Information and Communication Technologies (ICT) Programme

Project Nº: H2020-ICT-2016-1-732105

D5.1: CERBERO Holistic Methodology and Integration Interfaces (Final Version)

Lead Beneficiary:	AI
Work package:	WP5
Date:	28/08/2019
Distribution - Confidentiality:	[Public]

Abstract: This is an updated report on integration activities with emphasis on cross-layer and cross abstraction levels integration methodology as well as operational interfaces among tools. Here presented is a definition of CERBERO holistic design framework, the integration roadmap followed and the interfaces developed. This release of the deliverable is the final of three releases, of which the first D5.4 and D5.5 are already available.

Disclaimer

The CERBERO Consortium is the following:

Num.	Beneficiary name	Acronym	Country
1 (Coord.)	IBM Israel – Science and Technology LTD	IBM	IL
2	Università degli Studi di Sassari	UniSS	IT
3	Thales Alenia Space Espana, SA	TASE	ES
4	Università degli Studi di Cagliari	UniCA	IT
5	Institut National des Sciences Appliquees de Rennes	INSA	FR
6	Universidad Politecnica de Madrid	UPM	ES
7	Università della Svizzera italiana	USI	CH
8	Abinsula SRL	AI	IT
9	Ambisense LTD	AS	UK
10	Nederlandse Organisatie Voor Toegepast Natuurwetenschappelijk Ondeerzoek TNO	TNO	NL
11	Science and Technology	S&T	NL
12	Centro Ricerche FIAT	CRF	IT

For the CERBERO Consortium, please see the http://cerbero-h2020.eu web-site.

Document Authors

The following list of authors reflects the major contribution to the writing of the document.

Name(s)	**Organization Acronym**
Maria Katiuscia Zedda	AI
Giuseppe Meloni	AI
Michael Masin	IBM
Evgeny Shindin	IBM
Francesca Palumbo	UNISS
Karol Desnos	INSA
Julio Oliveira	TNO
Luca Pulina	UNISS
Claudio Rubattu	INSA/UNISS
Tiziana Fanni	UNICA
Daniel Madroñal	UPM
Alfonso Rodriguez	UPM
Leonardo Suriano	UPM
Rafael Zamacola	UPM

Document Revision History

Date	**Ver.**	**Contributor (Beneficiary)**	**Summary of main changes**
01/07/2019	V0.1	AI	First draft
08/07/2019	V0.2	UNICA, UNISS and UPM	Updated PAPIFY-MDC section, added Spider-PAPIFY-MDC section
09/07/2019	V0.2	AI and UPM	New structure with more direct connections included and connection with external tools
15/07/2019	V0.3	UPM	New sections

23/07/2019	V0.4	IBM	Refactoring of sections 4 and 5 related to CIF, new structure and content.
25/07/2019	V1.0	AI, UNISS	Review
26/07/2019	V1.1	UPM	Fix Section 6
29/07/2019	V1.2	AI	Refactoring of section 7

Table of Contents

1. Executive Summary

This is the final version of the integration activities and complete framework characteristics with emphasis on interfaces among tools.
After the first release, D5.4, that has defined CERBERO methodology, following the iterative development scheme of the project, during the Phase II the methodology has been deeply revised, according to possible technical requirements updates, and the verification and fine tuning of the integration process, these updates have been reported in the second version of the deliverable D5.5. Final reporting on integration activities and the complete framework characteristics and interfaces are included in the present deliverable D5.1. Please note that this non-chronological numbering of the deliverable is due to a recent amendment.

In order to speed up and ease the reading and review process the text of sections and paragraphs that have NOT been significantly updated and revised are in dark gray. Sections that have been deeply updated and revised are 4.2, 4.3, 4.4, 5 and 6. Furthermore, a new section 7 has been added.

1.1. Structure of Document

The document starts by an introduction to modeling of CPSs and vision of CERBERO integration. Then state of the art methodologies are reported. Then the report discusses CERBERO design framework integration approach with required interfaces between the myriad of tools across all layers of the toolchain (model, system of system design and analysis and computational level design and implementation). This last part is devoted to explaining the methodological approach related to the tool integration reporting also a number of successful integration endeavors already achieved by the consortium members. The last section summarized the lesson learned during the integration process, and the contribution of the present deliverable to achieve CERBERO operational objectives.

1.2. Related Documents

- D2.2: CERBERO Technical Requirement
  - The activities behind D5.5 contribute to satisfying the requirements 0011 and 0012 listed in D2.2.
- D5.4 – D5.5: CERBERO Holistic Methodology and Integration Interfaces
  - The proposed holistic methodology reported in the D5.4 and D5.5 has been the basis of the D5.1
- D5.2 – D5.2: CERBERO Framework Components
  - The deliverable reports the component/features of CERBERO design environment that have been integrated
- D5.7: CERBERO Framework Demo
  - After phase II, the verification of the proposed process has allowed identifying the correct methodology for the framework integration and to define the semantic integration approach based on CIF (CERBERO Interoperability Framework).

1.3. Related CERBERO Requirements

Deliverable D2.2 of the CERBERO project defines a list of CERBERO Technical Requirements (CTRs) the project should achieve. Each of them is referenced with a unique identifier ranging from 0001 to 0020.

The CERBERO holistic methodology described in the current document address 4 CTRs, as described in the Table 1-1. It is important to note that most of the requirements related to this deliverable are covered by the tools integrated in the framework and are not reported in the following table.

CTR id	**CTR Description**	**Link with the D5.1 document on CERBERO holistic methodology and integration interfaces**
0001	CERBERO framework/technology SHOULD increase the level of abstraction at least by one for HW/SW co-design and for System Level Design.	Users can benefit from the integration of lower level HW-development oriented tools with higher abstraction level ones. The entry point for custom accelerators developments are mainly high level dataflow specifications.
0002	CERBERO framework SHOULD provide interoperability between cross-layer tools and semantics at the same level of abstraction.	The semantic integration at the same level of abstraction and the interoperability between cross-layer tool has been achieved with the definition and development of CIF.
0003	CERBERO framework/technology SHOULD provide incremental prototyping capabilities for HW/SW co-design.	Direct integration of computing level development tools allowed the possibility to complementary and incrementally integrate/assess new features on the systems. Staring from pure SW implementation monitoring capabilities allows to understand what can/should be moved to HW. Different versions of this latter can be customized with a small user effort and time having the same C code as an entry point, leading to a new set of prototypes to be evaluated.
0009	CERBERO SHALL develop integration methodology and framework.	All the integration related activities, both direct and CIF-based, concurs to the final development of the framework.

Table 1-1: List of CTR addressed by the D5.1

2. Introduction

Cyber-Physical Systems (CPS) are engineered systems comprising interacting physical and computational components. In CPS, computation and communication are deeply embedded in and interacting with physical processes to add new capabilities and characteristics to physical systems.

In this report, we investigate how the various system components could be integrated into a holistic operational framework respecting the requirements imposed by the overall system and seeking a new foundation for CPS design, integration and operation. The goal of the project is to deliver a model-based, heterogeneous and robust solution that is going to be customizable (upon scenario needs) and generalizable (to different scenarios).

2.1. Modelling for Systems Engineering

Systems Engineering (SE) dictates the design process associated with the development of large-scale products through defining systems and subsystems requirements, their architecture, and critical parameters. With the recent increase in product complexity and business competition, mere intuition of system design engineers has proven insufficient for finding feasible, safe, reliable and affordable designs [1]. Hence modelling started to emerge to overcome these shortcomings and provide a well-thought-out plan for a solid design and smooth implementation and refinement.

In the context of systems engineering, models are created to deal with complexity. In doing so they allow us to understand an area of interest or concern and provide unambiguous communication amongst interested parties, models are leveraged in nearly all stages of the development process. During analysis, models provide an abstract representation of the desired solution, e.g. in terms of dynamic behaviour diagrams such as sequence diagrams and state machines. In the design phase, the software architecture could be abstracted through UML (Unified Modeling Language) component structure diagrams and class diagrams. In addition, ports and interfaces facilitate modelling of data flows between components. From these models, code can be partially generated in the implementation phase. Moreover, many models can be used at several levels of specification of the system. For example, Finite State Machines are useful at the system, and component levels, both for HW and SW. For testing purposes, models are used to generate test cases. To sum it up, Model-Based Systems Engineering (MBSE) or, more generically, Model-Driven Engineering (MDE) has models as the primary data source. Model Driven Development uses the activities associated with modelling to drive the whole development process [2].

From that perspective, specialized modelling tools come into play to increase productivity. There is a plethora of modelling languages such as SysML (a UML extension), AADL (Architecture Analysis & Design Language), and modeling tools such as Excel, IBM Rhapsody, PTC Artisan Studio, Sparx Enterprise Architect, domain-specific tools (e.g. medical, avionics, automotive, marine, space, etc.), and simulation environments such as Simulink and Modelica [1]. The Functional Mock-up Interface (FMI) standard facilitates the exchange of simulation models between suppliers and OEMs. FMI adopts a tool-independent approach for both model exchange and co-simulation of dynamic models using a combination of XML-files and compiled C-code [3]. Its success could be attributed, at least partially, to a minimalistic API allowing high flexibility for tool providers.

There are also black-box integration tools that help with reducing complexity, improving efficiency and cutting development time. FRONTIER - by ESTECO - provides an innovative optimization environment with modular, profiled-based access. ESTECO's integration platform for multi-objective and multi-disciplinary optimization offers a seamless coupling with third party engineering tools, enables the automation of the design simulation process and facilitates analytic decision making. ModelCenter Integrate - by Phonix Integration - allows users to automate any modelling and simulation tool from any vendor, integrate these tools together to create repeatable simulation workflows, set simulation parameters, and automatically execute the workflow. Hence ModelCenter Integrate increases productivity by enabling users to execute significantly more simulations with less time and resources.

The benefits of using models in systems engineering are manifold: Building models is usually easier than building the actual system as a whole from the ground up. Modelling helps to capture, structure, and understand the system and to reveal, early enough, possible problems and potential bugs lurking at every stage of the system design and implementation. It has proven more appropriate for high-stakes and increasingly complex applications such as reconfigurable and adaptive cyber physical systems. Last but not least, integration per se is not a scientific activity, it is rather an engineering problem full of accidental issues (i.e. tools adopting different languages, running on different platforms, vendor-specific components, etc.) CERBERO is no exception; therefore, we decided to adopt a model-based, semantically oriented, approach.

2.2. Systems Integration from CERBERO Perspective

Systems integration is a process whereby a cohesive system is created from components that were not specifically designed to work together. Components of an integrated system are often systems in their own right and integration aims at interconnecting these components together in a layered fashion in order to provide a useful exchange of information, data and/or control between these sub-systems and assuring that the integrated system meets requirements and performs according to user expectations.

To facilitate systems interconnection, an interface is defined and created as a point of interaction between communicating systems. Interfacing may also mean using a common message format, or intermediate representation, to provide kind of a unified communication paradigm across the system entirely, or partially. The translation would be required from the interface of one component to the intermediate representation, or vice versa.

Generally, integration is done considering subsystems as black-boxes, hence creating a middleware to “glue” together these disparate subsystems without them needing to know anything about each other. We support the “open world” assumption, where each tool should assume that all objects may have more properties than it knows about. Inspired by FMI standard, we are developing as simple formalism as possible to exchange objects with properties that tools recognize with middleware support for simple property mappings when it is semantically clear.

A recommended integration process adopts an iterative – that is, continuous or constantly evolving - integration model rather than a static or fixed model. Hence, it is essential to create a holistic and customizable framework for subsystem integration as these subsystems and tools undergo continuous development. For tracking of integration progress, the integrated system must be verified and validated periodically against system and user requirements toward a mature integration framework for the overall platform or toolchain.

3. State of the Art

Systems Engineering governs the design process associated with the development of large-scale products through defining systems and subsystems requirements, their architecture, and critical parameters. The recent increase in product complexity and business competition dictates the necessity for finding feasible, safe, reliable and affordable designs. Consequently, three techniques have become popular in helping handle the complexity: layering design process into several levels of abstraction, separation of concerns, and using computerized tools for automation of modelling, optimization and analysis.

Embedded systems are commonly subject to data intensive processing applications where huge amounts of data are handled in a regular way by means of repetitive computations. These applications deal with intensive or massive parallelism either on the data level or on the task level. High-level analysis of data-intensive applications becomes a complex task necessitating a refinement step toward low levels of abstraction specifying both computation and communication costs in the system.

The following works address the challenge of abstracting system design in a layered fashion that maintains computation and communication specifications of the system and facilitates analysis and optimization hence provides a competitive business edge for the end cyber-physical system.

3.1. Usage of Formal Semantics

Model-Based Engineering of Cyber-Physical Systems needs correct-by-construction design methodologies, hence CPS specification languages require mathematically rigorous, unambiguous, and sound specifications of their syntax and corresponding model semantics. Cyber-physical systems are software-integrated physical systems often used in safety-critical and mission-critical applications, for example in automotive, avionics, chemical plants, or medical applications. In these applications sound, unambiguous and formally specified modelling languages can help developing reliable and correct solutions.

Traditional systems engineering is based on causal modelling (e.g., Simulink), in which components are functional and a well-defined causal dependency exists between the inputs and outputs. It is known that such a causal modelling paradigm is imperfect for physical systems and CPS modelling since physical laws are inherently acausal.

Recently, acausal modelling has gained traction and several languages have been introduced for acausal modelling (e.g., Modelica, bond graphs) [4]. Every time a new specification language is introduced, there is a natural demand to extend it to support as many features as possible. Unfortunately, this often leads to enormously large and generic languages, which have many interpretations and variants without a clear, unambiguous semantics. Because of the size of these languages, there is not much hope for the complete formalization of their semantics.

A fundamental problem is that generic languages provide support for many more features than a specific problem needs, still, they often lack support for some essential functions that would be otherwise needed. Thus, in most cases, it is more feasible to use Domain

Specific Modeling Languages (DSML) [5] [6], which are designed to support exactly the necessary functions. Additionally, because DSMLs are usually significantly smaller than generic languages, their formal specification is feasible.

The work in [7] discusses the challenges to develop the formal semantics of a CPS-specific modelling language called Cyber-Physical Modeling Language (CyPhyML). The paper formalizes the structural semantics of CyPhyML by means of constraint rules, and the behavioural semantics by defining a semantic mapping to a language for differential algebraic equations. The specification language is based on an executable subset of first-order logic, which facilitates model conformance checking, model checking and model synthesis.

3.2. Viewpoint Modelling

Viewpoint modelling is an effective approach for analyzing and designing complex systems. Splitting various elements and corresponding constraints into different perspectives of interests enables separation of concerns such as domains of expertise, levels of abstraction, and stages in the lifecycle. Specifically, in Systems Engineering different viewpoints could include functional requirements, physical architecture, safety, geometry, timing, scenarios, etc. The first development and refinement step are referred to as Engineering Modeling, and the second optimization and analysis step is referred to as Design Space Exploration (DSE). Consequentially, there are no automatic tools for holistic DSE based on libraries of previously developed and tested Analysis Viewpoints.

Despite partial inter-dependences, models are usually developed independently by different parties, using different tools and languages. However, the essence of Systems Engineering requires repetitive integration of many viewpoints in order to find feasible designs and to make good architectural decisions, e.g., in each mapping between consecutive levels of abstraction and in each design space exploration. This integration into one consistent model becomes a significant challenge from both modelling and information management perspectives.

The work in [1] suggests: (1) a unique modular algebraic viewpoint representation robust to design evolution and suitable for the generation of the integrated optimization/analysis models, and (2) an underlying ontology-based approach for consistent integration of local viewpoint concepts into the unified design space model. The paper shows also an example of an optimization model with different combinations of partially interdependent Analysis Viewpoints. Using the proposed modelling and information management approach the underlying viewpoints' equations can be applied without modification, making the approach pluggable.

3.3. MARTE and PiSDF

At the present time, embedded systems are commonly dedicated to data-intensive processing applications where huge amounts of data are handled in a regular way by means of repetitive computations. These applications deal with intensive or massive parallelism.

Indeed, parallel applications can implement two levels of parallelism: data parallelism and task parallelism. High-level analysis of data-intensive applications becomes a complex task necessitating a refinement step toward low levels of abstraction specifying both computation and communication costs in the system. Accurate performance numbers can be reached at the cost of very detailed modelling. On the other hand, a moderate effort for modelling leads to a high-level evaluation task, but the accuracy is lost.

The work in [8] proposes a new approach that takes advantage of Model-Driven Engineering (MDE) foundations and Modeling and Analysis of Real-Time and Embedded Systems (MARTE) profile. The paper defines a transformation to a new level of abstraction that alleviates the exploration and analysis tasks of real-time data-intensive processing applications. This level is based on a novel extension of the famous Synchronous Data Flow (SDF) Model-of-Computation (MoC), the Parameterized and Interfaced Synchronous Dataflow (PiSDF) model. PiSDF facilitates the specification, and especially the analysis of data-intensive applications as it gathers a lot of features including hierarchy, configurability and dynamism. This MoC introduces analysis techniques facilitating the design space exploration task. Then, a high-level analysis of the data-parallel application is performed using the PREESM rapid prototyping tool.

4. CERBERO Integration Methodology

4.1. Tool Suite

As a first step toward building CERBERO toolchain, IBM and AI in collaboration with the entire consortium have surveyed and agreed upon an array of tools that are either state-of-the-art or already exist within the consortium as fully mature technologies. This survey helped to collect and consolidate information about the functionality of each tool, setup requirements, documentation availability, maturity/readiness level, licensing, and usability domains, and would serve as a reference point for integration efforts. These tools cover the entire stack of a CPS: system model layer, application layer, OS or runtime layer, and hardware abstraction layer. This survey analysis has been the basis for defining a feasible CERBERO integration plan and to specify the interfaces for newly designed tools.

4.2. System Design and Operational Framework

For the model-based design of complex systems, such as CPS and CPSoS, a structured and well-defined design framework is essential to guarantee high-quality products that fulfil all requirements of the stakeholders and to enable handling the complexity of such systems by introducing different viewpoints or abstraction layers to model the system under development. Therefore, an important step to obtain a system implementation is the Design Space Exploration (DSE), where design decisions are taken based on goals and requirements defined in previous phases. The set of valid solutions may be restricted by constraints, which could be derived from previously defined requirements considering aspects such as functional requirements, physical architecture, safety, geometry, timing, etc. Valid solutions are rated with respect to defined goals to facilitate a decision toward the choice of final system implementation. Based on this abstract framework, concrete DSE methods can be implemented addressing different kinds of DSE problems that are relevant in industrial practice. In the CERBERO framework we intend to carry out DSE at different levels, at the system level to define the proper distribution among computing nodes, and within the single node to identify the optimal HW/SW partitioning and components set-up. The selected optimal design resulted from the system level DSE should be available (i.e., machine-readable) for HW/SW co-design to define, partially, its functional requirements.

CERBERO design framework aims at shifting designers’ work at a higher abstraction level and earlier in the design process, relieving them from manual and complex HW/SW tuning phases. It creates and promotes model-based cross-layer optimization, design, verification and simulation methods that aim at deeply modifying CPS system design approach, shifting from a V model paradigm to a ladder model paradigm, see Figure 4.1, thus slashing the time to market. In the model-based approach, all the properties and system characteristics are tackled concurrently right at the model level at design-time to allow the cross-optimization of physical components, as well as the cross-configuration of the HW and SW layers. Dealing with CPS and CPSoS poses challenges for the overall system assessment (dimensioning and characterizing communication channels, profiling power consumption, etc.) To cope with these challenges, CERBERO features adequate analysis tools and provide sufficient breakdown of all related layers of the system model to guarantee continuous validation.

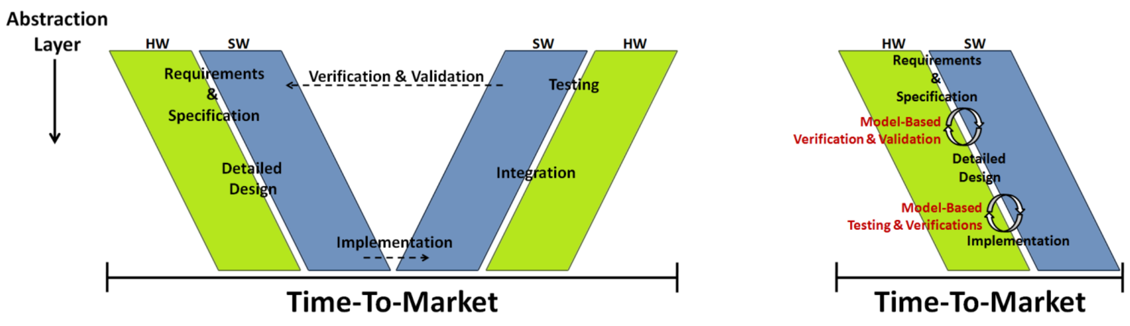

Abstraction Layer
HW
SW
Requirements & Specification
Verification & Validation
SW
HW
Testing
Detailed Design
Integration
Implementation
Time-To-Market
HW
SW
Requirements & Specification
Model-Based Verification & Validation
Detailed Design
Model-Based Testing & Verifications
Implementation
Time-To-Market

Figure 4.1: V-Model vs. Ladder System Design Approach

The CERBERO toolchain operates at different levels of abstraction, going from the computation level to the system (of systems) level, delivering also user-level features (i.e. requirements/model verification). Figure 4.3 illustrates the CERBERO tools that offer support at Design-Time as foreseen at M27:

- At Model/Verification Level SAGE suite leverages on formal methods to automatically check the consistency of a set of requirements provided by the user.
- At System (and System of Systems) Level, AOW solves optimization problems to return frontier of Pareto optimal solutions, while DynAA is based on discrete-component models and the related parallelism and physical aspects, finding optimal solutions by means of model simulations.
- At the Computation Level several tools, for design implementation, are present.
    - PREESM enables parallel-application development with design-time prediction, as well as code generation and re-use capabilities.
    - PAPIFY provides monitoring capabilities by means of an event library aimed at generalizing Performance API (PAPI) for heterogeneous architectures.
    - MDC is an automated dataflow-to-hardware framework for the generation of coarse-grain reconfigurable accelerators.
    - ARTICo^3 exploits a DPR-enabled multi-accelerator computing scheme, going to the user-defined application down to the system implementation.
    - JIT hardware composition refers to the ability to implement, at run-time, hardware accelerators on FPGAs without a pre-synthesized design. IMPRESS, belonging to the JIT HW design suite, is a TCL script-based tool for the automated generation of relocatable partial bitstreams under Vivado.

At design-time, using intermediate semantics for integration through the CERBERO Integration Framework (CIF) has allowed requirements analysis and verification at the model level. Integration of some tools at the computational level (both for SW and HW design), however, is more implementation oriented since interfaces between these tools have already been coded or currently being developed and tested in a DIRECT INTEGRATION tool-to-tool manner. In Figure 4.2 and Figure 4.3 connection achieved through the CIF are highlighted in yellow, while direct point-to-point connections are a highlight in green. Basically, with respect to M27, the final Design-Time support of the CERBERO framework is the one depicted in Figure 4.3, where you should notice that MDC has not been connected to SAGE. We opted for building the connection of MDC to CIF with the idea of providing MDC the possibility to connect to higher abstraction level tools and to assess, conversely from the CIF point of view, the possibility of using CIF also to make interoperable lower computation level framework components.

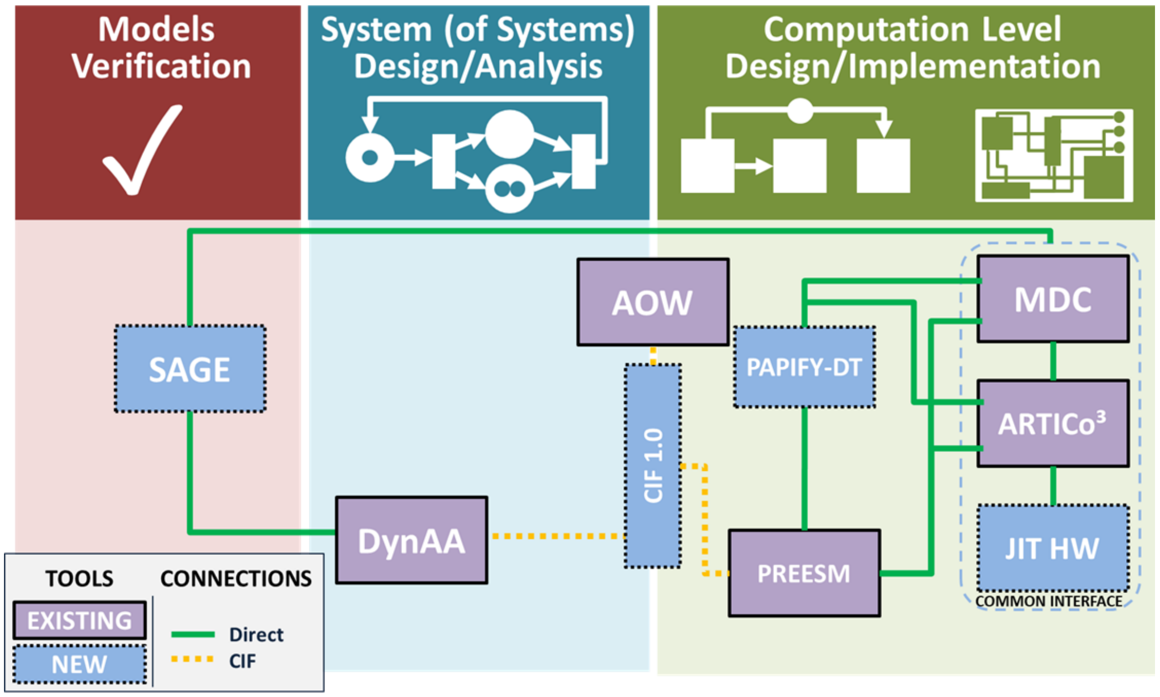


Figure 4.2: CERBERO toolchain for Design-Time Support (as foreseen at M18)

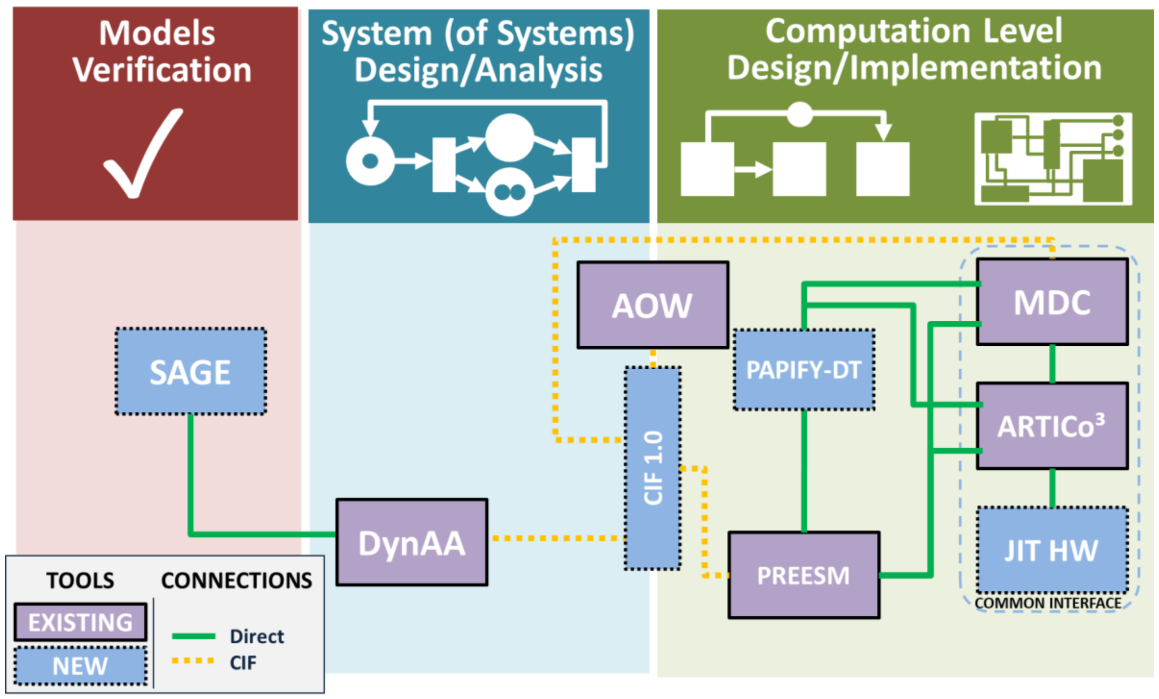


Figure 4.3: CERBERO toolchain for Design-Time Support (as foreseen at M27)

Figure 4.4 illustrates the CERBERO tools that offer support at Run-Time as foreseen at M27:

- At System (of Systems) Level, MECA improves the resilience of human-machine teams by providing system, environmental and human monitoring and diagnosis, and high-level decision support in cases of unforeseen conditions and events, while DynAA explores different solutions at run-time to provide direct interaction with signals that come from the system and the environment.
- At the Computation Level several tools, for run-time support are present.
    - SPIDER performs dynamic mapping and scheduling of dataflow applications on a parallel heterogeneous architecture.
    - PAPIFY is meant to provide a large set of run-time execution information to SPIDER.
    - MDC and ARTICo^3 deploy and configure proper engines over the physical substrate at design-time. These engines are used at run-time to execute all the actions needed to support run-time reconfiguration of the HW.
    - JIT HW composition addresses fine-grain reconfiguration, providing a way to map circuits at run-time by composing small HW components laid on an overlay architecture. IMPRESS, a tool to implement JIT composable HW, allows reconfigurable module composition to generate custom overlays on the fly, without the need of predefined floor planning and inter-module communication description.

It is important to highlight that the final Run-Time support of the CERBERO framework has not changed from M18 to M27. Furthermore, the connection between SPiDER and ARTICo^3 is not available at the moment due to low-level incompatibilities between tools.

However, the ARTICo^3 runtime library provides similar rescheduling capabilities for FPGA-based processing, and it is being used currently while SPiDER is modified to support heterogeneous HW/SW processing properly.

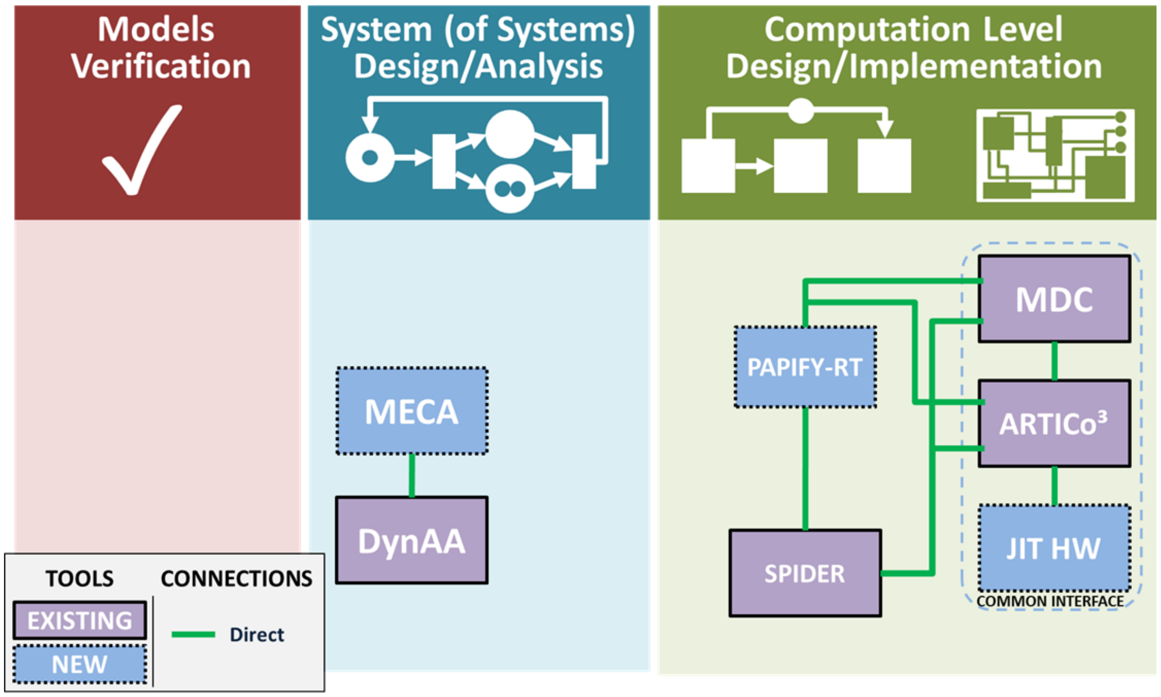


Figure 4.4: CERBERO toolchain for Run-Time Support (as foreseen at M18 and M27)

Integration is based on methods and interfaces in agile cycles performing cross-layer integration and combining modelling, simulation, verification and code generation activities. It turned out to be essential before attempting the modelling, and hence integration efforts, to plan the following system characteristics:

- Functional and non-functional requirements.
- Available system libraries, including models that require new developments.
- System structure and composition.
- A unified naming convention for all properties and attributes.
- Open world assumption (more tools, more viewpoints).

Consequentially, constraints and KPIs (such as jitter, delay, latency, KPI, QoS, energy consumption) are verified before starting integration activities. All those activities have involved partners, and in particular technology/tools provider, participating to WP2-5.

Integration also involves gap analysis in order to discover gaps and overlaps and reveal points of interoperability. Understanding the opportunities for integration or gap-filling informs holistic trade-off decisions about integrating systems and capabilities [9].

Execution of CERBERO integration methodology adopts a Continuous Engineering approach that guarantees reuse of design time models for generating novel and more accurate design and operational models up to the runtime environment. In this deliverable, we provide an update on the integration methodology adopted.

Semantic model-based integration supports cross-layer and cross-levels of abstraction modelling, usually dealing with system integration and system modelling. In this case, we leverage on ontologies "passing" relevant properties among layers and levels of abstraction. On the other hand, each layer (that can handle more than one model) has its own tools that usually much more compatible among each other and that should pass information by means of the same ontologies focusing on fast execution (important, e.g., for simulation). We implement a simplified version of ontology based on things with properties: each tool will look for relevant properties of intermediate models, constraints or KPIs from other tools (potentially in other layers/levels of abstraction); while the framework supports mapping model properties to tools/analysis namespace properties and provides shared directories for model exchange. The methodology has been co-refined together with project advance in cross-layer integration framework combining modelling, DSE, simulation, verification and code generation capabilities of CERBERO tools. Two iterations were planned during the total duration of the project, see Figure 4.5 , delivering a "beta" and a "final" version of CERBERO integration strategies and internal interfaces. This report refers to integration activities up to M27. We are still closing the implementation step of the second phase of the project.

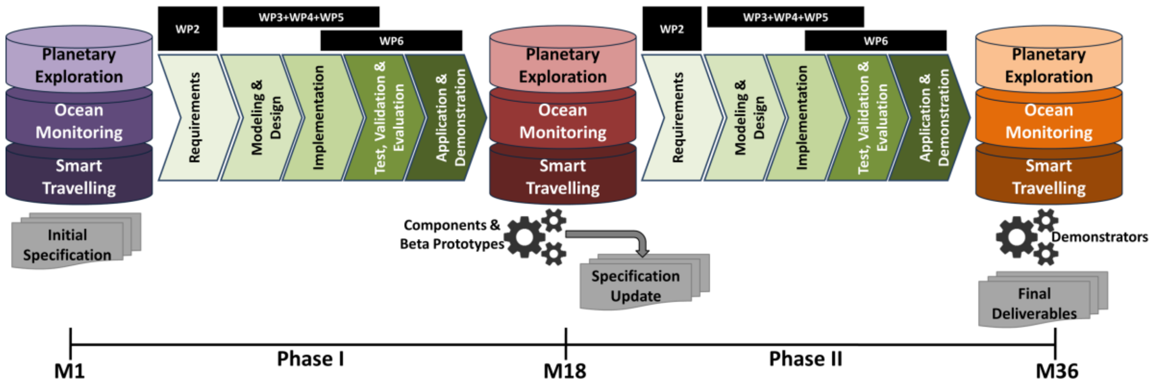


Figure 4.5: Iterative Development Scheme of CERBERO Integration Framework

4.3. Simplified Ontology-based Integration with Intermediate Representation

As mentioned earlier, integration is traditionally a complex engineering problem. It is characterized by several different accidental issues, as the usage of different modelling paradigm or languages, and thus require extreme effort to create and maintain necessary integration infrastructure. This is particularly true in the CPS environment where you need to combine components suitable for the different aspects of the CPS. These motivations led the designer to look for semantic integration of tools, and ontology-based integration is particularly suitable, in our opinion, to the case.

The term "ontology" derives from ancient Greek "onto", which means "being" and logos, which means, "discourse". Ontology -- or roughly the "science of stuff" and how it is represented -- used to be a rather obscure branch of philosophy. It still is in some cases, but it is also an important and growing area of computer science and the web of things (WoT). Then, ontology has assumed other relevant meanings, such as:

"A formal shared and explicit representation of a domain concept."

or:

"A method for formally representing knowledge as a set of concepts within a domain, using a shared vocabulary to denote the types, properties and interrelationships of those concepts."

or:

"A formal way to describe taxonomies and classification networks, essentially defining the structure of knowledge for various domains."

Ontology-based data integration involves the use of ontology(s) to effectively combine data or information from multiple heterogeneous sources. So, we have started CERBERO integration efforts by creating a kind of ontology, by which we mean a collection of terms that identify the real things and relationships that are relevant to CERBERO-supported domains and industry-driven requirements. Maintaining an ontology design facilitates keeping track of the terms and ensures integration efforts quickly get up to speed.

CERBERO features heterogeneous technologies and tools. As discussed above, we retained the idea that model-2-model transformation would not necessarily be the main mean of communication between tools (also, the feasibility of having fully automated model to model transformations from the system of system level down to the hardware is unlikely). Instead, each tool will manage its own model(s), and the intermediate representation will be used to exchange "cross-layers" and "cross-models" information between tools.

The intermediate format is, therefore, necessary to achieve the mediation between the application's class model conceptualization and the common domain ontology conceptualization since objects in the original format cannot be handled directly in the framework [9]. Thus, CEBERO Interoperability Framework (CIF) follows the Resource Description Framework (RDF)-like meta-model underlying common ontology.

CERBERO integrated framework aims at one model for the entire platform. That is, a single model that maintains combined information provided from different connected tools. The key feature of CERBERO integrated framework is an ability to enrich this model by information provided by connected tools in tool-specific formats and to produce views of this model that are readable by connected tools. This functionality provided internally by the two-layered model structure [20] that separates instances, properties and aggregations (lower level) from classes (upper level) (see Figure 4.6).

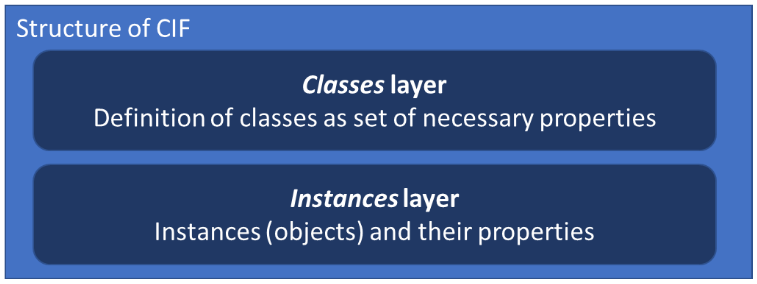

Structure of CIF
Classes layer
Definition of classes as set of necessary properties
Instances layer
Instances (objects) and their properties

Figure 4.6: Layers of the CERBERO Intermediate Format

Each instance in this model represents a thing that possesses one or more properties within corresponding namespaces. The property itself possess a value that can be either simple (integer, float, string, etc.) or object (another instance). Aggregations are special instances that serve to represent one-to-many relations between instances, so each aggregation can "contain" several instances. An example of instance-level CIF model is presented on Figure 4.7.

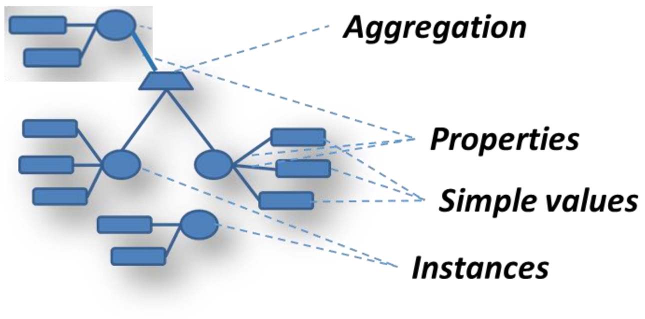


Figure 4.7: Example of CIF model

Classes are implemented using classification-by-property paradigm [20]. That is, any instance that possesses some predefined set of properties becomes an instance of the corresponding class. This predefined set of properties denoted as a class definition. The set of class definitions related to the specific namespace form ontology.

The system-level of abstraction can have a complete view of the system infrastructure. However, system level is not fully or directly aware of the "internals" of the underlying more detailed levels. Since the information available at the system level are relevant at lower levels, instances and properties in a model structure are inter-linked in such a way that allows navigation from one property in the system to another across different layers (see Figure 4.8).

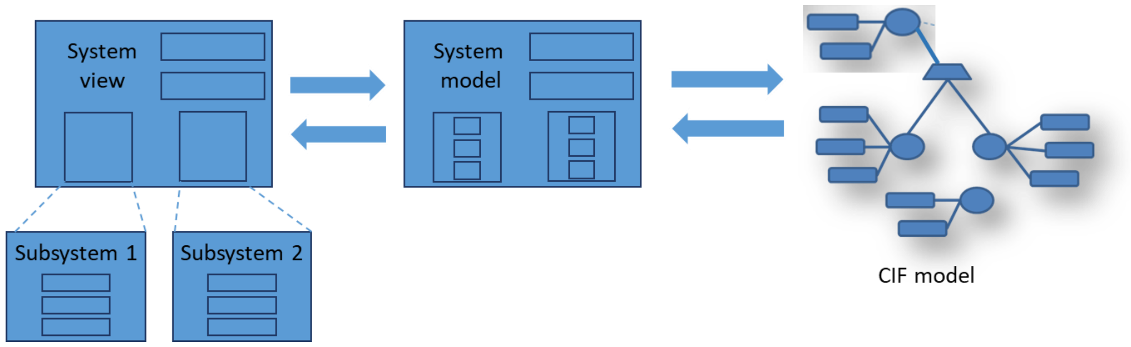


Figure 4.8: From system view to CIF model

Ontology helps with revealing meaning and relations of each property from the whole graph by referring to a property by its name. All properties relevant to the model are present in the ontology. Ontologies can be either simplified (i.e. system model features only a subset of all properties of the real system), or full ontology where all properties in the system model are presented in the ontology. To enable interoperability between different tools and preserve the integrity of holistic model mappings between ontologies are provided. These mappings expressed through equivalence rules between classes and define relations between instances, classes and properties coming from different namespaces. As a result, CIF database contain a single model combined from different viewpoints provided by different tools (Figure 4.9).

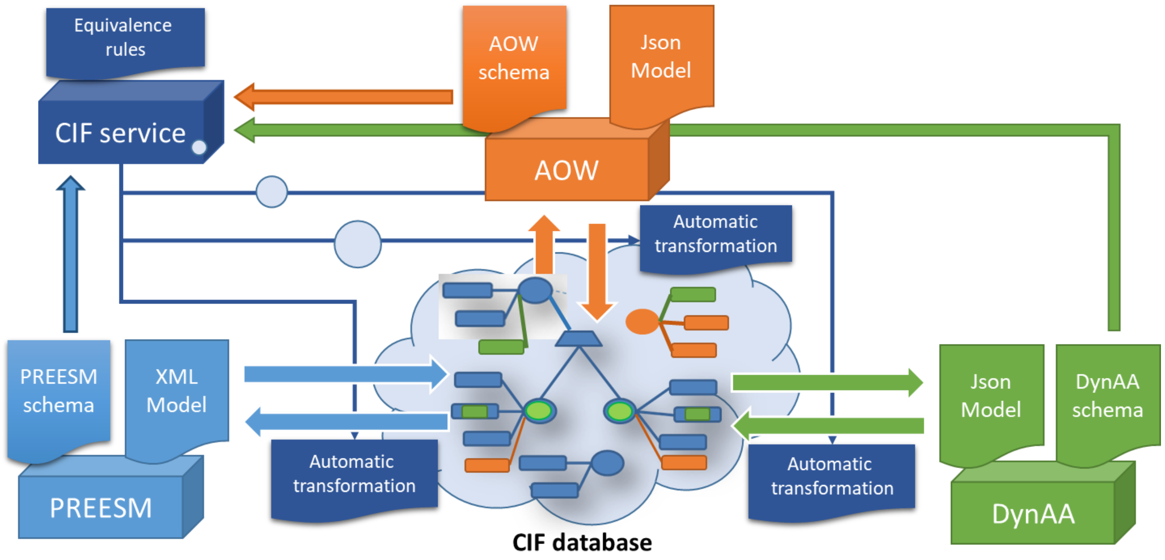


Figure 4.9: Example of ontology mapping between PREESM, AOW and DynAA

CERBERO proposes an application layer solution for interoperability. The key idea is to utilize semantics provided by existing specifications and dynamically wrap them in a middleware fashion into semantic services in such a way that automates interoperability without any modifications to existing standards, devices, or technologies, while providing to the framework user an intuitive semantic interface with services that can be obtained by executing all CERBERO technologies. In particular, a semantic layer is proposed for simple mapping of KPIs to model properties (i.e., connecting corresponding namespaces) as a semantic service; we may conventionally call that middleware the "CIF Service"
The architecture of CIF service represented on Figure 4.10.

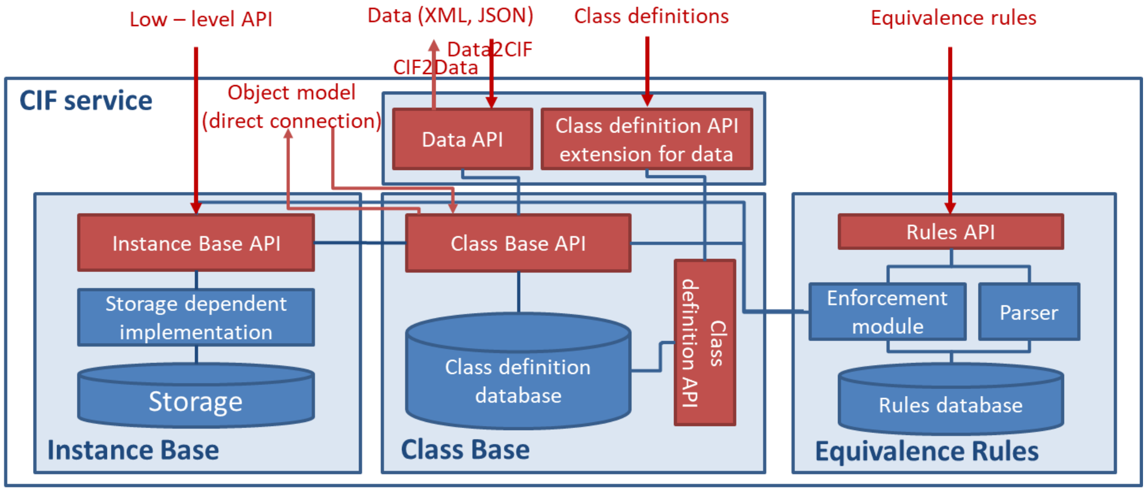


Figure 4.10: Architecture of CIF service

Tools can be connected to CIF directly by exposing their object models, or indirectly by translating their object models into suitable representation such as XML or JSON. In the second case one need additional layer translating data representation of object model into class base representation. This layer requires extensions of class definition semantic that is necessary for data-to-model translation. CIF service provide following APIs:

- Low-level API is designed to deal with low-level CIF model, i.e. add/remove/ change instances, properties and aggregations directly. This API is not supposed to use directly by external tools but can be useful for service extensions.
- Class definition API is designed to allow ontology definition through definition of related classes and classification by property paradigm.
- Class base API is designed for direct tool connection and allow import and export of part of tool model that is described using corresponding class definitions.
- Data API serves to import/export tool models from XML or JSON data files and requires from corresponding class definition extensions related to data-to-model translations.
- Equivalence rules API serve to maintain mappings between different ontologies (namespaces).

To sum it up, intermediate representation is the data or object model that one level of abstraction, layer or tool is going to produce and the other is going to consume. It may not necessarily be human understandable/comprehensible, but it’s necessarily interpretable from one level/layer to another. In addition, feedback can be communicated to the originating level/layer to facilitates self-adaptivity and reconfigurability. To connect different layers or tools to the CIF service one should provide ontology of the respective layer or tool. This ontology expressed in a form of class definitions that utilizing classification-by-property paradigm. When class definition provided respective models can be exported/imported into intermediate format automatically. Ones several tools or layers describe their ontologies, it is possible to provide mappings between these ontologies in a form of equivalence rules between corresponding classes. When corresponding equivalence rules are provided, CIF service will care on automatic model transformation between connected tools.

4.4. Mapping KPIs to Model

The evaluation of any Key Performance Indicator (KPI) out of its system model implies that designers must determine a way to calculate them. Such calculation can be expressed by means of a certain mathematical structure, or as we call it, an ***algebra***. *Algebras* provided as mathematical expressions between properties of different instances having specific relations. Thus, algebra defines its own model including instances, classes and properties. In a case when there is no plain mathematical structure and calculation of corresponding KPI or its part should be held by some algorithm this algorithm can be described as a function possessing input and output parameters that can be described w.r.t to the same model structure. Models of the KPIs can be naturally expressed as a set of parametrized class definitions. Ones all parameters are given one can instantiate these definitions adding them to the class definition database through corresponding API. Furthermore, these definitions can be mapped to the model by equivalence rule mechanism.

Thus, to map KPI to the model one should instantiate class definitions provided by KPI ontology and map these definitions to the model (see Figure 4.11). Ones mapping provided, application serving for corresponding KPI calculation can read the model, perform calculation and return the results back to the CIF.

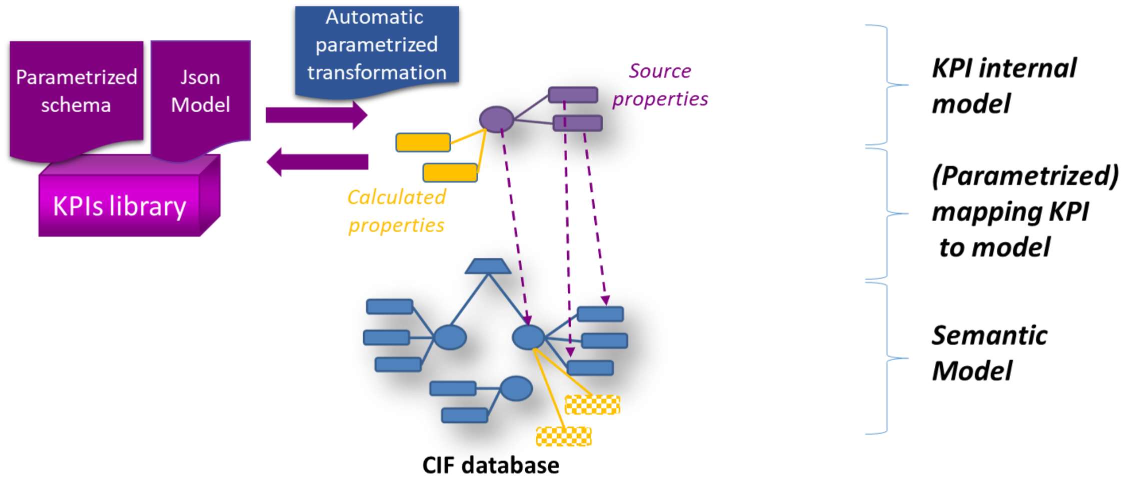


Figure 4.11: Mapping of KPI to the model

4.5. OS Qualified as a Toolchain Host

CERBERO array of tools and technologies widely support both Windows and Linux. However, Windows has been qualified by the consortium as the host operating system of choice for the project since it has a larger market share and is more familiar with respect to Linux, more feature-complete, enjoys commercial support, and is not subject to the disintegrity imposed by a wide range of different Linux distributions that are only maintained by their respective communities.

5. Internal Interfaces

Cyber Physical Systems (CPSs) are an evolution of embedded systems and are based on a tight combination of collaborating computational elements (i.e. micro computing units or embedded systems interconnected by a communication system) that control physical entities. Therefore, in CPS all types of smart equipment (i.e. sensors, actuators, devices, machines, robots) are interconnected creating a smart community with data capture and action capability from/to the physical world.

A CPS-based architecture for manufacturing is made of smart but independent manufacturing components without any knowledge of the role they have to play together in the real application. Ontologies can supply such kind of knowledge, playing a very crucial role in CPS design.

Ontology in computer terms is concerned with the meaning of and the relationship between entities. It derives its importance from its ability to organize raw or unstructured data by semantics (i.e. meaning, such as classes, properties or attributes, and relationships), rather than merely by strings or keywords, thus facilitating more efficient data operations (storing, querying, sending and receiving). An ontology is often referred to as a “schema”.

Ontologies are typically far more flexible than class representations and hierarchies as they are meant to represent information coming from all sorts of heterogeneous data sources. Class hierarchies, on the other hand, are meant to be fairly static and rely on far less diverse and more structured sources of data.

Therefore, CERBERO consortium postulates simplified ontologies as the right tool to implement cross-layer information sharing and data flow in order to implement internal interfaces and hence realize the CERBERO holistic integration methodology. As discussed above, by simplified ontology we mean using things with properties where all analysis viewpoints are defined by a set of properties and all objects that hold them.

5.1. Low-Level Instance Base Interface

As mentioned in Section 4.3 low-level interface intended for direct manipulation with basic CIF structures that includes instances, properties and aggregations.

Instance – represents a simple object that has unique identifier (UID) that allow to distinguish one instance from another. Instances can possess properties.

Property - represents a relation between instance (property owner) and property value, that can be either simple value, another instance or aggregation. Each property has name and namespace. Any instance can possess (be owner of) only one property with specific name within specific namespace.

Aggregation – generalization of the instance that served to represent one-to-many relations. May contain any number of members that can be either instances or simple values. In the current implementation cannot possess properties. The question on whether aggregation can or cannot possess properties is under investigation.

The low-level interface includes methods to get/create/delete instances, properties, aggregations and namespaces. It also includes additional methods that are created to fulfill functionality required by high-level APIs such as class base or equivalence rules. This interface in general is not supposed to be used by CIF connected tools directly, unless CIF connected tool do not introduce extension to CIF itself.

5.2. Ontology and Class Definition Interface

Many software packages are now available for creating ontologies, among which are:

- Stanford University's Protégé (https://protege.stanford.edu/) , a free, open-source ontology editor.
- TopBraid Composer from TopQuadrant. (https://www.topquadrant.com/products/topbraid-composer/).
- Generally, any text editor.

Apart from RDF/XML, RDF ontologies can also be expressed in human-readable formats, for example JSON that is also popular format for data exchange.

Before delving into a full example of a JSON-represented ontology, here below is a quick recall of the most relevant terminology:

- **JSON:** is a lightweight format for data exchange (as XML, but less verbose yet more human-readable). [By "data" we mean is the set of classes and attributes that will be shared by tools and technologies that interoperate within the CERBERO framework.].
- **JSON-Schema:** a JSON document according to which the ontology is defined. [For example, if there are required or mandatory attributes if they are of number datatype, if they can be null, etc.] In XML equivalence, that would correspond to an XML schema or a Document Type Definition (DTD).
- **Ontology:** formally defines a common set of terms used to describe and represent a domain (according to the JSON-Schema).
- **Instance of Ontology:** a specific element of an ontology.

Here we provide a simple Ontology such as representing an instance of DynAA task class that is defined as all instances having “name”, “type” and “params” properties within DynAA namespace. Here name and type are simple properties having values of string type and params is object property which value is an instance of DynAA param class that have integer property load and optionally integer property factor. An instance of this class looks like:

```
{
        "name": "Split0",
        "type": "Split",
        "params": {
                "factor": 8,
                "load": 2
        }
}
```

Following is a JSON Schema that describes the Ontology:

```
{
  "namespace": "dynaa",
  "class": "task_description",
  "schema": {
    "properties": [
      {
        "name": "name",
        "optional": false,
        "value": {
          "type": "str",
          "optional": false,
          "collection": null,
          "constrains": [],
          "default": null,
          "object": null
        }
      },
      {
        "name": "type",
        "optional": false,
        "value": {
          "type": "str",
          "optional": false,
          "collection": null,
          "constrains": [],
          "default": null,
          "object": null
        }
      },
      {
        "name": "params",
        "optional": true,
        "value": {
          "optional": false,
          "type": "object",
          "collection": null,
          "constrains": [],
          "default": null,
          "object": {
            "namespace": "dynaa",
            "class": "param_description"
          }
        }
      }
    ]
  }
}
{
  "namespace": "dynaa",
  "class": "param_description",
  "schema": {
    "properties": [
      {
        "name": "factor",
        "optional": true,
        "value": {
          "type": "int",
          "optional": false,
          "collection": null,
          "constrains": [],
          "default": 1,
          "object": null
        }
      },
      {
        "name": "load",
        "optional": true,
        "value": {
          "type": "int",
          "optional": false,
          "collection": null,
          "constrains": [],
          "default": 0,
          "object": null
        }
      }
     ]
  }
}
```

By using the schema, a computer program can validate, understand, and process the data file to purposes of creating a computational model, transferring data, analysis, simulation, and reporting. The use of schemas makes the modelling tool more generic, and independent of a specific file format. This is very important to increase the interoperability between tools, modelling layers (HW/SW), and modelling views.

CERBERO intermediate format introduce following format of class definitions (schemas):

CIF class definition file includes following fields:

- `"namespace"` – namespace for which belongs class definition.
- `"class"` – name of the class.
- `"schema"` – schema of the class.

Schema consists of properties definitions enlisted under `“properties”` key, where each property includes following fields:

- `"name"` – name of the property.
- `"namespace"` – optional. Should be provided if property has different namespace (not same as defined in the schema namespace).
- `"optional"` – true/false indicates if property is optional or not. Instance of respective class may not possess optional properties.
- `"value"` – describes value of the instance.

Value field includes following properties:

- `"type"` – describes type of property value may be either “str”, “int”, “float” or “object”, indicating that related property should have value of corresponding type.
- `"optional"`– true/false indicates that value is optional (i.e. can have “null”) value or not.
- `"collection"` – “set”/ “list”/null indicates that property points to collection of objects of corresponding type. Null value in this field indicates that property possess single value. Properties possessing collection values (having collection “set” or “list”) possessing aggregation value in the instance base.
- `"constrains"` – set of constraints on property value (functionality of this field does not implement yet).
- `"default"` – default value of the property. Optional. If provided and property has null value, then default value used instead of null.
- `"object"` – either null (for simple types) or object specification (for object type).

Object specification consist of two fields:

- `"namespace"` – namespace for which belongs object value.
- `"class"` – name of the class for which belong object value.

5.3. Class Base Interface for Directly-Connected Tools.

Class base interface introduce methods to deal with objects of specific classes. Each class should be described by providing corresponding class definition. As mentioned in section 4.3 classes are implemented using classification-by-property paradigm, i.e. each instance in the instance base that possess specific set of properties described in some class definition is the object of the corresponding class. In order to use class base interface direct-connected tools should expose their objects as Python dictionaries. Class base interface includes following methods:

- `"create_object(self, namespace, class_name, dct)"` – creates object of specific class within specific namespace from Python dictionary object. Fails if Python dictionary object does not include specific set of properties described in class definition.
- `"get_object(self, namespace, class_name, cif_instance)"` - obtains object of specific class within specific namespace from specific CIF instance. Fails if provided instance does not possess necessary set of properties described in class definition.
- `"get_objects(self, namespace, class_name)"` – obtain all objects that meets specific class definition within specific namespace.
- `"remove_objects(self, uids)"` – removes instances of corresponding objects including corresponding properties.
- `"update_object(self, uid, dct)"` – update object by setting property values according to values provided by Python dictionary object.

5.4. Data Serialization and Data Interfaces

Successful tool integration necessitates that system model data to be serialized or rendered into a preferably standard, format or syntax that can be parsed later and transformed into another format as per need of subsequent layers.

JSON representation of a set of *RDF* triples as a series of nested data structures has become increasingly popular as a data serialization format thanks to its more lightweight structure compared to XML, making it a useful format for data exchange in a way that requires less bandwidth than a bulky XML document. Thus, CERBERO choose JSON format as a primary format for data serialization. To enlarge compatibility with CERBERO tools that using XML as primary serialization format automatic XML-to-JSON and JSON-to-XML translators are provided.

However, JSON data format (especially after XML-to-JSON translation) are lacking several important features that are natural parts of underlining object models. For example, JSON format lacking possibility of referencing object that are defined in other parts of JSON document. To overcome difficulties introduced by the missing features we provide class definition extension for data. This extension provides following properties:

On the schema level:

- `"name" -` name of the schema. Since objects of the same class can be serialized with different data formats several different schemas can be provided to deserialize these objects. These schemas distinguishing by their names.
- `"representation"` – defines properties representation into JSON serialized object. Includes following sub-properties:
    - `"type"` – can be one of “key_value_base” / “property_base” / “mixed”. Describing data representation type: “key_value_base” is a native JSON representation, where JSON key describes property name and JSON value describes property value; “property_base” is a special representation often produced by XML-to-JSON converters, where each property defined by two JSON key-value pairs one having property name as its value and another having property value as its value; “mixed” is a representation where both types of representation are used.
    - `"base_key"` – parameter that defines JSON key which value defines class of JSON serialized objects. This value will be parsed according to corresponding schema all other JSON key-value pairs will be ignored.
    - `"key_prefix"` – parameter that defines prefix that should be stripped from JSON key on JSON-to-CIF conversion or added to JSON key on CIF-to-JSON conversion.
    - `"key_value_base"` – describes parameters specific for “key_value_base” representation of properties. Includes “base_key” and “key_prefix”.
    - `"property_base"` – describes parameters specific for “property_base” representation of properties. Includes “base_key”, “key_prefix” and two additional parameters:

    - `"property_name_key"` – describes which JSON key used by parser to identify key-value pair that defines property name as its value.
    - `"property_value_key"` – describes which JSON key used by parser to identify key-value pair that defines property value as its value.
- `"keys"` – define list of unique keys that allow to distinguish one object of corresponding class from another. Includes following sub-properties:
    - `"name"` – name of the key.
    - `"properties"` – defines list of properties which form unique identifier of the object of corresponding class.

On the property level:

- `"representation"` – defines representation of the specific property if schema has "mixed" type of representation.
- `"base_key"` – same meaning as "base_key" in representation description but applied only to specific property.
- `"key_prefix"` – same meaning as "key_prefix" in representation description but applied only to specific property.

On the object level:

- `"schema"` – defines name of the schema of the nested JSON object. Object will be parsed according to corresponding schema.
- `"extensible"` – true/false. Defines if nested object can represent a new object of corresponding class (true), or only reference to existing object of corresponding class (false).
- `"id_type"` – defines how nested object are identified and can be either:
    - `"object"` – nested object itself provided according to corresponding schema,
    - `"uid"` – reference to the existing instance in the CIF database provided as UID of the CIF instance,
    - `"key"`–indicates that only several properties are provided and set of provided properties includes at least properties enlisted in the unique key provided in the "id_key" property value,
    - `"key_property"` – indicates that provided value of a property that uniquely identify the object. In this case "id_key" property refers to key that based on one property only.
- `"id_key"` – required if "id_type" property has value "key" or "key_property" defines a name of the unique key in the corresponding schema.

CIF data interface expose exactly same methods as class base interface, however these methods gets schema name as additional parameter and using JSON or XML instead of Python dictionary objects.

5.5. Ontology Alignment and Equivalence Rules

Ontology alignment, or ontology matching, is the process of determining correspondences between concepts in ontologies. In the tool-integration context involving many tools providing their own ontologies, ontology matching has taken a critical place for helping heterogeneous tools to interoperate. CIF provides ontology alignment as set of the equivalence rules between objects of two or more classes. Equivalence rules allow automatic transformation of objects between different tools, levels and layers of abstractions. Set of the rules, describing all equivalence relations between objects of all classes of two different namespaces, represents a mapping between corresponding ontologies. Albeit, different tools and languages for ontology alignment exists, CERBERO found that existing tools and languages are not suitable for CIF simplified ontologies for various semantic and syntactic reasons. Thus, instead of trials to adopt existing tools and languages to ontology alignment CERBERO decided to develop new equivalence rules language based on mathematical background and language developed by IBM for metrics library. This choice supported by common principles that lies under SEMI developed by IBM and CIF developing by CERBERO. Moreover, specification of this kind of language can be further reused to develop KPI library language.

CIF provide following syntax of equivalence rules.

- Main rule syntax:
  `ns1:class1 operator ns2:class2 [*…] [ON …] [IMPLYING …];`

`ns1, ns2`	− names of namespaces
`class1, class2`	− names of classes
`operator`	− one of: ===, <==, ==>, <==>
`[*…]`	− optional multiplication part
`[ON …]`	− optional “on” part
`[IMPLYING …]`	− optional “implying” part
`;`	− termination symbol

- Optional multiplication part syntax:
  `* ns3:class3 | int_expression [*…]`

`ns3`	− name of namespace
`class3`	− name of class
`|int_expression`	− any integer expression that can be provided instead of `ns3:class3`
`[*…]`	− optional multiplication part

- Optional on part syntax:
  `ON (bool_expression [, bool_expression])`

`bool_expression`	− any bool expression
`[, bool_expression]`	− optional additional comma-separated bool expressions

- Optional implication part syntax:
  `IMPLYING (implication [, implication])`
- Implication syntax 1:

```
ns1:class1.property_expr1 operator
ns2:class2.property_expr2 [*…] [ON …] [IMPLYING …]
```

`ns1, ns2`	– names of namespaces
`class1, class2`	– names of classes
`operator`	– one of: ===, <==, ==>, <==>
`property_expr1, property_expr2`	– expressions defining (sub)properties names
`[*…]`	– optional multiplication part
`[ON …]`	– optional "on" part
`[IMPLYING …]`	– optional "implying" part

- Implication syntax 2:

```
ns1:class1.property_expr1 = gen_expression
```

`ns1`	– name of the namespace
`class1`	– name of the class
`property_expr1`	– expression defining (sub)property name
`gen_expression`	– general mathematical expression

Equivalence rule semantics.

Semantics of main rule operators (void multiplication part).

===	- means that corresponding classes are equivalent, i.e. each instance of class 1 is also instance of class 2 and vice versa.
==>	- means that class 2 equivalent to class 1, i.e. each instance of class 2 is also instance of class 1, but instance of class 1 is equivalent to instance of class 2 only if both met matching criteria provided in "on" part.
<==	- means that class 1 equivalent to class 2, i.e. each instance of class 1 is also instance of class 2, but instance of class 2 is equivalent to instance of class 1 only if both met matching criteria provided in "on" part.
<==>	- means that instance of class 1 is equivalent to instance of class 2 only if both met matching criteria provided in "on" part.

Semantics of multiplication part.

Multiplication part change equivalence rules operator semantics in the following sense:

- when multiplication part contains class reference this means that class 1 equivalent to cartesian product of instances of class 2 and class 3, each instance of class 1 has two different instances (one of class 2 and one of class 3) as his counterpart with respect to corresponding relation operator.
- when multiplication part contains integer expression this means that each instance of class 1 corresponds to number of instances of class 2, and this number defined by integer expression that may depend on properties of corresponding instances.

Semantics of "On" part.

"On" part can include several logical expressions that treated as matching criteria between instances that are equivalent according to corresponding rule. These expressions can be treated as one single expression with "and" operator between corresponding parts.

Semantics of implications.

Implication of kind 1 (syntax 1) can be treated as nested equivalence rule and define equivalence relations between property values of instances of corresponding classes. Implication of kind 2 (syntax 2) define property value that should be assigned during rule execution process. This value is a result of calculation of general mathematical expression.

6. Direct Tool Integration Holistic Methodology

Figure 4.3 and Figure 4.4 illustrate the CERBERO framework for design-time and run-time support respectively. Some of the integration was already described in previous deliverables. For the sake of conciseness those connections are listed here in after, but not explained any further:

1. DynAA – MECA (D6.10)
2. ARTICo^3 – MDC – CAPH (see D5.7)
3. PREESM – Spider – PAPIFY/PAPIFY-Viewer (see D5.7)
4. PREESM – ARTICo^3 (see D5.4)
5. MDC – HLS Tools (see D5.4)
6. SPIDER – MDC (see D5.5)

The rest of this Section illustrates new achieved direct tool-to-tool connections or advancement in the already illustrated ongoing ones.

a. PREESM – SPIDER – MDC
b. PAPIFY – MDC
c. SAGE-ReqV – DynAA
d. PAPIFY – ARTICo^3
e. IMPRESS - ARTICo^3
f. ARTICo^3 – PREESM
g. SPIDER – PAPIFY –MDC
h. ARTICo^3– MDC – PAPIFY
i. APOLLO – PREESM

In the previous deliverable, we foresaw the possibility of connecting SAGE with MDC, but as said before, we opted to connect MDC to CIF. We still believe that MDC could benefit from a connection to higher level abstraction tools, like SAGE (i.e. for automatic test generation functions) or AOW (i.e. to exploit continuous linear programming to perform a faster exploration of the design space). Nevertheless, the implementation of direct connections seemed to be too time consuming, since all the mentioned tools operate on different Models of Computation. Therefore, we invested time on creating proper schema to connect MDC with CIF to improve its interoperability with higher level tools.

Also, there is still another integration to be performed, based on combining the composability provided by ARTICo^3 with the capability of changing functionality in specific overlays obtained by JIT composition. In other words, whenever a JIT composed design (either using the deterministic approach or using the iterative/evolutionary options) is to be used, it will be embedded into a specific ARTICo^3 slot. An example overlay based on a block-based neural network is being presently developed and refined as an example and, for later versions of the demonstrations to be achieved in CERBERO, this overlay will be embedded in an ARTICo^3 slot and exchanged with other possibilities. This double level granularity will allow using different types of overlays, and exchange them according to specific needs.

6.1. Integrating PREESM and SPIDER with MDC

SPIDER (a dataflow-based runtime manager for multi-/many-core architectures) and MDC (a dataflow-to-hardware synthesizer for coarse-grained reconfigurable systems) have been combined in order to manage software and hardware reconfigurability at runtime. This work has seen the collaboration among INSA, UNICA and UNISS.

As depicted in Figure 6.1, the proposed design flow considers:

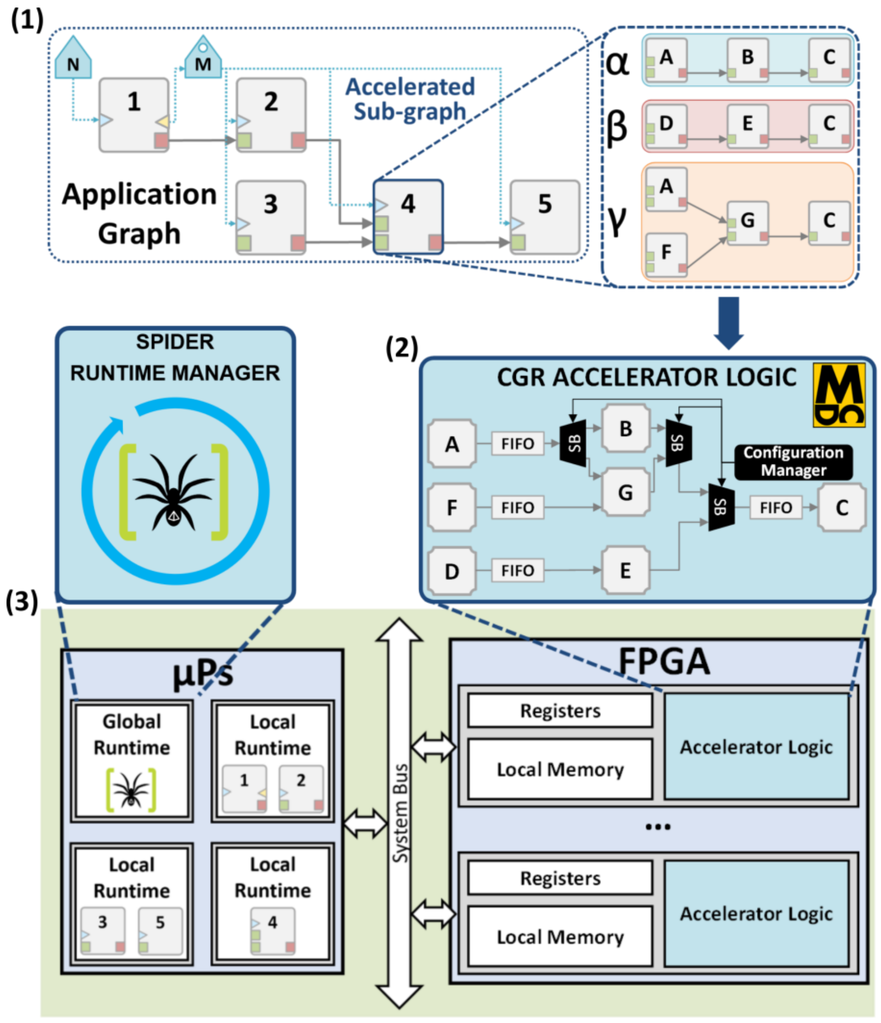


Figure 6.1: PREESM-SPIDER-MDC design flow

(1) a PiSDF-based description of the application developed using PREESM, that implies some reconfigurable workloads requiring hardware acceleration with low reconfiguration overheads (actor 4). The functionalities of these specific workloads (α,β,γ) can be implemented in hardware and handled by a parametric software actor in the high-level graph of the application.

(2) This actor initializes and exchanges data with the reconfigurable processing element, generated as a coarse-grained reconfigurable accelerator using MDC, and capable to perform all the single functionalities depending on the value of the dynamic input parameter (such as M) related to its configuration.

(3) One or multiple instances of such accelerator can be deployed onto an FPGA device implemented in a system on chip including micro-processors (such as Xilinx Zynq-7000). Thus, depending on the adaptation strategies, SPIDER schedules and maps at runtime the whole application graph, and sends job orders to the processing elements.

6.2. Integrating PAPIFY with MDC

The objective of this integration is to provide monitoring support to HW accelerators developed using MDC. As a result, both SW and HW Processing Element (PE) resources will be monitored through the same interface, PAPIFY. The PEs are resources where one or more actors are scheduled for the execution. It can be a SW core or a complete HW accelerator. PAPIFY provides an interface to access to performance monitoring information of the different PEs existing in the target platform (see D5.7). MDC, on the other hand, provides reconfigurable HW accelerators, which can be composed of one or several dataflow networks (see D5.7). Each actor of the dataflow network is implemented in HW as a custom Functional Unit (FU). To integrate them, both a PAPI component and the Performance Monitoring Counters (PMCs) have been developed for MDC (see Figure 6.2).

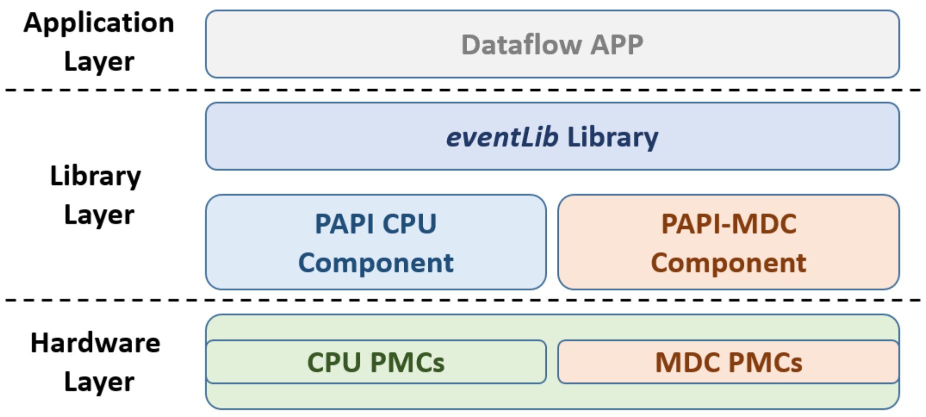


Figure 6.2: Custom PAPI component and Performance Monitoring Counters (PMCs) for MDC.

Specifically, the HW monitoring can be done on two levels of abstraction, as shown in Figure 6.3:

(1) accelerator-level: the monitoring is homogeneous for every accelerator that can be implemented using MDC, and in particular it keeps trace of important dataflow metrics during execution, such as the execution time, the number of input tokens and the number of output tokens.

(2) low-level: the monitoring is specific for the current accelerator, e.g. it is identifying the bottleneck FUs internally.

The accelerator-level monitors are automatically inserted by MDC, while the low-level monitoring still requires manual steps to be used within the code generation flow.

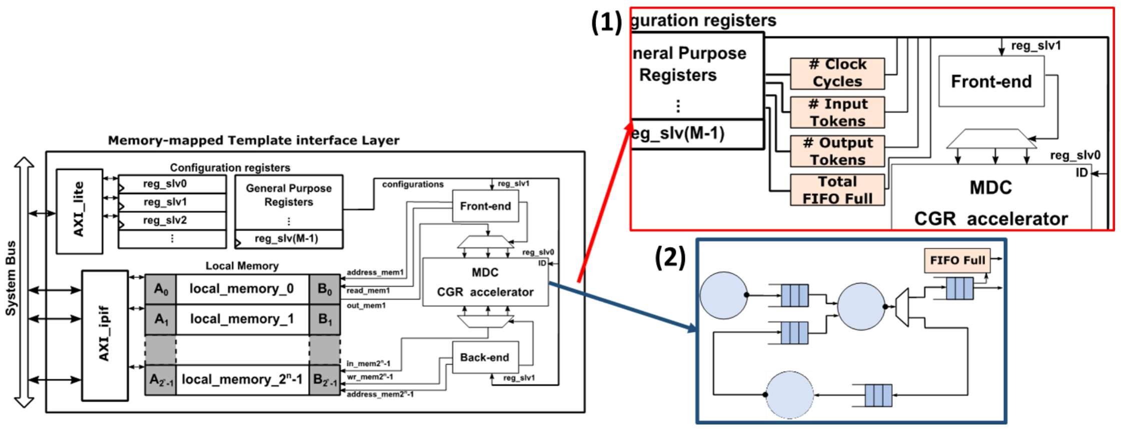


Figure 6.3: HW Monitoring at two level of abstraction: (1) accelerator level; (2) low-level.

Since the base address of the accelerator may change from one accelerator to one other, as well as the number and type of events to be monitored, we developed a configurable PAPI-MDC component that is automatically configured when the application is launched.

The PAPI-MDC component is compliant with the standard already existing SW components and can be naturally accessed by PAPIFY. The monitoring strategy is based on starting the monitoring, launching the actor execution and stopping the monitoring. In order to support this flexibility, the PAPI component is automatically configured, by means of an XML file, when the application is launched and the specific available monitors for the accelerator under evaluation are loaded. The XML file, automatically generated by MDC, specifies the physical base address of the accelerator to be monitored (baseAddress), the number of available events (nbEvents) and their type (event), but this approach can be easily extended to consider other variables.

```
<?xml version="1.0" encoding="UTF-8"?>
<mdcInfo>
        <baseAddress>0xADDRESS</baseAddress>
        <nbEvents>N</nbEvents>
        <event>
                <index>M</index>
                <name>MDC_EVENT_NAME</name>
                <desc>Event Description</desc>
        </event>
</mdcInfo>
```

6.3. Integrating SAGE-ReqV and DYNAA

The goal of the integration between ReqV and DynAA is the verification of the DynAA model with respect to the requirements introduced and verified in ReqV. The requirements in ReqV describe behaviors over time and can contains numerical constraints and boolean signals. ReqV formalizes such requirements in Linear Temporal Logic (LTL). The main issue for the verification of DynAA models is that it contains continuous signals, while most of the available verification tools for LTL are designed for discrete models.

The proposed solution is to use a well-established language, supported by off-the-shelf verification tools, to model the system at a high-level of abstraction, representing the discrete evolution of the system, and to provide a sound and semantically equivalent translation to DynAA Java code, letting the user refine the model with the continuous part. This idea is similar to the C++/Java code generators from UML diagrams.

Promela, the modeling language adopted for this purpose, is expressive enough to support the models related to CERBERO use cases and can be automatically analyzed by the Spin Model Checker.

In the following, we present a brief introduction to Promela and the proposed translation into DynAA components.

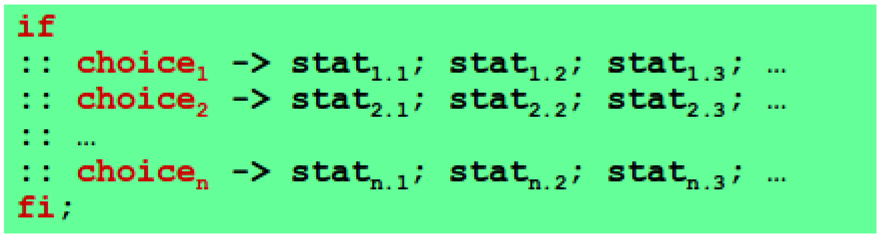
Promela And Spin

Promela (Process/Protocol Meta Language) is a verification modeling language that can be analyzed with the Spin Model Checker (http://spinroot.com) to verify the correctness of the modeled system with respect to a given specification. Promela has a C-like notation for specifying system design or its finite-state abstraction unambiguously. It is especially suited to model concurrent and parallel systems.

A Promela model consists of three types of objects: processes, channels and variables.

A process is identified with the **proctype** keyword and it can accept channels references or variables in input. Processes can be instantiated with the **run** command or the **active [N]** modifier (to automatically instantiate N objects of the same type). A process body can contain loops, if statements, expressions, assignments, etc.

The syntax and semantics of the if and do statements differ from the one used in C programs:



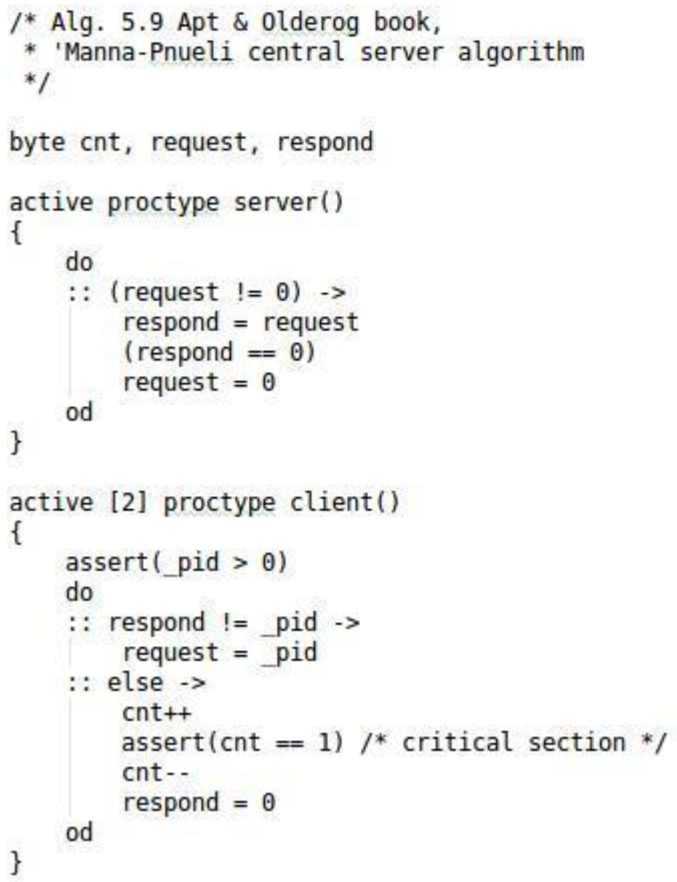
```
if
:: choice_1 -> stat_1.1; stat_1.2; stat_1.3; …
:: choice_2 -> stat_2.1; stat_2.2; stat_2.3; …
:: …
:: choice_n -> stat_n.1; stat_n.2; stat_n.3; …
fi;
```

Figure 6.4: If/do statement in Promela

The if/do statement is composed of one or more guards (choice_i), followed by a sequence of other statements. The program chose to execute non-deterministically one of the branches with executable guard. If no guard is executable, the if/do statement is blocked. The else guard became executable if none of the other guards is executable. Finally, the do statement behaves in the same way as the if statement, but at the end of the chosen list of statements it repeats the choice selection, until a break statement is found.

Channels are special objects that model communication channels and can be used to exchange messages between processes. A channel can be defined to be synchronous (i.e., *rendez-vous*), or asynchronous (i.e., buffered).

Variables declaration works as in C and the supported types are **bit, bool, byte, short, int, mtype** (symbolic user-defined constants) or arrays. More complex types can be declared with **typedef**, in the same way as C's structures.



```
/* Alg. 5.9 Apt & Olderog book,
 * 'Manna-Pnueli central server algorithm
 */

byte cnt, request, respond

active proctype server()
{
    do
    :: (request != 0) ->
        respond = request
        (respond == 0)
        request = 0
    od
}

active [2] proctype client()
{
    assert(_pid > 0)
    do
    :: respond != _pid ->
        request = _pid
    :: else ->
        cnt++
        assert(cnt == 1) /* critical section */
        cnt--
        respond = 0
    od
}
```

Figure 6.5: Example of a Promela Model with two proctypes, a client and a server, and three running processes (two clients and one server).

Figure 6.6 shows an example of a Promela program implementing the central server algorithm.

For a complete specification of the language syntax and semantics, we remand the reader to the official documentation (http://spinroot.com/spin/Man/promela.html).

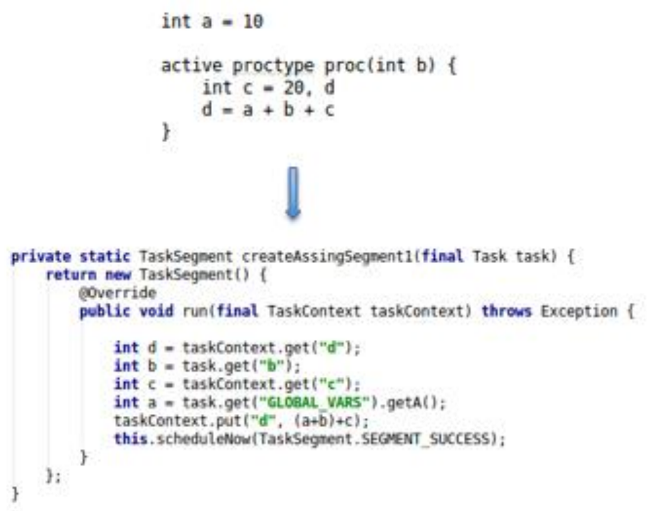
Given a model system specified in Promela, Spin can generate a C program that performs an efficient online verification of the system's correctness properties. The properties can be specified as invariants or linear temporal logic formulae. Spin also checks for the absence of deadlocks, unspecified receptions, and unexecutable code.

Promela to DynAA Translation

Every proctype in Promela is represented as a **Task** object in DynAA, and every Promela statement is represented as a DynAA specialized **Segment** (currently expressions, if/do statements, assignments and asserts are supported).

The variables defined in Promela are stored in different places, depending on their scope:

- **Global variables** are stored in the **DataContainer** shared object
- **Proctype parameters** are stored as **Task** properties
- **Local variables** are stored in the **TaskContext** object



```
int a = 10

active proctype proc(int b) {
    int c = 20, d
    d = a + b + c
}
```

```
private static TaskSegment createAssingSegment1(final Task task) {
    return new TaskSegment() {
        @Override
        public void run(final TaskContext taskContext) throws Exception {

            int d = taskContext.get("d");
            int b = task.get("b");
            int c = taskContext.get("c");
            int a = task.get("GLOBAL_VARS").getA();
            taskContext.put("d", (a+b)+c);
            this.scheduleNow(TaskSegment.SEGMENT_SUCCESS);
        }
    };
}
```

Figure 6.6: Example of Promela to Java translation provided by DynaaGen.

The translation is automated with a tool called **DynaaGen** (https://gitlab.sagelab.it/sage/dynaagen), part of the SAGE Verification Suite. The tool takes in input a Promela file and automatically generates the Java Source Code with the DynAA classes corresponding to the same modelled system (an example is shown in Figure 6.6).

In conclusion, requirements, checked and encoded into LTL formulae by ReqV can be used to verify the Promela model's correctness with the Spin model checker. The same model, then, can be translated with DynaaGen to build a correct-by-construction model in Java. At this point, the designer can further refine the model and run simulations with DynAA. Finally, this approach has also the desired side effect to reduce the size of model written by the designer because, as shown in the previous section, Promela has a much more compact syntax compared to the Java counterpart. This gives the designer the possibility to focus more on the functional behavior of the system and increase her confidence in the final system correctness.

6.4. Integrating PAPIFY with ARTICO^3

The objective of this integration is to provide access to PMCs located on (1) CPUs and (2) on the FPGA when ARTICo³ infrastructure is in charge of dispatching HW task on the different slots of the Programmable Logic.

In order to achieve the goal, the same approach adopted for the integration of MDC – PAPIFY (see section 6.2 of D5.1) was followed. Specifically:

1. A new ARTICo³ - PAPI component was designed;
2. An XML extension, consistent with the one adopted for MDC with PAPIFY, was designed, together with an XML parser;
3. Additional ARTICo³ runtime functions, embedded in a dynamic library, were developed.

The PAPI component developed does not have direct access to the PMCs. Instead, it calls the specific runtime function embedded within the ARTICo³ shared library. This step was necessary to keep the PAPI component independent from the hardware description (i.e. no HW address are managed within it). Moreover, in that way, the ARTICo³ software infrastructure is aware that an external library is accessing its registers and can manage the data to be read. In Figure 6.7 we show that the PAPI component develop is not directly connected with the HW and needs other low-level, hardware-specific functions. In other words, the PAPI components does not manage the hardware but delegate its management to a proper library.

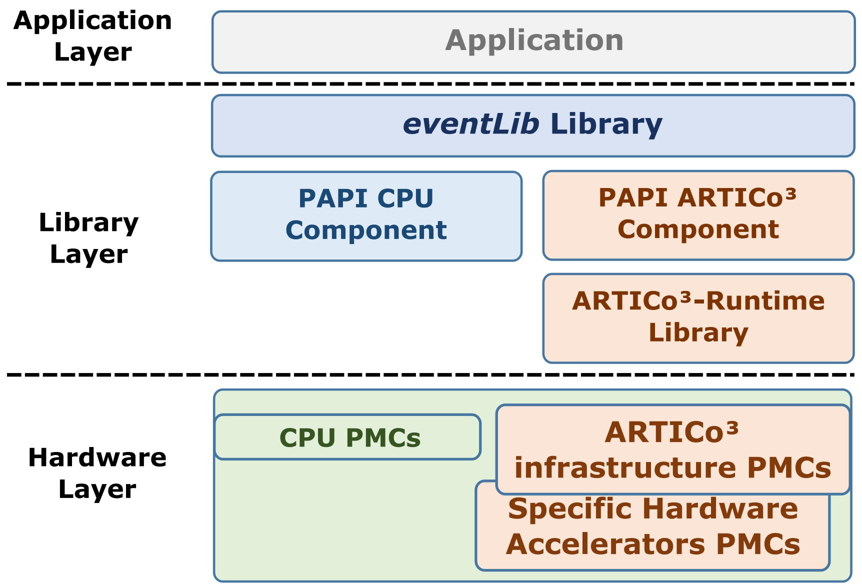


Figure 6.7: Custom PAPI component and Performance Monitoring Counters (PMCs) for ARTICo³ infrastructure.

Depending on the specific application to be monitored (note: the application can be Dataflow or not), the PMCs developed at design time may differ from one project to the other. For this reason, the same approach followed in section 6.2 was included in an extension of ARTICo^3-PAPIFY integration, as explained in [10], where the ARTICo³ component automatically configures itself by reading an XML file which describes the hardware monitors within:

1. The ARTICo³ infrastructure.
2. Every specific ARTICo³ hardware accelerator (that can be loaded at runtime on the available slots).

As already mentioned, following the same approach to provide a consistent methodology for accessing instrumented HW, an XML PAPI-ARTICo³ was also designed in order to be compliant with PAPI and PAPIFY from one side and with ARTICo^3 accelerators from the other. The following listing shows the structure of this xml description:

```
<?xml version="1.0" encoding="UTF-8"?>
<artico3Info>
        <nbEventsArtico3>N</ nbEventsArtico3 >
        <eventArtico3>
                <index>M</index>
                <name>ARTICo3_EVENT_NAME</name>
                <desc>Event Description</desc>
        </eventArtico3>
        <nbKernels>K</nbKernels>
        <kernel>
                <kernelName>ARTICo3_KERNEL_NAME</kernelName>
                <nbEvents>N</nbEvents>
                <event>
                        <index>M</index>
                        <name>ARTICo3_KERNEL_EVENT_NAME</name>
                        <desc>Event Description</desc>
                </event>
        </kernel>
        <nbEvents>N</nbEvents>
</artico3Info>
```

It can be noted that there are two different types of events:

- the generic events associated to the ARTICo³ infrastructure (always present) such as ERRORS and CLOCK-CYCLES.
- Specific kernel events associated to particular hardware accelerators. They are not part of the ARTICo³ infrastructure but, instead, they are part of the internal structure of the accelerators (when designed). For the specific case of MDC accelerators into ARTICo^3 containers, these registers may be accessed via an MDC into ARTICo^3 encapsulated XML file. This case will be shown in the PAPIFY-ARTICo^3-MDC section.

The concept is graphically explained in the following Figure 6.8, where we highlight the two different kind of event and the location of the associated PMCs.

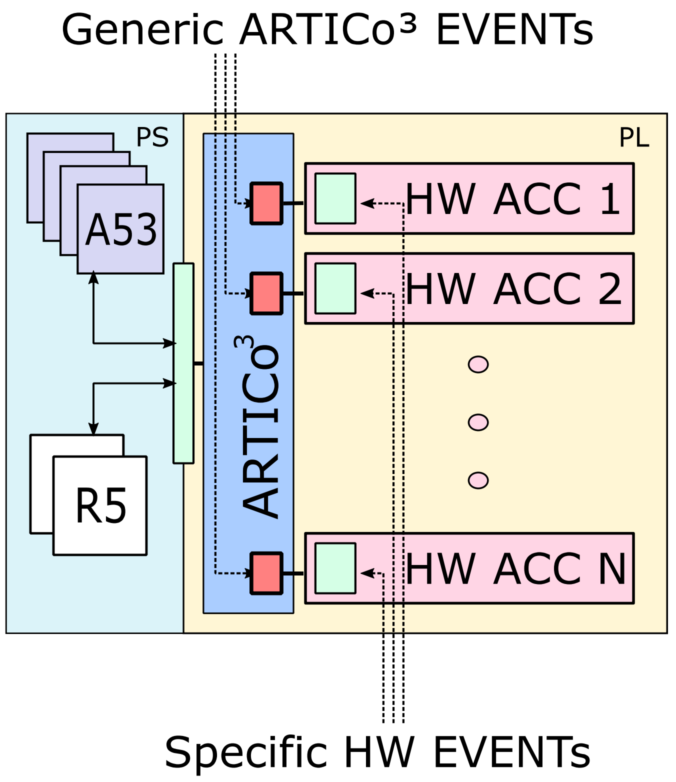


Figure 6.8: Generic Event and Specific Event within the ARTICo³ infrastructure

The two only functions that are embedded within the PAPI component are:

```
/**
 *
 * @param eventName : the name of the PAPI event used
 * @return          : the value read
 *
 */
uint32_t genericEvent(char * eventName)


/**
 *
 * @param eventName : the name of the PAPI event used
 * @return          : the value read
 */
uint32_t specificEvent(char * eventName)
```

As PAPIFY (EventLib) is naturally interfaced with the PAPI library, the ARTICo³ PMCs are easy accessible making use of it. The developed functions were needed to guarantee the compatibility with no action on the upper software layer.

6.5. Integrating IMPRESS with ARTICo^3

As explained in deliverable 5.2 IMPRESS is the tool that provides the reconfiguration features needed for just-in-time HW composition. It is possible to leverage the reconfiguration capabilities provided by IMPRESS to improve ARTICo^3 with the following features:

- **Relocation**. This feature permits to use one partial bitstream for each compatible slot (i.e., slots with the same resource footprint). For example, in a scenario where there are *m* kernels and *n* compatible slots, without relocation *m*n* partial bitstreams are needed. With relocation only *m* partial bitstreams are necessary. In the case of a multi-kernel scenario, relocation adds more flexibility to the scheduling and placement of the accelerators on the available reconfigurable slots.
- **Decoupling the static and reconfigurable designs**. In contrast to Xilinx reconfiguration flow, IMPRESS decouples the static and reconfigurable designs. This allows to have several pre-built static configurations of ARTICo^3. This way a user can use one of these predefined configurations and just generate a kernel independently, without re-implementing the static part.
- **Sub-clock region slots**. This allows to use slots with a finer granularity as multiple slots can be stacked in the same clock region or combining coarse (slot-based reconfiguration) with extended features, such as fine grain reconfiguration.

ARTICo^3

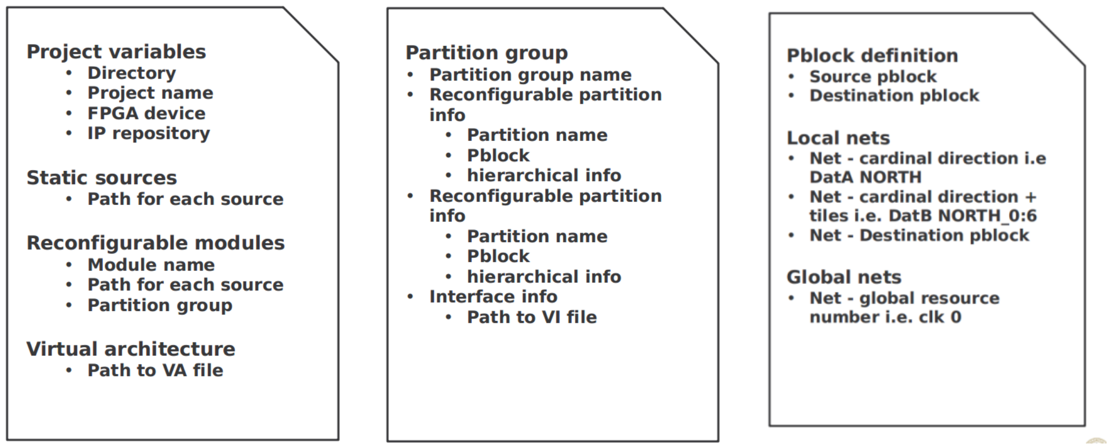
When a user wants to implement a kernel in ARTICo^3 the first step is to select a Reference Design, also known as template. The ARTICo^3 toolchain is highly modular, since it builds around a common core (based on Python scripts), but then relies on these templates to apply device- or board-specific customizations in the hardware designs. From all the files that define the template, two play a relevant role reconfiguration-wise:

- A constraints file that defines the floorplanning of the FPGA (i.e., the number and location of the reconfigurable slots).
- A set of TCL scripts that implement the reconfigurable multi-accelerator system using the Xilinx Partial Reconfiguration Flow and the information of the template.

IMPRESS

IMPRESS is a set of TCL scripts that extends the features offered by the Xilinx reconfiguration flow. IMPRESS needs three input files to define a reconfigurable design. The first file is called project_info and contains all the information related to the project (i.e., FPGA device, reconfigurable and static sources, etc.). The second file is called virtual_architecture and contains information of the floorplanning and compatible partitions. The third file contains the interface of each partition.

Figure 6.9 shows the contents of each file.



Project variables
- Directory
- Project name
- FPGA device
- IP repository

Static sources
- Path for each source

Reconfigurable modules
- Module name
- Path for each source
- Partition group

Virtual architecture
- Path to VA file

Partition group
- Partition group name
- Reconfigurable partition info
  - Partition name
  - Pblock
  - hierarchical info
- Reconfigurable partition info
  - Partition name
  - Pblock
  - hierarchical info
- Interface info
  - Path to VI file

Pblock definition
- Source pblock
- Destination pblock

Local nets
- Net - cardinal direction i.e DatA NORTH
- Net - cardinal direction + tiles i.e. DatB NORTH_0:6
- Net - Destination pblock

Global nets
- Net - global resource number i.e. clk 0

a) Project_info b) virtual_architecture c) interface

Figure 6.9: IMPRESS input files

ARTICo^3 + IMPRESS integration

In order to integrate both tools, the custom scripts that ARTICo^3 used to implement a reconfigurable design have been replaced by IMPRESS scripts. The floorplanning definition of each template has been changed for the virtual_architecture and interface files. As happened before the integration, these files have to be filled in manually by the user (although they could eventually be automatically generated by the core scripts in the ARTICo^3 toolchain). However, the project_info contents are now generated automatically by the ARTICo^3 toolchain as this information is already described in the ARTICo^3 project configuration.

To use the partial bitstreams generated by IMPRESS, it has also been necessary to modify Linux reconfiguration drivers to implement the reconfiguration engine used by IMPRESS.

6.6. Integration ARTICo^3 with PREESM

As described in section 3.1.1 of deliverable D4.3, it is more than frequent to have a tightly couple of FPGA accelerated HW functions together with SW, all wrapped with an operating system. In such architecture, the system can manage at the same time software threads as well as hardware threads. A big challenge within the CERBERO project is to have the possibility of easily design an application (a SW application running upon an operating system) that efficiently exploits the different HW computing fabrics (namely ARTICo^3, MDC and JIT). Specifically, this section addresses, the integration between ARTICo^3 and PREESM.

As a contribution of the collaboration between INSA and UPM, it is possible to account with newer versions of PREESM, where the capability to easily exploit the SW and HW tasks combination by offloading the chosen software task to the ARTICo^3 slot accelerators has been added. The heterogeneous execution of the whole application therefore, has:

- SW threads running on the available CPU described in the Architecture.
- SW threads that delegate the execution of “pieces of code” (i.e. instances of a Dataflow Actor) to the FPGA programmable logic.

The strength of PREESM is to give the possibility of independently describe (1) the algorithm (by using the PiSDF (see Desnos[11]) and (2) the architecture (by using the S-LAMin Pelcat [12]). Thus, PREESM will be in charge of finding an optimal solution by mapping and scheduling every instance of actors of the PiSDF up on the S-LAM.

Following, Figure 6.10 the possible elements of the S-LAM that can be used to describe any architecture are reported:

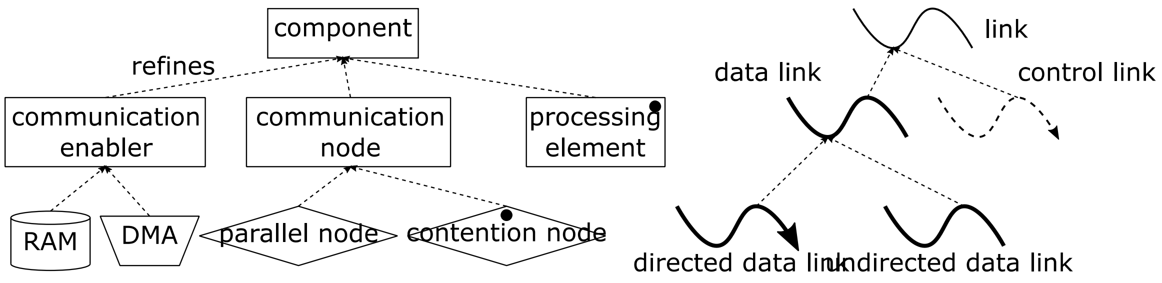


Figure 6.10: Elements of the S-LAM [12]

To better explain the concept, two examples are here given:

1. In Figure 6.11, a SLAM description of three ARM core plus one ARTICo³ processing element is shown (bottom left).
2. In Figure 6.12, a SLAM description of dual ARM core plus four ARTICo³ processing element is shown.

It is worth to note that no other elements were added to the S-LAM typology reported above in order to describe a heterogeneous system with hardware accelerators.

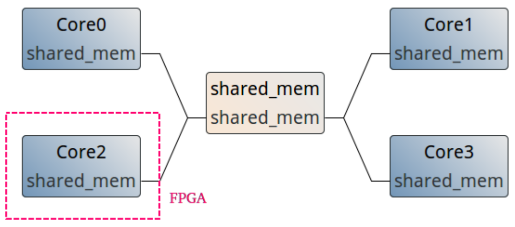


Figure 6.11: Three ARM cores and one ARTICo³ SLOT

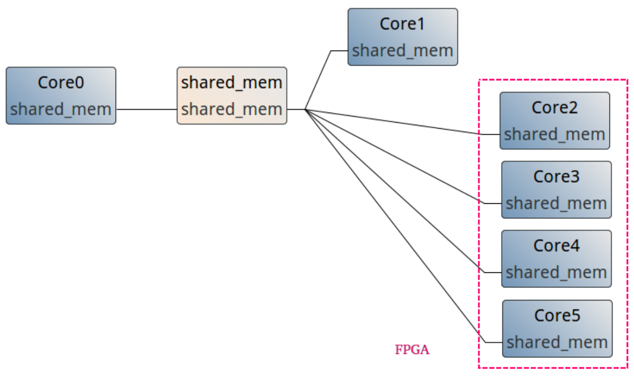


Figure 6.12: Two ARM cores and four ARTICo³ SLOT

From the user's standpoint, he/she should just describe the application (using the PiSDF representation [11]) and the architecture (using the S-LAM representation) including at least one core of ARTICo³ type in the S-LAM figures shown above.

If a new ARTICo³ infrastructure should be added in the S-LAM, these two main concepts must be followed:

1. Every slot of the architecture is a processing element within the S-LAM.
2. All the slots communicate with the main memory of the system.

The code generator that has been embedded into the last versions of PREESM 'prints' (produce the code for) all the necessary instructions to manage the accelerators, send and receive data. All the in-out buffers (ARTICo³ internal memory banks) are properly generated and automatically handled, and a similar process is also done with ARTICo³ registers. Heterogeneous executions were tested with several hardware/software configurations. From the user's standpoint, it is remarkable that very little hardware background is needed, since the process of 'printing' is achieved automatically.

To describe how the hardware code generator of PREESM works, it is worth to be noted that three of the inputs of the generator are:

1. The S-LAM model.
2. The Directed Acyclic Graph (DAG) of the application.
3. The Scenario (explained after).

The DAG is derived by means of graph transformations from the PiSDF, the S-LAM describes the architecture and, by means of the Scenario, some constraints can be added. In this case, we can specify which actor can be executed on the ARTICo³ architecture. Of course, only the actor with the corresponding hardware accelerators is going to be selected as possible candidate to be moved on the Programmable Logic (PL).

Within the Code Generator, an intermediate model is created. Every object of this intermediate model within the PREESM Code Generator, is associated to a specific "printer". The new object created for this purpose are:

- The *FPGA load bitstream object*: this printer is in charge of printing the instruction for charging the bitstream of the accelerator in the slot of ARTICo³.
- The FPGA register *setting*: this printer is in charge of printing the instruction necessary for setting up the register of the hardware accelerators when needed.
- The *FPGA data transfer*: this printer is in charge of printing the instruction necessary to move the data to/from the Processing System from/to the Programmable Logic.
- The *FPGA start execution*: this printer is in charge of printing the instruction necessary to give the start signal to the FPGA to start the computation using the data previously charged within the slots.

All the printed instructions are part of the ARTICo³ runtime library. This way, the PREESM's generated software will be in charge of automatically manage the SW-based Processing as well as HW-based processing (by interfacing with the ARTICo³ runtime library [13]). In Figure 6.13 , a schematic view of the heterogeneous system with SW-based Processing threads and HW-based Processing delegate-thread is shown:

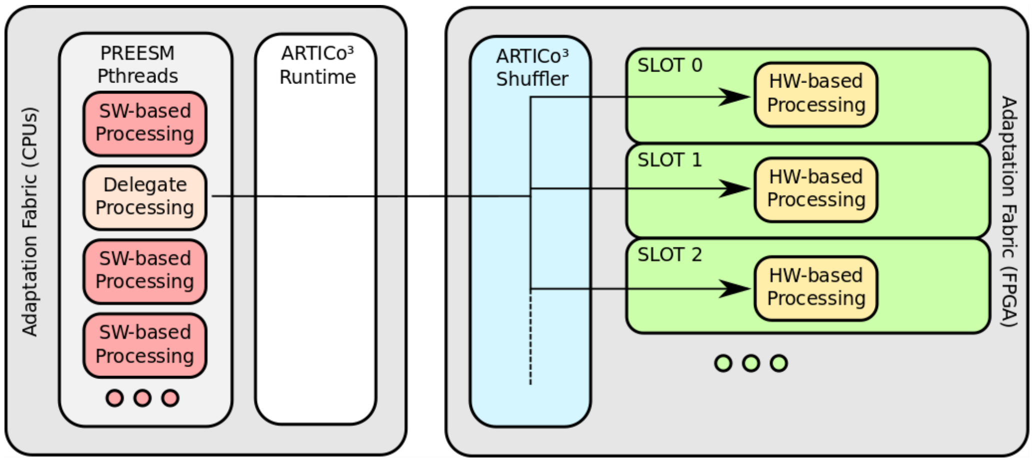


Figure 6.13: A graphical view of the Delegate SW thread offloading computation to the ARTICo³ slots

Thanks to the S-LAM semantics [12], an architecture making use of ARTICo³ can be described. The instance of the actors within the PiSDF is then mapped and scheduled upon the architecture described here.

6.7. Integrating Spider with PAPIFY and with MDC

The Spider – PAPIFY – MDC integration is the natural extension of the standalone integration of Spider-MDC, Spider-PAPIFY and PAPIFY-MDC.

In the SPIDER-MDC integration, SPIDER is capable to configure the CGR accelerators generated by MDC to compute their different functionalities. Depending on the adaptation strategy, SPIDER schedules and maps, at runtime, the whole application graph composed of software tasks (including those that manage the communication with the accelerators) and sends these latter to the slave processors.

In the SPIDER-PAPIFY integration, it has been demonstrated that, in an application monitoring configured using PREESM framework from which the code compliant with the SPIDER run-time manager has been generated, the user is able to decide how many CPU cores the system will use and to monitor the workload distribution during the system execution using PAPIFY-Viewer.

Additionally, it has been explained that PAPIFY-MDC integration has been conducted to offer a transparent monitoring approach from the user point of view (PAPIFY accesses both the SW and the HW PMCs in the same way). When we replaced the monitored MDC accelerators in applications adopted to validate the SPIDER-MDC integration, the same SW application was able to manage both the standard MDC accelerator and the monitored one (please, pay attention that, at this step, we did not read the monitors). After feeding PREESM with the XML file containing the PAPI-info updated to consider also the MDC-PAPI component, the new generated SW application, which includes also the required function calls to access the MDC monitors, was able to access the HW monitors in the

MDC accelerator as well as the standard PMCs in the CPU. Thus, the integration of SPIDER – PAPIFY – MDC came out without any further modification of the tools.

6.8. Integrating ARTICo^3 and MDC with PAPIFY

The integration of ARTICo³ and MDC was successfully achieved and explained in detail in D5.7 [14]. Besides, the integration of the monitoring strategy that makes use of the software layers PAPI and PAPIFY with (1) MDC and (2) ARTICo³ was also successfully achieved and discussed in previous sections. However, to make the three tools communicate, further modifications where necessary in the MDC *Kernel Adapter*, that wrap the logic generated by MDC into an ARTICo^3 compliant kernel. Indeed, the previous *Kernel Adapter* could not be aware of the extra logic inserted to monitor the selected event. Now that logic is kept into account, as well as the extra configuration registers exploited to communicate the values of the monitors.

The integration of PAPIFY with ARTICo³ evolved from the first version presented in [10] to the one above described where an approach that makes use of XML was adopted in order to be compliant with the new MDC PAPI component [15]. The advantage of such solution is to have a PAPI component that configures itself when the corresponding XML file is read.

The XML that describes the PMCs of ARTICo³ was designed keeping in mind that some slots of the hardware infrastructure may host an MDC accelerator. In that case, the XML description looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<artico3Info>
        <nbEventsArtico3>N</ nbEventsArtico3 >
        < eventArtico3>
                <index>M</index>
                <name>ARTICo3_EVENT_NAME</name>
                <desc>Event Description</desc>
        </ eventArtico3>
        <nbKernels>K</nbKernels>
        <kernel>
                <kernelName>ARTICo3_KERNEL_NAME</kernelName>
                <baseAddress>0xADDRESS</baseAddress>
                <nbEvents>N</nbEvents>
                <event>
                        <index>M</index>
                        <name>MDC_EVENT_NAME</name>
                        <desc>Event Description</desc>
                </event>
        </kernel>
        <nbEvents>N</nbEvents>
```

```
</artico3Info>
```

This means that:

- The description of generic events of the whole ARTICo³ infrastructure are, logically, not affected. No changes are needed for the generic ARTICo³ events.
- The description of specific events that belong to every specific kernel must reflect the hardware register structure. A specific case is given by the use of MDC accelerators: the registers are read/written using ARTICo³ API and, so, naturally accessible by PAPI and PAPIFY.

In summary, this section highlights the process that allows PAPI-PAPIFY to work with (1) MDC accelerators, (2) ARTICo³ architecture and, also, (3) the combined functionality of ARTICo³-MDC. The natural integration of the tools makes, so, PAPIFY a valid monitoring instrument for multi-grain reconfigurable architecture by means of this triple integration.

6.9. Integrating CERBERO Tools with External Tools

CERBERO tools have been also integrated with external tools (that means tools not developed in the CERBERO project.

APOLLO-PREESM

Dataflow applications are modeled as a set of *actors* interconnected through a set of FIFOs, used for sending and receiving information in a streaming fashion. One of the main advantages of modeling an application with a dataflow model of computation is that the inherent structural parallelism can be exploited. However, this parallelism extraction is kept within the limits of the application structure: since actors are considered as primitives, their behavior remains untouched. As a result, some optimization opportunities might be missed.

At this point is where the polyhedral model can become useful. This model is well-known for applying transformations to optimize computationally-intensive applications, focusing on aspects such as data locality or memory usage. However, the tight restrictions imposed by the polyhedral model force most of the tools to work at compile-time, as Polly [16], Graphite [17] or Pluto [18]. To the best of our knowledge, the only tool that extends the polyhedral scope to overcome these limitations is APOLLO (Automatic speculative POLyhedral Loop Optimizer) [19].

APOLLO applies polyhedral optimizations on-the-fly to loop nests (*for* loops or any kind of loop nests) that cannot be optimized at compile time. To do so, APOLLO relies on a speculative system that builds a prediction model to support dynamic transformations.

To efficiently combine PREESM with APOLLO, a mechanism called multi-versioning has been included in APOLLO to test different transformations and choose the most efficient one during the execution of the first iterations of the dataflow loop. In addition, to enable the execution in parallel of several actors, a thread-safe version of APOLLO has been made which includes libraries such as Pluto and Piplib.

Therefore, the combination of a dataflow framework as PREESM with a tool like APOLLO leads to exploit the hidden optimization possibilities within the actors in dataflow models. This integration has been achieved.

7. Conclusion

7.1. Lesson Learned

In the Table 7-1 below we have compared the two approached (CIF and Tool to Tool connection) used for tools integration in CERBERO in order to highlight the lesson learned and to give guidelines for the after-CERBERO project future.

The first lesson is that the two approaches are not in competition, but they have solved and simplify different issues related to integration of CPS tools. In general, CIF turned out to be effective when tools using different model of computation or placed at different level of abstractions needed to be connected to each other. On the contrary, mainly at the computing level, for code instrumentation or code generation direct integration of new back-ends into already existing tools was more straightforward. Moreover, it is a powerful instrument to allow accessibility to the toolchain. Indeed, any new complementary tool could be added and interact with one or more tools without the need of building specific dedicated interfaces. On the contrary, mainly at the computing level, for code instrumentation, code generation or direct hardware synthesis direct integration of new back-ends into already existing tools was more straightforward, normally in this cases information exchange at the model level is not needed. In fact:

Why tool to tool connection
PREESM/Spider-PAPIFY/PAPIFY Viewer: Here a tool-to-tool connection (without CIF) is needed because connections between the tools requires deep changes in the way PREESM works. Notably, the PAPIFY parameterization by the system developer needs to be made at a high level of abstration, integrated with the scenario edition GUI of PREESM. To connect the developed system with PAPIFY viewer, the code generated by PREESM must integrate specific code, which can only be done by integrating PAPIFY concerns directly into the code generation of PREESM. Using CIF for any of these concerns would be overly complex and largely inefficient, without any real benefits since these concerns are specific to the PREESM/PAPIFY connection.
PREESM/ARTICo^3 connection is also not compatible with CIF, since the connection is supported by integrating Artico concerns directly into the code generation of PREESM.
MDC-ARTICo^3: MDC generates HDL codes and Xilinx-compliant IP for Vivado Environment. On the other hand, ARTICo^3 takes as input HDL code. The abstraction and computation levels are the same, so no model transformations are required to exchange data between the two tools. In this case a direct connection was straightforward and avoided the presence of an intermediate layer among the tools.
SAGE-DynAA: in this case, a tool-to-tool connection has been implemented because the modelling stage of the discrete part of a model using Promela is placed on the top of the tool-chain, during the early phase of the design process. Given the compact synthax of Promela with respect to the Java counterpart needed by DynAA, the advantage of this solution is twofold. On the one hand, this let the designer to focus on the modelling avoiding programming details; on the other hand, off-the-shelf tools such as SPIN can be used to verify the model with respect to the requirements formulated using ReqV.
PAPIFY-MDC: given that PAPIFY has been designed to be able to access every PAPI-component, this integration relies on the insertion of properly designed performance monitoring counters in the MDC-generated accelerators, and in the development of a PAPI-compliant MDC-component. Done that, both the PAPI calls and the MDC calls to the accelerator can be inserted in the application, and there is no need of any further interface to make them exchange data.

PREESM/SPIDER-MDC: the integration is based on the possibility to control MDC-compliant accelerators by software tasks designed in PREESM and dynamically handled by SPIDER. The developer can exploit the drivers generated by MDC to call the acceleration from the C code description of the application blocks. Thus, the combination of these tools does not need any further layers. However, including accelerator in a high-level description in PREESM/SPIDER is application-dependent and a custom implementation is required.
IMPRESS + ARTICo^3: the modularity in the ARTICo^3 toolchain enables the replacement of the original TCL scripts (i.e., Xilinx Partial Reconfiguration flow) by IMPRESS. Since both tools operate at a very low implementation level, a direct connection between tools is required.
PAPIFY + ARTICo^3: Both tools work at low level (i.e.: hardware), so there is no need of providing access to higher layers., since PAPIFY provides access to HW counters and registers available at hardware architecture level
ARTICo^3 + MDC + PAPIFY: The integration of these three tools, taken two by two, are based on direct tool to tool links with no dependencies on upper layers. The access from PAPIFY to monitors and counters on both MDC and ARTICo^3 is done in a consistent way and, in case MDC accelerators are embedded into ARTICo^3 containers, an encapsulation method is used, which is solved by the runtime libraries of both reconfigurable fabric types.
Why connection through CIF
HW-SW co-design facilitated by PREESM uses AOW for Design Space Exploration (DSE) of optimal mapping and scheduling of functional actors on a heterogeneous hardware. The resulted design is further analyzed in DynAAsimulation. All tools are connected through the CIF interface. Moreover, MDC to CIF interface has been created to facilitate cross-layer interconnection and prospectively to complement MDC with new features offered by other tools, as automatic test generation or faster (and larger) design space exploration using e.g. AOW. In both examples, the usage of CIF in this case was needed to facilitate the access to tools leveraging on different models and semantics.

Table 7-1: Comparison between tool to tool connection and connection through CIF

7.2. Operational Objectives Deliverable Contribution

In the Table 7-2 a report on how the activities reported in the present deliverable have contributed to reach proposal operational objectives.

CH#	Operational Objectives	D5.1 Contribution
1.1	Provide reusable Libraries of KPIs, Cross-Layer Models and Adaptivity support.	PREESM/Spider connection with PAPIFY automates the monitoring of KPIs in synthesized CPS components, at a high level of abstraction (Dataflow MoC-level). Different flavours of disjoined and combined adaptivity run-time support have been enabled by the stand-alone and combined usage of MDC and ARTICo^3.
1.2	Provide a comprehensive framework, customizable upon the UC needs, extending and making interoperable a large set of tools.	The large effort spent on T5.4 in building all tool connections (direct and through CIF) has led to define the comprehensive set of tools to support all the needs of the CERBERO use-case that span across sectors and level of abstractions.
2.1	Reduce DSE by an order of magnitude.	Multi-view design requires multiple iteration between corresponding tools. Creation and maintenance of the connections between tools become a critical bottleneck in

		many cases resolved by CIF and direct connections described above.
2.2	Reduce by 50% the design efforts required to build a CPS of a given performance.	The large effort spent on T5.4 in building all tool connections (direct and through CIF) has led to define a comprehensive set of tools. Cross layer connections and design automation help achieving this purpose.
2.3	Reduce by 50% cost of maintenance.	Adaptivity could have a positive impact on both short-term and long-term maintenance. Having invested on integrating tools capable of offering different flavours of adaptivity indirectly contribute to reduce maintenance costs.
3.1	Provide a fully marketable version of the CERBERO modelling and design environment.	Not promoting a single tool, but an ensemble of tool. You can benefit from the integration having a more complete available environment. Raise the level of abstraction and training hiding many low level details. Moreover, CIF can be beneficial thanks to interoperability and openess to external new connections.
3.2	Foster Interoperability	CIF considerably reduces interoperability effort to create and maintain connections between design tools.

Table 7-2: D5.1 Contribution to CERBERO Operational Objectives

8. References

1) Michael Masin, Lior Limonad, Aviad Sela, David Boaz, Lev Greenberg, Nir Mashkif, Ran Rinat, “Pluggable Analysis Viewpoints for Design Space Exploration”, Procedia Computer Science, Volume 16, 2013, Pages 226-235.

2) MBSE case study, International Council on Systems Engineering, January 2012. incoseonline.org.uk.

3) Functional Mock-up Interface (FMI) Standard, fmi-standard.org.

4) Presentation of Modelica, Sébastien FURIC, INSA Lyon and LMS Engineering Innovation, France.

5) Janne Luoma, Steven Kelly and Juha-Pekka Tolvanen, "Defining Domain-Specific Modeling Languages: Collected Experiences", MetaCase, Ylistönmäentie 31, FI-40500 Jyväskylä, Finland.

6) Domain-Specific Modeling Languages: Moving from Writing Code to Generating It, Steven Kelly, Microsoft, December 2007. https://msdn.microsoft.com/en-us/library/cc168592.aspx.

7) Gabor Simko, David Lindecker, Tihamer Levendovszky, Sandeep Neema, Janos Sztipanovits, “Specification of Cyber-Physical Components with Formal Semantics – Integration and Composition”, Springer, Part of the Lecture Notes in Computer Science book series (LNCS, volume 8107).2013 pp 471-487.

8) Manel Ammar, Mouna Baklouti, Maxime Pelcat, Karol Desnos, Mohammed Abid. “MARTE to PiSDF transformation for data-intensive applications analysis”, Design & Architectures for Signal & Image Processing (DASIP), Oct 2014, Madrid, Spain. 2014.

9) Heiko Paulheim, Chapter 9, Ontology-based Application Integration, 2011, Springer, Berlin, ISBN 1461414296.

10) L Suriano, D Madroñal, A Rodríguez, E Juárez, C Sanz, E de la Torre ; “A Unified Hardware/Software Monitoring Method for Reconfigurable Computing Architectures Using PAPI”; 2018 13th International Symposium on Reconfigurable Communication-centric Systems on Chip; Lille (France), 2018.

11) Desnos K, Pelcat M, Nezan JF, Bhattacharyya SS, Aridhi S. Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration. In2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS) 2013 Jul 15 (pp. 41-48). IEEE.

12) Pelcat, M., Nezan, J.F., Piat, J., Croizer, J. and Aridhi, S., 2009, September. A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems.

13) Rodríguez A, Valverde J, Portilla J, Otero A, Riesgo T, de la Torre E. Fpga-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: The artico3 framework. Sensors. 2018 Jun;18(6):1877.

14) Tiziana Fanni, Alfonso Rodríguez, Carlo Sau, Leonardo Suriano, Francesca Palumbo, Luigi Raffo, Eduardo de la Torre; “Multi-Grain Reconfiguration for Advanced Adaptivity in Cyber-Physical”; Systems; 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)¸ 2018.

15) Madroñal Quintín, Daniel; Morvan, Antoine; Lazcano López, Raquel; Salvador Perea, Rubén; Desnos, Karol; Juárez Martínez, Eduardo y Sanz, César (2018). Automatic Instrumentation of Dataflow Applications using PAPI. En: "CF ’18: Computing Frontiers Conference", 8-11, 2018.

16) Grosser, Tobias, et al. "Polly-Polyhedral optimization in LLVM." Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT). Vol. 2011.

17) Pop, Sebastian & Cohen, Albert & Bastoul, Cédric & Girbal, Sylvain & Silber, Georges-André & Vasilache, Nicolas. (2006). GRAPHITE: Polyhedral analyses and optimizations for GCC. Proceedings of the GCC Developers' Summit 2006.

18) Uday Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model International Conference on Compiler Construction (ETAPS CC), Apr 2008, Budapest, Hungary.

19) Caamaño, Juan Manuel Martinez, et al. "APOLLO: Automatic speculative polyhedral loop optimizer." IMPACT 2017-7th International Workshop on Polyhedral Compilation Techniques. 2017.

20) Jeffrey Parsons, Yair Wand; Emancipating Instances from the Tyranny of Classes in Information Modelling, ACM Transactions on Database Systems, Vol. 25, No. 2, 2000.