A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimisation

D6.1 AMPERE ecosystem requirements and integration plan

Version 1.0

Documentation Information

Contract Number	871669
Project Website	www.ampere-project.eu
Contractual Deadline	30.07.2020
Dissemination Level	PU
Nature	R
Author	Sara Royuela (BSC)
Contributors	Eduardo Quiñones (BSC), Arne Haman (Bosch), Claudio Scordino (EVI), Delphine Longuet (TRT)
Reviewer	Marco Merlini (THALIT)
Keywords	Software Architecture, Integration plan, Interfaces among SW components

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 871669.

Change Log

Version	Description Change
V0.1	Initial draft by Sara Royuela and contributions by Eduardo Quiñones
V0.2	Contributions from EVI, BOSCH and TRT
V0.3	Reviewed by THALIT
V1.0	Ready for submission

Table of Contents

1. Executive Summary

This deliverable covers the work done during the first phase of the project within work package 6 (WP6) "*AMPERE System Design and Computing Software Ecosystem Integration*". The deliverable spans 7 months of work and describes the work done in Task 6.1 "*AMPERE ecosystem requirement specification*" to reach milestone MS1.

The deliverable describes the AMPERE software development ecosystem upon which the AMPERE use cases will be developed, deployed and executed. Concretely, it identifies the set of software components and tools that will form the AMPERE ecosystem, and it provides a short description of each component.

One of the key features of the AMPERE ecosystem will be its capability to instantiate multiple software architecture configurations, incorporating different software components. The objective of this feature is to cover different system necessities to fulfill performance requirements, while guaranteeing the non-functional requirements (NFR) imposed by the cyber-physical interactions. This key feature requires each software component to define a clear interface, described in this deliverable, to ensure a seamless integration with the rest of components.

Finally, this deliverable also provides the development and integration plan for the AMPERE project. It includes the tools, techniques and methodologies that will be shared among all partners in order to ensure the quality of the final product.

The first milestone of Task 6.1 has been carried out successfully and all objectives of MS1 have been reached and documented in this deliverable.

2. The AMPERE Software Development Ecosystem

This section describes the components forming the AMPERE software ecosystem and the relations among them.

2.1. Overview

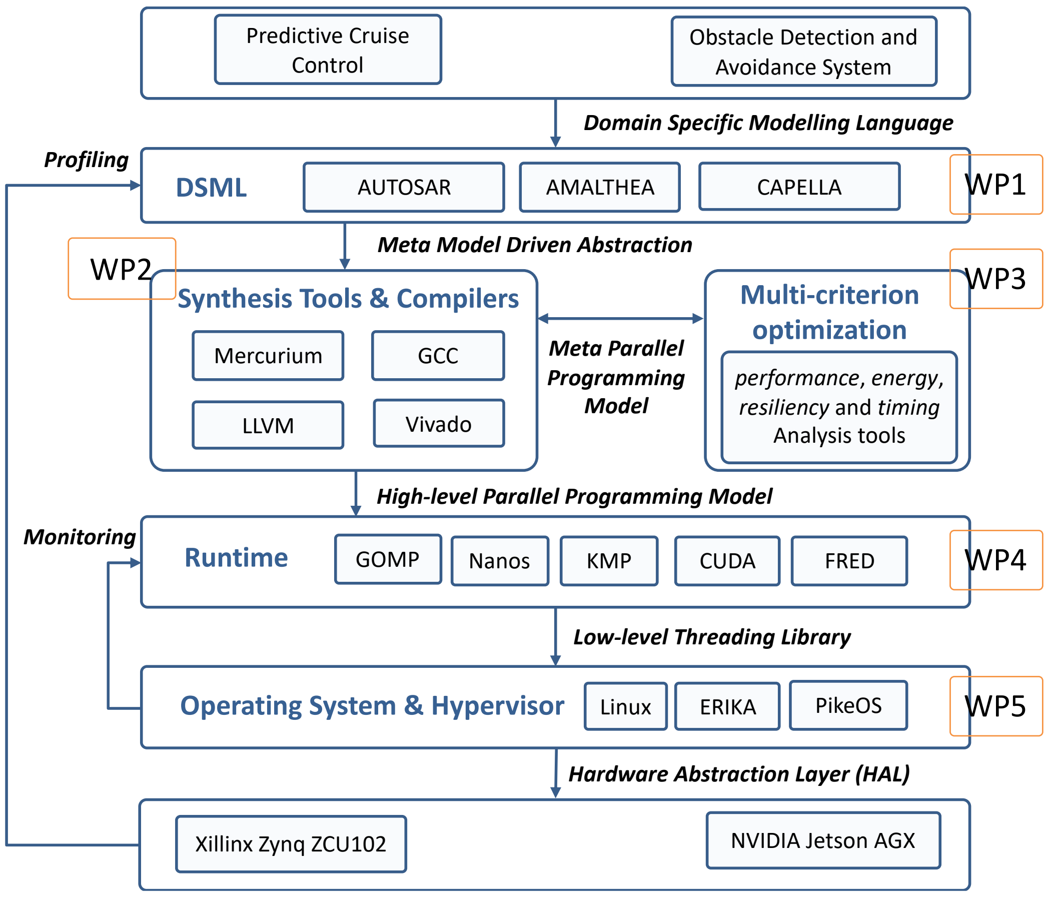
AMPERE addresses each of the requirements later defined in Section 3 by developing a software development ecosystem incorporating components from multiple computing areas, including embedded computing, cyber-physical systems (CPS), software engineering, data analytics, and high-performance computing (HPC). This unique (an heterogeneous) combination of software components aims to efficiently transform high-level model driven engineering systems into highly efficient and scalable parallel software including the non-functional requirements of the system at programming model, compiler, runtime, operating system (OS) and architectural levels.

Figure 1 shows the overall AMPERE software development ecosystem including the main software components. Concretely, it consists of:

- The **Domain Specific Model-Driven** layer (WP1). This layer contains the selected *Domain Specific Modelling Languages (DSMLs) interface* that will be used to develop the AMPERE use-cases. Additionally, it also defines the *meta model driven abstraction interface* for the underlying components to communicate with this layer. AMPERE aims to consider the following DSMLs: AMALTHEA [1], CAPELLA [2] and AUTOSAR [3] and AUTOSAR ADAPTIVE [4].
- The **code synthesis tools** and **compilers** layer (WP2). This layer contains the software tools (compilers like Mercurium [5], GCC [6] and LLVM [7], and FPGA frameworks like Vivado [8]) in charge of generating the optimized parallel code based on the requirements specified in the DSML and the information gathered by the tools for multi-criterion analysis. For this purpose, it also defines the *meta parallel programming model interface* gathering the information exposed in the meta model driven abstraction and the results of the analysis of the multi-criterion optimization

layer. The meta parallel programming model will be then transformed to the underlying *high-level parallel programming models (PPMs)* supported by the processor architecture. The actual transformation will be performed by the compilers. The high-level PPMs act as the interface of the underlying parallel runtime frameworks responsible of orchestrating the parallel execution.

- The **multi-criterion optimization** layer (WP3). This layer corresponds to the tools in charge of statically analyzing the non-functional requirements (NFR) of the system, as described in the meta parallel programming models and the meta model driven abstraction interfaces. Based on this information, the previous layer will have the required information (through the meta parallel programming model interface) so the transformation to the final parallel code can be done according to the results of the analysis phase.
- The **runtime** layer (WP4). This layer corresponds to the parallel runtimes, including GOMP [9], Nanos [10], KMP [11], Vivado [8] and CUDA [12], and is in charge of fulfilling the requirements defined at analysis phase while ensuring the performance. This layer will rely on the low-level PPM interfaces supported by the underlying OS, being Pthreads [13] the most commonly used in Linux.
- The **operating system (OS)** and **hypervisor** layers (WP5). This layer corresponds to the OS, including Linux [14] and ERIKA Enterprise [15] (the latter implements the AUTOSAR middleware), and the hypervisor, PikeOS [16], in charge of providing safety and security system guarantees through isolation mechanisms. The hypervisor provides the *Hardware Abstraction Layer (HAL) interface* responsible of interacting with the underlying parallel processor architectures; in AMPERE, the Xilinx Zynq ZCU102 and the NVIDIA Jetson AGX will be supported (see Deliverable D5.1 [17] for further details).


Predictive Cruise Control
Obstacle Detection and Avoidance System
Domain Specific Modelling Language
Profiling
DSML
AUTOSAR
AMALTHEA
CAPELLA
WP1
Meta Model Driven Abstraction
WP2
Synthesis Tools & Compilers
Mercurium
GCC
LLVM
Vivado
Meta Parallel Programming Model
WP3
Multi-criterion optimization
performance, energy, resiliency and timing Analysis tools
High-level Parallel Programming Model
Monitoring
Runtime
GOMP
Nanos
KMP
CUDA
FRED
WP4
Low-level Threading Library
Operating System & Hypervisor
Linux
ERIKA
PikeOS
WP5
Hardware Abstraction Layer (HAL)
Xillinx Zynq ZCU102
NVIDIA Jetson AGX
Figure 1. AMPERE Software development ecosystem: components, relationships and interfaces.

2.2. Software Components

AMPERE has carefully selected the software components that will form the AMPERE ecosystem, prioritizing those owned by the AMPERE partners or offered as open source with a large community behind. By doing so, we envision to reduce the time-to-market and maximize the exploitation opportunities of the AMPERE ecosystem.

Table 1 identifies the initial set of software components that aims to be included in the AMPERE ecosystem, the WP in which the component will be evaluated, the owner and the license. This list will be revised (and updated if needed) in Deliverable D6.2 "Refined AMPERE ecosystem interfaces and integration plan", at MS2.

Table 1. Initial selection of AMPERE software components at MS1.

Software Layer / WP	Software component	WP	Owner	License
Domain Specific Modelling Language (DSML)	AUTOSAR	WP1	AUTOSAR	Proprietary
	AMALTHEA		BOSCH (BOS)	Open source
	CAPELLA		Thales Research and Technology (TRT)	Open source
Artificial Intelligence	TensorFlow	WP1	Google	Open source
Parallel Programming Models	OpenMP	WP2	OpenMP	Open source
	CUDA		NVIDIA	Proprietary
Compilers and Hardware Synthesis Tools	Mercurium	WP2	BSC	Open source
	GCC		GNU	Open source
	LLVM		LLVM	Open source
	Vivado		Xilinx	Proprietary
Analysis and Testing Tools	Multi-criteria Analysis and Testing Tools	WP3	AMPERE	Open source
Runtime Libraries (RTL)	GOMP	WP4	GNU	Open source
	Nanos		BSC	Open source
	KMP		LLVM	Open source
	Fred		SSSA	Open source
Operating Systems	Linux	WP5	Linux	Open source
	ERIKA Enterprise		Evidence (EVI)	Open source
Hypervisors	PikeOS		SYSGO (SYS)	Proprietary

Next subsections briefly introduce the different components of the AMPERE development ecosystem.

2.2.1. DSML

The development of the CPS is described using the DSML provided by the model driven framework. Different frameworks support different DSML to describe the software components and elements that form the system (at different granularity level) and the communication mechanisms existing among them. The most important software components of the four DSMLs considered in AMPERE are presented in Table 2 (see Deliverable D1.1 [18] for further details).

Table 2. Software components included in the considered DSML.

DSML	Main software components
AMALTHEA	An open-source modeling and development platform for automotive applications (compatible with AUTOSAR), which defines the following main software abstractions[1]: • Tasks / ISRs (Interrupt Service Routines) representing schedulable entities including activation information (Stimulus Model) • Constraint model for defining timing constraints, e.g. for tasks end event chains • Event model for describing causal relationships and end-to-end event chains • Runnables describing the actual executed code including information about execution time and accessed labels (=memory access patterns) • Labels (shared variables) exchanged by Runnables • Channels describing larger data elements stored in memory, useful for describing e.g. streaming data • Modes for defining different operational modes of the software • Mapping model describing the mapping of SW entities to HW elements • Scheduler model and Scheduler allocations to define scheduler policies and responsibilities.
AUTOSAR Classic/Adaptive	A methodology for component-based development of automotive software for electronic control units (ECUs). The relevant modeling concepts of AUTOSAR classic are compatible and covered in a very similar manner by AMALTHEA. Advanced topics addressed by AUTOSAR adaptive, mainly POSIX-based operating systems publish-subscribe messaging patterns, service-oriented middlewares, can also be covered by AMALTHEA in a similar fashion.
CAPELLA	An open-source methodology and tool for model-based system engineering, which provides SysML-inspired diagrams at different levels of abstraction to describe the architecture and the behaviour of a system. It offers extensions for dedicated modelisation, enabling for simulation and analysis, for example for timing requirements of multi-processor real-time systems, through the Tideal extension and the Time4Sys platform implementing the UML-MARTE meta-model. Examples of concepts supported by the real-time extension of Capella: • hardware resources (processors) • tasks with best- and worst-case execution times, deadlines, priorities, activation mode (periodic, sporadic, bursts…) • task dependencies • various scheduling policies (EDF, Round Robin, FIFO, Priority Based…)

2.2.2. Parallel Programming Models

The parallel programming models are the tools used to translate the information specified in the DSML into the executable code that better exploits the architecture while ensured the NFR. This deliverable considers four different parallel programming models: OpenMP, OmpSs, OpenCL and CUDA. The most important features of them are summarized in Table 3 and Table 4 (see D2.1 [19] for further details).

[1] Find more information at: https://www.eclipse.org/app4mc/help/app4mc-0.9.8/index.html#section3.14

Table 3. Comparison of the presented parallel programming models based on parallelism patterns and architecture abstraction.

	Parallelism			Architecture abstraction		
Parallel Programming Model	**Data parallelism**	**Asynchronous task parallelism**	**Host/device**	**Abstraction of memory hierarchy**	**Data and computation binding**	**Explicit data mapping**
OpenMP	parallel for simd	task/taskloop	Host and device (target)	OMP_PLACES, teams and distribute	proc_bind	map(to|from|tofrom|alloc)
OmpSs	for	task	Host and device (target/ implements)	ndrange(n, G1,…, Gn, L1,…,Ln)	-	copy_in/copy_out/ copy_inout/copy_deps
CUDA	<<<…>>>	Async kernel launch and memcpy, CUDA graphs	Device only	Blocks/threat shared memory	-	cudaMemcpy
OpenCL	kernel	clEnqueTask	Host and device	Work-group and work-item	-	bufferWrite

Table 4. Comparison of the presented parallel programming models based on synchronizations, mutual exclusions, language binding, error handing and tool support.

	Synchronizations			Mutual exclusion	Language library	Error handling	Tool support
Parallel Programming Model	**Barrier**	**Reduction**	**Join**				
OpenMP	barrier	reduction	taskwait	Locks, critical, atomic, single, master	C/C++ and Fortran based directives	cancel	OMPT interface
OmpSs	-	reduction	taskwait	critical, atomic	C/C++ and Fortran based directives	-	Extrae [20]
CUDA	_syncthreads	-	-	atomic	C/C++ extensions	-	NVIDIA profiling tools
OpenCL	work_group barrier	work_group reduction	-	atomic	C/C++ extensions	exceptions	System/vendor tools

2.2.3. Compilers and synthesis tools

The compiler and synthesis tools are in charge of translating the meta parallel programming model, containing the data extracted from the DSML and included by the analysis tools, into parallel code. The different tools taken into consideration are described as follows:

- Mercurium [21] is a free source-to-source compilation infrastructure aimed at fast prototyping. The compiler can be easily extended thanks to its plugin architecture, and the different plugins can be dynamically loaded according to the chosen configuration. Mercurium supports OpenMP and OmpSs, and has been extended to other programming models in the past. Figure 2 shows a high-level overview of the architecture of the compiler. The image exposes several key factors of this compiler: (1) Mercurium uses a common internal representation for C, C++ and FORTRAN, which enhances the portability of any analysis and optimization between the three languages; (2) the semantics of the parallel programming model are stored in the internal representation (IR) at high-level in order to be able to generate readable output code, hence the analysis works at the same level as the user; and (3) it can use different native compilation toolchains in order to generate the final executable, so specific compilers for FPGA, NVIDIA or SMP can be used.

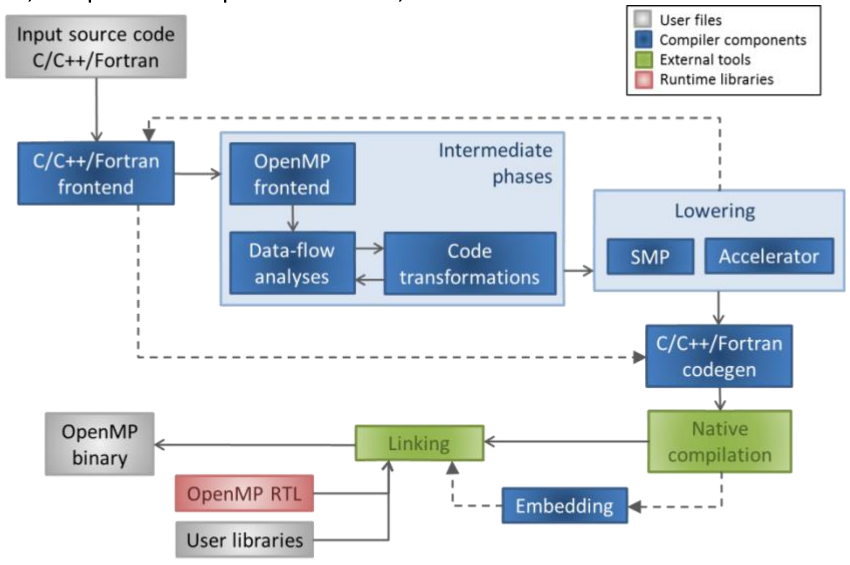


Figure 2. Mercurium architecture overview.

- GCC [6] is a portable compiler supporting a wide variety of platforms. It can also be used for cross-compiling and so producing executables for a different system from the one used by GCC itself. It provides multiple frontends, including C, C++ and Ada, and also includes support for OpenMP and OpenACC. Figure 3 shows a high-level overview of the GCC architecture: on the left, a zoom in to the insights of the compiler, and on the right, an overview of the components of the tool-chain. The IR used within the compiler, GIMPLE, is shared for all languages and, on top of it, the tool includes several analysis and optimization phases.

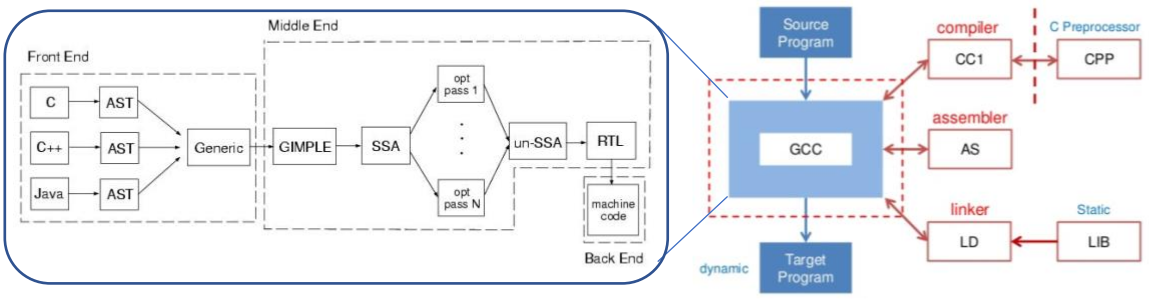


Figure 3. GCC architecture overview.

- LLVM [22] is a compilation tool-chain designed around a language-independent IR that serves as a portable, high-level language that can be optimized with a variety of transformations over multiple passes. It supports several languages, including C, C++ and Ada, as well as OpenMP. Furthermore, it can easily be extended by including new passes in the plugin architecture of the compiler (see Figure 4).

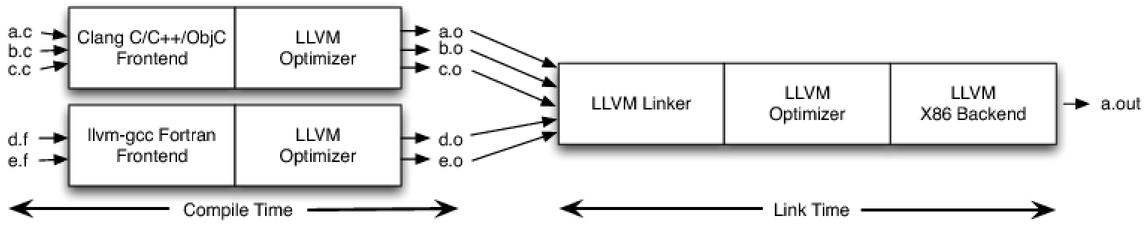


Figure 4. LLVM architecture overview.

2.2.4. Analysis and Testing Tools

A set of tools will be responsible of statically analyse at compile-time the fulfillment of non-functional requirements and, at run-time monitor that these requirements are guaranteed. If not, counter measurements must to be taken.

2.2.5. Runtime Libraries

The runtime libraries are in charge of orchestrating the parallel execution accordingly to the requirements defined at analysis time and the code generated at compilation time. The runtime libraries considered in this deliverable are described next:

- GOMP [23] is the OpenMP implementation for the C, C++ and Fortran compilers of GNU.
- Nanos [24] is the runtime developed at the Barcelona Supercomputing Center (BSC) to support OmpSs and also OpenMP.
- KMP [25] is the OpenMP subproject of LLVM, with compatibility for GCC and Intel.

2.2.6. Operating Systems and Hypervisor

The software architecture envisioned in the AMPERE project includes multiple operating systems run concurrently on the same platform.

To this aim, at the lowest level, the architecture is based on the PikeOS hypervisor by partner SYS [16] for controlling the allocation of the various hardware resources (e.g. cores, memories, peripherals) among the different OSs.

Then, on top of the hypervisor, two different operating systems are concurrently executed:

- A version of the general-purpose Linux OS [14], properly enhanced for improving real-time response times and power efficiency
- ERIKA Enterprise [15], a safety-critical RTOS designed by partner EVI. This RTOS has been designed for the automotive domain and is already used in production by renowned companies operating in this market. In particular, the RTOS is compliant with the AUTOSAR Classic standard and is being certified ISO26262.

2.3. Interfaces

The interfaces used to communicate the different components (represented in Figure 1 in italics and bold) are described next:

- The **DSML** and the **compiler(s)** will communicate through the *meta model driven abstraction interface*, which will identify (and generalize) the software components defined by the DSML (at different granularity levels), and the interaction existing among them. Moreover, it will capture the functional and non-functional requirements to ensure a correct execution of the system.

The compiler(s) will be then responsible of transforming the *meta model driven abstraction* into the *meta parallel programming model interface* represented in the form of a Direct Acyclic Graph (DAG), or Task Dependency Graph (TDG). The DAG or TDG is a graph containing the units of parallelism of a program as nodes, and the data and control dependencies between the tasks as edges. The TDG shall be able to expose the nature and the different levels of parallelism and so allow to identify the most suitable parallel hardware engine, as well as to guarantee the correctness of the program considering lack of deadlocks and data-races. **Error! Reference source not found.** shows a possible representation of the DAG using the JavaScript Object Notation (JSON) [26]. There, a DAG is represented as a series of nodes, where each node has information about:

Listing 1. JSON example of a DAG representing the meta model driven abstraction.

```
{
    “dag”:
        “nodes” : [
          {“id”: value, “data_in”: ítem-list, “data_out”: ítem-list,
               “successors”: node-list, “predecessors”: node-list,
               “locks”: lock-list},
          …
        ]
    }
}
```

- The **multi-criteria optimization** component will communicate with the **compiler** through the *meta parallel programming model interface,* by enriching the TDG, containing the functional and non-functional requirements of the system and captured by the meta model driven abstraction. This information will include an analysis of the of non-functional requirements, and the conditions to be fulfilled at runtime, e.g., execution time, worst case execution time (WCET), energy consumption, deadline, period, and priority, among others (see Listing 2).

Listing 2. JSON example of a DAG representing the meta parallel programming model.

```
{
    “dag”:
        “nodes” : [
          {“id”: value, “data_in”: ítem-list, “data_out”: ítem-list,
               “successors”: node-list, “predecessors”: node-list,
               “execution_time”: value, “WCET”: value,
               “energy”: value, “deadline”: value,
               “period”: value, “priority”: value},
          …
        ]
    }
}
```

- The **compiler** will communicate with the **runtime** through the parallel runtime API calls. As an illustration, Table 5 shows a representative set of the API of the libgomp OpenMP runtime.

Table 5. OpenMP interface between the compiler and the runtime components for libgomp RTL.

	OpenMP directive	#pragma omp parallel num_threads(…) proc_bind(…) firstprivate(…) …

Spawn parallelism	**Runtime call**	void GOMP_parallel(void (*fn) (void*), void* data, unsigned num_threads, unsigned int flags)
	Description	Spawn a team of (*num_threads)* threads and execute the code associated to the parallel region (fn) fulfilling the given memory model (*data)* and a set of additional features (*flags)*.
Work sharing	**OpenMP directive**	#pragma omp single
	Runtime call	bool GOMP_single_start()
	Description	Returns true if the executing thread is the one that shall execute the associated region of code
Task distribution (host)	**OpenMP directive**	#pragma omp task depend(…) priority(…) firstprivate(…)
	Runtime call	void GOMP_task(void (*fn) (void*), void* data, void (*cpyfn) (void*, void*), long arg_size, long arg_align, bool if_clause, unsigned flags, void **depend, int priority)
	Description	Create an OpenMP task with the associate code (*fn),* data environment (*data*, *arg_size* and *arg_align)*, dependencies (*depend*), *priority*, and additional features (*if_clause* and *flags*).
Task offloading (device)	**OpenMP directive**	#pragma omp target device(…) firstprivate(…) map(…) depend(…) nowait
	Runtime call	GOMP_target(int device, void (*fn) (void *), const void *unused, size_t mapnum, void **hostaddrs, size_t *sizes, unsigned char *kinds)
	Description	Map variables (*mapnum*, *hostaddrs*, *sizes*, and *kinds*) to a *device* data environment and execute the code associated with the target task (*fn*) on that device.
Partial synchronization	**OpenMP directive**	#pragma omp taskwait
	Runtime call	void GOMP_taskwait()
	Description	Forces the suspension of the current task region until all child tasks[2] generated before the taskwait complete execution.
Full synchronization	**OpenMP directive**	#pragma omp barrier
	Runtime call	void GOMP_barrier()
	Description	Forces all threads of the team executing the barrier to complete any previous task before continuing executing beyond the barrier.
Resiliency (Error model)	**OpenMP directive**	#pragma omp cancel construct-type
	Runtime call	bool GOMP_cancel(int which, bool do_cancel)
	Description	Activates cancellation of the innermost enclosing region of the type specified.
	OpenMP directive	#pragma omp cancellation point construct-type
	Runtime call	bool GOMP_cancellation_point(int which)
	Description	Introduces a user-defined cancellation point at which tasks check if cancellation has been activated.

- The **runtime** will communicate with the **OS** through the low level threading library. As an illustration, Table 6 shows a representative set of the API of Pthreads.

[2] An OpenMP *task* is a *child task* of its generating *task region*, i.e., the code encountered during the execution of the task.

Table 6. OS-threads interface between the runtime and the OS components for Pthreads.

API call	Description
int pthread_create(pthread_t* thread, const pthread_attr_t* attr, void* (*start_routine)(void*), void* arg)	Creates a new thread with identifier *thread* with certain attributes (*attrs*). The thread starts execution by invoking *start_routine* with *arg* as argument.
int pthread_join(pthread_t thread, void** retval)	Blocks the calling thread until the specified *thread* terminates.
int pthread_barrier_wait(pthread_barrier_t* barrier)	Synchronizes participant threads at *barrier*.
void pthread_exit(void* retval)	Terminates a thread.
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex)	Blocks the thread on a condition variable (*cond*).
int pthread_cancel(pthread_t thread)	Sends a cancellation request to thread. The effective cancellation depends on attributes specific to the thread: cancelability state and type.

- The **hypervisor** will communicate with the underlying parallel platform through the hardware abstraction layer (HAL), including.

3. Requirements of the AMPERE Software Development Ecosystem

Based on the Rational Unified Process® (RUP®) [27], "a *requirement* describes a condition or capability to which a system must conform; either derived directly from user needs, or stated in a contract, standard, specification, or other formally imposed document". To avoid overlooking valuable statements in the definition of the requirements of the system, a systematic approach is needed. FURPS+ [28] is a classification system that organizes requirements in five groups: Functionality, Usability, Reliability, Performance and Supportability, and introduces concerns about design, implementation, interface and physical requirements. The different groups are detailed next:

1. **Functional requirements**: these represent the main features of the product, but also include other aspects such as security, considering the services to protect access to certain resources or information, reporting, considering the access to reporting facilities, or auditing, considering the access to trails of system execution.
2. **Usability**: this aspect includes characteristics such as consistency, ease of use and aesthetics of the user interface.
3. **Reliability**: this includes characteristics such as availability (the amount of system "up time"), accuracy and the system's ability to recover from failures.
4. **Performance**: this includes characteristics such as throughput, response time, recovery time, start-up time, and shutdown time.
5. **Supportability**: this includes characteristics such as testability, adaptability, maintainability, compatibility, configurability, and scalability.

The next subsections describe the situation of the industrial ecosystem relevant to the AMPERE project, and introduce the business goals and the technical requirements based on that analysis and the classification of requirements just introduced.

3.1. AMPERE Ecosystem Business Goals (BG)

Based on the previous market analysis, we have identified the following BGs that the AMPERE software development ecosystem has to incorporate.

ID	**BG1**
Name	**Interoperability**
Type	Business goal
Description	The AMPERE ecosystem will ensure integration and interoperability among SW components by incorporating de-facto market standards. This is a key feature for a wider uptake of the AMPERE solution and for exploitation purposes. Main interests will be those applied in parallel computing, automotive and railway domains, and possible contributions to standardization are also addressed.
Rationale	Ensure integration and interoperability of AMPERE software components with existing solutions.
Involved stakeholders	Parallel processor architecture providers, OEMs and TIER1 automotive and railway companies.

ID	**BG2**
Name	**Ease of use**
Type	Business goal
Description	One of the main goals of the AMPERE solution is to help programmers in the development of advanced CPS with safety requirements. To do this, AMPERE will reduce the time of manually parallelization, by means of automatic transformation of DSMLs to parallel code.
Rationale	Reduce time to market of deploying and executing new advanced CPS and additional costs of training.
Involved stakeholders	OEMs and TIER1 automotive and railway companies, innovative SMEs providing advance services to automotive companies.

ID	**BG3**
Name	**High-performance**
Type	Business goal
Description	AMPERE aims to develop a novel software development ecosystem capable of exploiting the performance capabilities of the most advanced parallel and heterogeneous embedded processor architectures. To ensure high performance, AMPERE will adopt parallel programming models used in the HPC domain, such as OpenMP or OmpSs.
Rationale	The elaboration of information coming from multiple vehicle sensors with the objective of increase the safety and comfort of vehicles.
Involved stakeholders	OEMs and TIER1 automotive and railway companies and any industry developing or using advanced CPS with HPC requirements.

ID	BG4
Name	**Non-functional requirements**
Type	Business goal
Description	The AMPERE ecosystem will provide guarantees about the non-functional requirements inherited from the cyber-physical interaction. The "level of guarantees" will depend on the "criticality-level" of the functionality as imposed by the automotive and railway standards.
Rationale	Enable the use of AMPERE technology in computing environments with non-functional requirements, i.e. automotive and railway.
Involved stakeholders	OEMs and TIER1 automotive and railway companies and all stakeholders developing or using advanced CPS with non-functional requirements.

ID	BG5
Name	**Safety and security**
Type	Business goal
Description	The AMPERE ecosystem will guarantee the functional safety and security aspects required at each integrity level (as defined in the automotive and railway safety standards), which special interest on the isolation features required by each standard in terms of space/time segregation methods at component level.
Rationale	Ensure safety and security operation of the CPS processed by the AMPERE ecosystem.
Involved stakeholders	OEMs and TIER1 automotive and railway companies and all stakeholders developing or using advanced CPS with safety and security requirements.

3.2. Technical Requirements (TR) of the AMPERE Software Development Ecosystem

The BG identified in the previous section results in the technical requirements described below.

ID	REQ-SWARCH-TR1
Name	**Increase software productivity**
Type	Technical requirement
Description	AMPERE aims to promote productivity regarding the development and execution of advanced CPS, in terms of programmability, portability and performance, as one of the key competitive advantages of the AMPERE ecosystem. With such a purpose in mind, AMPERE promotes the use of high-level parallel programming models that provides the abstraction level needed to describe the parallelism of complex systems, while hiding the complexity of the underlying processor architecture.
Rationale	**Programmability**. The developer is responsible for defining the functionality of the CPS functionalities using the DSMLs. Then, the AMPERE ecosystem will transform it to the parallel programming model supported by the underlying processor architecture. The most advanced developers may also manually introduce parallelism with the objective of further optimizing the parallel code. In this case, the AMPERE ecosystem will guarantee the correctness of the parallel code by detecting potential race-conditions and deadlocks. Additionally, the AMPERE runtime framework will

	efficiently manage the underlying computing resources, while hiding the complexity of the parallel architecture. Overall, the developer is only responsible of defining what the application does, but not how it does it. **Portability.** The use of well-known parallel programming models in the HPC domain in order to express the parallelism exposed by the applications enables to execute the same application in multiple platforms. The performance portability, and so being able to execute in different platforms with significant performance loss, is also an important aspect. The underlying run-time systems included in the AMPERE ecosystem are responsible for dealing with the internals of each specific technology with the objective of maximizing the performance capabilities considering all possible configurations. **Performance.** The AMPERE ecosystem will be responsible of efficiently managing the execution of the CPS on the most advanced parallel and heterogeneous embedded processor architectures featuring GPU and FPGA accelerators. AMPERE will include the proper parallel programming models to exploit the capabilities of the architectures where the final functionalities will ride on. A number of parallel programming models will be supported in the AMPERE ecosystem, including OpenMP, and CUDA, which can exploit the benefits of each final architecture (multi-core, NVIDIA GPU, FPGA, etc.).

ID	**REQ-SWARCH-TR2**
Name	**Fulfillment of non-functional requirements**
Type	Technical requirement
Description	The AMPERE software development ecosystem aims to support the non-functional requirements usually found in cyber-physical systems, such as time predictability, energy-efficiency, resiliency, security and safety.
Rationale	In CPS, the interaction between the computing system and its environment needs to cope with the non-functional requirements inherited from the application domain. Control loops require guaranteed response times, and sensors and embedded computers require energy efficiency. Moreover, CPSs must guarantee a correct operation of the system and be resilient to potential errors introduced in the system due to SW or HW faults. AMPERE will include these non-functional requirements as first class entities in the software architecture. Developers will specify the requirements through the DSML, which can span from critical guaranteed requirements to best-effort approaches, depending on the criticality of the application functionality. The component deployment will be either statically or dynamically performed, to be able to provide the required quality-of-service (QoS). Critical guarantees will be provided through static deployment of resources, whilst softer requirements will be coped with a mix of static and dynamic adaptation approaches.

ID	**REQ-SWARCH-TR3**
Name	**Safety and security mechanisms**
Type	Technical requirement
Description	The AMPERE software development ecosystem will incorporate the mechanisms needed to fulfil the safety standards. It will take into account safety, considering the

	proper system design, and security, considering potential external threads, to guarantee a correct CPS operation each integrity level.
Rationale	AMPERE will include the adequate isolation features required by each standard, taken into account the impact that parallel execution has on composability, as well as developing the required space and time segregation methods.

Each TR addresses one or more BGs, as defined in Table 7.

Table 7. Relation between business goals and technical goals of the AMPERE project.

Business goals	**Technical goals**
BG1. Interoperability	REQ-SWARCH-TR1. Increase software productivity
BG2. Easy-to-use	
BG3. Scalability and performance	
BG4. Real-time requirements	REQ-SWARCH-TR2. Fulfillment of non-functional requirements
BG5. Safety and security	REQ-SWARCH-TR2. Fulfillment of non-functional requirements REQ-SWARCH-TR3. Privacy and security mechanisms to guarantee a legal framework

4. Software Development and Integration plan

The AMPERE project involves a distributed team of several people from different institutions and areas of expertise. This section defines the development and integration processes to be followed during the execution of the project, inspired in the Scrum methodology [29]. Scrum is an iterative and incremental framework for managing product development. It defines a flexible, holistic product development strategy where a development team works as a unit to reach a common goal, and enables teams to self-organize by encouraging close collaboration of all team members. The Scrum process is divided in Sprints. A sprint is a timeboxed effort restricted to a specific duration. Each sprint starts with a planning event that identifies the work to be done and makes an estimated forecast for the sprint goal. Each sprint ends with a sprint review and sprint retrospective to identify lessons and improvements for the next sprints.

The remainder of this section is organized as follows: Section 4.1 introduces the development and integration, as well as quality assurance processes for the AMPERE project; Section 4.2 defines the infrastructure to be used in the project to enable all teams to share and coordinate information, and Section 4.3 describes the standards and guidelines to facilitate the usage of the infrastructure.

4.1. Processes

This section introduces two main processes: (1) the development and integration process, and (2) the quality assurance process. The former focuses on providing means for a continuous development approach that reduces risks and facilitates the building and releasing procedures, and the latter focuses on guaranteeing high quality development results. These processes are specified in the following subsections, including the activities related to each of the processes.

4.1.1. Development and Integration Processes

The AMPERE project is an aggregation of different components, as defined Section 2, which cooperate to provide different functionalities. The software architecture integration process consists in a combination of all software components into a unique ecosystem, ensuring that all components work as defined in the functional requirements and, the software architecture, as a whole, provides the desired functionalities.

The AMPERE project is split in four phases with a milestone at the end of each phase. The different phases, defined in the project's Grant Agreement [30]. Based on that, Figure 5 shows the integration process of the different components of the AMPERE ecosystem. The process follows a tree structure and is done incrementally with different integration steps in each one of the phases. The integration steps of each of the phases are explained as follows:

- **Phase 1:**
  - SYSGO: Ensure compatibility of the selected architecture with the DSML, the PPM, the runtime and the hypervisor.
- **Phase 2:**
  - BSC: Integrate the meta model driven abstraction (DAG) and the meta parallel programming model (TDG) ensuring their compatibility.
  - ISEP: Integrate the NFR, in an isolated manner, with the DAG and the TDG.
  - BOS/THALIT: Ensure compatibility of the DAG and TDG with the use-cases (test suite at this point of the project) evaluating functional safety and security focusing on composability.
  - ETHZ: Integrate the NFR, in an isolated manner, with the runtime system.
  - SYS: Integrate the NFR, in an isolated manner, with the hypervisor.
- **Phase 3:**
  - BSC: Integrate proposals for extending the DSML and the PPM ensuring their compatibility.
  - ISEP: Integrate the NFR, in an holistic manner, with the synthesis tools that use the extended DAG and the extended TDG, as well as with the extended DAG.
  - BOS/THALIT: Ensure the compatibility of the extended DSML and the extended PPM with the use-cases ("final" use cases at this point of the project) evaluating functional safety and security considering NFR in a holistic manner.
  - SSSA: Integrate the NFR, in a holistic manner, with the runtime system.
  - SYS: Integrate the NFR, in a holistic manner, with the hypervisor.
  - EVI: Integrate the runtime and hypervisor systems, with holistic support for NFR, with the architecture.
- **Phase 4:**
  - TRT: Integration and evaluation of the complete AMPERE ecosystem.

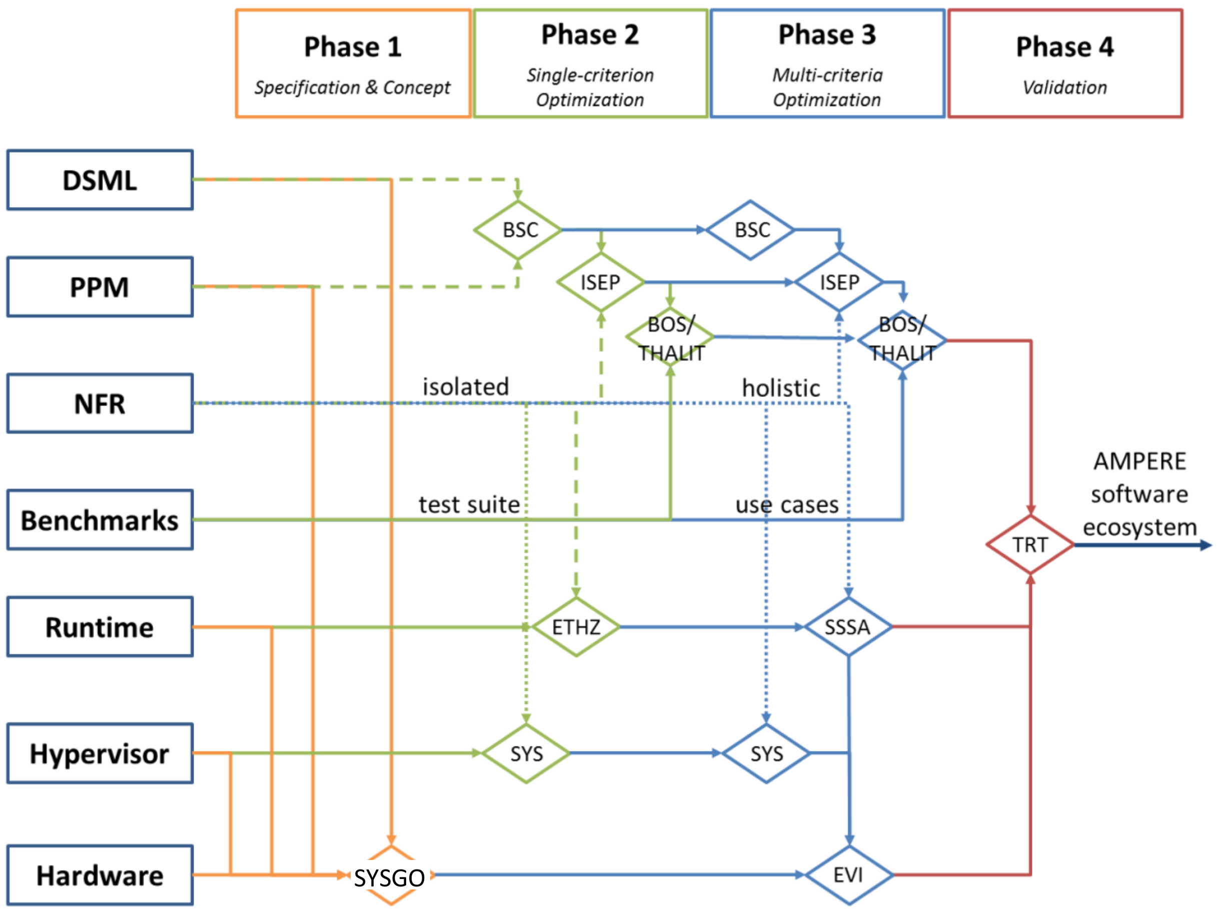


Figure 5. Tree-like diagram of the integration of the AMPERE ecosystem components.

4.1.2. Quality Assurance Process

The AMPERE software developers' team is a highly distributed team of teams. This scenario requires taking special care of the communication and interconnection among the different teams. With such a purpose, several activities will be carried at, so the project provides high quality standards in terms of faults and compliance with specified behavior. These activities will occur at different stages of the development process. Some of them are continuous and some others are recurring. This section summarizes the quality assurance activities performed in the development and integration process of the AMPERE project.

4.1.2.1. Scrum-based Methodology

The Scrum methodology reduces risks and removes dependencies between releases and integration activities, allowing the synchronization of the different integration parts and the final validation. An important aspect included in the methodology is the recognition that there will be unpredictable challenges. For that reason, Scrum adopts an evidence-based empirical approach focusing on how to maximize the team's ability to deliver quickly, to respond to emerging requirements, and to adapt to evolving technologies and changes in the project conditions.

In Phases 1 and 2, each team in AMPERE will define its own sprints regarding contents and duration, because each team is to work in isolation. After that, in Phases 3 and 4, all teams in the project will have to define a common structure for the sprints, according the duration and contents of each sprint.

4.1.2.2. Unit Testing

During Phases 1 and 2, synthetic benchmarks will be developed and used together with unit tests in order to validate the functionalities implemented in each component (this studies will be included in D1.2 "Analysis of functional safety aspects on single-criterion optimization and first release of the test bench suite", to be delivered in month 15). These unit tests will allow determine if individual units of code are correct, simplifying the later integration and facilitating future changes. Each time a new functionality is to be added to the software ecosystem, the entire unit test suite should be executed, ensuring that all new and existing tests run successfully upon code check in. To enhance the quality of the unit tests, a series of guidelines shall be used, e.g., name tests properly, keep tests small and fast, cover boundary cases, and prepare tests for code failing.

Initially, unit tests are to be tested manually by each component's developers. A tool like automake [31] can be used to automatize the execution of tests. In a second stage, in order for the unit testing to be useful, the tests are to be performed automatically by a continuous integration system, as explained in Section 4.2.2.2.

4.1.2.3. Regression Tests

During the integration, changes to the implementation may be needed. In such case, regression tests shall be run each time a modification is implemented in any part of the software ecosystem, so it can be determined whether the changes break anything that worked prior to the change. Regression tests, overall, consist on rerunning previous tests any time a modification is performed, as well as writing new tests when necessary. Adequate coverage is paramount when conducting regression tests. For that reason, a series of strategies and good practices must be followed, e.g., check possible side effects when fixing bugs, write regression tests for each bug fixed, and remove redundant tests.

4.1.2.4. Bug and Issue Tracking

Bug and issue trackers allow managing lists of bugs and issues. This kind of application is used to create, update and resolve project issues, providing the platform to maintain a knowledge base that includes information that may help to resolve the issues. During the integration phase this is a key aspect to ensure the correct collaboration and communication among the different teams that participate in the integration of each particular component. In general, issue trackers have proven to be an effective lightweight task management system, much more useful than real-time messaging o emailing. For this task, the AMPERE project will use different tools, as detailed in Section 4.2.3.1.

4.2. Infrastructure

This section describes the tools and platforms that we have identified to potentially be used within the AMPERE project. Infrastructure decisions are based on consortium agreements and are triggered by common development standards as well as in particular the quality assurance and integration processed described in Section 4.1.

4.2.1. Development Platform

Different components of the AMPERE software architecture will be implemented using different development platforms. These are described in the next paragraphs.

4.2.1.1. Integrated Development Environments (IDEs)

The different groups working on the APERE project will take benefit of different IDEs. This kind of platform provides several benefits:

- *Code completion or code insight*: IDEs recognize language's keywords and function names. This knowledge is typically used to highlight typographic errors, suggest a list of available functions based on the appropriate situation, or offer a function's definition from the official documentation.
- *Resource management*: IDEs manage resources such as libraries and header files, hence being aware of any required resource missing. By using this feature, errors can be spotted at the development stage and not later, in the compile or build stage.

- *Debugging tools*: IDEs allow thoroughly testing applications before release by means of assigning variable values at certain points, connecting different data repositories, or accepting different run-time parameters.
- *Compile and build*: IDEs allow automatic translation from high-level language to object code for languages that require a compile or build stage.

Some of the IDEs that will be used in the AMPERE project are listed as follows:

- Eclipse [32] comprises a Rich Client Platform (RCP) for developing general purpose applications, and includes a powerful plug-in system. Components such as AMALTHEA and PikeOS are developed on top of eclipse-based tools (i.e., App4mc and CODEO, respectively).
- Visual Studio Code [33] is a free cross-platform source-code editor. Partners including SSSA and BOSCH will use this platform.
- VIM [34] is a highly configurable and robust text editor. Partners including BSC, THALIT and ETHZ will use this platform.
- Other IDEs and editors that might be used by the AMPERE partners include Emacs [35], Visual Studio [36], and Sublime Text [37].

4.2.1.2. Software Configuration Management

Software configuration management systems provide means for distributed teams to work collaboratively together on shared documents. Within AMPERE, development will be carried out using Git [38]. This is a free and open source distributed version control system able to handle very large projects with speed and efficiency, because it has a tiny footprint and includes features like cheap local branching, convenient staging areas, and multiple workflows. Git will be used in AMPERE for sharing code source, as well as documentation such as deliverables, technical reports, papers and posters, among others. It is particularly convenient within the AMPERE project, where a very distributed team develops in parallel several components, because of its support for submodules: Git submodules allow to treat different projects as separate, yet still be able to use one from within the other.

4.2.1.3. Instant Messaging and Transparency

Communication is of paramount importance in the development and integration process, particularly, when working in remote teams composed of many people. For such a reason, not only communication but also transparency is needed, because everybody must be aware of what is happening in all sides in order to participate or even plan their own work.

The AMPERE partners will use Slack [39], a team communication tool that provides many benefits. The most significant to the project are listed below:

- Integrates all team communications in one place. Furthermore, the communications can be segmented into channels, organized by topics, and different users can be assigned to each channel, depending on the visibility the channel must have.
- Integrates other web services, e.g., GitHub, for notification and viewing code check-ins, and Dropbox and Google Drive, for file sharing.
- All content is searchable from one search box. Communications between several people can lead to large amounts of information that is later hard to find. Slack search filter options narrow the search on the conversations to specific channels, persons, or many other filters.
- Code snippets sharing. Slacks supports sharing code snippets with specific syntax highlighting. The platform also supports other members to download it, view it in a raw mode, or leave comments and modifications.

4.2.2. Integration Platform

This section describes the platforms that will be used for the integration of the AMPERE project.

4.2.2.1. Automated Build System

The different components of the AMPERE project will use different tools for automated build system, listed as follows:

- GNU Make [40] is a tool that controls the generation of executables and other non-source files of a program from the program's source files. A special file, *makefile*, details rules for building and installing a package, and abstracts decisions such as the order for updating files or the need for updating a particular file. The tool is not limited to any particular language or compilation tool-chain. Partners including BSC, EVI, SSSA, SYS, THALIT and ETHZ will use this software.
- Cmake [41] is a family of tools to build, test and package software. It is used to control the compilation process using platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in any particular compiler environment. Partners including EVI, BOS and ETHZ will use this software.
- Bitbake [42] is a build engine that follows recipes in a specific format in order to perform sets of tasks. It includes a scheduler that creates a dependency tree to order the compilation, schedules the compilation of the included code, and finally, executes the building of the specified, custom Linux image. EVI will use this software.
- Apache maven [43] is a project management and comprehension tool that manages project building, reporting and documentation from a central place. The primary goals of this platform are: (1) making the build process easy, (2) providing a uniform build system, (3) providing quality project information, (4) providing guidelines for best practices development, and (5) allowing transparent migration to new features. THALIT will use this software.

The automated build system to be used for the integration of the software components of the AMPERE project is yet to be defined.

4.2.2.2. Continuous Integration System

Continuous integration systems will be used to automatize the integration of the software components, as well as for testing. Such systems focus on two goals:

1. Building and testing software projects continuously. Jenkins provides a flexible continuous integration system, making it easy for developers to integrate changes to the project, and making it easy for users to obtain a fresh build. The automated, continuous build increases the productivity.
2. Monitoring executions of externally-run jobs. This includes jobs such as cron jobs or jobs that are run on remote machines. The results of these jobs are kept by Jenkins, as well as sent to developers by email. Any form of checking these results allows developers to notice when something is wrong faster and easier than using traditional testing mechanisms.

The AMPERE project partners will use two such systems: Jenkins [44] and Gitlab [45].

4.2.3. Quality Assurance Tools

This section covers the quality assurance tools used within the AMPERE project to provide means to test and control code and thus system quality.

4.2.3.1. Issue Tracking Tool

The AMPERE partners will different tools to track issues and feature requests, depending on the component. The most relevant are described as follows:

- GitLab [46] enables lean and agile project management from basic issue tracking to scrum project management. Specifically, it allows:
  - Manage and track issues: (1) collaborate and define specific business needs, (2) track effort, size, complexity, and priority of resolution, and (3) eliminate silos and enable cross-functional engagement.

- Visualize work with issue boards: (1) visualize the status of work across the lifecycle, (2) manage, assign and track the flow of work, and (3) enable Kanban and Scrum styles of agile delivery.
- Maintain traceability through the DevOps Pipeline: (1) link issues with actual code change needed to resolve issues, (2) visualize and track the status of builds, testing, security scans, and delivery, and (3) enable entire team to share a common understanding of status.

Partners including BSC, SSSA, SYS and THALIT will use this software.

- Jira [47] is a commercial software product for issue tracking and project management that allows agile software development. The most relevant features are the following:
  - Kanban boards: allow visualizing the status of the tasks of the full team.
  - Roadmaps: sketch out the big picture to ensure the roadmap connects to the team's work.
  - Reporting: provide reports with real-time insights into the performance of the team.
  - Connect issues to code: Connect information from a version control into the issue tracking.

  Partners including SYS, BOS and THALIT will use this software.
- Bitbucket [48] is a web-based version control repository compatible with Git. Among others, this tool includes:
  - Code review: allows creating merge request with designated approvers, and hold discussions in the source code with inline comments.
  - Continuous integration: allows building, testing and deploying with integrated continuous integration and continuous delivery (CI/CD).
  - Secure: saves the code in the cloud with IP whitelisting and required 2-step verification.

  THALIT will use this software.

4.3. Standards and Guidelines

Development guidelines provide a basic set of rules to enforce consistent and standardized coding practices. These guidelines are even more vital in a distributed software development project with teams at geographically separated locations. The guidelines in this section assure code quality and complement the processes defined in Section 4.1.

4.3.1. Design Patterns

Within AMPERE, developers will use design patterns when applicable. Design patterns [49] are time-tested solutions to recurring design problems and offer several benefits:

1. Provide solution to issues in software development using a proven solution.
2. Design patterns make communication between designers more efficient.
3. Facilitate program comprehension.

4.3.2. Code Comments

Code comments help to explain and describe the actions of a certain block of code, describing behaviors that cannot otherwise be clearly expressed in the source language and easing comprehension. AMPERE developers will comment crucial parts in the source code to help other developers understand their code. In spite of numerous benefits of having properly commented source code, comments can be misguiding if not used properly. Thus a few points worth consideration while writing comments are:

1. Comments can get out of sync with the code if people change the code without updating the comments. Thus, comments should always change together with code.
2. Good comments are hard to write and time consuming, but pay off in long term.
3. Adding comments can be counter-productive if the information provided by them is not relevant to the part of code where they are provided. Hence, inline comments should describe the next line of code.

4.3.3. Programming Style

Programming style is a set of rules or guidelines used when writing the source code. These guidelines include elements common to a large number of programming styles such as the layout of the source code, including indentation, the use of white space around operators and keywords, the capitalization of keywords and variable names, the style and spelling of user-defined identifiers, such as function, procedure and variable names; and the use and style of comments.

Since the AMPERE project will include several components that are already under development and follow their respective programming styles, developers in the frame of the AMPERE project will follow these styles. For those parts of code which purpose is integrating different components of the AMPERE ecosystem, the involved partners will define the programming style together the APIs, as introduced in Section 0.

5. Acronyms and Abbreviations

- API – Application Program Interface
- BG – Business Goals
- CAGR – Compound Annual Growth Rate
- CPS – Cyber Physical System
- CI/CD – Continuous Integration and Continuous Delivery
- CAGR – Compound Annual Growth Rate
- DAG – Direct Acyclic Graph
- DSML – Domain Specific Modelling Language
- HAL – Hardware Abstraction Layer
- IDE – Integrated Development Environment
- FR – Functional Requirement
- HPC – High-Performance Computing
- JSON – JavaScript Object Notation
- KPI – Key Performance Indicator
- MS – Milestone
- NFR – Non-Functional Requirement
- OS – Operating System
- QoS – Quality of Service
- RCP – Rich Client Platform
- RTL – Runtime Library
- TDG – Task Dependency Graph
- TR – Technical Requirement
- WP – Work Package

6. References

[1] C. Wolff, L. Krawczyk, R. Höttger, C. Brink, U. Lauschner, D. Fruhner, E. Kamsties and B. Igel, "AMALTHEA -- Tailoring tools to projects in automotive software development," in *8th International*

Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), 2015.

[2] P. Roques, Systems Architecture Modeling with the Arcadia Method: A Practical Guide to Capella, Elsevier, 2017.

[3] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa and K. Lange, "AUTOSAR -- A Worldwide Standard is on the Road," in *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, 2009.

[4] S. a. S. A. Fürst, "Autosar the next generation -- the adaptive platform," *CARS@ EDCC2015,* 2015.

[5] Programming Models @BSC, "Mercurium," June 2020. [Online]. Available: https://pm.bsc.es/mcxx. [Accessed June 2020].

[6] GNU, "GCC, the GNU Compiler Collection," June 2020. [Online]. Available: https://gcc.gnu.org/. [Accessed June 2020].

[7] "The LLVM Compiler Infrastructure," June 2020. [Online]. Available: https://llvm.org/. [Accessed June 2020].

[8] T. Feist, "Vivado design suite," *White Paper,* vol. 5, p. 30, 2012.

[9] GNU, "https://gcc.gnu.org/projects/gomp/," January 2020. [Online]. Available: https://gcc.gnu.org/projects/gomp/. [Accessed June 2020].

[10] Programming Models @BSC, "Nanos++," June 2020. [Online]. Available: https://pm.bsc.es/nanox. [Accessed June 2020].

[11] "LLVM OpenMP Runtime Library," September 2015. [Online]. Available: https://openmp.llvm.org/Reference.pdf. [Accessed June 2020].

[12] NVIDIA, "CUDA C++ Programming Guide," June 2020. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Accessed June 2020].

[13] B. Nichols, D. Buttlar and J. P. Farrell, Pthreads programming: A POSIX standard for better multiprocessing, O'Reilly Media, Inc., 1996.

[14] "Linux," June 2020. [Online]. Available: https://www.linux.org/. [Accessed June 2020].

[15] "Erika Enterprise RTOS v3 :: Erika3," June 2020. [Online]. Available: http://www.erika-enterprise.com/. [Accessed June 2020].

[16] SYSGO, "PikeOS Certified Hypervisor," June 2020. [Online]. Available: https://www.sysgo.com/products/pikeos-hypervisor/. [Accessed June 2020].

[17] AMPERE - WP5 (SYSGO), "D5.1 - Reference parallel heterogeneous hardware selection," 2020.

[18] AMPERE - WP1 (THALIT), "D1.1 - System models requirement and use case selection," 2020.

[19] AMPERE - WP2 (BSC), "D2.1 - Model transformation requirements," 2020.

[20] BSC Performance Tools, "Extrae Documentation release 3.8.0," June 2020. [Online]. Available: https://tools.bsc.es/doc/pdf/extrae.pdf. [Accessed June 2020].

[21] Programming models @BSC, "Mercurium," June 2020. [Online]. Available: https://pm.bsc.es/mcxx. [Accessed June 2020].

[22] "The LLVM Compiler Infrastructure," June 2020. [Online]. Available: https://llvm.org/. [Accessed June 2020].

[23] GNU, "GOMP," January 2020. [Online]. Available: https://gcc.gnu.org/projects/gomp/. [Accessed June 2020].

[24] Programming Models @BSC, "Nanos++," June 2020. [Online]. Available: https://pm.bsc.es/nanox. [Accessed June 2020].

[25] LLVM, "Support for the OpenMP language," January 2020. [Online]. Available: https://openmp.llvm.org. [Accessed June 2020].

[26] "JSON," January 2020. [Online]. Available: www.json.org. [Accessed June 2020].

[27] P. Kruchten, The rational unified process: an introduction, Addison-Wesley Professional, 2004.

[28] P. Eeles, "Capturing architectural requirements," *IBM Rational developer works,* 2005.

[29] K. Schwaber and M. Beedle, Agile Software Development with Scrum, vol. 1, Prentice Hall Upper Saddle River, 2002.

[30] Barcelona Supercomputing Center, Instituto Superior de Engenharia do Porto, Eidgenoessische Technische Hochschule Zuerich, Scuola Superiore di Studi Iniversitari di Perfezionamento Anna, Evidence SRL, Rober BOSCH GMBH, THALES SA, THALES ITALIA SPA, SYSGO , "AMPERE Grant Agreement," 2019.

[31] GNU, "Tests (automake)," June 2020. [Online]. Available: https://www.gnu.org/software/automake/manual/html_node/Tests.html. [Accessed June 2020].

[32] The Eclipse Foundation, "Enabling Open Innovation & Collaboration," 2020. [Online]. Available: www.eclipse.org.

[33] Microsoft, "Visual Studio Code," June 2020. [Online]. Available: https://code.visualstudio.com/. [Accessed June 2020].

[34] "Vim," June 2020. [Online]. Available: https://www.vim.org/. [Accessed June 2020].

[35] GNU, "Emacs," June 2020. [Online]. Available: https://www.gnu.org/software/emacs/. [Accessed June 2020].

[36] Microsoft, "Visual Studio," June 2020. [Online]. Available: https://visualstudio.microsoft.com/. [Accessed June 2020].

[37] Sublime HQ, "Sublime Text," May 2020. [Online]. Available: https://www.sublimetext.com/. [Accessed June 2020].

[38] Software Freedom Conservancy, "Git," 2020. [Online]. Available: https://git-scm.com/. [Accessed June 2020].

[39] Slack, "Slack," 2020. [Online]. Available: slack.com. [Accessed June 2020].

[40] GNU, "Make," June 2020. [Online]. Available: https://www.gnu.org/software/make/. [Accessed June 2020].

[41] "CMake," June 2020. [Online]. Available: https://cmake.org/. [Accessed June 2020].

[42] Yocto Project, "BitBake User Manual," January 2018. [Online]. Available: https://www.yoctoproject.org/docs/1.6/bitbake-user-manual/bitbake-user-manual.html. [Accessed June 2020].

[43] "The Apache Maven Project," 2020. [Online]. Available: https://maven.apache.org/. [Accessed June 2020].

[44] "Jenkins," 2020. [Online]. Available: https://jenkins.io/. [Accessed June 2020].

[45] Gitlab, "The first single application for the entire DevOps lifecycle," June 2020. [Online]. Available: https://about.gitlab.com/. [Accessed June 2020].

[46] Gtilab, "Gtilab," 2020. [Online]. Available: gitlab.com. [Accessed June 2020].

[47] Atlassian, "Jira. Issue & Project Tracking Software," June 2020. [Online]. Available: https://www.atlassian.com/software/jira. [Accessed June 2020].

[48] "Bitbucket. The Git solution for professional teams," June 2020. [Online]. Available: https://bitbucket.org/product. [Accessed June 2020].

[49] E. Gamma, Design patterns: elements of reusable object-oriented software, Pearson Education India, 1995.