Ref. Ares(2020)4082539 - 03/08/2020

D2.1 Model Transformation Requirements

Version 1.0

Documentation Information

Contract Number	871669
Project Website	www.ampere-project.eu
Contratual Deadline	31.07.2020
Dissemination Level	PU
Nature	R
Author	Sara Royuela (BSC)
Contributors	Eduardo Quiñones (BSC), Michael Pressler (BOSCH), Delphine Longuet (TRT) and Alexandre Amory (SSSA)
Reviewer	Jan Rollo, Enkhtuvshin Janchivnyambuu (SYSGO)
Keywords	Model Driven Engineering, Parallel Programming Models, Functional and non-Functional Requirements

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 version by Sara Royuela and contributions by Eduardo Quiñones (BSC)
V0.2	Contributions by Michael Pressler (BOSCH) on AMALTHEA
V0.3	Contributions by Delphine Longuet (TRT) on CAPELLA
V0.4	Contributions by Alexandre Amory (SSSA)
V1.0	Deliverable ready for submission

Table of Contents

1. Executive Summary 3
2. Introduction 3
3. Model driven engineering overview........ 4
  3.1. AMALTHEA and AUTOSAR 4
    3.1.1. General Description........ 4
    3.1.2. Parallelism exposed........ 7
  3.2. CAPELLA........ 7
    3.2.1. General Description........ 8
    3.2.2. Parallelism exposed........ 10
4. Parallel programming models........ 10
  4.1. CUDA 10
  4.2. OpenCL 12
  4.3. OpenMP 13
  4.4. OmpSs 14
  4.5. Summary: Programming models productivity........ 15
5. Preliminary analysis of DSML transformation: from AMALTHEA to OpenMP 17
  5.1. Base Language........ 17
  5.2. Execution Model 18
  5.3. Support for functional and non-functional requirements........ 20
    5.3.1. Functional safety and correctness 20
    5.3.2. Time predictability........ 21
    5.3.3. Energy........ 22
6. State-of-the-art synthesis tools 23
7. Summary and Conclusions........ 27
8. Acronyms and Abbreviations........ 28
9. References 28

1. Executive Summary

This deliverable covers the work done during the first phase of the project within WP2. The deliverable spans 7 months' work (including 1 extra month with respect of the Grant Agreement (European Comission and AMPERE beneficiaries, 2019) due to the COVID19 situation), and handles the work done in Task 2.1 "Model transformation requirements specification" to reach milestone 1 (MS1). Concretely, the deliverable covers the activities conducted within WP2 towards the implementation of a code synthesis component capable 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, Task 2.2 will define 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 selected in Task 5.1 (AMPERE, 2020).

The target at MS1 is: (1) the definition of the code synthesis tool for an efficient model transformation and parallel programming model support to express functional and non-functional constraints, and (2) identification of current state-of-the-art synthesis tools. The first milestone of Task 2.1 has been carried out successfully, and all objectives of MS1 have been reached and documented in this deliverable.

2. Introduction

WP2 aims to develop a *meta parallel programming abstraction* independent of the underlying processor architecture, capable of capturing all system functional and non-functional requirements, as well as incorporating the parallel semantics required to enable and efficient model transformation, optimized for performance, timing, resiliency, cyber-security and energy-efficiency. With such a purpose in mind, Section 3 introduces the general requirements of the model driven engineering tools, Section 4 introduces the requirements of the parallel programming models, and Section 5 discusses the compatibility between the domain specific modelling languages (DSML) and the parallel programming models (PPM).

The relation of this deliverable with other WP is shown in Table 1, considering deliverables and tasks that are involved with the study performed in this deliverable.

Table 1. Relationship between D2.1 and other WPs.

Deliverable	Leader	Task	Description
D1.1	THALIT	T1.1	System models requirements specification and use case selection
D3.1	ISEP	T3.1	Multi-criteria optimization requirements specification
D4.1	SSSA	T4.1	Runtime requirement specification
D5.1	SYSGO	T5.1	Reference parallel heterogeneous hardware selection

3. Model driven engineering overview

Model Driven Engineering (MDE) is a structured development paradigm focused on models as central artifacts to describe CPS requirements, capture the overall CPS architecture (including HW and SW elements if needed), and specify functional behavior and interfaces. MDE methods provide the capability to reason about the properties and relationships between the components of the system while abstracting their complexities. Particularly, MDE presents the following benefits:

- MDE allows defining components with clear interfaces facilitating the integration of new features by means of *composability*, i.e., the non-functional requirements fulfilled when developing new components in isolation is maintained at system integration.
- MDE enables the use of code synthesis methods and tools that transform the system model description into source code to be compiled and executed in the target platform. These tools entail a *correct-by-construction* paradigm in which consistency between the transformed code and the system models can be ensured.
- MDE enables the development of *domain specific modelling languages* (DSML) that facilitate the description of the cyber physical interactions characteristics from each domain.

For these reasons, MDE is a methodology suitable to build complex systems with conflicting properties such as high dependability, high performance and competitive cost. Examples of such systems are those used in the automotive and the railway industries, as the use cases proposed in the AMPERE project (AMPERE, 2020).

Unfortunately, current MDE synthesis tools, like AUTOSAR used in the automotive domain, transform DSMLs into a sequence of concurrent code suitable only for single-core execution. Other tools like Simulink Coder provide a limited support for parallelism, allowing for multi-core execution of models but without exploiting the possible parallelism inside the model subsystems.

This section briefly introduces the three state-of-the-art DSML considered in the AMPERE project, focusing on those relevant aspects of the DSML execution model that can impact on the parallel execution of the system. See Deliverable D1.1 (AMPERE, 2020) for a complete description of the DSMLs addressed by the AMPERE project.

3.1. AMALTHEA and AUTOSAR

AMALTHEA and AUTOSAR are two different state-of-the-art standards for distributed model-based development in the automotive industry. A general description of the models and the analysis of the opportunities for exploiting parallelism in the model are detailed next.

3.1.1. General Description

AMALTHEA (BOSCH, 2020) is an open source tool platform for engineering embedded multi- and many-core software systems proven in the automotive sector by Bosch and their partners. The development of the AMALTHEA data model and the platform is part of the Eclipse APP4MC project (Eclipse Foundation, Inc, 2020). AMALTHEA offers certain compatibility with AUTOSAR (Fürst, y otros, 2009).

AUTOSAR (AUTomotive Open System ARchitecture) is an alliance of key industrial players in the automotive industry, establishing a de-facto open industry standard for the development and

execution of automotive software architecture. The standard defines two main platforms: the *Classic Platform* for embedded systems with hard real-time and safety-critical constraints, and the *Adaptive Platform* for fail-safe high-performance computing ECUs for use cases such as autonomous driving.

AMLTHEA includes two different models at the top level: the system model, and the trace model. The former is used for designing and modeling, while the latter is used for testing and providing feed-back to the system model. WP2 aims to focus on the system model because it is the one that includes the information about the parallel nature of the system and the functional and non-functional requirements of system components. Concretely, The AMALTHEA system model is composed by a series of data models that allow defining: (1) the units of (concurrent) work and the functionalities implemented in the system (in terms of execution time, memory access, etc.), (2) the elements of the hardware platform (e.g., processors, memories, etc), and (3) the mapping between the software and the hardware elements (i.e., schedulers, work mapping, etc.). The most relevant data models for the model transformation, together with their components and functionalities, are introduced next:

a. *Software model*. Among others, it defines:
    - *Task* (process): The unit of concurrency at the operating system (OS) level scheduler composed of a sequence of functionalities (runnables) that run sequentially. It is defined by a priority, a preemption model, a deadline, a multiple time activation (MTA) and a call tree (or activity graph).
    - *Runnable* (function): An atomic functionality not visible by the OS scheduler. It is defined by the number of cycles it takes to execute and the access it does to variables (e.g., labels, frequency, etc.)

b. *Stimuli model*: defines the activation of tasks using a set of attributes like the type of activation (e.g., periodic, sporadic, etc.), the offset and the recurrence.

c. *Constraints model*. Among others, it defines:
    - *Runnable sequencing constraints*: define the sequential ordering among runnables based on, for example, data exchange.
    - *Event chains*: relate stimuli to events, and allows defining sequences, parallel paths and combinations of both.
    - *Timing constraints*: define restrictions for the execution of the events like repetition and delay.

These three models allow defining the components relevant to determine the units of concurrency and the relationships between these units, as well as several non-functional requirements of the system regarding performance and time-predictability. Moreover, AMALTHEA includes two complementary models related to the underlying platform that are particularly interesting for WP2, which are:

a. *Hardware model*. Among others, it defines:
    - Processing units: number and features like frequency, scratchpad, flash, etc.
    - Memory: available units, latency, etc.

b. *OS model*. Among others, it defines the scheduler with attributes like the algorithm, and the delay of context switch, among others.

AUTOSAR provides similar abstractions to those supported by AMALTHEA system model. In fact, the design of AMALTHEA has been highly inspired in AUTOSAR. From a *software model* perspective, an AUTOSAR application is composed of a set of *runnables* that communicate with each other, and *AUTOSAR tasks*, which agglomerate runnables with the same release period and are the unit of scheduling of the AUTOSAR run-time environment (RTE), as depicted in Figure 1.

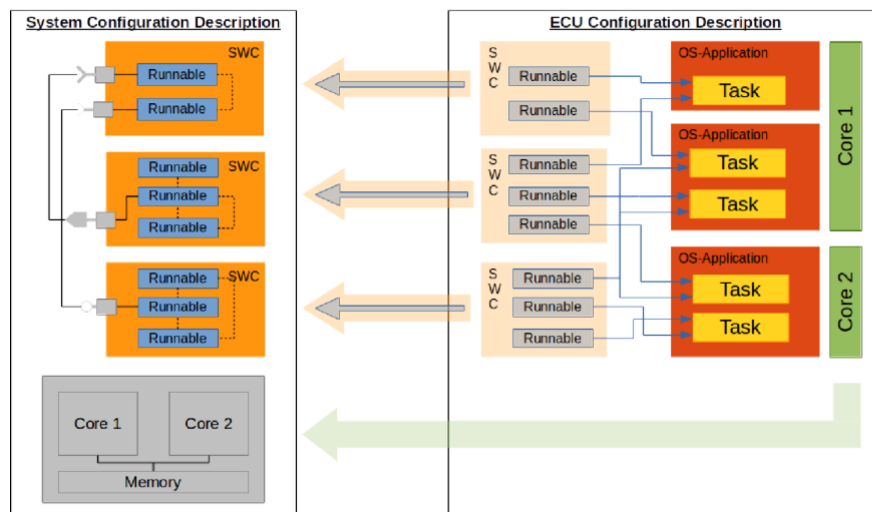

Figure 1. Content of AUTOSAR Configuration Descriptions (Sailer, 2014).

For a *constraints model* perspective, the execution order of runnables is constraint by two kinds of precedences: (1) simple precedence and (2) extended precedence, which result from the communication among runnables with the same or different release period, respectively. As a result, simple precedences only exist within tasks, while extended precedences only exist among tasks.

AUTOSAR provides rich communication method, described by a virtual function bus (VFB) and software-component (SW-C) model. Two communication mechanisms for the exchange of a single data element between runnables are defined: inter-runnable-variable (IRV) and sender-receiver (SR) communication, used between runnables from the same or from different SW-Cs, respectively. The run-time environment (RTE) guarantees data consistency for both mechanisms. The developer can define two modes: by default, communication is explicit, i.e. a precedence is imposed from producer to consumer, defining a strict order of execution, and the consumer uses the most recent value of the producer; optionally, communication can be defined as implicit, in which data is distributed to all consumers after the producer execution has finished. On the consumer side data is buffered and calculations are performed on a copy. As a result, concurrent execution of runnables is possible, because data are buffered and delivered with a delay. This is a form of asynchronous communication. AMALTHEA also allows expressing sender-receiver communication. Although the model is much more limited in this case, the dynamic execution model allowed by AUTOSER is similar to AMALTHEA.

AUTOSAR also provides support for the specification of the timing behavior of applications through the AUTOSAR Timing Extensions (TIMEX) (Peraldi-Frati, Blom, Karlsson, & Kuntz, 2012). This is used in the AUTOSAR Classic Platform to associate timing requirements and specifications (e.g., worst-case response time of an end-to-end service chain) to (1) functional blocks of the

specification during the early-stage specification phases, and (2) the intermediate elements of the function networks obtained during the functional decomposition. For example, TIMEX allows for the specification of worst-case execution times and worst-case transmission times for communications.

Unfortunately, regarding the possibility to run components on parallel and multi-core platforms, AUTOSAR and AMALTHEA lack a detailed model because they have tackled the execution semantics from an OS point of view. The complexity of modern automotive systems increases the importance of this inadequacy.

3.1.2. Parallelism exposed

AMALTHEA and AUTOSAR define three execution scopes in which parallelism can be potentially exploited with different granularity levels:

1. *Among tasks*: this is the only option currently available to exploit through the OS scheduler. In case of AMALTHEA, it provides a set of synchronization mechanisms among tasks to ensure the correct order of execution.
2. *Among runnables*: this option is currently not supported because the model forces runnables to execute sequentially within a task. This limitation in AMALTHEA is a legacy aspect of the model, inherited from AUTOSAR. To accomplish concurrency among runnables, they must be enclosed in different tasks. It is worth mentioning that Bosch has proposed to create dedicated tasks only to transfer data from/to the GPU and execute GPU kernels concurrently from the rest (Wurst, y otros, 2019).
3. *Inside runnables*: this option is transparent to the AMALTHEA and AUTOSAR models since the internals of the runnables are not exposed to the model and it is not visible to the OS scheduler.

As a result of our analysis, we conclude that: (1) parallelism among tasks is a coarse-grained level of parallelism suitable for being exploited by the OS, as it is now; (2) parallelism among runnables is a fine-grained level of parallelism suitable for being exploited by the high-level parallel programming model in the shared-memory machine (or symmetric multi-processing, SMP); and (3) parallelism inside runnables is an even more fine-grained level of parallelism suitable for being exploited by the parallel programming model (not necessarily the same) in the SMP as well or in dedicated accelerators (e.g., GPU or FPGA).

Currently, AMALTHEA only supports the three-levels of parallelism described above by wrapping all runnables suitable for concurrency in different tasks, using the task as the unit of parallelism. Then, these newly created tasks must be properly synchronized with the rest of the tasks, as well as be assigned to a specific scheduler to ensure the correct order of execution. This forces the programmer to do the additional exercise of including certain runnables to “artificial” tasks.

3.2. CAPELLA

Capella (Roques, 2017) is an open-source tool for model-based systems engineering developed by Thales. A general description of the model and an overview of the opportunities for exploiting parallelism in the model are detailed next.

3.2.1. General Description

Capella provides SysML-inspired diagrams for graphical modelling of systems, hardware and software architectures. It implements the principles and recommendations defined by Arcadia (Voirin, 2017), the Thales standard systems engineering method. Capella is implemented on top of the Eclipse IDE platform, based on the PolarSys solution.

The Arcadia method enforces an approach structured on different engineering perspectives. These perspectives are clearly separated between system contexts and need modeling (operational need analysis and system need analysis) and solution modeling (logical and physical architectures). The perspective dedicated to the need understanding help the system engineer to define what the users of the system need to accomplish and what the system has to accomplish for the users. The perspectives for the solution architectural design show how the system will work in order to fulfil its expectations and how the system will be developed and built. The perspective of interest for the AMPERE project is the physical architecture, at the lowest level of abstraction, which describes the finalized solution, with enough details to provide unambiguous contracts towards downstream engineering teams.

The Capella workbench can be enhanced or specialized for a given business need, according to the concept of viewpoint. In the context of the AMPERE project, the relevant viewpoint is Tideal, which allows system architects to capture the performances properties and requirements of complex real-time embedded systems.

The Tideal viewpoint extends diagrams used in the physical architecture perspective (Physical Architecture Blank diagrams and End-To-End Flow Scenario diagrams). It adds new concepts such as Timing Design Scope that allows to select all the elements that need to be analyzed and schedulers (see Figure 2) and End-To-End Flow timing constraints (see Figure 3). It also extends existing Capella elements such as Physical Component, Physical Functions and Executions to define their role in the real-time system and their timing properties.

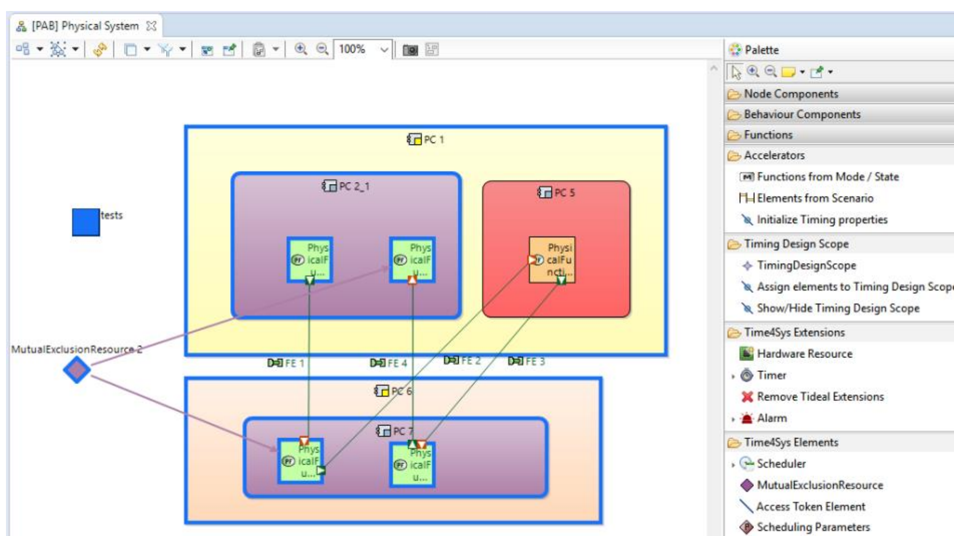

Figure 2. Physical Architecture Blank diagram extension

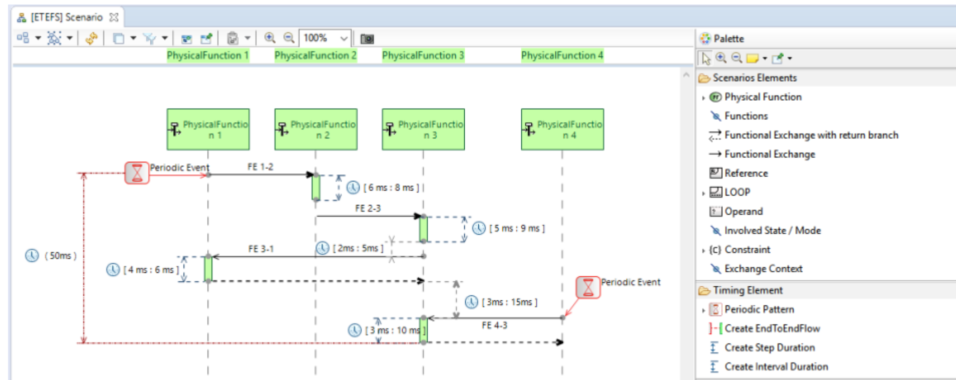

Figure 3. End-to-End Flow diagram extension

Since the aim of the AMPERE project is to guarantee by construction that the resulting system fulfils its functional and non-functional requirements, the design models should allow for analysis and verification. However, Capella does not propose behavioral semantics, even within the Tideal extension. The Time4Sys[1] open-source platform bridges the semantic gaps between existing system modelling editors and real-time analysis tools. It implements model transformations in order to translate the Capella/Tideal design models into verification models that can be taken as inputs by verification tools. Time4Sys is not limited to a specific design framework or a specific tool. Instead, it provides ways to extend the import (resp. export) mechanisms in order to connect any existing modelling language (resp. analysis tool).

Time4Sys implements a dedicated editor based on its own implementation of UML-MARTE meta-model (Object Management Group, 2019). This editor is basically not used when the design model is directly generated from Tideal but the analysis of a problem may require a real-time expert to check and adapt the design model (e.g. a deadline is missed); this short trial/error loop could then be done directly in Time4Sys without having to edit the upstream architecture model.

Time4Sys is composed of two building blocks, the design and the verification (or analysis) pivot models, as well as a set of transformation rules between them. It allows representing a synthetic view of the system design model that captures all elements, data and properties impacting the system timing behavior and required to perform timing verification (e.g. tasks mapping on processors, communication links, execution times, scheduling parameters, etc.). More precisely, the design model consists of the following elements:

- Resources:
  - Hardware resource: a processor, with a scheduling policy, to which is allocated a set of tasks.
  - Software resource: a task, with a (relative) deadline.
  - Bus: a communication medium to which is allocated a set of communication channels.
- Applications:
  - Execution or communication step: characterized by a best- and worst-case execution times and a priority. Execution steps correspond to tasks, communication steps correspond to communication channels.

[1] https://www.eclipse.org/time4sys/

- Event: the activation policy for an execution step (periodic, sporadic, burst, sliding window, once…)
- Relations between (execution or communication) steps:
  - Dependency: the completion of a step on a resource can trigger the activation of a step on the same or on another resource.
  - Mutual exclusion: the steps involved cannot be executed concurrently since for instance, they need access to the same memory resource.

Various scheduling policies are supported by Time4Sys, including:

- First-in-First-out (FIFO): tasks are treated in the same order as they are activated.
- Fixed priority (FPS): each task on a given processor comes with a given static priority and, upon completion of a task instance, the highest priority task instance in the waiting queue is selected for execution.
- Preemptive FPS: a version of FPS where a lower priority task can be temporarily stopped to execute a newly activated instance of a higher priority task.
- (Preemptive) RMS: FPS where tasks with shorter periods necessarily have a higher priority.
- Earliest Deadline First (EDF): the task with the closest deadline is executed first.
- Shortest job first (SJF): the shortest task instance is selected first.
- Time-Division Multiple Access (TDMA) and Round Robin: tasks are allocated slots in a cyclic manner to execute (part of) their instances, one after the other.

3.2.2. Parallelism exposed

Capella and Time4sys define a way to express parallelism using tasks running on the same hardware resource. This constitutes a high level view of parallelism but currently no finer modeling can be done in Capella or Time4Sys. Shared memory accesses can be expressed using mutual exclusion. Currently, a task releases a mutex when it stops running, even if the task has not completed (e.g., it has been preempted by a higher priority task). This means that it has currently no interests for tasks running on the same hardware resource. If necessary, a richer model for shared memory could be developed.

4. Parallel programming models

This section introduces basic information about the execution model and the memory model supported by the parallel programming models considered for the AMPERE project and supported by the two parallel processor architectures selected: The NVIDIA Jetson AGX and the Xillinx UltraScale+ (see Deliverable D5.1 (AMPERE, 2020) for further information).

4.1. CUDA

CUDA (NVIDIA®, 2020) is a parallel programming model designed to naturally map the parallelism within an application to the massive parallelism of the stream multiprocessors (SMs) implemented in NVIDIA devices. The CUDA platform is accessible through CUDA-accelerated libraries, compiler directives (e.g., OpenACC), and extensions to industry-standard programming languages (e.g., C, C++). Additionally, interfaces including OpenCL and OpenGL are also supported.

A CUDA program is a serial program that calls parallel *kernels*, i.e., functions or full programs. Each kernel executes across a set of parallel threads organized in a hierarchy of grids of thread blocks. A *thread block* is a set of concurrent threads that can cooperate through synchronization and shared access to a memory space private to the block. A *grid* is a set of threads block that may execute in parallel with other grids. Parallelism is determined explicitly by specifying the dimensions of a grid and its thread blocks when launching a kernel.

Parallel execution and thread management are automatic. All thread creation, scheduling and termination are handled by the underlying system, mostly directly in hardware. Per block thread synchronization is accomplished calling the *__syncthreads()* intrinsic.

Threads may access data from multiple memory spaces: per-thread *local* memory, per-block *shared* memory and *global* memory. Global memory is managed via the *cudaMalloc()* and *cudaFree()* runtime calls. To support a heterogeneous system architecture combining a CPU and a GPU, each with its own memory system, CUDA programs must copy data between the *host memory* and the *device memory*. Unified memory is a component of CUDA that provides *managed memory* to bridge the host and device memory spaces by defining a memory space in which all processors see a single coherent memory image with a common address space.

```
/* kernel.cu */
__global__ void matrix_mul(float A[N][N], float B[N][N], float C[N][N],
int wA, int wB)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    float th_value = 0;
    for (int k = 0; k < wA; ++k)
       th_value += A[row * wA + k] * B[k * wB + col];

    C[row * wA + col] = th_value;
}

/* main.c */
int main()
{
    …
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N/threadsPerBlock.x, N/threadsPerBlock.y);
    matrix_mul<<numBlocks, threadsPerBlock>>(A, B, C, wA, wB);
    …
}
```

Listing 1. CUDA matrix multiplication example.

Listing 1 shows a simple example of a CUDA parallel program. There, the kernel matrix_mul is specified by means of the *__global__* specifier. The *threadIdx*, *blockIdx* and *blockDim* are built-in variables that allow accessing each thread, block and block dimension respectively.

The concurrency among kernels is managed using CUDA streams. A stream is a sequence of operations that execute in issue-order on the GPU. Using several streams allows the concurrent and synchronous or asynchronous execution of kernels.

Newer versions of CUDA include a new paradigm, CUDA graphs, offering two main characteristics: (1) it allows expressing work as graphs rather than single operations, and (2) it

enables a *define-once-run-repeatedly* execution flow. A graph consists of a series of operations, such as memory copies and kernel launches, connected by dependencies and defined separately from its execution. The programming interface offers

4.2. OpenCL

OpenCL (Khronos OpenCL working group, 2020) is an open standard for writing programs that execute across heterogeneous platforms including CPUs, GPUs, DSPs, FPGAs and other accelerators. Naturally, OpenCL pursues portability while considering programmability.

The OpenCL architecture consists of one *host* (CPUD-based) that controls multiple *compute devices* (CPUs and GPUs). Each of these consists of multiple *compute units* (equivalent to stream multiprocessors in NVIDIA, and stream cores or SIMD engines in AMD) and the latter contain multiple *processing elements*, each of them executing OpenCL *kernels*. So, the kernel is the basic unit of parallelism. Kernel bodies are instantiated once per *work item* (equivalent to a *CUDA thread*), and each work item gets a unique global id. Work-items are wrapped in *work-groups* (equivalent to a *CUDA thread block*).

OpenCL offers fine-grained data- and thread-parallelism (at the work-item level) nested within coarse-grained data- and task-parallelism (at the work-groups level). Synchronization in the form of memory fences is possible within threads in a work-group, as well as synchronization barriers for threads at the work-item level. Additionally, the host can use blocking API operations to wait for completion of certain events.

Listing 2 shows a simple example of a OpenCL parallel program. There, the kernel matrix_mul is specified by means of the *__global__* specifier. The *threadIdx*, *blockIdx* and *blockDim* are built-in variables that allow accessing each thread, block and block dimension respectively.

OpenCL has an advantage over CUDA, and is that it can be executed, not only in any GPU including the library, but also in the host. Hence, schedulers could decide to execute a given task with a unique OpenCL implementation in the host based on the availability of the resources or the performance expected for the given device (Wen, Wang, & O'boyle, 2014).

```
/* kernel.cl */
__kernel void matrix_mul(__global float A*, __global float B*, __global
float C*, int wA, int wB)
{
   int col = get_global_id(0);
   int row = get_global_id(1);

   float th_value = 0;
   for (int k = 0; k < wA; ++k)
      th_value += A[row * wA + k] * B[k * wB + col];

   C[row * wA + col] = th_value;
}

/* main.c */
int main() {
    …
    clGetDeviceIDs(platform_ids[0],
       gpu ? CL_DEVICE_TYPE_GPU : CL_DEVICE_TYPE_CPU,
       1, &device_id, NULL);
    context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
    commands = clCreateCommandQueue(context, device_id, 0, &err);
    lFileSize = LoadOpenCLKernel("kernel.cl",
       &KernelSource, false);
    program = clCreateProgramWithSource(context, 1,
       (const char **) & KernelSource, NULL, &err);
    err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    kernel = clCreateKernel(program, "matrix_mul", &err);
    …
    err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&d_C);
    err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&d_A);
    err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&d_B);
    err |= clSetKernelArg(kernel, 3, sizeof(int), (void *)&wA);
    err |= clSetKernelArg(kernel, 4, sizeof(int), (void *)&wB);
    err = clEnqueueNDRangeKernel(commands, kernel, 2,
        NULL, globalWorkSize, localWorkSize, 0, NULL, NULL);
    err = clEnqueueReadBuffer(commands, d_C, CL_TRUE, 0,
        mem_size_C, h_C, 0, NULL, NULL);
}
```

Listing 2.OpenCL matrix multiplication example.

4.3. OpenMP

OpenMP (OpenMP ARB, 2018) is an API aiming at facilitating parallel programming in shared-memory systems first, and also in heterogeneous systems based on the extensions of later specifications. The model allows expressing parallelism in a *fork-join* fashion. Parallelism is spawned when a parallel construct is reached, creating a team of threads, and joined when the implicit barrier at the end of a parallel region in found. Furthermore, parallelism can be spawned following two different paradigms: the *thread-based model*, which allows for data parallelism, and the *task-based model*, which allows for task parallelism.

Synchronization of threads occurs when a barrier construct is found, and also based on flush directives. Synchronization of tasks occurs based on taskwait and taskgroup directives, and also on task dependencies, hence enabling a data-flow model.

OpenMP offers a relaxed-consistency, shared-memory model that defines three different views of the memory: a shared space accessible to all threads called *memory*; a *temporary view* of memory for each thread; and a *threadprivate* memory, private to each thread that cannot be accessed by any other thread.

The *accelerator model*, based on the tasking model, is an extension that pursues portability between devices with different ISAs, as well as programmability, by easing the burden of defining data movements between the host and the accelerator, and performance boosted by inserting accelerated parts in the applications. This is a host-centric model where a host device offloads computation to one or more target devices with their own local storage.

The OpenMP accelerator model defines a thread hierarchy where *OpenMP threads* (equivalent to *CUDA threads*), are wrapped into *teams* (equivalent to CUDA *thread blocks*), which are in turn wrapped into *leagues* (equivalent to CUDA *grids*). The parallel for construct can be used to exploit the former, while the teams and distribute constructs are used to exploit the two latter.

```
/* main.c */
int matrix_mul(float A[N][N], float B[N][N], float C[N][N]) {
   #pragma omp target device(0) map(to:A[0:N*N], B[0:N*N])
                           map(from:C[0:N*N]
   #pragma omp teams distribute parallel for private(i,j,k)
   for (i=0; i<n; i++)
      for (j=0; j<n; j++)
         for (k=0; k<n; k++)
            C[i][j] += A[i][k]*B[k][j];
}

int main() {
   …
   matrix_mul(A, B, C);
   …
}
```

Listing 3. OpenMP accelerator model matrix multiplication example.

Listing 3 shows an example of the computation of a matrix multiplication using the OpenMP accelerator model. The user code remains the same as for a sequential version of the benchmark. Just additional directives are inserted so the compiler knows how to do the transformation to exploit that computation in the accelerator.

4.4. OmpSs

OmpSs (BSC Programming Models, 2019) is a task-based parallel programming model built on top of a set of C/C++ and Fortran language directives and a runtime API. It aims at fast-prototyping and offers a simple yet complete set of directives and runtime options that allows covering both homogeneous and heterogeneous architectures without the need for changing the code.

Ompss defines a thread pool based execution model, meaning that the OmpSs application defines a pool of threads at the beginning of the program, while the application is initially executed just by one of them. Then, parallelism is distributed using tasks. Compared to OpenMP, it offers interesting features such as richer dependency clauses or the `implements` construct, which allows defining different implementations (e.g., C, CUDA and OpenCL) for the same kernel.

The memory model is very similar to that of OpenMP, although the rules to define the data-sharing attributes in OmpSs are slightly different (e.g., in OmpSs, the variables appearing in the dependency clauses are shared by default).

OmpSs supports heterogeneity with the `target` construct. The region inside includes the kernel to be executed, and the implementation shall match the type of device specified in the `device` clause (e.g., if the `device` clause receives the value *cuda*, then the kernel should be written in CUDA). This hybrid programming approach allows easily taking advantage of already existing CUDA/OpenCL kernels, while offering a good programmability to offload them and manage data.

Additionally, OmpSs also has support for FPGAs (Programming Models @BSC, 2020), by using the *fpga* value in the `device` clause. In this case, the kernel to be offloaded to the FPGA is transformed using the Accelerator Integration Tool (AIT) to generate FPGA bitstream. Listing 4 shows an example of the computation of a matrix multiplication using OmpSs and CUDA. There, the OmpSs model is used to define parallelism across CUDA kernels and handle the offloading, including data copies between the host and device memories.

```
/* kernel.cl */
#pragma omp target device(cuda) ndrange(2,N,N,16,16) copy_deps
#pragma omp task inout([N*N]C) in([N*N]A,[N*N]B)
__global__ void matrix_mul(float A[N][N], float B[N][N], float C[N][N],
int wA, int wB)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    float th_value = 0;
    for (int k = 0; k < wA; ++k)
       th_value += A[row * wA + k] * B[k * wB + col];

    C[row * wA + col] = th_value;
}

/* main.c */
int main() {
   …
   matrix_mul(A, B, C);
   …
}
```

Listing 4. OmpSs+GPU matrix multiplication example.

4.5. Summary: Programming models productivity

The programming models used within the AMPERE project must facilitate the expression of the required levels of concurrency to exploit all hardware resources in the underlying heterogeneous architecture. Table 2 shows a comparison in terms of forms of parallelism and architecture abstraction features available in the parallel programming models just presented. Further, Table 3 compares the models based on synchronization, mutual exclusion, language binding, error handling and tool support. Overall, OpenMP and OmpSs provide the most comprehensive set of features to support a wide range of parallelism patterns, synchronizations and architectures, on both the host and the device, by allowing modelling the memory hierarchy. Additionally, OpenMP offers two advantages over the rest of programming models. First, it defines an

emerging error model that includes features for cancelling parallel execution, i.e., aborts an OpenMP region and causes executing tasks to proceed to the end of the canceled region. Proposals to extend this model with further extend this model with support for call-backs, and other resiliency mechanisms already exist (Wong, y otros, 2010). Second, it allows binding the computation with the data by defining a binding policy (i.e., *master*, to assign all threads to the same place[2] as the master thread; *close*, to assign threads close to its parent; and spread, to create a sparse distribution of the threads of a team among the set of places of the parent's place partition[3]) to a parallel region.

Table 2. Comparison of the presented parallel programming models based on parallelism patterns and architecture abstraction (extended from (Yan, Chapman, & Wong, 2015)).

	Parallelism			Architecture abstraction		
Parallel Programming Model	**Data parallelism**	**Asynchronous task parallelism**	**Host/device**	**Abstraction of memory hierarchy**	**Data and computation binding**	**Explicit data mapping host/device**
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 3. Comparison of the presented parallel programming models based on synchronizations, mutual exclusions, language binding, error handing and tool support (extended from (Yan, Chapman, & Wong, 2015)).

	Synchronizations						
Parallel Programming Model	**Barrier**	**Reduction**	**Join**	**Mutual exclusion**	**Language library**	**Error handling**	**Tool support**
OpenMP	barrier	reduction	taskwait	Locks, critical, atomic, single, master	C/C++ and Fortran based directives	cancel	OMPT interface/ Extrae (BSC Performance Tools, 2020)
OmpSs	-	reduction	taskwait	critical, atomic	C/C++ and Fortran based directives	-	Extrae (BSC Performance Tools, 2020)

[2] In OpenMP, a *place* is an unordered set of processors on a device.

[3] In OpenMP, a *place partition* describes the places currently available to the execution environment for a given parallel región.

CUDA	_syncthreads	-	-	atomic	C/C++ extensions	-	NVIDIA profiling tools
OpenCL	work_group barrier	work_group reduction	-	atomic	C/C++ extensions	exceptions	System/vendor tools

Nonetheless, for the exploitation of the accelerator devices, OpenMP might not provide the best performance compare to dedicated languages such as CUDA for NVIDIA GPUs. In these cases the programmability might be harmed by the difficulty of using CUDA. In this regard, there is a proposal (Yu, Royuela, & Quiñones, 2020) that use the OpenMP programming language to define workflows that can later be transformed into CUDA graphs for enhanced performance opportunities. In a similar line, the OmpSs programming model also runs on GPUs and FPGAs; for the former, it uses kernels written with OpenCL and CUDA, and for the latter it uses high-level OmpSs C/C++ and compiler transformations (Filgueras, y otros, 2014) for lowering the code to the target device.

Overall, OpenMP is very suitable for parallelizing applications thanks to the features it includes to fine-tune the parallelization process, as well as its support for both host and accelerator execution. OpenMP offers great programmability because it is based on compiler directives that can incrementally be inserted in the sequential source code to achieve better performance. Moreover, OpenMP include a nice interoperability with CUDA, OpenCL and FPGA kernels, and has several mechanisms to control the runtime behavior.

5. Preliminary analysis of DSML transformation: from AMALTHEA to OpenMP

In order to make an efficient and effective use of the parallel programming models from DSMLs transformations, it is of paramount importance that the two representations are compatible. This section provides a preliminary analysis of the compatibility of OpenMP and AMALTHEA, from three different angles: (1) the base language, (2) the execution model and (3) the non-functional information that both models can define. The same analysis is currently being conducted with CAPELLA and OpenMP. This analysis, not included in this deliverable, is following the same reasoning as the one presented here.

5.1. Base Language

In the automotive and railway domains, C and C++ are the most widely used languages. The Motor Industry Software Reliability Association has even developed specific guidelines, MISRA C (Hatton, 2007) and MISRA C++ (Motor Industry Software Reliability Association and others, 2008), meant to promote safety best practices for automotive software, are accepted worldwide for developing safety-critical software in C and C++. Additionally, the AUTOSAR C++ Coding Guidelines (AUTOSAR, 2017) have been created by AUTOSAR to support the development of adaptive platform components that must complain with the stringent functional safety requirements of ISO 26262 (ISO, 2011) using modern C++. All considered PPM (OpenMP, OmpSs, CUDA and OpenCL) are built on top of C and C++.

5.2. Execution Model

To illustrate the compatibility of the current capabilities of the AMALTHEA model to describe parallelism with OpenMP, we consider the Cholesky decomposition benchmark shown in Listing 5. *Cholesky* is composed of multiple invocations to four different kernels (*portf_tile*, *trsm_tile, gemm_tile*, and *syrk_tile*) inside different loops controlling the flow, so the number of invocations of each kernel depends on the number loop iterations. This listing also includes the OpenMP directives for the parallel execution in italics, showing the data dependencies existing between kernels (through the **depend** clause) and the offloading of the *portf_tile* kernel to a GPU (through the **target** clause).

Figure 4 shows the parallelism exposed by the Cholesky benchmark in the form of the Task Dependency Graph (TDG) extracted from the **depend** clauses. The orange box represents the *cholesky* and the inner nodes represent the different invocations to the kernels, i.e., *portf_tile*, *trsm_tile, gemm_tile*, and *syrk_tile.*

```
void cholesky(float *A, int ts, int nt) {
  for (k = 0; k < nt; k++) {
    #pragma omp target map(to:A, from:A) depend(out:A[k][k]) device(GPU)
    potrf_tile(A[k*nt+k], ts, priority + ((nt-k)+10000));
    for (i = k + 1; i < nt; i++)
      #pragma omp task depend(in:A[k][k]) depend(out:A[k][i])
      trsm_tile(A[k*nt + k], A[k*nt + i], ts, priority + (nt-(i-k))+100);
    for (i = k + 1; i < nt; i++) {
      for (j = k + 1; j < i; j++)
        #pragma omp task depend(in:A[k][i], A[k][j]) depend(out:A[j][i])
        gemm_tile(A[k*nt + i], A[k*nt+j], A[j*nt+i], ts,
                  priority + (nt-(i-k)+10));
      #pragma omp task depend(in:A[k][i]) depend(out:A[i][i])
      syrk_tile(A[k*nt + i], A[i*nt + i], ts, priority + (nt-(i-k))+100);
    }
  }
}
```

Listing 5. Cholesky computation (in italics, the OpenMP directives for the parallel execution is shown).

In the AMALTHEA model, Cholesky can be described as follows:

- The *cholesky* function corresponds to an AMALTHEA task.
- The different kernel invocations inside the *cholesky* function correspond to different runnables. Since AMALTHEA does not allow including control flow within tasks, the different loops have to be unrolled.

AMALTHEA only allows describing the parallelism among tasks. Therefore, the program must change the system description and include the kernels invocations, i.e., the runnables, inside tasks, and the dependencies among the tasks shall be defined using the AMALTHEA event chains described in the constraints model (see Section 3.1.1). Furthermore, to offload the execution of *portf_tile*, this runnable has to be further divided into three different runnables: *host-to-gpu copy*, *gpu offloading*, and *gpu-to-host copy* (Wurst, y otros, 2019). Figure 5 shows how this behavior can be represented in the current AMALTHEA specification. Each kernel (runnable) has to be inserted inside a task to exploit parallelism, as well as the different runnables generated by splitting the kernel to be offloaded to the accelerator.

Overall, the model currently defined in AMALTHEA requires using the task as an abstract container in order to exploit parallelism. Additionally, the offloading of tasks to accelerator devices is also modeled by splitting the runnable into three tasks (2 for copies and 1 for actual computation). These requirements of the representation forces designers to shape their system focusing on how the functionalities have to be parallelized, rather than what functionalities can be parallel.

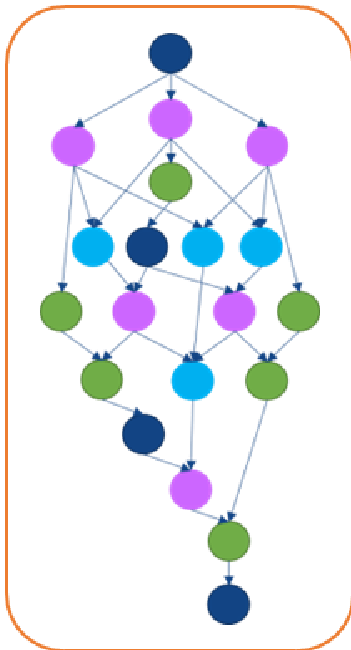

Figure 4. Task Dependency graph of the Cholesky benchmark in ***Error! Reference source not found.*** *.*

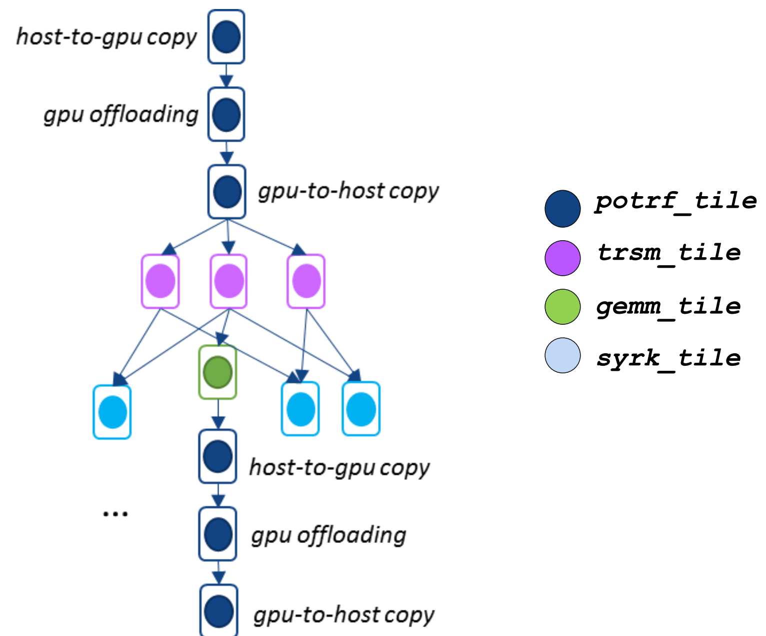

Figure 5. AMALTHEA model representing the TDG in Figure 4.

The objective of AMPERE is to incorporate model transformation techniques to automatically generate code capable of efficiently manage the parallel execution of AMALTHEA applications at different granularity levels, i.e., among runnables and within a runnable. This is indeed, a possible enhancement of the AMALTHEA model, allowing parallelism within tasks, and also between runnables of the same task. Interestingly, AMALTHEA already supports a data-model that allows expressing the input and output data of the runnable. In that regard, there are similarities between runnables and the OpenMP tasks that AMPERE aims to explore. Moreover, OpenMP tasks can implement CUDA kernels that are offloaded to the GPU.

It is of paramount importance that the DSML is able to describe the execution model of the PPM. In that regard, the scheduling model supported by OpenMP is compatible with those supported by AMALTHEA or AUTOSAR. Despite AMALTHEA and AUTOSAR are agnostic of the underlying scheduler, it is very common the use of limited preemption schedulers due to their good schedulability and time predictability properties. Interestingly, OpenMP define task-based models with a limited preemptive execution model based on **task scheduling points** (TSPs), i.e., *a point during the execution of a task region at which it can be suspended to be resumed later; or the point of task completion, after which the executing thread may switch to a different task region*. Furthermore, OpenMP allows defining priority-driven schedulers to ensure that task-

critically is properly handled (Serrano, Royuela, & Quiñones, Towards an OpenMP specification for critical real-time systems, 2018).

Table 4 summarizes the similarities of the abstractions provided by AMALTHEA and OpenMP described above, with the objective of exploiting the parallel opportunities exposed by AMALTHEA software.

Table 4. Matching components between the AMALTHEA and the OpenMP models.

AMALTHEA	OpenMP
Task	OpenMP program
Runnable	`task` construct
Runnable offloaded to an accelerator device (e.g., FPGA, GPU)	`target` construct
Runnable sequencing constraints	`depend` clause (associated to the `task` and `target` constructs)
Preemption strategy supported: Non-preemption, limited-preemption, fully preemption	Preemption strategy supported: Non-preemption, limited-preemption

5.3. Support for functional and non-functional requirements

OpenMP originally targets HPC systems, being its main focus to expose features for exploiting performance. There are however several works that push the introduction of OpenMP into other domains such as high performance real-time embedded systems. For such a purpose, the OpenMP has to be adapted to meet functional safety and time-predictability. Several features and techniques have been proposed targeting these aspects. Following paragraphs describe these proposals.

5.3.1. Functional safety and correctness

The *functional safety* of the OpenMP specification has been analyzed (Royuela, Duran, Serrano, Quiñones, & Martorell, 2017). This work shows that, although certain features might jeopardize the analyzability of the system, minimal limitations on the available features together with the use of two new directives for enabling full-system analysis even in the existence of third-party libraries, may cover most of the possible sources of non-determinism introduced by the specification.

Several *correctness* techniques aiming at delivering fault-free OpenMP systems have been developed. These mainly target dead-locks and data-races. Regarding the former, there are techniques that apply to different programming models, like Sherlock (Eslamimehr & Palsberg, 2014) and Chord (Naik, Park, Sen, & Gay, 2009), targeting effectiveness. More interestingly, there is a sound technique for detecting dead-locks in C/Phtreads programs (Kroening, Poetzl, Schrammel, & Wachter, 2016) that can be easily applied to OpenMP. Regarding the latter, there are techniques that retrieve data races in specific subsets of OpenMP, like a fixed number of threads (Ma, y otros, Symbolic analysis of concurrency errors in OpenMP programs, 2013), or using affine constructs (Basupalli, y otros, 2011). More general approaches also exist, providing

no false negatives (Lin, 2005), or at least providing one race when races are present (Banerjee, Bliss, Ma, & Petersen, 2006).

Programmability (and productivity) has also been largely tackled in OpenMP. In this regard, different works consider the use of compiler-analysis techniques for relieving the user from defining certain information that can be automatically derived (Royuela, Duran, Liao, & Quinlan, 2012) (Royuela S. a., 2012) (Ma, y otros, Symbolic analysis of concurrency errors in OpenMP programs, 2013). This works enhance not only the programmability of the model, but also the correctness expectations, because of two reasons: (1) they automatize certain tasks avoid possible human errors, and (2) they include compiler analysis techniques that can be used to check the correctness of the system based on the users definitions.

Resiliency is also an aspect that has been considered in OpenMP. In order to enhance the reliability of the framework, different proposals for including an error model in the specification have been provided (Duran, y otros, 2007) (Wong, y otros, 2010). These aim at providing programmers with the tools for recovering the system at certain points where the parallel execution might fail. In this regard, the OpenMP specification includes one mechanism targeting resilience (and also performance), which is *cancellation*. Two directives allow defining points at which the parallel execution can be resumed and the regions to be cancelled (e.g., a parallel region or a taskgroup region).

5.3.2. Time predictability

Although OpenMP has not been designed for providing timing guarantees, previous works have tackled this aspect. The mainly focus on the OpenMP tasking model because the TDG resembles the Direct Acyclic Graph (DAG) scheduling model used in real-time systems for verifying the timing constraints of the tasks, and tackle both tied (Sun, Guan, Wang, He, & Yi, 2017) and untied tasks (Serrano, y otros, 2015).

OpenMP lacks however the concept of time. Based on the previous works, different extensions to the OpenMP specification regarding tasks have been proposed (Serrano, Royuela, & Quiñones, Towards an OpenMP specification for critical real-time systems, 2018) in order to consider time:

- *Recurrency*: in real-time systems, tasks are either periodic or sporadic triggered by an event. In this sense, a new clause, named `event`, containing the event that triggers a task has been proposed.
- *Deadlines*: the criticality of a task can be related to the point in time at which the task has to be finished. Several schedulers, like earliest deadline first (EDF) and least laxity (LL), use this information to prioritize the tasks. A new clause, named `deadline`, containing the expression that determines the time instant at which the task must finish has been proposed.
- *Time management* in the runtime: the control loop used in real-time systems to trigger tasks has to be implemented in the OpenMP runtime. An extension derived from this is the concept of *persistent task* (Pop & Cohen, 2011).

The scheduling decisions are paramount for the time predictability of the system. In this sense, the use of work-conserving schedulers is paramount to avoid incorrect or too pessimistic timing analysis (Serrano, y otros, 2015) (Sun, Guan, Wang, He, & Yi, 2017). Work conserving policies can be ensured within OpenMP teams, but there is a limitation when different parallel regions can run in parallel: the specification states that the number of threads in an OpenMP team cannot

vary during the life of the parallel region to which it was associated. In this regard, there is a proposal that considers the cooperation between different OpenMP teams (different OpenMP parallel regions) in order to avoid idle cycles in threads from one team when there is work to do in a different team, but works at an OS-thread level and is limited but the aforementioned limitation. The thread pool based execution model defined by OmpSs does not have this limitation because the scheduler has flat access to all executing threads.

Amalthea addresses many concepts mentioned above. Recurrency is supported via the stimuli model. Stimuli are responsible to activate processes, and can define different recurrency patterns: single, periodic, variable rate, and event related, among others. Additionally, a task can include several attributes to define its timing constraints and aspects related to the scheduling approach like a priority, a preemption strategy and a deadline. Finally, the OS model includes features to describe the scheduler including the particular algorithm to be used. In this regard, AMALTHEA recognizes several different scheduling algorithms like fixed priority (e.g., deadline monotonic, fixed priority preemptive, rate monotonic, etc.) and dynamic priority (e.g., earliest deadline first, priority based round robin, etc.) among others, and also allows user-defined algorithms (this information is a placeholder that needs to be implemented in the tools consuming the model, e.g., simulators). Schedulers can be composed in a hierarchy association, and tasks can be assigned to a specific scheduler.

Capella through the Tideal viewpoint also supports a large part of these concepts. The activation policy of a task is modeled by an event triggering the task. The supported activation policies include periodic and sporadic activations, but also burst (several activations in a short time, that will repeat after a while) or sliding window (no more than a certain number of activations in a sliding time window). A task is defined by a number of timing constraints: a priority, a deadline, a best- and worst-case execution time. Finally, a task is allocated to a processor with a given scheduling policy among fixed priority, rate monotonic, earlier deadline first, first in-first out, round robin, etc. All these constraints can be used for schedulability analysis and simulation by external verification tools, thanks to the Time4Sys platform.

5.3.3. Energy

Power and so energy management is also an aspect that has been considered in OpenMP, and the importance of power management has already been noted (Chapman, y otros, 2009). There is a proposal for extending the OpenMP specification in order to allow addressing the issue of energy consumption and power management (Alessi, Thoman, Georgakoudis, Fahringer, & Nikolopoulos, 2015). This work provides also the compiler and runtime systems that fulfill these constraints. The extensions proposed include multi-objective optimization goals with a clause that allow providing the goals in terms of execution time, power, energy and quality of service. A different proposal tackles the energy consumption from a cost-per-operation point of view, and defines extensions to model to allow defining the accuracy of the floating point operations (Rahimi, Marongiu, Gupta, & Benini, 2013).

Amalthea allows modeling the power and frequency of the system in the hardware model, but offers no option for defining these non-functional requirements at task level.

Capella offers the capabilities to model some of these power consumption aspects. As of today, there is no tool that can be used to analyze this data and provide feedbacks to the developer.

6. State-of-the-art synthesis tools

Synthesis methods are widely adopted techniques in multiple computing domains, including HPC, artificial intelligence and embedded computing because of the benefits they provide: (1) they allow non-computer experts (e.g., physicists) to effectively use diverse computing resources to solve complex problems while hiding low-level details of the system, and (2) they facilitate the verification of the system. Next we describe the state-of-the-art synthesis frameworks available in the mentioned domains.

<table>
<tr><td rowspan="2">HPC</td><td>Data Flow Language (DFL) (Fernández, Beltran, Mateo, Patejko, & Ayguadé, 2014)</td><td>DFL is a framework for the design and implementation of DSMLs for distributed heterogeneous HPC systems.
It is composed of (1) a DSML that abstracts the concepts needed to implement efficient HPC applications, and (2) a code synthesis mechanism based on Lightweight Modular Staging (LMS) (Rompf & Odersky, 2010) that transforms the DSML into an OmpSs (Alejandro, y otros, 2011) program, a parallel programming model designed at BSC.
The concepts included in DFL are: (1) buffers, abstracting the concept of data, (2) tasks and kernels, representing computations written in C++ and OpenCL, and (3) high-level operations, such as map, reduce, divide and conquer, used to exploit distributed systems without exposing low-level details.
DFL features a data-flow design matching that defined by OmpSs. Additionally, it includes mechanisms to reuse C/C++ libraries (e.g., FFT (Frigo & Johnson, 1998) or VTK (Sima, 1996)) to enhance the productivity of the system and allow compiler use already existing libraries.</td></tr>
<tr><td>Delite Compiler Framework and Runtime (Brown, y otros, 2011)</td><td>Delite is an end-to-end system for building, compiling and executing DSL application on parallel heterogeneous hardware based on LMS.
The framework (1) lifts embedded DSL applications to an intermediate representation (IR), (2) performs generic, parallel, and domain-specific optimizations, and (3) generates an execution graph along with multiple kernel variants that target multiple heterogeneous hardware devices to achieve performance portability. The supported languages are C++, CUDA and Scala (Odersky & Spoon, 2010), the latter supporting transformation to OpenCL (Passerat-Palmbach, Reuillon, Mazel, & Hill, 2013).</td></tr>
<tr><td>IA</td><td>Distributed Multiloop Language (DMLL) (Brown, y otros, 2016)</td><td>DMLL is an intermediate language based on common data-parallel patterns that captures the necessary semantic knowledge to efficiently target distributed heterogeneous architecture.
The language models high-level data-parallel patterns as multiloops, a loop abstraction that captures the high-level structure of the loop and its outputs. It also provides mechanisms</td></tr>
</table>

		to efficiently distributing the computation by partitioning data. This language is implemented on top of the Delite framework, and hence reuses its heterogeneous code generators for C++, CUDA and Scala, and the compiler optimizations like code motion and common subexpression elimination.
	Halide compiler (Ragan-Kelley, y otros, 2013)	The Halide optimizing compiler synthesizes high performance implementations using the Halide open-source DSML for complex image processing pipelines and vision applications. The compiler lowers a functional representation of an imaging pipeline to imperative code. It does so by applying a series of transformations, including flattening, vectorization and unrolling, and then generates code via LLVM (The LLVM Compiler Infrastructure, 2020). The code generator produces parallel vector code for x86 and ARM CPUs with SSE/AVX and NEON, and graphs of CUDA kernels for hybrid CPU-GPU execution, and so targets data-parallel models.
	Rewriting rules (Steuwer, Fensch, Lindley, & Dubach, 2015)	This is an approach for the transformation of high-level functional expressions to high-performance OpenCL Code. The framework receives high-level algorithmic primitives representing a program (local/global, to indicate where to store the results of a given function; vectorization, for exploiting SIMD instructions, etc.) and automatically generates low-level hardware primitives using rewrite rules (e.g. reduce rules, for reductions, cancellation rules to eliminate operations equivalent to the identity, etc.). The framework targets code portability and high-performance, and uses an OpenCL code generator to demonstrate its capabilities.
	NOVA (Collins, Grewe, Grover, Lee, & Susnea, 2014)	NOVA is a polymorphic functional language, a compiler for CPUs and GPUs and a multi-core runtime. The language includes support for nested parallelism, recursion and type polymorphism, and offers high-level operations including *map*, *reduce* and *scan*. The compiler includes different optimizations that allow generating code for a variety of target platforms, synthesizing sequential C, parallel C and CUDA codes. The multi-core runtime for parallel C is a straightforward implementation that creates a number of threads and assigns an equal share of the input to the process.
	Compact Components (CoCo)	The CoCo framework, from the SSSA AMPERE partner, is a component based multicore system designed targeting the creation of visuo-haptics applications.

	(Ruffaldi & Brizzi, 2016)	CoCo defines different components (i.e., callbacks, input and data ports, declarative attributes and operators). The execution of components is scheduled using periodic (with fixed rate) or sporadic (based on events) approaches. CoCo is integrated with Robot Operating System (ROS), a de-facto standard for robotics that includes drivers, algorithms, and developer tools.
Embedded computing	**Matlab and Simulink** (MathWorks, 2020)	Matlab is a tool for analyzing data, developing algorithms and creating mathematical models based on a programming language that expresses matrix and array mathematics directly. Simulink is a tool for running simulations, generating sequential C/C++ code and register-transfer level (RTL) code to be executed on an FPGA, and testing and verifying embedded systems. Used in the automotive domain.
	Gedae (The Gedae Development Environment, 2020)	Gedae is a development environment that includes the Idea Text Language and Compiler. The Idea language combines data-flow language abstractions, high level algebra and math similar to Malab, and control similar to UML. An additional architectural modeling language used to create the hardware model of the architecture. The Idea compiler targets efficiency and scalability, as well as portability and correctness by implementing several optimizations. It uses partitioning and mapping techniques based on a flow graph representation of the application.
	MPSoC Application Programming Studio (MAPS) (Ceng, y otros, 2008)	MAPS is an integrated framework for the user-directed parallelization of C applications for MPSoCs. The parallelization process is done in three steps: analysis, partitioning and code emission. The first step considers sequential C code and a description of the target platform. Then, the code is profiled in order to extract information about possible parallel tasks. The approach is orthogonal to the programming model. Instead. Instead, the framework has been integrated with the TCT framework (Urfianto, Isshiki, Khan, Li, & Kunieda, 2008), which uses the Tightly-Coupled-Thread (TCT) programming model (Isshiki, Urfianto, Kahn, Li, & Kunieda, 2006).
	ASCET (ETAS, 2020)	ETAS ASCET-DEVELOPER is a tool for developing applications for embedded systems. It includes graphical models, like the block diagram and state machine editors, and textual programming annotations, like the Embedded Software Development Language (ESDL) and C-code editors. The framework provides a Code Generator that translates function models into highly efficient and safe embedded C-code (ISO26262 and IEC61508 TÜV-certified) for AUTOSAR and non-AUTOSAR applications.

		Used in the automotive domain.
	DaVinci suite (Vector, 2020)	DaVinci is a tool for designing the architecture of software components (SW-C) for AUTOSAR ECUs. The tool allows creating interfaces, define the internal behavior with runnable entities and link SW-C to one another. It provides special functions for automatically generate data mapping between SW-C, as well as the analysis of communication relationships. It also includes a Contract-Phase Generation tool that allows generating header files and implementation templates for C-based applications. Used in the automotive domain.
	SCADE suite (Ansys, 2020)	The SCADE suite is a model-based development environment used to design critical software. It provides several capabilities, including: (1) model-base design for data-flow and state machine design, (2) model analysis to assess safety requirements, (3) debugging and simulation to examine variables and build full-system prototypes, and (4) automatic code generators for C and Ada qualified to the highest level of safety across different domains including automotive and rail transportation applications. Used in the railway domain.
	MagicDraw (NoMagic, 2018)	MagicDraw is a process, architecture, software and system modeling tool to facilitate the analysis and design of object oriented (OO) system with support for Java, C++, C#, CL (MSIL) and CORBA IDL programming languages. Used in the railway domain.
	IBM Engineering Systems Design Rhapsody (Rational Rapsody) (Gery, Harel, & Palachi, 2002)	Rhapsody is a solution for modeling and system design. It is integrated with the IBM Engineering portfolio, offering a design and test environment that supports UML, SysML, UAF and AUTORSAR. The key technologies offered with Rhapsody are: (1) model-code associativity, to leverage the benefits of DMSL without losing access to the implementation; (2) automated implementation generation, with support for C, C++, Java, COM and CORBA; (3) execution framework, providing APIs for the different languages, used to perform manipulations at model abstraction level; (4) model execution, enabling a runtime model that allows tracing and controlling execution at a high abstraction level; and (5) model-based testing, used for testing and failure detection. Used in the railway domain.
	APP4MC (Eclipse Foundation, Inc, 2020)	On top of the Amalthea data model the APP4MC platforms offers a variety of tools like migration, visualization tools and framework supporting model-2-model and model-2-text transformations. The APP4MC transformation framework encapsulates various

		technologies: *Java*, as a programing language; Eclipse extension points, as a mechanism to load the configuration; *xtend2*, as a template definition language Templates, with Java like syntax and good support for lambda expressions, which are converted into java code which can be debugged during execution; and *Google Guice*, as a dependency injection mechanism to provide flexibility to hook customer templates (containing specific transformation rules) and override the definitions of platform templates. It provides more flexibility for the developers to develop their application e.g. built in mechanism for caching objects, clear separation w.r.t. configuration and templates) and separate the transformation templates as platform and customer templates. This framework acts like a wrapper around model transformation technologies *(like Xtend2, ATXL)* and provides the complete infrastructure for easily specifying meta-models, hooking loaders for the models, caching mechanism, defining transformation code, building update sites or command line products and testing of the transformation code. Within Bosch, the framework is used to generate synthetic code that imitates time and memory accesses described in AMALTHEA. It also generates the necessary code and configuration stubs to hook the synthetic code to communicate with and activate existing AUTOSAR Adaptive applications. Within AMPERE, we will extend this framework towards generating ROS2 nodes based on an Amalthea model.

7. Summary and Conclusions

This deliverable analyzes different DSML and parallel programming models, considering their capabilities to fulfill functional and non-functional requirements, as well as the mapping possibilities between the AMALTHEA DSML and the OpenMP parallel programming model. The deliverable also covers the state-of-the-art code synthesis tools used in different computing domains, including HPC, AI and EC.

Moreover, this deliverable provides a preliminary analysis of the compatibility of OpenMP and AMALTHEA, from the base language, the execution model and the non-functional information perspective. Regarding AMALTHEA, the way parallelism is exposed might be cumbersome for defining the parallelism exposed by models like OpenMP. Regarding OpenMP, the language lacks features for defining non-functional requirements like energy consumption, ensuring reliability or specifying timing requirements. Fortunately, there are several proposals that already push to extend the language in this direction. Furthermore, there is a synergy between the representation of the system in AMALTHEA and OpenMP by means of the TDG. This is of paramount importance for the definition of the meta model driven abstraction that will communicate the two components.

8. Acronyms and Abbreviations

- AIT – Accelerator Integration Tool
- API – Application Program Interface
- AUTOSAR – AUTomotive Open System ARchitecture
- BSC – Barcelona Supercomputing Center
- CoCo – Compact Components
- CPU – Central Processing Unit
- DFL – Data Flow Language
- DMLL – Distributed Multiloop Language
- DSML – Domain Specific Modelling Language
- DSP – Digital Signal Processor
- EDF – Earliest Deadline First
- FPGA – Field-Programmable Gate Array
- GPU – Graphics Processing Unit
- HAD – Highly-Automated Driving
- HPC – High-Performance Computing
- IRV Internal Variable
- LL – Least Laxity
- MAPS – MPSoC Application Programming Studio
- MBSE – Model-Based Systems Engineering
- MDE – Model Driven Engineering
- MS – Milestone
- OO – Object Oriented
- OS – Operating System
- PPM – Parallel Programming Model
- RTE – Run-time environment
- SM – Stream Multiprocessor
- SW-C – Software-Component
- TDG – Task Dependency Graph
- TSP – Task Scheduling Point
- VFB – Virtual Function Bus

9. References

European Comission and AMPERE beneficiaries. (2019). *Grant Agreement Description of Action.*

Yan, Y., Chapman, B. M., & Wong, M. (2015). *A Comparison of Heterogeneous and Manycore Programming Models*. Recuperado el June de 2020, de https://www.hpcwire.com/2015/03/02/a-comparison-of-heterogeneous-and-manycore-programming-models/

Yu, C., Royuela, S., & Quiñones, E. (2020). OpenMP to CUDA graphs: a compiler-based transformation to enhance the programmability of NVIDIA devices. *23rd International Workshop on Software and Compilers for Embedded Systems.* Sankt Goar, Germany.

OpenMP ARB. (Nov de 2018). *OpenMP 5.0 Specification*. Recuperado el June de 2020, de https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

Khronos OpenCL working group. (Apr de 2020). *The OpenCL™ Specification v3.0.1-provisional*. Recuperado el June de 2020, de https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf

NVIDIA®. (June de 2020). *CUDA C++ Programming Guide*. Recuperado el June de 2020, de https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf

BSC Programming Models. (Oct de 2019). *OmpSs Specification*. Recuperado el June de 2020, de https://pm.bsc.es/ftp/ompss/doc/spec/OmpSsSpecification.pdf

BOSCH. (June de 2020). *AMALTHEA*. Recuperado el June de 2020, de http://www.amalthea-project.org/

Eclipse Foundation, Inc. (May de 2020). *Eclipse APP4MC*. Recuperado el June de 2020, de https://www.eclipse.org/app4mc/

Hatton, L. (2007). Language subsetting in an industrial context: A comparison of MISRA C 1998 and MISRA C 2004. *Information and Software Technology, 49*(5), 475--482.

AMPERE. (2020). *D1.1. System models requirements and use case selection.*

BSC Performance Tools. (June de 2020). *Extrae Documentation release 3.8.0.* Recuperado el June de 2020, de https://tools.bsc.es/doc/pdf/extrae.pdf

Filgueras, A., Gil, E., Jimenez-Gonzalez, D., Alvarez, C., Martorell, X., Langer, J., . . . Vissers, K. (2014). OmpSs@ Zynq all-programmable SoC ecosystem. *ACM/SIGDA international symposium on Field-programmable gate arrays.*

Motor Industry Software Reliability Association and others. (June de 2008). *MISRA C++: 2008: guidelines for the use of the C++ language in critical systems.* Recuperado el June de 2020, de http://tlemp.com/download/rule/MISRA-CPP-2008-STANDARD.pdf

AUTOSAR. (March de 2017). *Guidelines for the use of theC++14 language in critical andsafety-related systems.* Recuperado el June de 2020, de https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_RS_CPP14Guidelines.pdf

ISO. (2011). 26262-1: 2011 Road Vehicles--Functional Safety--Part 1: Vocabulary. *Berlin: ISO*.

AMPERE. (2020). *D5.1. Reference parallel heterogeneous hardware selection.*

Fürst, S., Mössinger, J., Bunzel, S., Weber, T., Kirschke-Bille, F., Heitkämper, P., . . . Lange, K. (2009). AUTOSAR-- A Worldwide Standard is on the Road. *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden.*

Sailer, A. (2014). *Timing Simulation of Multi-Core Systems.* Recuperado el June de 2020, de https://www.timing-architects.com/fileadmin/user_upload/knowledge/autosar-timing-simulation.pdf

Peraldi-Frati, M.-A., Blom, H., Karlsson, D., & Kuntz, S. (2012). Timing modeling with autosar-current state and future directions. *Design, Automation \& Test in Europe Conference \& Exhibition (DATE).*

Wurst, F., Dasari, D., Hamann, A., Ziegenbein, D., Sañudo, I., Capodieci, N., . . . Burgio, P. (2019). System Performance Modelling of Heterogeneous HW Platforms: An Automated Driving Case Study. *22nd Euromicro Conference on Digital System Design (DSD).*

Serrano, M. A., Royuela, S., & Quiñones, E. (2018). Towards an OpenMP specification for critical real-time systems. *International Workshop on OpenMP.*

Fernández, A., Beltran, V., Mateo, S., Patejko, T., & Ayguadé, E. (2014). A Data Flow Language to Develop High Performance Computing DSLs. *Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing.*

Rompf, T., & Odersky, M. (2010). Lightweight modular staging: A pragmaticapproach to runtime code generation and compiled DSLs. *Proceedings of the ninth international conference on Generative programmingand component engineering, GPCE.*

Alejandro, D., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., & Planas, J. (2011). OmpSs: A Proposal for Programming Het-erogeneous Multi-Core Architectures. *Parallel Processing Letters, 21*(2), 173--193.

Sima, V. (1996). Algorithms and lapack-based software for subspace identifica-tion. *Proceedings of the IEEE International Symposium on Computer-Aided Control System Design.*

Frigo, M., & Johnson, S. G. (1998). FFTW: An adaptive software architecture for the FFT. *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing.*

Brown, K. J., Sujeeth, A. K., Lee, H., Rompf, T., Chafi, H., Odersky, M., & Olukotun, K. (2011). A heterogeneous parallel framework for domain-specific languages. *International Conference on Parallel Architectures and Compilation Techniques.*

Odersky, & Spoon. (2010). *Programming in Scala, Second Edition.* Artima.

Passerat-Palmbach, J., Reuillon, R., Mazel, C., & Hill, D. R. (2013). Prototyping parallel simulations on manycore architectures using Scala: A case study. *International Conference on High Performance Computing \& Simulation (HPCS).*

Brown, K. J., Lee, H., Romp, T., Sujeeth, A. K., De Sa, C., Aberger, C., & Olukotun, K. (2016). Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns. *IEEE/ACM International Symposium on Code Generation and Optimization (CGO).*

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., & Amarasinghe, S. (2013). Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices, 48*(6), 519--530.

The LLVM Compiler Infrastructure. (June de 2020). Recuperado el June de 2020, de llvm.org

Ruffaldi, E., & Brizzi, F. (2016). Coco - A framework for multicore visuo-haptics in mixed reality. *International Conference on Augmented Reality, Virtual Reality and Computer Graphics.*

Steuwer, M., Fensch, C., Lindley, S., & Dubach, C. (2015). Generating performance portable code using rewrite rules: From high-level functional expressions to high-. *ACM SIGPLAN Notices, 50*(9), 205--217.

Collins, A., Grewe, D., Grover, V., Lee, S., & Susnea, A. (2014). NOVA: A functional language for data parallelism. *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming.*

MathWorks. (June de 2020). *MATLAB & Simulink*. Recuperado el June de 2020, de www.mathworks.com

The Gedae Development Environment. (June de 2020). Recuperado el June de 2020, de www.gedae.com

Ceng, J., Castrillón, J., Sheng, W., Scharwächter, H., Leupers, R., Ascheid, G., . . . Kunieda, H. (2008). MAPS: an integrated framework for MPSoC application parallelization. *Proceedings of the 45th annual Design Automation Conference.*

ETAS. (June de 2020). *ASCET-DEVELOPER - Software Products & Systems*. Recuperado el June de 2020, de https://www.etas.com/en/products/ascet-developer.php

Vector. (June de 2020). *DaVinci Developer*. Recuperado el June de 2020, de https://www.vector.com/int/en/products/products-a-z/software/davinci-developer/

Ansys. (June de 2020). *Scade*. Recuperado el June de 2020, de https://www.ansys.com/products/embedded-software

Royuela, S., Duran, A., Serrano, M. A., Quiñones, E., & Martorell, X. (2017). A Functional Safety OpenMP* for Critical Real-Time Embedded Systems. *International Workshop on OpenMP.*

Royuela, S., Duran, A., Liao, C., & Quinlan, D. J. (2012). Auto-scoping for OpenMP tasks. *International Workshop on OpenMP.*

Royuela, S. a. (2012). Compiler automatic discovery of OmpSs task dependencies. *International Workshop on Languages and Compilers for Parallel Computing.*

Ma, H., Diersen, S. R., Wang, L., Liao, C., Quinlan, D., & Yang, Z. (2013). Symbolic analysis of concurrency errors in OpenMP programs. *42nd International Conference on Parallel Processing.*

Wong, M., Klemm, M., Duran, A., Mattson, T., Haab, G., de Supinski, B. R., & Churbanov, A. (2010). Towards an error model for OpenMP. *International Workshop on OpenMP.*

Duran, A., Ferrer, R., Costa, J. J., Gonzàlez, M., Martorell, X., Ayguadé, E., & Labarta, J. (2007). A proposal for error handling in OpenMP. *International Journal of Parallel Programming, 35*(4), 393--416.

Kroening, D., Poetzl, D., Schrammel, P., & Wachter, B. (2016). Sound static deadlock analysis for C/Pthreads. *31st IEEE/ACM International Conference on Automated Software Engineering.*

Eslamimehr, M., & Palsberg, J. (2014). Sherlock: scalable deadlock detection for concurrent programs. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.*

Naik, M., Park, C.-S., Sen, K., & Gay, D. (2009). Effective static deadlock detection. *IEEE 31st International Conference on Software Engineering.*

Ma, H., Diersen, S. R., Wang, L., Liao, C., Quinlan, D., & Yang, Z. (2013). Symbolic analysis of concurrency errors in OpenMP programs. *42nd International Conference on Parallel Processing.*

Lin, Y. (2005). Static nonconcurrency analysis of OpenMP programs. *International Workshop on OpenMP.*

Banerjee, U., Bliss, B., Ma, Z., & Petersen, P. (2006). A theory of data race detection. *Workshop on Parallel and distributed systems: testing and debugging.*

Basupalli, V., Yuki, T., Rajopadhye, S., Morvan, A., Derrien, S., Quinton, P., & Wonnacott, D. (2011). ompVerify: polyhedral analysis for the OpenMP programmer. *International Workshop on OpenMP.*

Sun, J., Guan, N., Wang, Y., He, Q., & Yi, W. (2017). Real-time scheduling and analysis of openmp task systems with tied tasks. *IEEE Real-Time Systems Symposium (RTSS).*

Serrano, M. A., Melani, A., Vargas, R., Marongiu, A., Bertogna, M., & Quinones, E. (2015). Timing characterization of OpenMP4 tasking model. *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES).*

Pop, A., & Cohen, A. (2011). A stream-computing extension to OpenMP. *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers.*

Alessi, F., Thoman, P., Georgakoudis, G., Fahringer, T., & Nikolopoulos, D. S. (2015). Application-level energy awareness for OpenMP. *International Workshop on OpenMP.*

Chapman, B., Huang, L., Biscondi, E., Stotzer, E., Shrivastava, A., & Gatherer, A. (2009). Implementing OpenMP on a high performance embedded multicore MPSoC. *IEEE International Symposium on Parallel \& Distributed Processing.*

Rahimi, A., Marongiu, A., Gupta, R. K., & Benini, L. (2013). A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters. *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS).*

Isshiki, T., Urfianto, M. Z., Kahn, A. U., Li, D., & Kunieda, H. (2006). Tightly coupled thread: A new design framework for multiprocessor system-on-chips. *Design Automation Symposium.*

Urfianto, M. Z., Isshiki, T., Khan, A. U., Li, D., & Kunieda, H. (2008). A multiprocessor SoC architecture with efficient communication infrastructure and advanced compiler support for easy application development. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences.*

Gery, E., Harel, D., & Palachi, E. (2002). Rhapsody: A complete life-cycle model-based development system. *International Conference on Integrated Formal Methods.*

NoMagic. (April de 2018). *MagicDraw*. Recuperado el June de 2020, de https://www.nomagic.com/products/magicdraw

Roques, P. (2017). *Systems Architecture Modeling with the Arcadia Method: A Practical Guide to Capella.* Elsevier.

Voirin, J.-L. (2017). *Model-based System and Architecture Engineering with the Arcadia Method.* ISTE Press - Elsevier.

Object Management Group. (April de 2019). *UML Profile for MARTE.* Recuperado el July de 2020, de https://www.omg.org/spec/MARTE/

Programming Models @BSC. (June de 2020). *OmpSs@FPGA*. Recuperado el June de 2020, de https://pm.bsc.es/ompss-at-fpga

Wen, Y., Wang, Z., & O'boyle, M. F. (2014). Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. *21st International conference on high performance computing (HiPC).*