ON THE INDUSTRIAL FITNESS OF WCET ANALYSIS

Enrico Mezzetti¹, Tullio Vardanega¹

Abstract

The process requirements that govern the development of high-integrity real-time systems make timing analysis an ineludible concern. Conceptually, the problem space of timing analysis includes the determination of the best, average and worst-case bounds for the execution time of the program parts of interest. As the problem space is vast, as often is the program to analyse, industry seeks tools and methods that can address its need effectively, that is to say, with a decent cost-benefit ratio. Static analysis is widely acknowledged as the most authoritative means to derive safe bounds on the worst-case execution time (WCET). The WCET in turn is the prerequisite input to feasibility analysis. Without WCET, feasibility analysis is just pointless. In terms of cost-benefit ratio, the value of feasibility analysis must be not inferior to the joint cost of obtaining the WCET values, ensuring the compliance of the system (at least in the worst case) to the analysis model, and running the analysis itself. It is not a given that this equation always holds in practice. When it does not, it is important to understand what are the impediments and how they can be slashed. Static WCET analysis is exposed to known fragilities in terms of cost efficiency and value tightness. Yet, the important progress achieved in the research around it suggests that the “WCET problem” is virtually solved, and quite satisfactorily so for simple single-processor architectures. The industrial ground, however, is the sole terrain where the truth of that claim can be ascertained. In this paper we discuss lessons learned from an experiment, massive for size, duration and effort, aimed to the timing analysis of a significant component of the software application embedded onboard a commercial satellite system. We discuss the limitations which we incurred in our application of static WCET analysis, highlighting those which we consider intrinsic to the method itself when confronted with the challenges of industrial-scale systems.

1. Introduction

In high-integrity real-time software systems, not providing a required service within the expected deadline equates to a system failure. The computation of a safe bound to the worst-case execution time (WCET) of each system activity is a prerequisite to the sound application of feasibility analysis, which provides dependable guarantees on the timing behavior of the system. However, determining safe and tight WCET bounds by static analysis or measurements is almost invariably a challenging and onerous activity, which adds to the already labour-intensive element of verification and validation required of critical systems. In the last decade, massive research effort has been devoted to the WCET problem and several approaches have been proposed to increasingly tighten safe bounds. The important advancements achieved in both theory and technology feed the widespread belief in the research community that the WCET analysis problem is virtually solved for simple single-processor architectures. The wavefront of research is consequently moving towards advanced processor architectures and multi-processors.

¹University of Padua, Department of Pure and Applied Mathematics, Padua, Italy; Email: {emezzett, tullio.vardanega}@math.unipd.it
At the same time, however, several industrial case studies have reported issues in the application of WCET analysis techniques to industrial systems [16, 5, 7, 11]. Whereas some studies report fully successful application of those techniques [17], a lesson commonly drawn from them suggests that, in spite of significant theory advances and the availability of powerful tools [1, 15], analysing the WCET behavior of real industrial-scale programs remains an extremely onerous and challenging task, even for comparatively simple hardware architectures, regardless of the used method and technique.

The difficulty in analysing industrial-scale systems stems from intrinsic limitations in the currently available WCET analysis techniques [9, 10], which seldom become apparent in laboratory experiments. The central problem is the poor scalability of the assumptions on which WCET analysis techniques rest, which break when faced with scorching reality. The conclusion we draw in this regard is that a not negligible gap still exists between state-of-the-art claims and state-of-practice facts.

In this paper we report on issues we experienced in attempting the static timing analysis of a real, complex on-board satellite software system. Based on that effort, we discuss misconceptions and pitfalls with WCET analysis techniques from an industrial standpoint. The remainder of this paper is organized as follows: Section 2 briefly surveys the state-of-the-art WCET analysis techniques and illustrates the key requirements from the industrial development process; Section 3 presents the industrial case study we carried out; Section 4 discusses a number of misconceptions and pitfalls in WCET analysis; Section 5 concludes by elaborating on some promising approaches to facilitate WCET analysis in industrial-scale settings.

2. Industrial-level WCET Analysis

In the high-integrity industry, the software development must comply with the process requirements set forth by the applicable domain-specific standard (e.g.: RTCA/DO-178B in avionics systems). As a result, a score of design, implementation, verification and validation activities must be carried out in order that the system can be declared predictable and dependable. Regardless of the specific application domain, the determination of the WCET bound for every critical task is a mandatory precondition to the execution of feasibility analysis, which sanctions the predictability of the system.

Two main approaches are currently available for the computation of WCET bounds or estimates: static analysis and measurement-based methods. Static WCET analysis techniques [18] try to compute a WCET bound for a given program from the binary and/or source code of the software and an abstract model of the hardware. The combination of the results from hardware timing analysis and program flow analysis can produce a safe WCET bound, contingent on the correctness of the underlying models.

The availability of accurate hardware models is increasingly defied by novel hardware architectures. This has been the main reason of the recent rise of interest in hybrid and measurement-based approaches. Hybrid techniques [4], in particular, attempt to compute realistic (hence tight but not necessarily safe) WCET estimates by first measuring the execution times of small program fragments (essentially equivalent to basic blocks) on the real execution platform, and then using control-flow analysis to derive from the resulting values the sought WCET estimate for the program.

This notwithstanding, static WCET analysis is to date the only approach that can compute safe WCET bounds and, as such, it is the single most natural candidate for use in critical systems.

The WCET bound, however, should not only be safe but also tight, that is to say, with as little over-
estimation as possible in comparison to the longest execution time observed at run time. The more pessimism in the WCET bound the less the usefulness of it. Static analysis in general suffers from inherent sources of overestimation, which impair the tightness of the computed bounds. The principal difficulties come from the need to abstract the execution contexts the analysis has to consider, and the inclusion of infeasible execution paths in program flow analysis. To counter the induced overestimation, the user is required to provide manual annotations (a.k.a. flow-facts) to assert infeasible paths, define branch and indirect call targets, refine loop bounds and support path-analysis in general.

Although state-of-the-art static WCET analysis techniques have been successfully integrated in industrial quality commercial tools, such as aiT [1] from AbsInt GmbH, their penetration in the development processes of high-integrity industry is still insufficient primarily because of the fundamental misalignment between the "mind-setting" of those tools and the mundane industrial needs, requirements and capabilities. People in industry acknowledge the fact that static analysis would be the solution to WCET determination, but they very well perceive – and fear – the costs that factual application of it would add to an already onerous development process. With efficiency and return on investment as main concerns, industry will not budge and not embrace a new, onerous, practice without confidence that the attainable benefits exceed the induced costs.

In this respect, the common industrial perception of static WCET analysis is that the quality of its results does not match the costs it incurs for large complex systems. On the one hand, as no fully automated solution is attainable, large effort and important skills are called upon to aid the analysis. Providing trustworthy manual annotations, in fact, requires very profound knowledge and understanding of the program behavior, which are both fragile (owing to incorrect beliefs) and volatile (owing to personnel mobility) assets even in high-integrity industry. On the other hand, the unavoidable overestimation incurred by static analysis is likely to increase with the complexity of the program to be analysed, thus producing exceedingly pessimistic bounds that rapidly lose industrial value. Consequently, the most common industrial practice to obtain WCET values still relies on software simulation and testing, prudently reinforced by the application of (supremely un-scientific) "safety margins".

In a recent, long-duration experiment on the static timing analysis of a satellite on-board software system we encountered difficulties that largely confirmed us in the opinion that state-of-the-art WCET analysis approaches are not up to real-life complex industrial software as yet. In the following sections, we first provide a characterization of a typical on-board software, both in terms of dimension and real-time features (which both vastly affect WCET analysability); then, on the basis of the actual case study, we discuss some of the limitations we observed in the static analysis methods.

3. Zooming on a Real-Scale Satellite On-Board Software

For obvious reasons of self-protection, industry is reluctant to disclose the internals of their products to outsiders, academic or professional that they may be. On the rare occasion in which this happens, the value of the availed information goes much beyond the case in question. Depending on the application domain, in fact, software systems are known to exhibit distinctive features (including architecture and functionality) that are generally shared among products of the same family, so that each product is in a way representative of the whole category.

This paper draws from the experience gathered in statically analysing the WCET behavior of a real-scale sample of a complex satellite on-board software (OBSW), made available by ThalesAlenia Space/France (TAS/F). In our experiments, we were granted unrestricted access (source code and
executable) to the GB2 OBSW, recently developed by TAS/F for the second-generation of the Globalstar\(^2\) Low Earth Orbit (LEO) communication satellite constellation. The GB2 OBSW runs on top of the 32-bit SPARC V8 LEON2 [2] processor, which is a relatively simple hardware architecture from the WCET analysis standpoint. For the timing analysis of the AOCS part of the GB2 OBSW we used *aiT for LEON2* by AbsInt [1].

The GB2 OBSW is a complex system that caters for the full range of attitude control, command, and operation services of the satellite. In architectural terms, the GB2 OBSW can be regarded as a layered composition of several company-standard, application-dependent and domain-specific functional blocks. Figure 1 depicts a high-level view of the GB2 OBSW architecture and highlights its four principal macro-layers and functional blocks, whose chief purpose is to facilitate software reuse across systems. The topmost layers, Mission Control and Application, provide for system and equipment management, and sub-system functionalities respectively. The Middleware layer just beneath is related to I/O and run-time support and rests directly upon the Hardware Software Interface layer. It is worth noting that, from the timing analysis perspective, all these layers are likely to be involved in the computation of the WCET behavior of the OBSW core tasks.

A communication channel abstraction, termed Software Bus, is responsible for all data transfers between all software modules, whereas communication between the spacecraft and the ground station is provided by means of Telemetry and Telecommand services, which comply with satellite link standards that vary (though only slightly) with the end customer.

![Figure 1. Schematic view of the GB2 OBSW architecture.](image)

The sheer scale of the GB2 OBSW, its real-world architecture and code structure are highly representative of typical software in the aerospace domain. More specifically, the software system reflects architectural principles that are consolidated across the whole industrial domain. It therefore presents aggressive challenges to the application of WCET analysis techniques and tools. A large part of the GB2 OBSW is the product of automated code generation from high-level models. Its overall source code base spans over 1,700 files, each with some 240 lines of code on average, for a total of 410K lines and almost 9 MB of executable. The large number of files is one of the direct consequences of automated code generation. The GB2 OBSW is implemented in Ada but it also includes some C and assembly code files, the latter related to legacy code and board-specific components. All source code has been compiled with the GNAT Pro 5.03a for LEON and ERC32 gcc-based compiler by AdaCore\(^3\).

The GB2 OBSW run-time architecture is intensively concurrent and uses complex patterns of inter-task interaction. The provision of the system functionalities is apportioned to some 30 tasks, which follow the classic taxonomy of periodic, sporadic and aperiodic tasks, and share a common address


\(^3\)http://www.adacore.com.
space. Inter-task communication is prevalently performed via shared resources, whether logical or physical. Inter-processor communication uses the MIL-STD-1553B serial bus, which imposes strict serialization on data-oriented synchronization among remote tasks.

The GB2 OBSW tasks often are the top of complex call graphs, which in some cases involve more than 1,700 procedures. As it is typical of control algorithms, the code structure at subprogram level is mainly sequential and rich with deeply-nested branches. Loops, though present, typically iterate over small bounds; only the data handling component, which serves the commands issued from ground, uses loop-intensive constructs to navigate complex space-efficient data structures.

The underlying real-time kernel supports priority-based preemptive scheduling, along with task synchronization, interrupt handling and exception management.

For our investigation we selected the Attitude and Orbit Control System (AOCS), on account of both its hard real-time nature and computational intensiveness. The AOCS is multi-moded, hence it serves the full set of “modes” which the satellite system can operate in. When in nominal mode, the functional behavior of the AOCS consists in a time-critical periodic control loop, which reads data from sensors (e.g.: gyroscopes, star trackers, GPS receivers, sun sensors), computes corrective actions to the satellite attitude, orbit and velocity, and sends commands to actuators (e.g.: reaction thrusters, momentum wheels, magnetic torquers). The AOCS activities are performed by a single periodic task whereas the interaction with the sensor and actuator equipments uses the I/O subsystem and is carried out by a set of sporadic and aperiodic tasks.

4. Misconceptions and Pitfalls of WCET Analysis

Let us now discuss the limitations of static WCET analysis techniques as we experienced them in our experiment. In doing so, we do not imply that those limitations are exclusive of static analysis techniques: we are well aware in fact that some of them are suffered by measurement-based approaches too. We leave out from our discussion those issues that may originate from unattended diligence to critical prescriptions of WCET analysis techniques or faulty applications of them. We therefore assume that a correct abstract processor model is available (for as much as this is increasingly less obvious) and abstract away from tool-specific considerations. We focus on what we consider to be the intrinsic limitations of WCET analysis and show how they are exacerbated by the complexity and costs incurred by applying it to industrial-scale software systems.

**Computational tractability** The first hurdle is the sheer dimension of the problem at hand. Computational tractability problems for static analysis are known to originate from the presence of complex hardware features (e.g.: caches, out-of-order pipelines), which cause the state space of the abstract processor model to explode in the attempt to fully and accurately account for the inner state of all the hardware features with bearing on the WCET.

Our experience in this regard is that the computational tractability problem arises even for simple processor models, such as the LEON2 processor used in our experiment. In contrast with other reported field studies (e.g.: [5] in the automotive domain) the program we analysed features a massive call graph, which involves in excess of 1,300 procedures (again largely a result of automated code generation) and, when drawn by an analysis tool, looks like the maze shown in Figure 2. In our case, the complexity hits the roof as the joint consequence of the very large number of execution contexts and the additional complication incurred in inter-procedural analysis. And the latter, unfortunately, can only be attenuated at the cost of unacceptable pessimism in the analysis results.
Besides tractability in space, we observed that even the computation time required by the analysis steps may dramatically increase. Selecting the entry point for the analysis fixes the combined super-graph\(^4\) to be traversed: its dimension and complexity determine the time to be spent in the WCET computation. Even if the super-graph was large, this would not be much of a problem if the analysis could be fully automated and it was sufficient to execute it just once without the need for any further iteration. In actual fact instead, the user is often forced into an extremely onerous (and frustrating) trial-and-error process of uncovering and fine-tuning the annotations required by the analysis.

The scale of the job at hand also poses the problem of figuring the more economical and feasible strategy to attack the super-graph, between starting from either the program root or the leaves. Both approaches, in fact, have pro and cons. Trying to analyse a program from the outermost root procedure makes it possible to best exploit the information automatically gathered by the analysis tool (e.g.: from inter-procedural data-flow analysis), but also exposes to the need of running several time-consuming analysis iterations – over the whole state space – as the tool itself will very likely fail multiple times asking the user for more annotations at each iteration. The alternative strategy, which consists in providing manual annotations upward from the leaf subprograms in the graph, allows the analysis to operate on a reduced state space. At the same time, however, the user is likely to provide pessimistic annotations owing to the lack of automatically extracted call-context information.

**Information gathering** An inescapable assumption in WCET analysis is that the user should be able to stipulate flow facts, in the form of manual annotations to the program, to assist the analysis process. User annotations are required when the analysis automaton is not able to reconstruct the call graph or to bound the loops, or just to shed pessimism off intermediate results.

Unfortunately, the assumption that the user can be a trusted source of flow facts is extremely fragile for large, complex and long-lived industrial programs. This is what we experienced in our experiment: determining flow facts for the GB2 OBSW, which is not old and obsolete code, but very central to current development, was extremely difficult if at all possible. The main difficulties we encountered in this regard are ascribable to the three following traits of the timing information.

- **Opacity**: the information, even if ultimately present, is often not available in forms intelligible to the average user. Leaving aside the object-code inserts which serve to adapt legacy code to specific platforms, the code generation engine of compiler back ends is a prolific source of idioms, which for example determine how the program control flow is reorganized, introduce loops and branches not traceable to the source code (as for e.g.: array slice assignment). It is well known that those idioms can vastly complicate static analysis in terms of the level of intelligence required and of command of source-to-object mapping of user annotations;

\(^4\) A joint representation of the call graph and the control-flow graph of each sub-procedure.
- **Non-locality**: the resolving information is not always local to where the analysis needs it. For example, the information may be hidden in a complex hierarchy of user-defined types or packages. Without prior knowledge of the overall architecture of the code (which, be it understood, is almost never localized in a single individual for a real-scale system and tends to gel instead in complex build commands) retrieving the needed information requires a massive search over large slabs of the code base, the cost of which grows inordinately with the complexity and articulation of the system. A typifying example of this problem is the mode management software, which encompasses whole libraries of specifications for the range of operational phases of the satellite lifetime, whose intelligence is necessarily scattered all over the place to reflect mode hierarchies and permit overriding and specialization throughout them;

- **Unavailability**: besides the plain lack of information that stems from black-box third-party components, the sought information may not even be available at all in the program at hand as it may depend on higher level design choices, far removed from it. Data and code structures whose size and range are determined by global variables or global effects fall in this category. In systems for which memory is a scarce resource it is not rare that buffers are statically allocated and then used with context-dependent bounds: if you look at the static bound you may be vastly pessimistic, but the actual source of the local bound may be very difficult to track down.

The most annoying aspect in the chore of gathering flow facts retrospectively is that a large proportion of that information had almost certainly been dissipated during the development process. To make things worse, rebuilding that knowledge base is not just costly but also limited in accuracy and completeness. It has long been acknowledged that the concern of timing analysis should accompany the development process from the outset, however there always seems to be legacy software that defies that countenance. Furthermore, with the growing recourse to automated code generation off high-level models, it is becoming increasingly evident that the logic of code generators follows principles that antagonize timing analysis and predictability\(^5\). For example, dispatching calls that can be statically solved within the model and thus simplify timing analysis may stay dynamic because that better suits the style of code generation.

**Code Analysability** The quality of the results of static WCET analysis depends on the analysability of the user program and the accuracy of the user annotations. Several constructs are known to be difficult to analyse and thus detrimental to the results of static WCET analysis. The effects of dynamic memory allocation, dynamic references, indirect calls, irreducible loops or recursion are only marginally contained by the use of increasingly elaborate annotation languages [8]. It has long been suggested that the code constructs that exceedingly complicate analysis and may incur overestimation should be informedly avoided. However, the tacit assumption or naive belief that such offending constructs are not present in real-life industrial software are just equally ill based. Dynamic references are typically adopted for data manipulation; recursion and indirect calls are naturally required by generic functionalities (e.g.: Failure, Detection Isolation and Recovery); and irreducible loops may be deftly introduced by the compiler, depending on the level of optimization.

The problem is quite evidently in the fact that no coding rule and practice, however intelligent it may be, operate retroactively: they have no effect whatsoever on the large volume of legacy code, which is a most valuable industrial asset and as such it is preserved across products. Legacy code is cryptic because slabs of it may have been written so many years before that insufficient knowledge has been retained on its internals. Yet this very same legacy is extremely precious, for it represents

\(^5\)This does not generally hold for purely functional code generated from simple models (e.g.: Simulink, SCADE).
intellectual heritage, proven by use, of prior development, which is therefore – ironically – much more trustworthy than new code. It is consequently obvious that industry is not willing to endure the cost, effort, time and potential regressions that may arise from the re-engineering of legacy, unless the expected benefits are far superior and guaranteed.

**Genericity vs. predictability**  
Technical and economic reasons push high-integrity industrial developers toward the reuse of software across homogeneous products. Software reuse, as a fundamental requirement of industrial software, is pursued at both architecture and code level. One of the resulting traits is the adoption of coarse interfaces between architectural building blocks or the use of configurable components and generic constructs (e.g.: the Ada generic packages). All of those programming techniques affect WCET analysis. The use of coarse interfaces hides the information on data and types, which may induce static analysis to pessimism, for example, in bounding data-dependent loops. In our experiment we hit this in the analysis of procedures related with the Software Bus component, which, as an abstract communication channel, cannot expose a strictly typed interface.

A similar effect is caused by the use of generic package constructs. In this case, data-dependency in the program is increased by the fact that, depending on the specific compiler in use, distinct instantiations may partially share object code. An example is the implementation of distinct static lists whose size depends on the type of the elements they should contain. The use of generic package instantiation in combination with coarse data types does not only compromise the tightness of analysis results but also seriously affects its complexity, as it increases the number of execution contexts that have to be accounted for.

**System level timing analysis**  The effects of task interactions and interleaving is certainly not a negligible issue in complex preemptive systems. Static WCET analysis is typically kept separate from system level issues and is focused on the timing behavior of individual tasks, under the simplifying assumptions that the analysed task runs in isolation. It is up to feasibility analysis to account for the costs of context switching and external interference suffered by the task owing to the effects of preemptive concurrency on history-dependent shared physical resources (e.g.: the cache).

This separation between intra- and inter-task analysis is very delicate and the placement of the boundary between the local and the global worst-case (in the absence of anomalies) is exceptionally difficult when the execution scenario becomes as complex as we outlined in Section 3. Despite the progresses made in calculating the context-switch cost [3], self-suspension as well as the firing of timer alarms and watchdogs on history-sensitive hardware are difficult to apportion to intra- or inter-task analysis, either of which must be very sophisticated to handle them all satisfactorily. One might argue that static scheduling, architectural partitioning and isolation should mitigate those problems. In fact, they are no panacea, because what they gain in analysability they lose in rigidity and underutilization, for no global win.

**Industrial fitness**  The final question we posed ourselves at the end of our experiment is whether the application of static WCET analysis techniques and tools meets the industrial requirements and fits in the industrial development process. The WCET analysis method, in fact, should be able to accommodate the peculiarities of the industrial development process. At least two such characters appear to be specific to the space domain:

- **Incrementality:** The development process proceeds incrementally in the implementation, integration and qualification of the system. Incrementality is prerequisite to return on investment
from the reuse of architectural building blocks. This vision of incrementality was the exact rationale of the architecture of the OBSW we analysed.

- **Patchability**: Satellite systems have a long operational life, which may span 7 to 15 years on average, sometimes even longer. During their lifetime, dynamic reconfiguration and code patches are both normally and exceptionally applied to the onboard system.

Static timing analysis currently leaves both needs unattended. Static analysis in fact is inherently not incremental as it critically relies on the availability of information that can only be safely determined on the final executable, near the end of the development process (e.g.: actual code and data layout). Moreover, the state of the art on component-level WCET analysis [14, 12] is still largely immature; as a consequence, **timing composability** [13] among software modules is not attainable with common hardware and software architectures, as yet. Incrementally adding a module to a software system may thus affect the timing behavior of preexisting modules. Hence the WCET bound obtained on a previous release for a given module is likely to become invalid in a subsequent incremental release.

The same holds true for in-flight code patching as any little modification in the program code may require to reapply large part of the analysis. This results from the fact that WCET analysis, which operates at a very low level of abstraction, is extremely sensitive to even seemingly minuscule program and system changes and must address many if not all of those changes with new annotations or the modification of preexisting ones, which both cause unwelcome additional costs.

5. **Conclusion**

Inherent limitations in combination with the intrinsic complexity of real-world systems still hinder the extensive industrial application of static WCET analysis. From the perspective of the industrial segment we investigated, static WCET analysis can only be fully embraced if its cost-benefit ratio turns to positive. As a result of all the issues and limitations we discussed, a score of conservative (i.e., pessimistic) assumptions, annotations and stubbing are the inevitable price to pay in carrying out the analysis to the end. This incurs the risk that the results may be scarcely satisfactory and difficult to defend against the more familiar maximum-observed execution time. Furthermore, the integration of static analysis itself in the industrial development process continues to be problematic. Some high-priority common industrial requirements, such as software reuse, incremental development and in-flight modification, are still difficult to account for.

Although no easy solution exists to reconcile WCET analysis and development practice, promising approaches are maturing. From a software perspective, more insightful approaches to automated code generation may help to bridge the gap between the theoretical assumptions of WCET analysis and the factual reality of industrial software. Indeed, a large part of the flow fact information (e.g.: on dynamic calls and loop bounds) is very likely – and economically – present at design time but is lost along the development process. The educated use of automated code generation from high-level models may feed flow facts directly from the outset of the information. Automated code generation is also the most effective way of imposing predictable code patterns and coding styles that guarantee code analysability by construction. The frightful effort of re-engineering the mass of legacy code might be much attenuated by the fit-by-iteration gear of fast turn-around automated code generation. Likewise, the drawbacks of coarse interfaces and generic programming could be eliminated by expressing genericity at model level instead of code level.

With respect to the complexity problem, instead, two possible approaches can be devised. The more
conventional one aims at reducing the sources of timing variability (and thus the state space) by moving towards more deterministic hardware designs [6]. It is worth noting, however, that the increase in predictability comes at the cost of reduced performance, which is still a non-negligible issue in high integrity system (although not the primary one). A more innovative approach, instead, promotes a paradigm shift that aims to eradicate the dependencies of timing behavior on the execution history, while at the same time benefiting from complex hardware acceleration features. This approach, which builds on probabilistic hardware and analysis techniques, is currently being studied as part of the EU funded PROARTIS project⁶.

Acknowledgments

The authors wish to acknowledge Gérald Garcia and Yann Gouy from TAS/F Cannes for their help, support and intense discussions, and AbsInt GmbH for the gracious provision of aiT for LEON2.

References


⁶http://www.proartis-project.eu/.