Ref. Ares(2021)4338066 - 03/07/2021

.AMPERE

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

D6.2
Refined AMPERE ecosystem interfaces and integration plan

Version 1.0

Documentation Information

Contract Number	871669
Project Website	www.ampere-euproject.eu
Contratual Deadline	30.03.2021
Dissemination Level	[PU]
Nature	R
Author	Sara Royuela, BSC
Contributors	Thomas Vergnaud (TRT)
Reviewer	Björn Forsberg (ETHZ)
Keywords	Software ecosystem, requirements, interfaces, integration

The AMPERE 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 version by BSC
V0.2	Contributions by TRT
V0.6	Review by ETHZ
V1.0	Final review addressing comments by BSC

Table of Contents

Executive summary

This deliverable covers the work done during the second phase of the project within *WP6*, *AMPERE System Design and Computing Software Ecosystem Integration*. The deliverable spans 11 months of work and describes the work done in *T6.2*, *Refined AMPERE ecosystem requirement specification* to reach MS2.

This deliverable is a refinement of *D6.1*, *AMPERE ecosystem requirements and integration plan*, which described (a) the AMPERE software development ecosystem for developing, deploying and executing the use cases; (b) the set of software components and tools that will form the ecosystem; and (c) the integration plan.

As recommended by the experts in the consolidated interim report resulting from the project overview, this deliverable also includes:

1. Refined explanation of the common interfaces among the WPs and software components.
2. Pointers to the deliverables explaining in detail the selected tools.
3. Better mapping of use case requirements in D1.1 [1], the model transformation requirements in D2.1 [2] and the solution software components and tools for AMPERE ecosystem in D6.1 [3].
4. Gantt diagram detailing the integration process and plan.
5. Concrete targets that the AMPERE ecosystem should fulfill with regard to increased software productivity, portability, performance, and ease of use.

The second milestone of Task 6.1 has been carried out successfully and all objectives of MS2 have been reached and documented in this deliverable.

1 Introduction

WP6 is devoted to the integration of the different components of the AMPERE software ecosystem. In particular, *T6.1, AMPERE ecosystem requirement specification* encloses the work performed for producing this deliverable, and concerns all components of the AMPERE software architecture. The general pipeline of the AMPERE components is depicted in Figure 1 (further details will be provided Chapter 3).

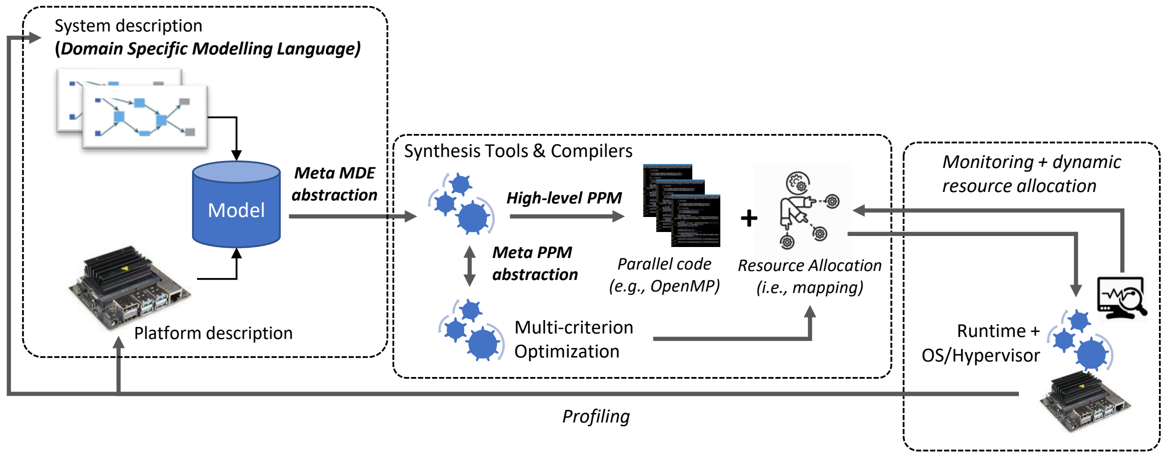

System description
(Domain Specific Modelling Language)
Model
Meta MDE abstraction
Platform description
Synthesis Tools & Compilers
High-level PPM
Meta PPM abstraction
Parallel code (e.g., OpenMP)
+
Resource Allocation (i.e., mapping)
Multi-criterion Optimization
Monitoring + dynamic resource allocation
Runtime + OS/Hypervisor
Profiling

Figure 1: AMPERE software architecture pipeline.

Table 1 lists the deliverables and tasks producing results that are subject to discussion in this deliverable.

Table 1: Tasks and deliverables related to D6.2

Deliverable	D. leader	Task	T. leader
D1.1. System models requirement and use case selection	THALIT	T1.1. System model requirement specification and use case definition	THALIT
D2.1. Model transformation requirements	BSC	T2.1. Model transformation requirements specification	BSC
D6.1. AMPERE ecosystem requirements and integration plan	BSC	T6.1. AMPERE ecosystem requirement specification	TRT

To reach the goals of WP6, this deliverable continues the work started in D6.1, and contributes as follows: Chapter 2 integrates the requirements described in D1.1 [1] and D2.1 [2] with the technical requirements described in D6.1 [3] for the overall AMPERE ecosystem, and includes the metrics to evaluate the objectives of the project; Chapter 3 augments the information provided in D6.1 [3] regarding the AMPERE software development ecosystem, including the selected tools and the interfaces to communicate the different components; Chapter 4 extends the integration plan already presented in D6.1 [3]; Chapter 5 provides the conclusions extracted from the work done for this deliverable.

2 Refined AMPERE ecosystem requirements

Deliverable *D6.1, AMPERE ecosystem requirements and integration plan* introduced the FURPS+[1] classification system and, based on this, described the Business Goals (BG) to be included in the AMPERE software development ecosystem, and the relation of these BG with the Technical Requirements (TR) of the ecosystem. However, these requirements where not properly related with the those of different components analyzed during the first phase of the project, as pointed by the reviewers in the First Technical Review of the project. To cover this lack of unification and provide a consolidated view of the requirements of the whole project, this section relates BG and TR (from D6.1 [3]) with the specific functional and non-functional requirements of the use cases (from D1.1 [1]), and the requirements of the model transformation (from D2.1 [2]). For more detailed information of the requirements themselves, refer to the corresponding deliverables.

Figure 2 illustrates the requirements of the project at different levels of abstraction (i.e., FURPS+, BG, TR, use-cases and model transformation), and relates the requirements expected for each level with those of the previous and next levels. The figure starts with the FURPS+ classification, which organizes requirements in five groups (in yellow): *functionality*, *usability*, *reliability*, *performance* and *supportability*. Next, the figure shows the five BG of the project (in green), i.e., *interoperability*, *high performance*, *non-functional requirements* (NFR), *ease of use*, and *safety and security*, and relates these BG with the FURPS concepts. For example, FURPS *performance* is considered in two different BG: the throughput belongs to the *high performance* BG, and the response time to the NFR. Then, the figure presents the three TR of the project (in blue), i.e., *productivity* (including *programmability*, *performance* and *portability*), *non-functional requirements* and *safety and security*, and relates these concepts to the BG, e.g., *productivity* is related with *interoperability* and *high performance* BGs. After that, the requirements of the use cases, as defined in D1.1 [1], are summarized in six groups (in orange), i.e., development *efficiency*, *energy*, *SW/HW platform*, *timing*, *safety* and *integrity*, and accordingly related to the TR. Finally, the figure incorporates the requirements of the model transformation (in pink), which include the *AMALTHEA modeling tool*, the *code synthesis tool* and the *OpenMP parallel programming framework*. These last requirements, all of them, are related to all previous requirements, as all three must provide the features needed for the requirements to be propagated from the design to the implementation.

[1]FURPS+ is the acronym for Functional, Usability, Reliability, Performance, Supportability and the extension (+) includes three additional categories: Constraints, Interface Requirements and Business Rules.

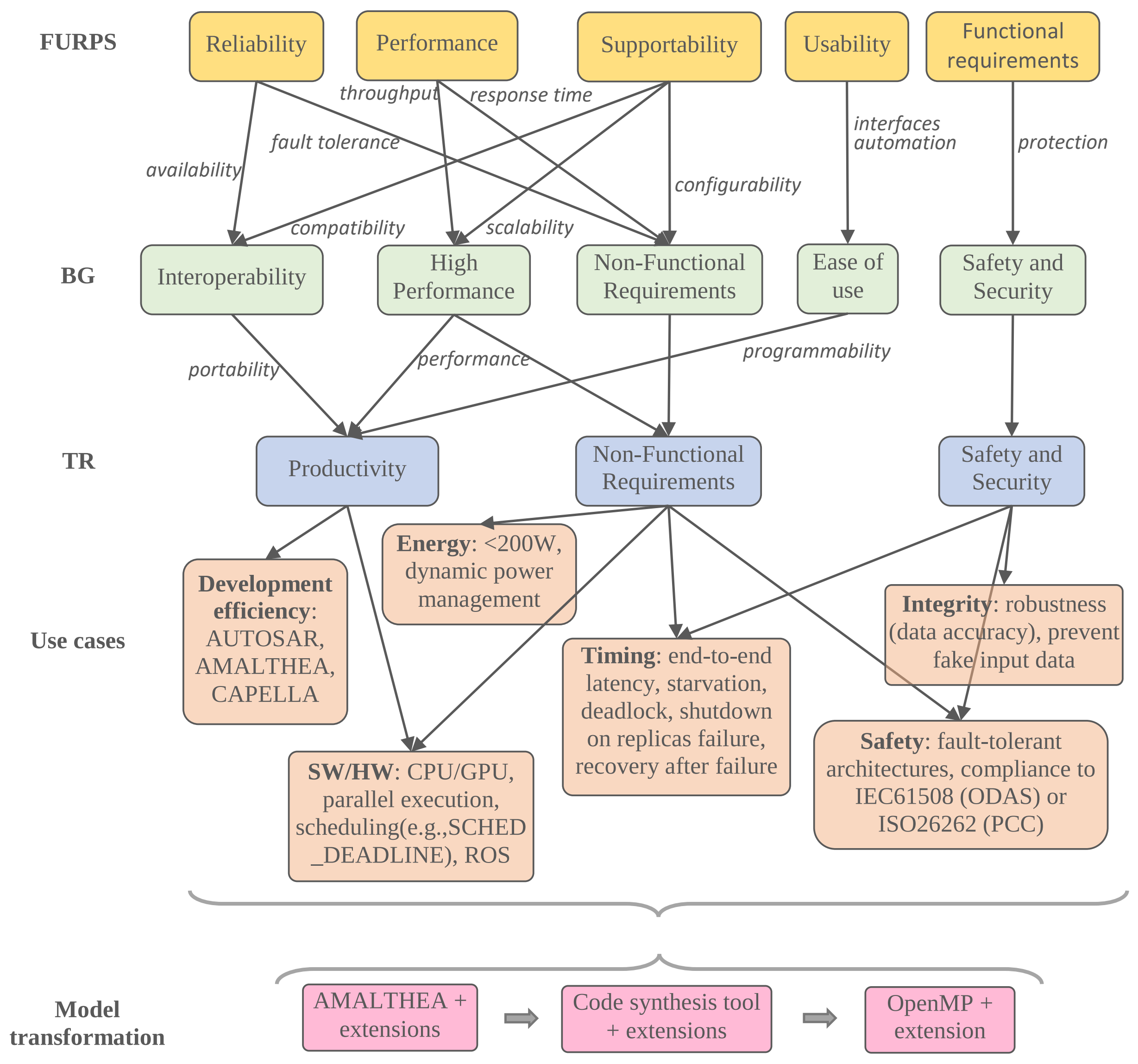

FURPS
Reliability
Performance
Supportability
Usability
Functional requirements
throughput
response time
fault tolerance
availability
interfaces automation
protection
configurability
compatibility
scalability
BG
Interoperability
High Performance
Non-Functional Requirements
Ease of use
Safety and Security
portability
performance
programmability
TR
Productivity
Non-Functional Requirements
Safety and Security
Use cases
Development efficiency: AUTOSAR, AMALTHEA, CAPELLA
Energy: <200W, dynamic power management
Integrity: robustness (data accuracy), prevent fake input data
Timing: end-to-end latency, starvation, deadlock, shutdown on replicas failure, recovery after failure
Safety: fault-tolerant architectures, compliance to IEC61508 (ODAS) or ISO26262 (PCC)
SW/HW: CPU/GPU, parallel execution, scheduling(e.g.,SCHED_DEADLINE), ROS
Model transformation
AMALTHEA + extensions
Code synthesis tool + extensions
OpenMP + extension

Figure 2: Requirements of the AMPERE ecosystem: a hierarchical description.

3 Refined AMPERE software development ecosystem

This section summarizes the software components composing the AMPERE software architecture at MS2, as depicted in Figure 3, and details the interfaces that will be used to communicate these components.

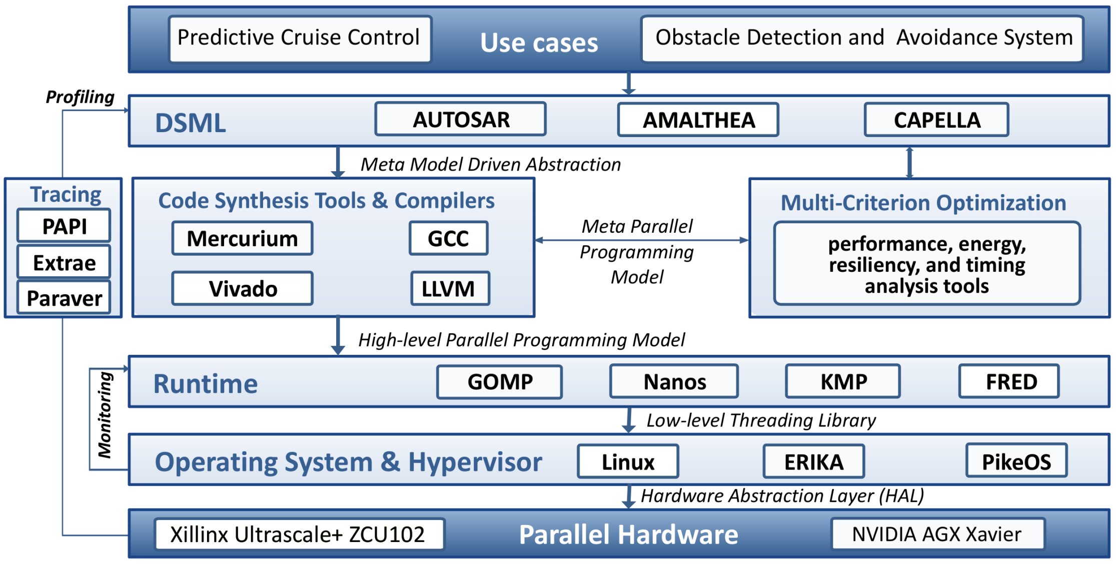

Predictive Cruise Control
Use cases
Obstacle Detection and Avoidance System
Profiling
DSML
AUTOSAR
AMALTHEA
CAPELLA
Meta Model Driven Abstraction
Tracing
PAPI
Extrae
Paraver
Code Synthesis Tools & Compilers
Mercurium
GCC
Vivado
LLVM
Meta Parallel Programming Model
Multi-Criterion Optimization
performance, energy, resiliency, and timing analysis tools
High-level Parallel Programming Model
Monitoring
Runtime
GOMP
Nanos
KMP
FRED
Low-level Threading Library
Operating System & Hypervisor
Linux
ERIKA
PikeOS
Hardware Abstraction Layer (HAL)
Xillinx Ultrascale+ ZCU102
Parallel Hardware
NVIDIA AGX Xavier

Figure 3: AMPERE software architecture.

3.1 Software components

The components of the AMPERE software architecture we previously defined in D6.1, including a set of analysis tools for tracing. Table 2 lists all components, relating them with the deliverable where the components are described in detail.

3.2 Interfaces

This section includes refined interfaces for the communication of the different components of the AMPERE ecosystem. At MS2, the communication of the components has been refined, as reflected in Figure 4. The figure is a zoom-in in the box labeled as *Synthesis tools and Compilers*, from Figure 1, and shows components as boxes, and communications as edges; the edges are further annotated with the interface used for the communication.

The different interfaces defined in the AMPERE project are explained next. Each interface corresponds to one (or more, like for example in the case of the TDG) annotated edges from Figure 4.

Amalthea The *system design* communicates with the *code generator* using Amalthea (further explained in D1.1 [1] and D2.1 [2]). Figure 5b provides the Amalthea model corresponding to the application workflow in Figure 5a. The Predictive Cruise Control (PCC) use case [1] will be implemented using Amalthea, while the Obstacle Detection Assistance System (ODAS) use case [1] will be implemented using Capella. For the latter, a binding between Capella and Amalthea is being implemented, and further details can be found in D6.3 [4].

Sources The code generator, APP4MC SLG (further explained in D2.2 [5]), generates C sources annotated with OpenMP directives, based on the Amalthea model extended with support for non-functional requirements. Figure 5c shows the code currently generated by the APP4MC SLG from the model in Figure 5b.

Table 2: Deliverables detailing each component of the AMPERE software architecture

Domain	Component	Deliverable
Domain specific modeling language	AMALTHEA AUTOSAR CAPELLA	D1.1. System models requirement and use case selection
Parallel programming models	OpenMP CUDA	D2.1. Model transformation requirements
Compilers and hardware synthesis tools	Mercurium GCC LLVM Vivado	D6.1. AMPERE ecosystem requirements and integration plan
Analysis and testing	Multi-criteria analysis tools PAPI Extrae Paraver	D3.2. Single-criterion energy optimisation framework, predictable execution models and software resilient techniques
Runtime libraries	Nanos GOMP KMP FRED	D6.1. AMPERE ecosystem requirements and integration plan
Operating systems	Linux Erika	D5.2. Single-criterion operating systems and hypervisor software
Hypervisor	PikeOS	D5.2. Single-criterion operating systems and hypervisor software

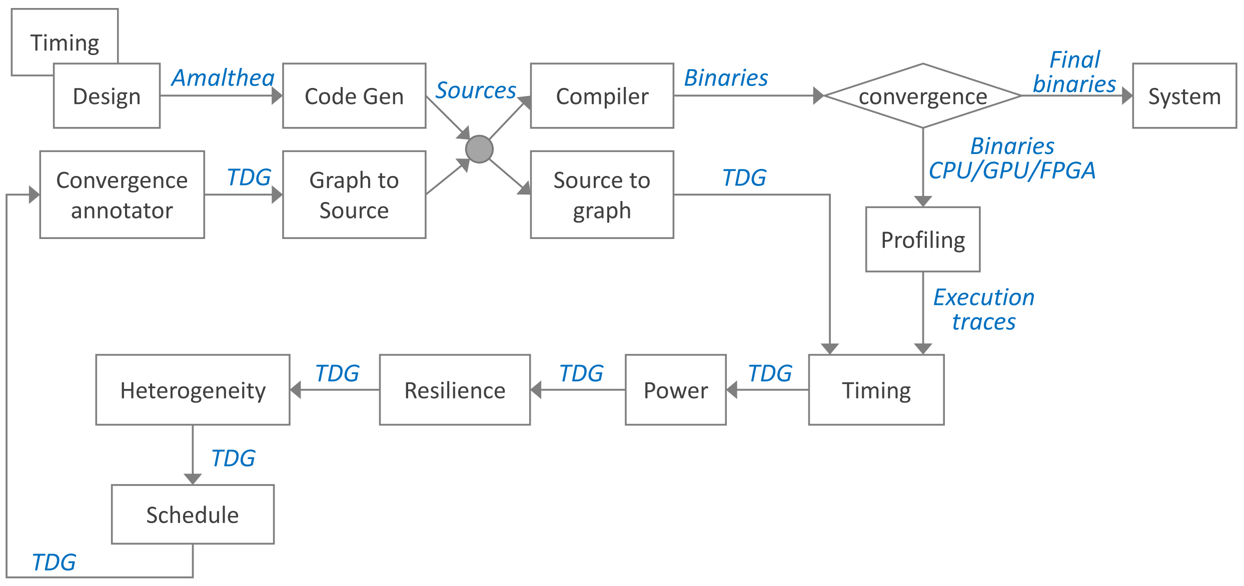


Figure 4: AMPERE's offline pipeline (from modeling to binary generation).

Traces Based on the binaries generated by the compiler, particularly GCC, AMPERE will use tracing and profiling tools, i.e., PAPI [6], Extrae [7] and Paraver [8], to obtain runtime data from the application. Extrae stores the execution information in three different files: (a) *.prv, with the time stamps and the events recorded, (b) *.pcf, with the configuration responsible for translating values contained in the trace into a more human readable values, and (c) *.row, a file containing

the distribution of the application across the cluster computation resources.
The format of the *.prv file [9] is as follows:

- Header:
  "#Paraver ($dd/mm/yy$ at $hh : mm$):$total\ time$:
  $\#nodes$($\#cpus$):$\#applications$:$app\ id$($\#cpus$:$\#nodes$),0"
- State record: intervals of the thread status.
  "1:$cpu\ id$:$app\ id$:$task\ id$:$thid$:$begin\ time$:$end\ time$:$state$"
- Event record: punctual events
  "2:$cpu\ id$:$app\ id$:$task\ id$:$thid$:$time$:$event\ type$:$eventvalue$"

The format of the *.pcf file is as follows:

```
DEFAULT_OPTIONS
LEVEL                THREAD
UNITS                NANOSEC
LOOK_BACK            100
SPEED                1
FLAG_ICONS           ENABLED
NUM_OF_STATE_COLORS  1000
YMAX_SCALE           37

DEFAULT_SEMANTIC
THREAD_FUNC           State As Is


STATES
0    Idle
1    Running
2    Not created
... // List of states

STATES_COLOR
0    {117,195,255}
1    {0,0,255}
2    {255,255,255}
... // List of colors (one for each state)

EVENT_TYPE
0 60000001  Parallel (OMP)
VALUES
0 close
1 DO (open)
2 SECTIONS (open)
3 REGION (open)

... // Descriptions of all event-type/event-values gathered during the tracing
```

Finally, the format of the *.row file is as follows:

```
 LEVEL CPU SIZE N_CORES
name_core_1
name_core_2
... // Names of each core

LEVEL NODE SIZE N_NODES
node_name_1
... // Names of each node

LEVEL THREAD SIZE N_THREADS
name_thread_1
name_thread_2
... // Names of each thread
```

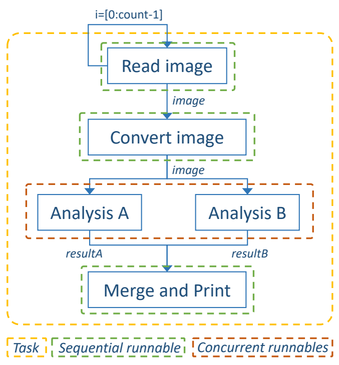
Figures 5d, 5e and 5f show the *.prv, *.pcf and *.row files generated for an execution of the code Figure 5c in a 4-core shared-memory machine.
Extrae can gather information not only from the application source (e.g., OpenMP task creation and OpenMP task instantiation), but also from hardware counters using the PAPI API. Furthermore, Extrae traces can be visualized with Paraver [8], as shown in Figure 5g. There, the execution of code in Figure 5c runs on 4 threads, although only 2 are used. Each value/color corresponds to one task (e.g., 1/blue corresponds to run_read_image, 2/green corresponds to $run\_convert\_image$, etc.)

TDG The compiler, particularly Mercurium [10], is in charge of transforming the source code generated by the APP4MC SLG into a Task Dependency Graph (TDG) that will be used and augmented by the Multi-criterion optimization tools for NFR analysis. Figure 5h shows the TDG currently generated by Mercurium and refined by a Python script (further details in D4.2 [11]) for the model in Figure 5c, and further augmented with information from the execution obtained with Extrae. The format of the TDG is as follows:

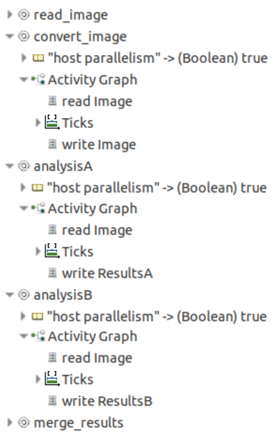
```
struct tdg_node{
        unsigned long id;               // Task instance ID
        unsigned short offin;           // Starting position in tdg_ins
        unsigned short offout;          // Starting position in tdg_outs
        unsigned char nin;              // Number of input dependencies
        unsigned char nout;             // Number of output dependencies
        unsigned int pragma_id;         // Identifier of the task contruct
        // The following members are filled with tracing information
        int execution_time;
        int papitot_counter_vals[8];
};
unsigned short tdg_ins[] = {...};     // Array of input dependencies
unsigned short tdg_outs[] = {...};    // Array of output dependencies
```

Currently, we recognize the execution time and a set of interesting hardware counters (like energy consumption). There is probably other data that will be needed in the TDG for the analysis of non-functional requirements. For this reason, and with the aim of making the TDG easily extensible, we have decided to add, for each node of the TDG (i.e., each Amalthea runnable, or OpenMP task), an array of elements (i.e., $papitot\_counter\_vals$), which can be easily tuned and extended if necessary.

Binaries This is the executable code generated by the compiler, either temporary (i.e., used during the analysis phase), or final (i.e., deployed in the system). The lowering of the OpenMP directives into runtime calls was already defined in D6.1 [3].



(a) Application's workflow.



```
▸ read_image
▾ convert_image
   ▸ "host parallelism" -> (Boolean) true
   ▾ Activity Graph
        read Image
      ▸ Ticks
        write Image
▾ analysisA
   ▸ "host parallelism" -> (Boolean) true
   ▾ Activity Graph
        read Image
      ▸ Ticks
        write ResultsA
▾ analysisB
   ▸ "host parallelism" -> (Boolean) true
   ▾ Activity Graph
        read Image
      ▸ Ticks
        write ResultsB
▸ merge_results
```

(b) AMALTHEA model.

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out:Image)
  run_read_image("");

  #pragma omp task depend(inout:Image)
  run_convert_image("");

  #pragma omp task depend(in:Image) \
        depend(out:ResultsA)
  run_analysisA("");

  #pragma omp task depend(in:Image) \
        depend(out:ResultsB)
  run_analysisB("");

  #pragma omp task depend(in:ResultsA, ResultsB)
  run_merge_results("");
}
```

(c) APP4MC SLG automatically generated code.

```
#Paraver (09/06/2021 at 12:25):8001972503_ns:1(4)
        :1:1(4:1),0
...
1:1:1:1:1:900570:902629:1
2:1:1:1:1:900570:60000018:3:60000118:3
2:4:1:1:4:900771:40000018:1
1:4:1:1:4:900771:902614:1
2:4:1:1:4:900771:60000018:3:60000118:3
2:2:1:1:2:901338:40000018:1
1:2:1:1:2:901338:903275:1
2:2:1:1:2:901338:60000018:3:60000118:3
2:3:1:1:3:901341:40000018:1
1:3:1:1:3:901341:903271:1
2:3:1:1:3:901341:60000018:3:60000118:3
1:4:1:1:4:902614:904986:7
2:4:1:1:4:902614:60000002:6
...
```

(d) Extrae trace (*.prv).

```
STATES
0      Idle
1      Running
2      Not created
...
STATES_COLOR
0      {117,195,255}
1      {0,0,255}
2      {255,255,255}
...
EVENT_TYPE
0 60000001    Parallel (OMP)
VALUES
0 close
1 DO (open)
2 SECTIONS (open)
...
```

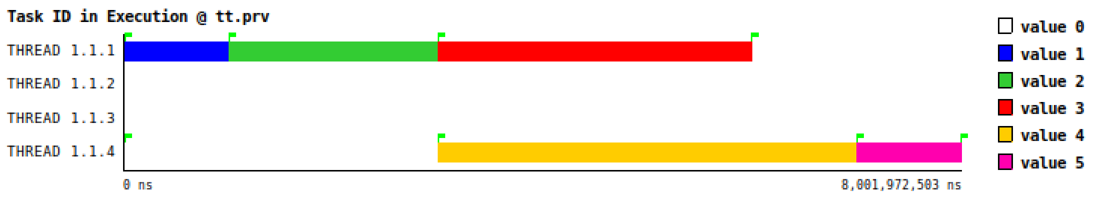
(e) Extrae trace (*.pcf).

```
LEVEL CPU SIZE 4
1.ampere_node
2.ampere_node
3.ampere_node
4.ampere_node

LEVEL NODE SIZE 1
ampere_node

LEVEL THREAD SIZE 4
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
```

(f) Extrae trace (*.row).



(g) Paraver visualization of the Extrae trace.

```
// Information obtained from the Extrae generated *.pcf file
int papitot_counter_flags[8] = {42000050, 42000059, 42000000, 42000002, 42000008, 42000055,
      42000046, 42001047};

struct tdg_entry tdg[5] = {
  {.id = 1, .offin = 0, .offout = 0, .nin = 0, .nout = 1, .pragma_id = 1, .execution_time = 434684,
           .papitot_counter_vals = {1609349,1529047,681,3091,547366912845266355 2,0,0,0}},
  {.id = 2, .offin = 0, .offout = 1, .nin = 1, .nout = 2, .pragma_id = 2, .execution_time = 741706,
           .papitot_counter_vals = {1638966,1776289,2022,8773,5473669128452663552,0,0,0}},
  {.id = 3, .offin = 1, .offout = 3, .nin = 2, .nout = 1, .pragma_id = 3, .execution_time = 473707,
           .papitot_counter_vals = {1620428,1563059,1122,4125,5473669128452663552,0,0,0}},
  {.id = 4, .offin = 2, .offout = 4, .nin = 2, .nout = 1, .pragma_id = 4, .execution_time = 5895201,
           .papitot_counter_vals = {1628180,1451409,1599,5407,5473669128452663552,0,0,0}},
  {.id = 5, .offin = 3, .offout = 4, .nin = 2, .nout = 0, .pragma_id = 5, .execution_time = 5895201,
           .papitot_counter_vals = {1645239,1452896,420,3210,5473669128452663552,0,0,0}}
};
unsigned short tdg_ins[] = {0, 1, 1, 2, 3};
unsigned short tdg_outs[] = {1, 2, 3, 4, 4};
```

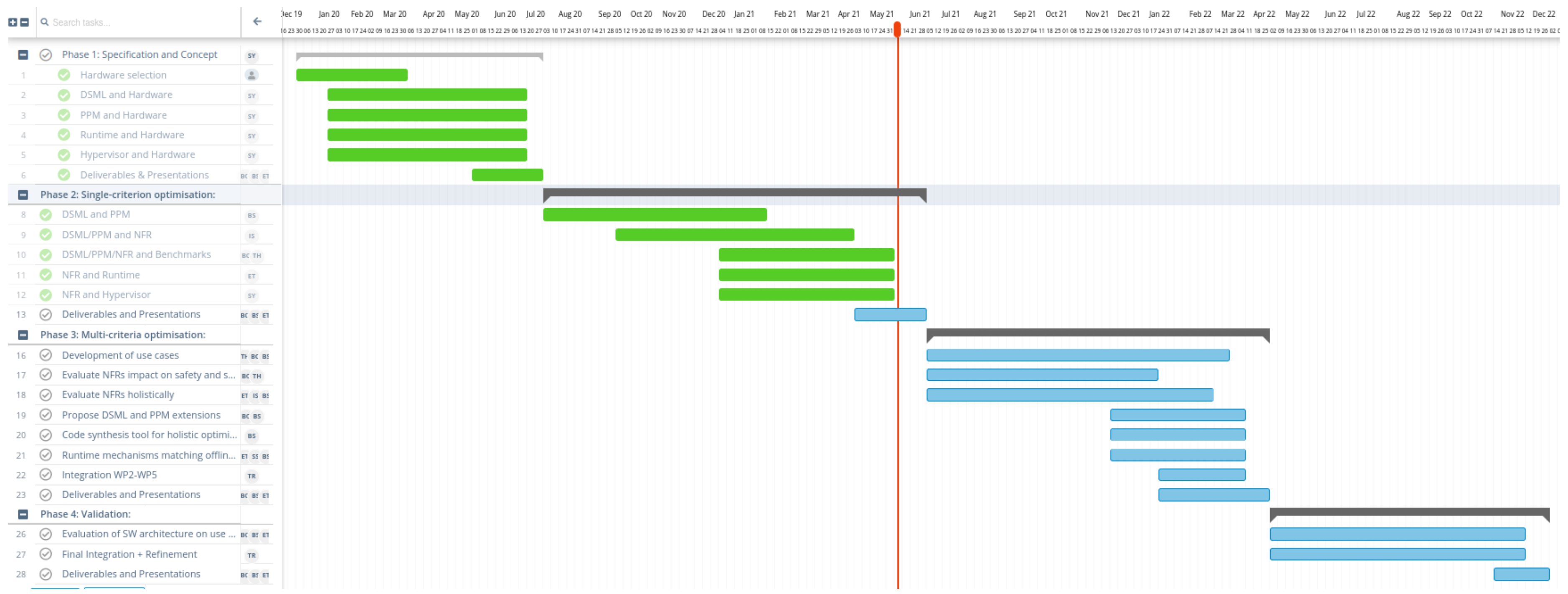
(h) TDG from Mercurium + script + Extrae tracing information

Figure 5: AMPERE's interfaces pipeline example.

4 Refined software development and integration plan

D6.1 already defined the software development and integration plan thoroughly. Since then, the AMPERE partners have just decided not to follow a Scrum based methodology, but an Agile [12] methodology. The reason is that the Scrum methodology implies tight sprints, the definition of a scrum master, and several other mechanisms that will be very difficult to fulfill in the distributed and heterogeneous environment conformed by the partners of the AMPERE project. Instead, Agile is a more flexible set of rules (which are the origin of methodologies like Scrum), to ensure the efficiency and the quality of the development process and the final product. These rules include: (a) iterative, incremental and evolutionary development, (b) efficient face-to-face communication, (c) short feedback loop and adaptation cycle, and (d) focus on the quality. To follow this methodology, all partners meet every week alternating meetings for discussing the advancements on the software architecture components and communication, and the advancements on the use cases. Besides these meetings, we also have ad hoc meetings for specific tasks when necessary.

Figure 6 includes the Gantt diagram detailing the integration process and plan. It is based on the tree-like diagram for the integration of the different components already presented in D6.1 [3] (specifically, in Figure 5 of that deliverable).



	Phase 1: Specification and Concept	SY
1	Hardware selection	
2	DSML and Hardware	SY
3	PPM and Hardware	SY
4	Runtime and Hardware	SY
5	Hypervisor and Hardware	SY
6	Deliverables & Presentations	BC BS ET
	Phase 2: Single-criterion optimisation:	
8	DSML and PPM	BS
9	DSML/PPM and NFR	IS
10	DSML/PPM/NFR and Benchmarks	BC TH
11	NFR and Runtime	ET
12	NFR and Hypervisor	SY
13	Deliverables and Presentations	BC BS ET
	Phase 3: Multi-criteria optimisation:	
16	Development of use cases	TH BC BS
17	Evaluate NFRs impact on safety and s...	BC TH
18	Evaluate NFRs holistically	ET IS BS
19	Propose DSML and PPM extensions	BC BS
20	Code synthesis tool for holistic optimi...	BS
21	Runtime mechanisms matching offlin...	ET SS BS
22	Integration WP2-WP5	TR
23	Deliverables and Presentations	BC BS ET
	Phase 4: Validation:	
26	Evaluation of SW architecture on use ...	BC BS ET
27	Final Integration + Refinement	TR
28	Deliverables and Presentations	BC BS ET

Figure 6: Gantt diagram of the AMPERE project.

5 Conclusions

D6.2 extends D6.1 taking into consideration the opinion of the experts expressed in the consolidated interim report, and the advancements made during Phase 2 of the project. At this point, all tools and systems for the integration of the project are already defined and agreed by all partners.

During this phase, AMPERE partners have improved their interconnection significantly by way of weekly meetings, allowing to keep their own developments aligned with the overall AMPERE objectives. During the next phase, the tools defined for the integration of the different software components (defined in D6.1 [3]) will be used during to ensure a smooth transition from isolation to integration.

List of Acronyms

API	Application Program Interface
BG	Business Goal
D	Deliverable
HW	Hardware
MS	Milestone
NFR	Non-Functional Requirement
PCC	Predictive Cruise Control
SLG	Synthetic Load Generator
SW	Software
T	Task
TDG	Task Dependency Graph
TR	Technical Requirement
WP	Work Package
FURPS	Functionality, Usability, Reliabilty, Performance, Supportability
SLG	Synthetic Load Generator
ODAS	Object Detection and Avoidance System

6 References

[1] AMPERE, "D1.1. System models requirements and use case selection," 2020.

[2] ——, "D2.1. Model transformation requirements," 2020.

[3] ——, "D6.1 AMPERE ecosystem interfaces and integration plan," 2020.

[4] ——, "D6.3 Single-criterion AMPERE ecosystem," 2021.

[5] ——, "D2.2. First release of the meta parallel programming abstraction and the single-criterion performance-aware component," 2021.

[6] Innovative Computing Laboratory, University of Tennessee, "Performance Application Programming Interface," 2021, icl.utk.edu/papi/.

[7] Barcelona Supercomputing Center, "Extrae," 2021, tools.bsc.es/extrae.

[8] ——, "Paraver," 2021, tools.bsc.es/paraver.

[9] ——, "Paraver: trace file description," 2001, tools.bsc.es/doc/1370.pdf.

[10] ——, "Mercurium compiler," 2021, pm.bsc.es/mcxx.

[11] AMPERE, "D4.2. Independent runtime energy support, predictability, segregation and resilience mechanisms," 2021.

[12] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries *et al.*, "Manifesto for agile software development," 2001.