ASSUME

ITEA 3

Affordable Safe & Secure Mobility Evolution

State-of-the-art and technology Y2

Deliverable D6.6.2

Deliverable Information			
Nature	Document	**Dissemination Level**	Public
Project	ASSUME	**Project Number**	14014
Deliverable ID	D6.6.2	**Date**	28.11.2016
Status	Final	**Version**	0.6
Contact Person	Moharram Challenger	**Organisation**	UNIT
Phone	+90 (232) 339 6633	**E-Mail**	Moharram.challenger@unitbilisim.com

Author Table

Name	Company	Email
Moharram Challenger	UNIT	Moharram.challenger@unitbilisim.com
Reinhold Heckmann	AbsInt	heckmann@absint.com
Didem Unat	Koc University	dunat@ku.edu.tr
Serdar Taşıran	Koç University	stasiran@ku.edu.tr

Change and Revision History

Version	Date	Reason for Change	Affected sections
0.1	03.11.2015	Initial version	
0.2	20.11.2015	Extension of initial version	2, 4.1, 5
0.3	26.11.2015	Modification and further extension	
0.4	01.12.2015	Updated version Submitted to Reviewers	
0.5	15.12.2015	Review finished	
0.6	28.11.2016	Additions of second year's SotA	1,3,4.2,5

Table of Contents

1. Executive Summary

In order to consider and keep up with the up to date science and technology, related work and tools are analysed in each technical work package. In this deliverable, the state of the art and technology are collected and published for the first and second year of the ASSUME project. This document will be updated based on the new studies and technologies in the next years of the project.

2. Scalable Zero-Defect Analysis for Single-Core Systems (WP2)

Avionics and automotive software development features a rich and multi-step validation and verification (V&V) process. It is however essentially based on conventional testing techniques, for which required coverage metrics and requirements are defined in international standards (e.g., ISO 26262 for automotive applications). Conventional V&V requires a significant and ever growing portion of the overall development effort. With rising system complexity, it is on the brink of becoming the bottleneck of today's processes.

2.1. Static analysis of run-time errors

Sound static analysis (SSA) is a promising technique to improve the situation. It allows the analysis of software on unit level. In contrast to testing, it achieves complete control and data coverage of software by employing conservative over-approximations [1]. Thus SSA allows, under favourable circumstances, to prove the total absence of certain kinds of errors, in particular run-time errors (RTE) [2].
Most SSA tools are limited in scalability and precision. A single analysis run can take several days, limiting their application to components of small size. The results may include thousands of false (spurious) alarms, leading on some projects to economic ineffectiveness due to high efforts inspecting by hand these alarms.
The state of the art in SSA for RTE on embedded C programs is Astrée, an analyzer developed by ENS and industrialized by AbsInt [3][4]. The limit on the precision of Astrée has its origins in the necessity for making approximate (abstract) computations, in order to scale up to large programs. In the past, it has been shown that by tailoring the abstractions to a specific class of properties and programs, the goal of zero false alarms can be achieved for synchronous embedded avionic and space software [5][6]. More research is necessary before generic libraries of abstractions are available to handle other common cases found in embedded software.

2.2. Analysis of interactions

Faults in complex industrial systems may result from complex hidden dependencies between interacting components. Existing tools do not allow for architecture and design verification of complex interactions (e.g. where dependencies between components are hidden in a communication layer or where call-back mechanisms are used). Therefore, to achieve the zero defect goal, architecture and design principles have to be improved and their fulfilment verified using new more powerful static analysis tools. Moreover, with the recent development of cyber-physical systems in safety relevant areas, the amount of interactions with the system context grows tremendously. Consequently, future systems will have to ensure safety and security to a much greater extent. While safety analysis focuses on the reliability and correctness of the software, approaches to security analysis have to examine the software against risks resulting from interactions through high level and low level software interfaces. Today's analysis tools do not provide sufficient support for safety and security analyses, although it is highly demanded.

2.3. Model-based development and integration with static analysis

Model-driven development has been used to a rising degree in automotive industries, including functional models (in Simulink or ASCET) and meta-models which capture relevant meta-data. AUTOSAR and also the meta-model of the project AMALTHEA are prominent examples. Model-driven development is also common in the avionics industry (LUSTRE and SCADE). The model

information is often ignored by SSA tools, leading to needlessly difficult analysis problems and a loss of precision. There are however some examples for the integration of model-based code generators and static analysis tools. An integration between AbsInt's WCET analyser aiT and Esterel's SCADE generator has been established in the projects INTEREST and INTERESTED [7]. In ALL-TIMES, a first integration between aiT, Astrée, and TargetLink from dSPACE has been set up [8].

Moreover, there is a need to check beyond published modelling guidelines (such as MAAB or MISRA) and company specific rules, to include quality criteria such as maintainability, changeability and expandability. Analyses for coupling, cohesion and encapsulation are already available for non-model based development, but not for model-based development. They are needed to prevent the introduction of defects resulting from side effects or insufficient understanding of the software system while modifying the code. In model-based development of large and complex models the same risks occur even more dramatically since the availability of software engineering principles in this field is very limited. Advanced methodology as well as convenient tool support is required for the quality analysis of models to prevent the introduction of defects during future development and maintenance activities.

3. System engineering methodology and standards (WP3)

Automotive system engineering is founded on a wide set of well established, proven and tested processes ranging from requirements elicitation to system verification and validation. Many of these processes comprise dedicated engineering approaches targeting particular system quality aspects like e.g. correctness, safety, security, and many more. Even though these aspects are often combined together, synergies between these approaches are seldom recognized. Significant benefit is thus expected from coherently applying these engineering techniques continuously throughout the development process, i.e. from requirements elicitation to system verification and validation. We provide state of the art practice in system engineering methodology and standards but more detailed discussion can be found in D3.1.

3.1. *"Roadblocks"*

For example, at Daimler, as an OEM, automotive system development starts at system level where the realization and deployment of functions is not clear at the beginning. The validation and verification of system requirements is executed in a multi-step process and supported by several tools and models. These models show certain aspects of the modelled system. The purpose of this approach is to improve the system understanding. Models serve also as source for verification. Since models typically are much simpler than their final source code representation verification tasks become better realizable.
However, the verification of a model with respect to a certain requirement does not guarantee that the implementation does not violate that requirement. To ensure this, the code or at least certain parts have to be generated from these models with a sound code generator or some verification technologies have to be applied. In certain cases higher-level models are extractable from the source. An example here is the extraction of the task model to prove the absence of raise conditions. In other cases such a higher-level model is very hard to extract. The code that is generated from Matlab/Stateflow is an example here.
Today many analysis tools work on code or even binary code level. For the efficient verification they often lack of information that is present but not easily accessible at this level. Due to this fact and due to technological borders, the application of static analysis tools today requires a high effort for setup and parameterization. Nevertheless many false alarms are produced causing significant effort for rework. Accompanied with the mentioned development process is the requirement of traceability. Available traceability solutions today are very limited and usually show only some aspects. Hence sound automatic impact analyses are difficult to execute. The pervasive traceability of requirements as well as faults requires the seamless integration into the development lifecycle of software-based vehicle functions running on multi-core embedded systems. This comprises data models, description languages, tools and methodology.
Today, the source to gain performance is parallelization. Single core CPUs have nearly reached their limits in that respect. Multi- and many-core CPUs are state of the art in hardware technology. However, the development or the migration of existing software to concurrent application that exploits the CPU resources is an art itself and not well supported by tools. Hence, the effort to migrate an existing application to a multi-core processor causes much effort today.

3.2. *Requirement Formalization & Impact Analyses*

Key results relevant for the ASSUME have been created within the ARTEMIS project CESAR, addressing the lack of requirements quality that often leads to additional efforts, cost overrun and

schedule drifts in downstream development activities. One means to improve requirements quality is to formalize requirements using boilerplates, domain ontologies and patterns in order to allow automatic analysis and test generation. Key results of ITEA2 SAFE, relevant for ASSUME, are the methodology and pervasive consideration to analyses on functional safety for electric / electronic architectures of vehicles in the concept phase, and represented by static architectural models. This includes formalized safety requirements engineering and –management, the derivation of the safety case, pervasive traceability from requirements to detailed hardware models running the embedded software and the analysis and evaluation of this hardware in terms of fulfilling the safety requirements using industrial standards as E.g. AUTOSAR, EATOP and PREEvision.

WP3 develops new patterns to blend functional requirements with timing requirements. We improve the consistency analysis to capture these new patterns. In near future, we further develop new formalization and analysis techniques to meet the industrial needs and investigate new pattern's integration to the existing tools such as BTC EmbeddedSpecifier and IBM DOORS.

3.3. Interfaces of Tools & Traceability

Quality assurance is integral part in model-based SW development. Today, several tools are applied to address the broad range of quality requirements. Proper tracking of product quality requires much manual work and is thus error-prone. Tight analysis integration would provide means to compile quality results in a uniform and centralized fashion taking into account not only design and modelling tools, but also analysis tools for determining different properties of a system under development and proving correctness of the system under various aspects such as functional behaviour, timing and safety. Consequently, the different models, generated source code and analysis results have to be related in order to ensure traceability of the development artefacts created during the process.

3.4. Standards for Semantic Interoperability

Further needs arise in semantic interoperability between methods and tools. Some standards and exchange formats (e.g. AUTOSAR [O7]) exist, which facilitate the integration of architecture and behaviour modelling tools, and code generators. While analysis tools usually support interfaces to such standards, the integration of the analysis tools themselves is often considered using ad hoc solutions. There have been efforts in different research projects like MBAT, ARAMiS and Amalthea to come up with a more systematic integration approach. In MBAT a prototypical tool coupling between BTC's EmbeddedTester and Astrée has been developed, with the goal to applying model-based testing to automatically find test cases for alarms reported by the static analysis. In the SAFE project, the data models of AUTOSAR, the initiative EATOP, tools from Dassault Systemes and PREEvision along with the respective tools were combined to facilitate pervasive traceability and analysis in architectural models. In the ARAMiS project the interoperability of design and analysis tools for multi-core systems was addressed. An option consists in developing in-house integration platforms, generally based on internal and proprietary point-to-point solutions. A second option consists in relying on commercial integration platforms implemented by well-established tool providers, e.g., PTC Integrity, IBM Rational Jazz, Siemens PLM Teamcenter, Dassault Enovia, Tasktop Sync, or Systemite System Weaver.

The CESAR project offered customizable systems engineering providing interoperability of existing or emerging technologies. This project constitutes a milestone for a European standardization effort. Reference Technology Platform (RTP) defines basic services and their interfaces to perform specific design steps. RTP led the development of Interoperability Specifications (IOS) enabling seamless implementation of the whole design flows. Similar interoperability challenges are addressed by two other German projects, namely SPES 2020 and ARAMiS. These two projects aim to define common vocabulary for software-oriented systems engineering.

4. Synthesis of predictable concurrent systems (WP4)

4.1. Verification of compilers and code generators

Compiler and automatic code generators are essential tools to bridge the gap between models and executables. Sequential code generation from a synchronous language like Scade 6 can be formalized as a series of source-to-source and traceable transformations that progressively eliminate high-level programming constructs (hierarchical automata, activation conditions, sequences) down to a minimal data-flow kernel which is further simplified to a generic intermediate representation for transition functions, and ultimately turned into C code. These tools are vulnerable to miscompilation risks: a bug in the compiler or code generator causing it to produce incorrect object code from a correct source program. These risks are difficult to address in the context of critical embedded software qualified at the highest assurance levels: a few code generators have been qualified at level A of DO-178B (e.g. the Scade KCG6 generator), but no optimizing C compiler. A radical way to eradicate the miscompilation risk and provide high assurance is to formally verify the compilers and code generators themselves, using program proof. The flagship of this approach is the CompCert C compiler, developed at Inria Gallium: an optimizing C compiler that is proved to be free of miscompilation bugs using the Coq proof assistant. CompCert provides provably correct mechanisms to trace properties of the source program down to the machine code, and is now in the pre-industrial phase via a collaboration with Airbus. The CompCert compiler has been licensed by AbsInt for further extensions of its capabilities and full industrialization. The full formal verification of a code generator from a modelling language such as Scade remains to be done.

4.2. Relaxed memory models

Sequential consistency (SC), coined by Lamport [28], is an idealized semantic model for describing the behaviours of concurrent programs. It describes executions of concurrent programs as total orders over the set of program statements in which the program orders of the individual threads/processes are preserved. Although this definition gives us a clear and easy understanding, it is not realistic. Many modern hardware architectures (including Intel-x86, PowerPC ARM and GPUs) and programming language specifications (like C, C++, 2011) allow more behaviours than SC ones due to performance reasons. Hence, their semantics are relaxed with respect to SC.

The documentation that describes memory subsystems of modern processors often lack formal precision and they are even inconsistent with the actual behaviours of the system at some points due to incorrectly implemented hardware. Extensive tooling was developed to perform model-based testing of processors and compilers, leading to the discovery of hardware bugs (acknowledged as such by the manufacturers) in the Power5 and Cortex A9 processors. Hence, there has been a notable effort to develop precise semantic models for these systems. They can be classified under two groups. Axiomatic models ([15], [29]) describe executions as unions of some relations over events and/or memory accesses and memory models as restrictions on the relations that define executions. Authors in [29] introduce a formal hierarchy of SC, RMO (relaxed memory order), PSO (partial store order), TSO (total store order) and Alpha memory models based on axiomatic models and proofs developed on Coq proof system. On the other hand, operational models ([21], [30], [31]) depict the behaviour of actual hardware components, abstracting them through data structures, such as queues. Most of the current research [13-21]

formalizes semantics of relaxed memory models of the system they study as an example of one of these classes.

Since SC is clear and powerful enough to reason about concurrent programs, it is desirable by the programmers. Adve and Hill [32] coined the term weak ordering as an interface between hardware and software. Given a restriction on the shared memory accesses of the programs as a synchronization model, hardware is weakly ordered with respect to this synchronization model if all the programs that obey the synchronization model show only SC behaviours. Hence, if the programmer writes a program obeying the synchronization model of a weakly ordered hardware, then s/he can reason this program as if it is SC. Similar definition of weak-ordering exists for programs and it is called robustness. A program is robust (or stable) if every weak memory behaviour of it corresponds to some SC behaviour. [29] and [33] propose a method for checking robustness. It characterizes robustness as acyclicity of a particular happens-before relation in the axiomatic model. However, the method in [33] is incomplete in the sense that it may label a program as non-robust although it is robust. [16] provides a complete decision procedure for checking robustness in terms of TSO programs.

In some applications, correctness is much more important than the performance. In this situation, the programmer may agree to sacrifice performance to get rid of non-SC behaviours of the program, which might be unprecedented and erroneous. For this reason, a line of research developed for enforcing robustness on the programs by using synchronization primitives. The most commonly used primitives are memory fences which force programs to wait until some memory accesses become visible to all other processing units. Since the fence causes processing units to wait, it may degrade the program performance. Therefore, inserting as few fences as possible is crucial for the minimum performance degradation. Initial theoretical results for finding minimal fence insertions that forces robustness date back to 1988 [34]. Authors in [29] extend this algorithm to particular weak memory models and fence types. Authors in [16] propose an optimal fence insertion algorithm as a modification of their robustness check algorithm, which minimizes a particular cost function. Another group provides a dynamic and efficient fence insertion algorithm, which is neither complete nor optimal [35].

There are recent studies on the verification of programs running on weak memory models. Successful methods has been developed and used for verification of SC programs (like Owicki-Gries, reduction, concurrent separation logic etc.) for a long time. There are recent attempts to extend these techniques to weak memory settings. [17] provides an Owicki-Gries kind of reasoning model for weak memory programs. [20] develops the relaxed separation logic (RSL) which can be used to verify programs in release/acquire fragment of C11 specification. A novel approach for verifying compiler optimizations is presented in [31]. This study considers possible statement rewritings or reorderings as compiler optimizations. Correctness of these optimizations depends on the underlying memory model of the platform that the program will run. For instance, reordering consecutive global read and write statements by the same thread is allowed by TSO memory model. Hence performing this reorder during the compilation period does not add any new behaviour to the program and it is valid for TSO. However, this reordering is not allowed by SC and it cannot be allowed as a valid optimization on an SC platform. To prove validity of given transformations on given memory models, the authors provide necessary conditions to be checked.

Important preliminary studies on verified code generation for weak memory models has begun to emerge recently [13], however, such work often assumes that inter-thread and inter-task interference has been already ruled out through other verification tools. Verified refinement of programming language code to executable machine code for weak memory models remains an unsolved problem.

4.3. Synthesis of critical real-time software for multi-processor architectures

Much of the classical work on real-time scheduling (both in research and the industry) relies on a process where the implementation is derived by manual transformations. Implementation is followed by verification and validation phases where timing analysis and schedulability analysis guarantee the respect of non-functional requirements. But today, the complexity of the multi-processor execution targets and the complexity of the functional and non-functional specifications increase rapidly, which makes it difficult to preserve a manual process (for cost, time-to-market, and/or confidence issues related to the number of errors introduced by human coders). Some important advances in this direction have largely automated the construction of task code and even the generation of full real-time implementations without providing schedulability guarantees or optimized mapping algorithms aimed at providing such guarantees. Work on optimized mapping still has to be integrated in standard industrial tooling. INRIA proposed methods and tools in this direction, namely the AAA methodology and the SynDEx and LoPhT tools for optimized real-time mapping of synchronous/reactive specifications onto multi-processor (distributed/multi-/many-core) targets.

4.4. Automotive applications

During the last years multi-core µC have entered the automotive domain. The arising challenge is to bring all existing and future SW from single core implementations and development processes into the new highly concurrent world. In industrial setting this transformation is still done by manual injection of inter-process communication and synchronization code. In addition, the mapping of runnable entities to different cores is also done manually. This injection of primitives is manual work, and hence prone to errors, and the runnable distribution has a huge impact on computation efficiency. Formal models of computation exist, but are currently not used for the multi-core SW engineering. Static analysis is in most cases restricted to the analysis of non-concurrent SW.

5. Zero-defect analysis for multi-core systems (WP5)

5.1. Static analysis of concurrent multi-core applications

Sound static analyzers (such as Astrée) have been successfully applied to check run-time errors in safety-critical sequential software, but far less tools are available for the analysis of concurrent software. Polyspace Code Prover can identify shared variables accessed by concurrent threads, but cannot precisely identify data races and lacks OS support so that OS-related information has to be provided manually. Earlier versions of the state-of-the-art industrial analyzer **Astrée** have been restricted to analyzing sequential code and did not support natively task-interleaving. To overcome this restriction ENS has developed **AstréeA**, a research prototype extending Astrée to check for run-time errors in multi-task C software consisting of millions of lines of code [9]. Concurrency effects like preemptions, task priorities, and critical sections can be soundly and precisely taken into account. AstréeA provides mechanisms to model operating systems by mapping the OS functionality to efficient stub libraries. In the course of the FORTISSIMO project the AstréeA mechanisms have now been transferred to Astrée and have been further enhanced. Currently OS support is provided for avionic software running under an ARINC 653 OS [10], and automotive software running under OSEK and AUTOSAR OS [11]. In ASSUME, the AUTOSAR support, which had been limited to system specifications in .oil format, has been extended to the service libraries, as, e.g., CAN and DEM. Furthermore some OSEK/AUTOSAR OS mechanisms like phases of execution, the priority inheritance protocol, and enabling/disabling interrupts have been modelled in the FORTISSIMO project. Detection of deadlocks has been added in ASSUME. Now, Astrée can find concurrency-specific faults, including detection of data races, deadlocks, and priority inversions.

The static analyzer framework **Goblint** is another emerging academic tool for concurrent programs. It has been elaborated in the scope of the MBAT project to prove the absence of data races in concurrent code as well as in interrupt-driven OSEK applications. The resulting prototype was imprecise with respect to global data, and not able to precisely model sophisticated synchronization primitives such as sending and receiving of events or suspending and resuming of tasks (often employed in embedded software to enforce scheduling policies). In ASSUME, the precision has been improved by better handling of casts between different types, support for context-sensitive warnings has been added, and the regression test infrastructure has been improved.

A third tool, the **MEMICS** analyzer, was developed in the ARAMiS project to detect race conditions in concurrent software by bounded model checking. Its focus is on the analysis of low-level code (close to machine code). It incorporates an elaborate memory model including `malloc` and `free` and deals well with pointer structures.

Another outcome of the ARAMiS project is the **Gropius** analyzer, a static analysis tool focused on concurrency errors arising in automotive software. In ASSUME, the tool was tested with real industrial code, which showed limitations in the tool design. A redesign of the tool in ASSUME led to an increase in efficiency of the tool and a reduction of the number of false positives during the analysis of industrial code.

5.2. Deductive methods

Program proofs for concurrent programs were pioneered by the Calvin and QED tools. Current tools include VCC, which operates on concurrent C programs annotated with specifications and

invariants and proves them correct using the Z3 SMT solver; Chalice, a modular verification tool for a dedicated concurrent language; and CIVL (Concurrency Intermediate Verification Language), which verifies refinement for concurrent programs in various different languages after translation into a common intermediate format. VCC and Chalice base their invariant reasoning on objects, object ownership, and type invariants. VCC does not support refinement and Chalice does so only for sequential programs; neither support movers nor reduction reasoning. Finally, concurrent separation logic reasons on concurrent programs without explicit non-interference checks. State-of-the art tools are able to blend this logic with explicit non-interference reasoning.

5.3. Dynamic race detection

Runtime verification and dynamic analysis fill an important gap between static analysis and testing. While static tools are conservative which may lead to a large false alarm rate, testing catches errors late, making it difficult to find their cause. Runtime verification, on the other hand, provides early error detection during execution. For instance, dynamic race detection tools, such as Goldilocks and FastTrack, instruments a program with code that detects data races while the program is running. However these tools often suffer from significant execution slowdown. To reduce this slowdown, a variety of techniques have been explored. Some approaches improve performance by sacrificing precision, i.e., missing some races. They accomplish this by sampling the accesses performed, e.g. ThreadSanitizer [36] and RACEZ [37]. Speeding up race detection and/or replay by parallelization has also been explored, e.g. in the GPU-accelerated split race checker Kuda and DoublePlay (parallelizing sequential logging and replay) [40]. Others, e.g., HARD (Hardware-Assisted lockset-based Race Detection) [38] and Paralog (enabling and accelerating online parallel monitoring of multithreaded applications) [39] make use of custom hardware to accelerate race detection and similar parallel program monitoring techniques.

The following commercial tools can be used or adapted to detect races on some particular embedded computing platforms:

- Intel Inspector XE, PIN dynamic instrumenter
- Valgrind DRD
- Helgrind
- Parallocity ZVM-K (ARM)
- Google ThreadSanitizer

While these tools have been used in commercial applications with some success, the algorithms underlying them are often not precisely documented and each of them may need some adaptation and modifications before they can be used on any particular code base and application.

Dynamic race detection for embedded systems has unique challenges. These include the mixed use of variables of different, often quite small, bit lengths [22], the use of task-based concurrency with priorities and interrupts [23-25] rather than threads and concurrency libraries, and issues relating with the platform on which development and testing is performed to the one on which the applications will finally run [26-27].

In particular, [22] proposes a dynamic race detection algorithm based on vector clocks by considering the granularity of program data (i.e.; words, bytes, bits, etc.) that is common in embedded systems. The main motivation is to improve data race precision as opposed to other

race detection solutions, which do not consider various data sizes in the program. In [23] an on-the-fly technique that efficiently detects apparent data races in interrupt-driven programs without false positives is presented. The technique combines a tailored lightweight labelling scheme to maintain the logical concurrency between a program and every instance of its interrupt handlers with a precise detection protocol that analyzes conflicting accesses to shared memories by storing at most two accesses for each shared variable.

Interrupt-driven programs where inconsistent ordering (races) of interrupt events could result in non-determinism in the program. To detect these kinds of races, the algorithm in [24] *sequentializes* the program and applies model checking. However, this solution does not focus on multi-threads programs because the program under consideration is single-threaded event-driven. A failure that is caused by interruption handler that modifies a certain variable between a reference or modification to the variable and a later reference to the variable is defined as a race in [25]. The proposed solution in [25] is to generate an interrupt at the instruction points that possibly cause race conditions and replace input value from external device to control interrupt handlers. This covers all possibilities of sharing memory between the interrupt handler and other routines that would cause data races.

To improve race detection performance in embedded systems, [26] employs hardware registers originally added to processors, for debugging by, to watch the traffic along the data and instruction buses. This improves analysis of races compared to software instrumentation based techniques.

Testing method for identifying faults in multitasking applications for embedded systems is proposed in [27] where intra and inter task analysis to generate test cases is used to improve the observability of faults.

In ASSUME, we aim at a race detection approach that can be adapted to a variety of platforms and applications, including possibly interrupt-driven ones, and one whose overhead-precision trade-off can be adjusted by the programmer.

5.4. Worst-case execution time (WCET)

Worst-case execution time (WCET) analysis on multi-core architectures has been considered in recent projects: Predator, T-Crest, Certainty, and parMerasa. In ARAMiS, an approach was proposed for computing an interference-sensitive Worst-Case Execution Time (isWCET) taking into account variable access delays due to the concurrent use of shared resources in multi-core processors [12]. The state of the art can now handle single-core executions without interference or when the number and kind of interference points can be determined. For time composable architectures, this is sufficient to obtain an overall WCET. There have also been recommendations for hardware configurations increasing predictability and composability.
Recent results by INRIA showed that precise and scalable timing analyses can be achieved on selected parallel applications (using for instance the Heptane WCET analyzer). The analysis has the precision and scalability of classic IPET-based WCET analysis.
Timing analysis on concurrent task execution at the system level can also be used to reason about potential race conditions, as part of the concurrency defect analysis. Two classes of analyses can be identified. Analytical methods determine performance characterizations, such as response times of task chains, by solving fixed point equations. Popular approaches include

SymTA/S and the Real-Time Calculus. Periodic resource models provide compositional methods, focusing on partitioned resources. Computational methods, on the other hand, rely on model-checking techniques, where the system behaviour is represented as a state transition system. For example, the model checker UPPAAL can be used for scheduling analysis, as well as the related TIMES tool. While computational methods typically provide better results, e.g. a reduced number of false positives, they also lack in scalability due to computational complexity. Scalability improvements have been proposed, e.g. as part of the COMBEST project. However, no computational analysis exists with integrated methods reliably preserving appropriate precision of the results.

6. Related Projects

Name Program Period	Technical Focus	Relationship
CompCert French ANR 2005-2009	Formal verification of compilers	First explorations of compiler verification using Coq.
ES_PASS ITEA 2 2007-2009	Embedded Software Product-based Assurance	Improvement and integration of the Astrée tool used in ASSUME.
COMBEST FP7 IST STREP 2008-2010	Computational and analytical models for non-functional properties of embedded systems. Methods and tools for rigorous embedded systems design.	Combination of different analysis techniques and tools.
PARSEC FUI 2009-2012	Model-driven engineering for critical distributed systems	Collaboration with Thales SA towards defining a development environment for critical distributed embedded systems requiring certification according to strict standards such as DO-178B (avionics) or IEC61508 (transportation).
ARAMiS German BMBF 2011-2014	ARAMIS develops methods and techniques for optimized use of Multi-Core architectures with respect to development standards in the transportation domain such as ISO 26262.	ASSUME will develop models and interchange formats for the analysis of single and multi-core software. Functional as well as non-functional properties will be taken into consideration. The ARAMIS meta-model for scheduling and timing will be taken into account to enrich the interfaces of the ASSUME platform. ARAMIS methods regarding the analysis of multi-core systems will be developed further in the ASSUME project including the MEMICS tool.
Amalthea(4public) ITEA 2 2011-2014, and 2014-2017	AMALTHEA4public will built a continuous **development tool chain platform** for automotive embedded **multi-core systems** based on results of various public funded projects by using the AMALTHEA methodology.	ASSUME extends the scope of AMALTHEA beyond timing and HW resource modeling and simulation. ASSUME derives a methodology to analytically calculate data for the AMALTHEA meta-model (in contrast to measuring and simulation).

Name Program Period	Technical Focus	Relationship
MBAT ARTEMIS 2011-2014	Combination of model-based analysis and testing.	Traceability between Requirements, Design and V&V artefacts. Extensions of the Astrée and Goblint tools used in ASSUME.
ParMerasa FP7 2011-2014	The objective of parMERASA (Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability) is a timing analysable system of parallel hard real-time applications running on a scalable multi-core processor.	The idea of analysable systems with regard to timing will be expanded in ASSUME by the analysis of functional and various non-functional properties in multi-core systems.
PHARAON FP7 2011-2014	Parallel and Heterogeneous Architectures for Real-Time Applications	Parallelization of soft real-time programs for low-power embedded architectures, based on task-parallel data-flow languages and model-driven engineering.
SAFE ITEA 2014-2017	The SAFE project brings solutions to demonstrate the compliance to the ISO26262 functional safety standard for the development of safe automotive applications based on the AUTOSAR architecture.	While SAFE focuses on architecture modelling in the concept phase of system development ASSUME will target the synthesis and analysis of implementation and behaviour models. Interfaces to SAFE will be explored regarding the traceability from concept models to implementation models in the development of safety-relevant functionality.
Verasco French ANR 2012-2015	Joint verification of compilers and static analyzers	Collaboration with Airbus towards the industrialization of CompCert.
ESPRESSO Swedish FFI 2012-2015	Modelling and analysis methodology, Guidelines and tool recommendations for model-based engineering of embedded system at Scania, Application and evaluation of the developed concepts	Traceability across the engineering phases (based on a use case from Scania)
CRYSTAL ARTEMIS 2013-2016	Interoperability of System Engineering Methods	Requirement Formalization, CCC (Correctness, Completeness, Consistency).

Name Program Period	Technical Focus	Relationship
AstréeA French ANR 2012-2015	Static analysis of concurrent programs.	Developed a research prototype of the AstréeA tool used in Assume.
Fortissimo German BMBF 2014-2016	Formal Analysis and Verification of Concurrent Hardware and Software	Develop the AstréeA tool as part of the commercial Astrée analyser. Addition of execution phases and priorities to the concurrency model of Astrée.
CAPACITES French Investissements d'Avenir 2014-2017	Parallel computing for safety-critical real-time applications	Collaboration with Kalray SA, Airbus, SAGEM, Dassault Aviation, MBDA on the construction of many-core safety-critical real-time applications. This includes mixed-criticality applications with focus on the spatial partitioning enforced by the architecture, and latency-critical applications expressed in an idiomatic form of OpenCL that enables worst-case response time analysis.
EMC2 ARTEMIS	EMC² finds solutions for dynamic adaptability in open systems, provides handling of mixed criticality applications under real-time conditions, scalability and utmost flexibility, full scale deployment and management of integrated tool chains, through the entire lifecycle.	ASSUME complements EMC² by analysis methods supporting the (1) validation of functional safety concepts including dynamic system adaptation and the (2) verification of technical safety concepts (against these functions) based on mixed-criticality multi-core platforms, thereby providing the needed arguments for certification purposes.
SPES_XT BMBF 2012-2015	The Project SPES_XT develops a seamless integration platform for modelling and analysis techniques for embedded systems.	The methods and techniques developed in ASSUME will be constructed to support the SPES methodology. They will provide new building blocks for the requirements viewpoint and the functional viewpoint in the SPES matrix. The demand of SPES_XT to implement the results in the industrial practice will be subsequently supported by the use case driven approach of ASSUME.

7. Conclusions and Discussion

This deliverable presents the state of the art and technology regarding the ASSUME project. The related technologies for the main work packages of the project including WP2 (Scalable Zero-Defect Analysis for Single-Core Systems), WP3 (System engineering methodology and standards), WP4 (Synthesis of predictable concurrent systems), and WP5 (Zero-defect analysis for multi-core systems) are discussed. Also, the related projects for ASSUME are elaborated in the document.

References

[1] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 238–252, New York, NY, USA, 1977. ACM Press.

[2] 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 Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03), pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[3] 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.

[4] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, X. Rival. Astrée: Proving the Absence of Runtime Errors. Embedded Real Time Software and Systems Congress ERTS², Toulouse, 2010.

[5] D. Delmas and J. Souyris. ASTRÉE: from Research to Industry. In Proc. 14th International Static Analysis Symposium (SAS2007), number 4634 in LNCS, 2007.

[6] O. Bouissou, É. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, É. Goubault, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, M. Turin. Space software validation using abstract interpretation. In Proc. of the International Space System Engineering Conference on Data Systems in Aerospace (DASIA 2009), volume SP-669, 7 pages, Istanbul, Turkey, May 2009.

[7] C. Ferdinand, R. Heckmann, T. Le Sergent, D. Lopes, B. Martin, X. Fornari, F. Martin. Combining a High-Level Design Tool for Safety-Critical Systems with a Tool for WCET Analysis on Executables. 4th European Congress ERTS - Embedded Real Time Software, Toulouse, 2008.

[8] D. Kästner, C. Rustemeier, U. Kiffmeier, D. Fleischer, S. Nenova, R. Heckmann, M. Schlickling, C. Ferdinand. Model-Driven Code Generation and Analysis. SAE World Congress 2014, available at http://papers.sae.org/2014-01-0217/

[9] 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.

[10] 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.

[11] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, C. Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. 8th European Congress ERTS - Embedded Real Time Software, Toulouse, 2016. To appear.

[12] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, M. Schmidt. Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. 26th Euromicro Conference on Real-Time Systems (ECRTS '14), pages 109-118. IEEE Computer Society, 2014.

[13] Jagannathan, S., Laporte, V., Petri, G., Pichardie, D., & Vitek, J. (2014). Atomicity refinement for verified compilation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *36*(2), 6.

[14] Jagannathan, S., Laporte, V., Petri, G., Pichardie, D., & Vitek, J. (2014). Atomicity refinement for verified compilation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *36*(2), 6.

[15] Sarkar, S., Sewell, P., Nardelli, F. Z., Owens, S., Ridge, T., Braibant, T. & Alglave, J. (2009, January). The semantics of x86-CC multiprocessor machine code. In *ACM SIGPLAN Notices* (Vol. 44, No. 1, pp. 379-391). ACM.

[16] Bouajjani, A., Derevenetc, E., & Meyer, R. (2013). Checking and enforcing robustness against TSO. In *Programming Languages and Systems* (pp. 533-553). Springer Berlin Heidelberg.

[17] Lahav, O., & Vafeiadis, V. Owicki-Gries Reasoning for Weak Memory Models, In ICALP 2015: 41st International Colloquium on Automata, Languages, and Programming.

[18] Hawblitzel, C., & Petrank, E. (2009, January). Automated verification of practical garbage collectors. In *ACM SIGPLAN Notices* (Vol. 44, No. 1, pp. 441-453). ACM.

[19] Yang, J., & Hawblitzel, C. (2010, June). Safe to the last instruction: automated verification of a type-safe operating system. In *ACM Sigplan Notices* (Vol. 45, No. 6, pp. 99-110). ACM.

[20] Vafeiadis, V., & Narayan, C. (2013, October). Relaxed separation logic: A program logic for C11 concurrency. In *ACM SIGPLAN Notices* (Vol. 48, No. 10, pp. 867-884). ACM.

[21] Burckhardt, S., & Musuvathi, M. (2008, January). Effective program verification for relaxed memory models. In *Computer Aided Verification* (pp. 107-120). Springer Berlin Heidelberg.

[22] **"Efficient Data Race Detection for C/C++ Programs Using Dynamic Granularity"** Young Wn Song; Yann-Hang Lee, 2014 IEEE Parallel and Distributed Processing Symposium,

[23] **"Verification of Data Races in Concurrent Interrupt Handlers"** Guy Martin Tchamgoue, Kyong Hoon Kim, and Yong-Kee Jun, International Journal of Distributed Sensor Networks, 2013

[24] **"Data Race Detection for Interrupt-Driven Programs via Bounded Model Checking"** Xueguang Wu, Yanjun Wen, Liqian Chen, Wei Dong, Ji Wang, SERE-C '13 2013 IEEE Conf. on Software Security and Reliability, 2013

[25] **"An effective method to control interrupt handler for data race detection"** Makoto Higashi, Tetsuo Yamamoto, Yasuhiro Hayase, Takashi Ishio, Katsuro Inoue, AST '10, Proceedings of the 5th Workshop on Automation of Software Test, 2010

[26] **"E-RACE, A Hardware-Assisted Approach to Lockset-Based Data Race Detection for Embedded Products "** Lily Huang, Michael Smith, Albert Tran, James Miller, 24th IEEE Symposium on Software Reliability Engineering (ISSRE), 2013

[27] **"An approach to testing commercial embedded systems"** Tingting Yu, Ahyoung Sung, Witawas Srisa-An, Gregg Rothermel, Journal of Systems and Software archive Volume 88, February 2014

[28] Lamport, L. (1979). **How to make a multiprocessor computer that correctly executes multiprocess programs.** *IEEE transactions on computers*, *100*(9), 690-691.

[29] Alglave, J. (2010**). *A shared memory poetics*** (Doctoral dissertation, Université Paris 7).

[30] Boudol, G., & Petri, G. (2009, January). **Relaxed memory models: an operational approach.** In *ACM SIGPLAN Notices* (Vol. 44, No. 1, pp. 392-403). ACM.

[31] Burckhardt, S., Musuvathi, M., & Singh, V. (2010). **Verifying local transformations of concurrent programs.**

[32] Adve, S. V., & Hill, M. D. (1990, May). **Weak ordering—a new definition**. In *ACM SIGARCH Computer Architecture News* (Vol. 18, No. 2SI, pp. 2-14). ACM.

[33] Alglave, J., & Maranget, L. (2011, July). **Stability in weak memory models.** In *International Conference on Computer Aided Verification* (pp. 50-66). Springer Berlin Heidelberg.

[34] Shasha, Dennis, and Marc Snir. "**Efficient and correct execution of parallel programs that share memory**." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10.2 (1988): 282-312.

[35] Liu, F., Nedev, N., Prisadnikov, N., Vechev, M., & Yahav, E. (2012). **Dynamic synthesis for relaxed memory models.** *ACM SIGPLAN Notices*, *47*(6), 429-440.

[36] Konstantin Serebryany and Timur Iskhodzhanov. 2009. **ThreadSanitizer: data race detection in practice**. In Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09). ACM, New York, NY, USA, 62-71.

[37] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. 2011. **RACEZ: a lightweight and non-invasive race detection tool for production applications**. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, NY, USA, 401-410.

[38] P. Zhou, R. Teodorescu and Y. Zhou, "**HARD: Hardware-Assisted Lockset-based Race Detection**," *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, Scottsdale, AZ, 2007, pp. 121-132.

[39] Evangelos Vlachos, Michelle L. Goodstein, Michael A. Kozuch, Shimin Chen, Babak Falsafi, Phillip B. Gibbons, and Todd C. Mowry. 2010. **ParaLog: enabling and accelerating online parallel monitoring of multithreaded applications**. In Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS XV). ACM

[40] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. **DoublePlay: parallelizing sequential logging and replay.** SIGPLAN Not. 46, 3 (March 2011), 15-26