifies a variable in a monotonic way, and to discover relational locks invariants.

Memory consistency. When several threads access a shared memory, it is important to determine the underlying consistency model ensured by the hardware and compiler. The simplest model, *sequentially consistent memory* [20], assumed implicitly in our examples above, states that, in an interleaving of thread executions, each thread reads back from the shared memory the value stored by the last thread to write into the memory. This is unfortunately not realistic: modern hardware introduce memory hierarchies, buffers and cache, and compilers introduce optimizations that invalidate this view, as several copies of a variable may reside in the system. Modern language specifications, such as C11, introduce weaker memory models to take such effects into account. As weak memories feature non sequentially consistent executions, an analysis tool

designed solely for sequential consistency is not sound with respect to a weak memory model. In contrast, Astrée is designed to be sound for a variety of memory models, based on the choice of which abstractions are used for thread interferences. For instance, the flow-insensitive non-relational abstraction used in the above example has been proven [23] to be sound for very lax memory models, while the soundness of the abstraction able to infer the monotonicity of shared variables requires a model such as *total store ordering* adopted by several popular processors, such as x86 [32].

4 Operating System Support

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 (e.g., the most critical software), the sofware runs directly on top of 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.

Library stubs. Astrée provides facilities to concisely write stubs that model functions at an abstract level using C code with additional primitives, including non-deterministic variable modifications and checked assertions (using arbitrary C boolean expressions). A typical stub first checks the validity of its arguments (using assertions), then performs necessary side-effects (such as modifying an argument passed by reference) and finally constructs a valid return value. For instance, the `sin` stub function only checks that its argument is a not a special floating-point number and returns a non-deterministic value assumed only to be in $[-1,1]$. Astrée comes with a complete set of stubs for the C library, weighting 9 Klines. It is based on the C99 standard [17], not on a specific implementation; as a result, the analysis results are sound whatever conforming C library implementation is used.

Concurrency stubs. With the addition of concurrency, new libraries have been added, including POSIX threads [16] and the ARINC 653 standard used in avionics [4]. These leverage the low-level concurrency primitives offered by Astrée and its internal notion of threads and mutexes, but often need to wrap them into more complex objects maintained in C arrays and structures. For instance, a POSIX thread is an Astrée thread together with attributes and a state (such as a cleanup routine, a return value, etc.). Additionally, the core set of Astrée objects is reused to model the wide variety of objects offered by such systems; e.g., asynchronous signal handlers are assigned an Astrée thread, mutexes are reused to implement read-write locks, etc. Around 3 Klines of the 9 Klines of C library are devoted to POSIX concurrency primitives, while the model of ARINC 653 occupies 4 Klines. More details on these models are available in [25].

OSEK/AUTOSAR support. Astrée has recently added support for OSEK/AUTOSAR operating systems [1], a widely used standard in automotive. 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 resources 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. Note that, due to the abstractions employed by Astrée to achieve scalability, some aspects of scheduling are not currently analyzed in a precise way. For instance, Astrée does not currently track which threads are in a stopped or started state, and assumes that every thread is possibly started at any point. As a result, interrupts enable and disable operations are not precisely handled. We plan to address this limitation in future work by simply adding new abstractions without changing the model.

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/AUTOSAR is that all system resources, including tasks, are not created dynamically at program startup. Instead they are hardcoded into 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

Size	Added	Select.	Time	Mem.
2.1 M	5.2 K	99.94%	24 h	27 GB
1.9 M	2.4 K	99.56%	154 h	18 GB
2.2 M	2.3 K	99.52%	160 h	23 GB
31.8 K	2.2 K	97.28%	50 mn	0.6 GB
33.1 K	1.2 K	97.18%	35 h	2.5 GB

Figure 4: Avionics case studies from [25], with the original size (in lines), the size (in lines) of added stubs, the selectivity (percentage of lines proved correct), the analysis time and memory consumption.

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.

The set of errors detected by Astrée includes runtime errors and data-race, but also a new alarm category *invalid usage of OS service*. As an example the OSEK stub automatically checks that the application calls OSEK services according to the specification. In case of API errors the analysis of an OSEK application will raise alarms from this new category, including: invalid task, alarm, or resource identifiers, calling a service from an ISR with incorrect level, improper nesting of resource acquisition and release (lock/unlock problems), or failure to release all the acquired resources before terminating a task.

5 Practical Experiments

The concurrency support built into Astrée has been tested in a variety of analysis experiments.

Name	Size	Select.	Time	Mem.
HiTechnic	162	100%	0.4 s	11 MB
NXT GT	302	97%	1.2 s	20 MB
NXTway-GS	439	98%	4.1 s	20 MB
NXT Cesar	4500	95%	6 mn	435 MB

Figure 5: Preliminary OSEK experiments on nxtOSEK samples [2, 14].

5.1 Avionics Software – ARINC 653

The support for ARINC 653 was first designed as a research experiment extending Astrée to analyze medium-sized to large concurrent industrial avionics C software. Astrée was later extended with a subset of POSIX threads, also used in avionics software. The results of these experiments are reported in details in [25] and summarized in Fig. 4. To sum-up, this study shows that Astrée can handle complex, realistic concurrent programs with a sufficient level of precision (a selectivity near 100%, indicating that very few lines exhibit an alarm) and adequate performance in the context of software validation (where tests, the usual validation method, can take weeks).

5.2 Automotive Software – OSEK

In the following we summarize experimental results obtained on OSEK applications: some small C programs designed for Lego Mindstorm NXT robots under the nxtOSEK system [2], and three real-life automotive applications. For reasons of confidentiality the results on industrial automotive projects have been anonymized. The results show that Astrée can be successfully applied on real-life industrial software projects. Moreover, the analysis runs on standard PC hardware and is reasonably fast.

Lego Mindstorm. As a proof-of-concept, our initial tests of the OSEK support in AstréeA were performed on simple, freely available C programs designed for the Lego Mindstorm OSEK platform. The results are shown in Fig. 5. The first three programs, of a few hundred lines, are sample programs included in the nxtOSEK distribution. The last program is the NXT Cesar robot developed at the iCube laboratory [14]. This program performs non-trivial floating-point computations, on which Astrée reports possible overflows and invalid operations; indeed the software elects to perform computations without checking operator arguments, and fix the result only after the computation, by replacing any infinity and not-a-number with zero.

Automotive 1. The first real-life application is a small project consisting of two tasks comprising 177 576 lines of preprocessed C code (without blank lines and without

comments). The project is configured by an `.oil` file automatically processed by Astrée. Astrée reports 698 alarm locations with alarms of the following type:

Alarm Category	#Loc
Invalid range of pointers and arrays	17
Division or modulo by zero	58
Invalid ranges and overflows	617
Read/write data race	6

The analysis takes 38min with full precision and consumes 7.5 GB RAM. It reaches 78% of the code. The 6 alarms about read/write data races were all confirmed to be justified, there were no false alarms about data races.

Automotive 2. The second real-life application consists of 358 335 lines of preprocessed C code (without blank lines and without comments). The configuration is given by an `.oil` file which can be automatically processed by Astrée to produce all relevant data structures and access functions. The project consists of 4 tasks, 30 ISRs (interrupts) and 3 alarms (timers). Astrée reports 1 796 code locations with alarms of the following types:

Alarm Category	#Loc
Division or modulo by zero	58
Invalid usage of pointers or arrays	460
Invalid ranges and overflows	1 278

With reduced precision settings the analysis reaches 97% of the code, the analysis time is 7h13min, and memory consumption 3.1GB. The resulting selectivity is above 99%; these alarms include run-time errors caused by the effects of data races, e.g., overflows or invalid pointer accesses, but not the data races itself. This distinction is reasonable since there may be data races which do not actually induce erroneous behavior.

Furthermore Astrée reports 15 967 code locations with alarms from the newly introduced concurrent alarm categories:

Invalid Concurrent Behavior	#Loc
Read/write data race	8 024
Write/write data race	7 941
Invalid usage of OS service	2

A data race alarm is produced for every access contributing to a race, i.e. for each shared variable subject to a data race several alarms will be issued. This increases the number of of alarms reported but helps users to distinguish between correct accesses and accesses contributing to a race. A further analysis of the data races shows that in this project most of the synchronization is done via explicit enable/disable interrupt calls. Currently such calls are not precisely handled, but dedicated abstractions for them are under development. Also task priorities have been exploited to optimize the application: write operations are mostly done by the highest-priority task which enables light-weight synchronization mechanisms. As explained in Sec. 6 the support of the priority ceiling protocol currently is in development, too, so to enable a sound result, task priorities are currently not taken into account. With both extensions finished we expect the number of data race alarms to be significantly reduced.

Automotive 3. The third real-life project is an OSEK application with 1 655 384 lines of preprocessed C code (without blank lines and without comments), again configured by an `.oil` file. The project consists of 24 tasks, 34 ISRs and 12 alarms. Astrée reports 1 743 code locations with alarms from the following categories:

Alarm Category	#Loc
Division or modulo by zero	4
Uninitialized variables	27
Invalid usage of pointers and arrays	310
Invalid ranges and overflows	1 402

With reduced precision settings the analysis reaches 46% of the code, analysis time is 3h7min, the required memory consumption is 5.4GB. The reason of the low percentage of reached code is incomplete environment information, and also the lack of some parts of the application which have not been available to us.

In total the number of alarms about invalid concurrent behavior is 5 759:

Invalid Concurrent Behavior	#Loc
Read/write data race	3 152
Write/write data race	2 599
Invalid usage of OS service	8

Also this project uses enable/disable interrupt calls as a synchronization mechanism and exploits task priorities to implement lightweight synchronization. When support for these mechanisms is finished we expect the number of data race alarms to be significantly reduced.

6 Related work

Applying formal methods to the verification of concurrent programs and systems has a long history. We will focus on recent work and refer the reader to [31] for a survey and historical perspective. The theoretical foundation of Astrée is based on the abstract interpretation theory [7]. We refer the reader to [8] for an in-depth comparison of abstract interpretation techniques with other formal methods. Other tools based on abstract interpretation include Polyspace [9] which can detect shared variables and take task interleavings into account. However, to the extent of our knowledge, it does

not report data races nor lock/unlock defects and lacks a direct support for OSEK applications so that users have to manually specify the concurrency setup. By comparison, in addition to reporting all potential data races and lock/unlock defects Astrée provides a complete and automated support for OSEK, including a stub OS library, a toolchain allowing the analysis to be automatically configured by an OIL file, the automatic detection of the entry points of all the tasks and interrupts, as well as the detection of critical sections. The modeling of concurrent embedded operating systems for use in the analysis of applications has been considered before in [11]. We report in [25] the use of Astrée for avionics application and detail the modeling of the ARINC 653 OS specification.

The thread-modular semantics employed to achieve a scalable analysis of concurrent programs is inspired from the rely-guarantee principle, introduced in proof methods [18]. Note that, unlike proof-based verification tools, Astrée automatically infers memory invariants as well as interferences and does not rely on the programmer to provide them.

Another popular method to verify concurrent systems is model checking. Model checking can suffer from the state explosion problem, particularly acute when considering concurrent systems. It has been partially addressed by partial order reduction methods [12]. In practice, the SPIN model checker has been used [13] to check for data-races and deadlocks in concurrent code from NASA. The analysis of C code was however limited to fragments of a few hundred lines. The study also mentions that C code up to 45 KLoc could be handled by analyzing a hand-crafted 1 Kloc model. By contrast, Astrée scales to million-line codes. It does not require building a model by hand, and can analyze directly full C applications without the need to extract small, self-contained parts, which is time-consuming and error-prone. Another recent proposal to improve the scalability of model-checking is to analyze a system only up to a fixed, generally small number of context switches [28]. While this method can be useful to find bugs, it is unsound and only covers a small fraction of the possible behaviors, and is thus not adequate according to the most stringent certification processes used in embedded critical software (such as avionics software [30]). By contrast, Astrée is sound and will find all run-time errors and data-races.

Sequentialization [29] suggests a reduction from concurrent programs to equivalent sequential ones in order to apply existing sequentialization verification methods. The method has been applied in particular to the static analysis by abstract interpretation of interrupt-driven programs [33]. The method is however limited to specific scheduling policies, as a higher priority task must complete before the control is returned to a lower priority task. Unlike Astrée, it does not permit arbitrary preemption (as found for instance in ARINC or POSIX threads), and is thus less general.

With the rise of multi-core applications, formal methods have been updated to take into account weakly memory models. Astrée is also aware of weakly memory models, through a careful selection of which abstractions are employed during the analysis. Similar results concerning the influence of the abstraction on the soundness in weak memory models can be found in [5].

Future work. Future work on Astrée is planned to address its current limitation. Firstly, we plan to add a deadlock detector. Secondly, we plan to improve the handling of thread priorities, including dynamic priorities and the priority ceiling protocol implemented in OSEK, to improve the precision. We plan to add a precise, flow-sensitive tracking of interrupt enable, which will also improve the precision by removing spurious interferences from disabled interrupts. Thirdly, we wish to improve our support for multi-core applications since our current support for multi-core requires, for soundness, ignores the priority of threads and assuming arbitrary preemption. The focus on multi-core will also encourage us to seek more precise abstractions of weakly consistent memory models.

7 Conclusion

Safety requirements mandate that critical software is exempt from run-time errors. The rising predominance of concurrent software architectures puts a strain on classic validation methods, such as testing or code reviews, that hardly cope with the non-deterministic nature of concurrent programs, the huge number of interleavings, and the difficulty to uncover errors in extremely rare but possible cases. We have presented Astrée, a tried static analysis verification tool based on abstract interpretation, and its recent extension to the sound analysis of concurrent C programs, efficiently covering all possible interleavings and uncovering all run-time errors and data races. We have explained how Astrée can support programs for various operating systems and concurrency libraries (POSIX threads, ARINC 653, OSEK/AUTOSAR) and presented encouraging experimental results. Ongoing work includes further experimentation (in particular on automotive applications under the OSEK/AUTOSAR system), support for more systems and concurrency models, as well as the design of additional abstractions to improve both the precision and the scalability of the analysis.

Acknowledgement. The work presented in this article has been supported by the German BMBF project FORTISSIMO and the project ANR-11-INSE-014 from the French Agence nationale de la recherche.

References

[1] AUTOSAR (AUTomotive Open System ARchitecture). http://www.autosar.org.

[2] nxtOSEK/JSP. http://lejos-osek.sourceforge.net/.

[3] AbsInt. *The Static Analyzer Astrée– User Documentation for AAL Annotations*, 2015.

[4] Aeronautical Radio Inc. ARINC 653. http://www.arinc.com.

[5] J. Alglave, D. Kroening, J. Lugton, V. Nimal, and M. Tautschnig. Soundness of data flow analyses for weak memory models. In *Proc. of the 9th Asian Symp. on Programming Languages and Systems (APLAS'2011)*, volume 7078 of *LNCS*, pages 272–288, Dec. 2011.

[6] 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.

[7] 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.

[8] P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, D. Monniaux, and X. 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.

[9] A. Deutsch. Static Verification of Dynamic Properties. *ACM SIGAda 2003 Conference*, 2003.

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

[11] A. Gamatié and T. Gautier. Synchronous modeling of avionics applications using the SIGNAL language. In *Proc. of the 9th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS'03)*, pages 144–151. IEEE Computer Society, 2003.

[12] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. PhD thesis, University of Liege, Computer Science Department, 1994.

[13] G. J. Holzmann. Mars code. *Commun. ACM*, 57(2):64–73, Feb. 2014.

[14] ICube. NXT CESAR project. http://icube-avr.unistra.fr/en/index.php/NXT_CESAR.

[15] IEEE Computer Society. Standard for binary floating-point arithmetic. Technical report, ANSI/IEEE Std. 745-1985, 1985.

[16] 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.

[17] ISO/IEC JTC1/SC22/WG14 working group. C standard. Technical Report 1124, ISO & IEC, 2007.

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

[19] D. Kästner and J. Pohland. Program Analysis on Evolving Software. In M. Roy, editor, *CARS 2015 - Critical Automotive applications: Robustness & Safety*, Paris, France, Sept. 2015.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 1978.

[21] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, pages 54–63. ACM, Jun. 2006.

[22] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[23] 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.

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

[25] 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.

[26] MISRA-C:2004 Guidelines for the use of the C language in critical systems, Oct. 2004.

[27] MISRA-C:2012 Guidelines for the use of the C language in critical systems, Mar. 2013.

[28] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Proc. of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

[29] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *Proc. of the ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'04)*, pages 14–24. ACM, June 2004.

[30] Radio Technical Commission for Aeronautics. RTCA DO-178C. Software Considerations in Airborne Systems and Equipment Certification, 2011.

[31] M. C. Rinard. Analysis of multithreaded programs. In *Proc. of the 8th Int. Symp. on Static Analysis (SAS'01)*, volume 2126 of *LNCS*, pages 1–19. Springer, Jul 2001.

[32] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Comm. ACM*, 53, 2010.

[33] W. Wu, L. Chen, A. Miné, D. Dong, and J. Wang. Numerical Static Analysis of Interrupt-Driven Programs via Sequentialization. In *Proc. of the 15th International Conference on Embedded Software (EMSOFT'15)*, pages 55–64. IEEE CS Press, Oct. 2015.