Coming out of the niche?

David Monniaux
CNRS / VERIMAG

November 1, 2010

Abstract

The strongest successes of static analysis so far have been, on the one hand, sound analysis on safety-critical embedded systems, a rather niche market, on the other hand static analysis for finding bugs in more ordinary programs. Can the gap be bridged?

1 Introduction

In the 1970s, C.A.R. Hoare [6] and others argued that programs should be proved correct before use, much like conjectures are proved before being used as theorems. This view was not well-received in all quarters [4], and many argued that formal proofs would not "scale up" from toy examples to real applications: not only real applications are huge in comparison to, say, a small sorting procedure (a common example for proof methods), but they are also considerably less well-specified. In fact, building and maintaining a comprehensive specification for a large program, even if possible, is a work of the same order of magnitude as that of the program itself.

Four decades have passed, and the current state of affairs is that almost no programs are proved correct. Some essential libraries and tools, such as compilers (see e.g. the CompCert project [7]), or some crucial parts of safety-critical embedded systems [11], are proved correct. Even with respect to the minimal implicit specification that a program should not crash with a division by zero, bad memory access or similar runtime errors, the only examples known to us are again in safety-critical embedded systems.

We shall therefore see what is so special about safety-critical embedded systems, then see how this is really a niche in computing. In light of these facts, we shall then investigate what could be done with respect to both the industry and the static analysis research field in order for static analysis to break out of the niche.

2 Safety-critical systems

Critical computing systems are those, typically driving machinery or vehicles, whose failure may result in unacceptable human losses. Examples of critical systems include fly-by-wire controls on aircraft or on manned spacecraft, radiation therapy equipment, nuclear power plant safety systems. To this list we may add in the future steering-by-wire and brake-by-wire systems on motor vehicles.

2.1 Fly-by-wire

The fly-by-wire controls on civilian airliners are currently one of the most spectacular applications of formal methods, which is why we shall give them special attention.

Let us first define "fly-by-wire". In very light airplanes, there is a mechanical connection between the pilot and the control surfaces of the plane (ailerons, elevators, rudder, etc.). On larger aircraft this mechanical connection must be supplemented by the aeronautical equivalent of power steering: pressurized fluids that help the pilot push the surfaces; yet there is still a direct correspondence between the position of the control yoke or pedals and that of the surfaces. On fly-by-wire aircraft, this mechanical connection is replaced by a fully electric one, with analog or digital computers controlling the surfaces according to the pilot's orders.

The benefits of fly-by-wire are manifold. Electrical connections are lighter than mechanical ones, and may be easier to maintain. Computers, especially digital ones, are flexible and can support advanced functions, such as protections against some forms of unsafe flights, or emergency maneuvers with reaction times below those of human pilots. For instance, the plane can automatically react to a stall — an insufficient airspeed, with possible dire consequences — by making a quick dive to regain speed.[1]

Unfortunately, with benefits come dangers and costs. The more sophisticated a system is, the likelier it contains bugs. Therefore, the use of software in critical embedded systems is strongly regulated: civilian airliners sold on the international market are to obey the DO-178B [10] standard on avionic

[1] Such an emergency dive maneuver appears to have been erroneously triggered twice on an Airbus A330, Qantas flight 72, on 7 October 2008; some passengers were wounded. The Australian air safety authorities suspect that a dysfunctional ADIRU (a unit processing airspeed and inertial data) fed incorrect data into the fly-by-wire system, and that this system, failing to detect that the data was incorrect, acted upon it [1].

software.[2] This standard divides avionic software into five levels of criticality, from A (safety-critical, with possible loss of life) to E (no importance for safety). Fly-by-wire is at level A, and thus is governed by the most stringent rules.

DO-178B contains a number of prescriptions [11]. For instance, the manufacturer should provide the general specifications of the system, and demonstrate that the executed code implements these specifications and nothing more. Obviously, a program crashing with a runtime error (e.g. divide by zero or array access out of bound) violates its specification, and thus the manufacturer should demonstrate that execution errors never occur. The manufacturer should also demonstrate that all hypotheses upon which the safety of the system reside are fulfilled; for instance, if the system is supposed to execute a task every 1 ms, then this task should be shown to always take less than 1 ms. DO-178B does not impose any specific method for fulfilling these prescriptions, except testing.

Aviation certification authorities tend to require that simple solutions be used; complex or error-prone software designs need to be justified on grounds that simpler designs would be infeasible. For these reasons, industrial programming rules generally ban dynamic memory allocation, recursion and complex usage of pointers in level A code.

So far, formal methods have been applied to fly-by-wire systems in the following three directions:

1. Proof of absence of runtime errors. Since the absence of runtime errors is implicit in avionic software specifications (and, indeed, in almost all software specifications), and some major causes of runtime errors (e.g. complex usage of pointers) are prohibited by coding rules, runtime errors are an obvious target for verification.

   The author knows of two commercial products able to analyze programs fully automatically and to produce an exhaustive list of possible runtime errors (barring, of course, bugs in the analyzer): the *Polyspace Verifier*[3] and the *Astrée* analyzer[4].

[2] DO-178B is an American standard; the same document as a European standard is referred to as ED-12B. DO-178B is to be replaced soon by a new revision called DO-178C.

[3] The PolySpace Verifier was developed under the scientific leadership of the late Alain Deutsch by PolySpace, a small company based near Grenoble, France. PolySpace is now a unit of The Mathworks, best known for producing Matlab and Simulink.
`http://www.polyspace.com/`

[4] *Astrée* [3], with the final syllable pronounced similarly to the one in *entrée*, was developed at LIENS in Paris, a joint laboratory between the National center for scientific research (CNRS) and École normale supérieure (ENS), by a group led by Patrick Cousot

2. Proof that the system fulfills functional properties, that is, properties concerned with the correctness of the computations performed by the program. Contrary to the absence of runtime errors, these properties have to be explicitly stated, for they differ according to what the program is supposed to be doing. For instance, one functional property of a sorting program is that it produces a sorted output; another is that its output is a permutation of its input.

   Proving that functional properties hold is generally considered a harder problem than proving the absence of runtime errors, if only because of their variety and complexity. The analyzers listed above for runtime error analysis are capable of a restricted form of verification of functional properties, for they check that user-given assertions (provided e.g. using the `assert` construct of the C language) are valid for all executions. Yet, since they are not designed for such properties, they are likely to provide false alarms on but the simplest of them.

   Currently, the best approach for functional properties seems to be assisted theorem proving, supplemented by efficient automated decision procedures: ideally, simple properties are proved automatically, and more complex properties need user intervention. An example of a platform for such proofs is *Frama-C*[5]

3. Proof that a given task always executes within a given time. If the design of a control system relies on a task being executed cyclically with a period of, say, 1 ms, then the manufacturer should demonstrate that this task always takes less than 1 ms (or, more realistically, less than 1 ms minus a margin of safety). *Worst-case execution time* (WCET) analysis, given a piece of code, provides an upper bound on its execution time; an analysis is all the more precise that the upper bound it provides is closer to the WCET.

   WCET analysis is generally easy and precise for programs that do not use function pointers or computed jumps, running on very simple architectures where a given instructions takes a defined number of clock

and including the author. `http://www.astree.ens.fr/`

Later, development was also supported by INRIA, and the software was licensed for commercial development to Absint GmbH of Saarbrücken, Germany.
`http://www.absint.de/astree/`

[5]Frama-C was developed at the Saclay center of the French Commissariat à l'énergie atomique (CEA). Many plugins were however developed at other locations, especially at the LRI joint laboratory between CNRS and Université Paris-Sud at Orsay, with support from INRIA. `http://www.frama-c.com/`

cycles: it boils down to counting clock cycles in all basic blocks and combining the results by integer linear programming (ILP). The task is however made considerably more complex by features of modern architectures such as pipelines and caches: the number of cycles for performing an instruction depends on the state of these components. An architecture may exhibit *timing anomalies*: a suboptimal execution time for an instruction may have the beneficial effect of speeding up later instructions. Furthermore, the most commonly used cache replacement policies may produce *domino effects*: whether or not some data was in the cache at some instant may have long-lasting consequences.

Absint's *aiT* tool analyzes WCET for a variety of architectures.[6]

2.2 Critical assessment: a niche market

The success stories of formal methods and especially static analysis by abstract interpretation on fly-by-wire software are enlightening, since all currently successful automatic methods rely on some important peculiarities of such critical programs and the organizations that produce them.

First of all, the stringent requirements of certification make avionic manufacturers natural customers for techniques that could lighten their burden. Not only do they have to develop programs with no bugs, but they also have to convince authorities of this absence of bugs. In contrast, most regular software products contain many bugs, and their producers often run "bug tracking" systems with thousands of entries. They have to select which bugs should be treated first, according to priority criteria such as the practical consequences of the bug (e.g. loss of data, crash of computer, or minor user interface problem) and the number of customers impacted. We have heard anecdotes that such software companies are uninterested in static analysis, even analysis that provides no false positives and actual error traces, because it would only add new bugs of unclear priority to their already considerably long lists.

Formal methods tend to perform poorly in the presence of dynamic control flow (computed jumps, virtual functions, etc.) and complex uses of pointers. If an analyzer cannot obtain a precise result on the possible targets of a jump, then, for soundness, it has to assume that a very large set of targets may be possible, and to follow the control flow to all these targets.

[6] aiT was developed by Absint in collaboration with Reinhard Wilhelm's group at the University of Saarlandes, Saarbrücken, Germany. `http://www.absint.de/ait/`

Similarly, if an analyzer cannot compute a precise result on the possible targets of a write to memory, it has to assume that a very large set of targets may be touched by the write, thus losing information about many memory locations. Sound analysis methods for runtime errors therefore tend to be well-adapted to programs that use pointers sparingly, e.g. for implementing call-by-reference, but tend to perform poorly on programs manipulating dynamic data structures with nontrivial invariants — but such manipulations anyway tend to be prohibited by coding guidelines for critical systems.

WCET analysis suffers from similar drawbacks. Cache analysis relies crucially on knowing the addresses of code and data being accessed. In a critical system, usually, the program is loaded at a fixed address, with no dynamic loading, dynamic linking or dynamic code generation; thus one has few difficulties with code references. Assuming that all variables are statically allocated (fixed addresses) or on the stack, and that pointers are only used for call-by-reference, again as usual in critical systems, then the only difficulties are arrays accesses.

Most difficulties in WCET analysis arise from the architecture (pipelines, caches, busses) [12]. A first difficulty is obviously that the bigger the cache or the pipeline, the more states the system has, even after abstraction, thus the costlier the analysis, in both time and space. A more profound problem is that hardware architectures are generally optimized for the "average case"[7], which is not the same as optimizing for the worst case. On an architecture optimized for the average case, the performance of a code fragment may decrease significantly in some rare conditions (for instance, for some specific configurations of data in cache and pipeline when the fragment is started). Because these conditions are rare, the "average" execution time is barely impacted, but the WCET is. The "average" case is what matters in almost all software applications, including soft real time (e.g. audio or video decoding). WCET only matters for hard real time, and rigorous WCET analysis is needed only for critical systems.

The more an architecture contains features meant to boost "average" performance (deep pipelines, caches, prediction, speculative execution...), the more likely it is to exhibit rare cases where performance is significantly degraded, and the less possible it is to compute tight bounds on WCET. The counter-intuitive consequence is that it may be preferable to use a simpler, less "performant" architecture (performance being considered in the "average" case), in lieu of a complex one, because the computed upper bound on the

[7]The word "average" here involves no rigorous definition from the theory of probabilities, but rather an informal notion of amortized "typical" case.

WCET on the simpler architecture may be less than on the more complex architecture. Architectures used for critical hard real-time systems should thus ideally exhibit characteristics fairly different from those generally considered desirable for other applications. Unfortunately, it is economically difficult, even infeasible, to design an architecture specifically for critical systems.

Commonly found software / hardware architectures are designed for "average case" performance, not for ease of analysis or reproducibility. As an example, due to various reasons [9], the result of floating-point computations can depend on compiler optimizations and code unrelated to the computation. Some computer arithmetic experts even advocate that in the future, standards could be relaxed so as to allow for better performance, at the expense of reproducibility of computations. When advised by the author that lack of reproducibility greatly hindered debugging of "corner cases" as well as formal proofs on critical systems, they answered that critical systems are a "niche market".

This summarizes the predicament of sound static analysis of programs: it is, at present, only suitable for a *niche market* of critical systems where safety, and more precisely provable safety, is paramount.

3 Coming out of the niche

The author, being involved in the development of the Astrée static analyzer, considered founding a start-up company to commercialize this system. He and colleagues chose against this, for they thought that the targeted market was too small. Furthermore, each new client would likely bring new classes of code, which would need specific developments (at least at the beginning — one could hope that after a while, sufficient classes of code would be covered). Costs would thus increase linearly with the number of clients. This contrasts with the business model of off-the-shelf software companies (e.g. Microsoft), which develop a single software for a large number of clients. Thus, the company would probably have to charge the high prices usually associated with custom software. Would prospective buyers be still interested? The question was, and still is: *how can program analysis be usable for a wider range of applications*?

Unsound analysis So far, the most successful answer has been to relax the requirements of what static analyzers do. Instead of *proving* the absence of specification violations, one can instead try to *search* for such violations, without any guarantee that all of them would be found (and also without

guarantee that those found truly are violations). As notable examples of such *bug-finding* tools, one can cite Dawson Engler's work on the analysis of system-level C code and that of his company, Coverity[8].

Unsoundness can be deliberately incorporated into otherwise sound analyzers so as to reduce the number of false alarms. We have mentioned that imprecise results of pointer analysis could dramatically decrease the precision of all subsequent analysis: if we do not know where a pointer points to, any write to this pointer will destroy useful information in the analyzer. Thus, one can choose to be unsound with respect to pointer aliasing, as in the Clousot[9] tool.

Facilitation Another direction is to persuade the designers of systems that could benefit from analysis to design them so as to facilitate it. With respect to WCET, this means less complex pipelines, least-recently-used (LRU) cache replacement policy, and avoidance of timing anomalies. [12] With respect to software, this means high-level, strongly typed programming languages, avoidance of pointer arithmetic, avoidance of uselessly dynamic control flow, avoidance of shared-memory parallelism etc. In the case of code generated from high-level languages (e.g. Scade or Simulink), it could also be preferable to analyze the source code instead of the target code.

We are especially concerned about multi-threading, especially since current multi-core architectures make it sound like a good idea. Shared-memory multi-threading, with the usual mutual exclusion primitives, is notoriously difficult to get right. Bugs in preemptive multi-tasking are often very hard to reproduce, making debugging difficult. We suspect that, unless higher-level languages or primitives are used, debugging and verification will be increasingly hard as multi-threading is used massively.

An early start Unfortunately, currently, verification generally comes as an afterthought. This may be due to development methodologies that keep a wall of separation between the design and testing teams, so that the biases and lapses of the design group do not contaminate the testing group.[10] Program verification, being considered a new-fangled kind of testing, is therefore

[8] `http://www.coverity.com/`

[9] Clousot is a static checker for Microsoft's "code contracts" framework [5], now available commercially.

[10] When one designs a system, one tends to test it the way it is supposed to be used according to one's design, not according to what should be possible according to the documentation. Thus, having an unrelated team perform final tests is likely to expose bugs that would not have been exposed by a group closer to development.

performed very late in the development process.

By the time verification is performed, the design is therefore fixed. Barring a "smoking gun" bug, the system will not change because of analysis — and such "smoking guns" are rare anyway: since most sound analysis tools proceed by over-approximation, it is often difficult to obtain an execution trace actually violating the specification. In fact, even if such a bug is found, the system designers may prefer to leave it in as a known hazard rather than design a workaround that could have adverse effects (for instance, one may prefer to run the chance of a rare race condition rather than add a lock which could possibly introduce a deadlock or other adverse effect). In such a context, static analysis has limited interest.

We therefore advocate that static analysis should be used early on in the system design. This would make it less costly to change the design, when a simple change with no serious adverse effects for the efficiency or safety of the system could ease analysis.

No magic bullet Some approach static analysis with unrealistic expectations, then, inevitably, are disappointed and consider that it "does not work".

On one occasion, the author's team was visited by a company that had trouble maintaining some multi-threaded program. The system used shared memory and locks, but some locks were removed for efficiency because "manual analysis" had established they were not really needed. Unfortunately, because of design and personnel changes over the course of years, the reasons why the system worked, even if originally correct, were forgotten and perhaps no longer valid. It seems unreasonable to expect that a fully automated analyzer could magically discover reasons why a multi-threaded system should run safely, whereas an engineering team with access to original design documentation is not able to.

In the current state of the art, program analyzers are not "push button" solutions: often, some feedback from the user is needed to refine the analysis and remove false alarms. Prospective users should thus be informed that some amount of customization could be needed. Often, some modest changes may bring dramatic improvements.

The need for examples As we explained before, it is our opinion that analyzers should be developed with actual tests cases, preferably representative of the class of programs of interest. Unfortunately, it is surprisingly difficult to obtain such examples. A company may be interested in an ana-

lyzer, but may be unwilling to release code samples to analyzer developers, even under a non-disclosure agreement, for fear of intellectual property and confidentiality issues. One solution to this problem is for the company to prepare "representative" code cases — programs that are not in any production device (or maybe old ones), but that are similar to those in actual devices.

An obvious alternative is to run the analyzer on free or open-source software. The Linux kernel, the GNU Compiler (gcc), etc. are favorite test cases. One difficulty, however, is that these programs are generally written in full C (complete with dynamic memory allocation and pointer arithmetic) and thus a suitable analyzer is needed (it is no coincidence that successful analyzes being reported on the Linux kernel are unsound). Furthermore, these programs may not be representative of the intended targets: system utilities typically make many string and memory manipulations, but very little and trivial floating-point computation, which is the reverse of control systems.

4 Better cooperation within the static analysis research community

It is always very difficult to predict what avenues of technical or scientific research will be fruitful. Here are however the directions that we consider most promising for the next five years or so.

Convergence of analysis methods Currently, there are markedly different approaches to automatic analysis:

- Abstract interpretation by Kleene iterations, often accelerated with widenings, as exemplified by Astrée.
- Predicate abstraction with CEGAR (counterexample guided abstraction refinement) [8].
- Construction of "candidate invariants" by heuristics, then bounded model-checking in order to prove they truly are invariants, as exemplified by the Boogie tool [2].
- Exact acceleration, in the cases where some invariants may be computed directly from the form of the transition relation (Astrée uses some form of this, for numerical filters).

Often, tools implement only one of them (with the exception of exact acceleration tools, often coupled with Kleene iterations), whereas some communications would probably improve the situation. For instance, a common issue with CEGAR using Craig interpolants is that the predicates that they find may be not general enough (e.g. $x = 1 \wedge y = 1$ followed by $x = 2 \wedge y = 2$ etc. instead of $x = y$); this is not so surprising, since they are based on finite abstract traces and thus infer predicate suitable for suppressing spurious counterexample traces of a given length, but not necessarily of any length. In contrast, the whole idea of the abstract union and widening operators is to generalize conditions in the hope that they will be suitable for an unbounded number of iterations.

We thus expect to see more work in the direction of combined approaches. As an example, we have proposed recently a method for finding optimal invariants in certain lattices ("traditional" abstract interpretation) using a method not based on Kleene iterations, in which bounded model checking has an essential role.

Standardized test cases The satisfiability modulo theory (SMT) research community has a standard format for analysis problems (SMT-LIB[11]). For all the criticism one could raise against this standard (e.g. SMT-solvers being optimized to solve the standard problems and not actual problems from real-life use), it allows comparisons between implementations. In contrast, the static analysis community often makes comparisons impossibles: the test cases are often proprietary, badly identified, the properties being checked are often not clearly listed, as well as the precision of the results; furthermore, the implementations are often unavailable. In short, the benchmarks typically given at the end of static analysis publications are often little meaningful. This contrasts with the standard practice in other fields of science, where claims have to be independently verifiable to be admissible for publication.

It seems more difficult to build a repository of analysis test cases than of SMT formulas. One reason is that the SMT decision problem is well-defined, whereas the problem of obtaining a "nice" invariant is not. This difficulty can be circumvented by providing reachability or co-reachability properties, thus obtaining a well-defined decision problem.

The "rich models" project[12] is a step in this direction. There are, however, considerable difficulties in such an endeavor: for once, the definitions of the

[11] `http://www.smtlib.org/`

[12] `http://www.richmodels.org/`

memory model and associated semantics depends on the source language (e.g. C is not the same as Java). Perhaps it would be sensible to begin with the least delicate language features, that is, arithmetic and basic control flow (even though, even with arithmetic, there are difficulties such as finite-word integer arithmetic and floating-point).

Open platforms We have emphasized the need to test analyzers on actual program examples. However, in order for an analyzer to run on real-life C or Java code, it has to embark a compiler front-end: a parser (or byte-code reader) and a type analysis. Furthermore, it also needs some pointer analysis, and, for Java, a class analysis (in the case of C, pointers are very commonly used, if only to implement call by reference; Java enforces object orientation). In short, before being able to test a new idea on, say, numerical abstractions, one has to expend many man-months[13] developing a basic analysis platform.

We thus think that the static analysis research domain would benefit from the availability of some analysis platforms, capable of operating on real-life code (C, Java or other industrial languages), under a free license.

5 Conclusion

If we (as in, the static analysis research community) want to have our methods and tools accepted by a larger segment of the industry, we need to make some efforts:

- On collaboration: open platforms, open benchmarks, more integration.
- On education of industry professionals, who seem to oscillate between the rejection of static analysis as some academic fantasy on the one hand, or expect it as a magic bullet on the other hand.
- Obviously, on science.

Otherwise, we might get stuck in the niche of safety-critical systems, which is not so large.

References

[1] *Aviation occurrence investigation AO-2008-070, interim factual report no 2: In-flight upset 154 km west of Learmonth, WA, 7 october 2008.* ATSB (Australian Transport Safety Bureau), November 2009.

[13] We know that they are somewhat mythical, but still...

[2] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO (Formal Methods for Components and Objects)*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005. ISBN 3-540-36749-7. doi: 10.1007/11804192_17.

[3] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation (PLDI)*, pages 196–207. ACM, 2003. ISBN 1-58113-662-5. doi: 10.1145/781131.781153.

[4] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, 1979. ISSN 0001-0782. doi: 10.1145/359104.359106.

[5] Manuel Fahndrich and Francesco Logozzo. Static contract checking with abstract interpretation. presented at FoVeOOS, 2010. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=138696`.

[6] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. ISSN 0001-0782. doi: 10.1145/363235.363259.

[7] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL)*, pages 42–54. ACM, 2006. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111042.

[8] Kenneth L. McMillan. Applications of Craig interpolants in model checking. In *TACAS (Tools and Algorithms for the Construction and Analysis of Systems)*, volume 3440 of *LNCS*, pages 1–12. Springer, 2005. ISBN 3-540-25333-5. doi: 10.1007/11494744_2.

[9] David Monniaux. The pitfalls of verifying floating-point computations. *Transactions on programming languages and systems (TOPLAS)*, 30(3): 12, May 2008. ISSN 0164-0925. doi: 10.1145/1353445.1353446. URL `http://hal.archives-ouvertes.fr/hal-00128124/en/`.

[10] DO-178B. *Software considerations in airborne systems and equipment certification (RTCA/DO-178B, EUROCAE ED-12B)*. RTCA, Inc., December 1992.

[11] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In *FM 2009: Formal Methods*, volume 5850 of *LNCS*, pages 532–546. Springer, 2009. ISBN 3642050891. doi: 10.1007/978-3-642-05089-3_34.

[12] Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. Static timing analysis for hard real-time systems. In *VMCAI (Verification, Model Checking, and Abstract Interpretation)*, volume 5944 of *LNCS*, pages 3–22. Springer, 2010. ISBN 3642113192. doi: 10.1007/978-3-642-11319-2_3.