



Funded by
the European Union



MYRTUS

Multi-layer 360° dYnamic orchestration and interopeRable design environmenT for
compute-continUum Systems

Deliverable:

D7.1 - MYRTUS DPE v1

Due date of deliverable: (30-04-2025)

Actual submission date: (28-04-2025)

Start date of Project: 01 January 2024

Duration: 36 months

Responsible: TUD

Revision: draft

Dissemination level		
PU		x
DEC		
SEN		

Funded by the European Union, by grant No. 101135183. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.



DOCUMENT INFO

Author

Author	Company	E-mail
Jeronimo Castrillon	TU Dresden	jeronimo.castrillon@tu-dresden.de
Guilherme Korol	TU Dresden	guilherme.korol@tu-dresden.de
Jiahong Bi	TU Dresden	jiahong.bi@tu-dresden.de
Francesca Palumbo	UNICA	francesca.palumbo@unica.it
Francesco Ratto	UNICA	francesco.ratto@unica.it
Amina Moussaoui	Softeam	amina.moussaoui@softeam.fr
Juan Cadavid	Softeam	Juan.Cadavid@softeam.fr
Alessandra Bagnato	Softeam	Alessandra.Bagnato@softeam.fr
Andrés Otero	UPM	joseandres.otero@upm.es
Juan Gallego	UPM	j.gallegor@upm.es
Alfonso Rodríguez	UPM	alfonso.rodriguez@upm.es
Francesco Regazzoni	USI	francesco.regazzoni@usi.ch
Subhadeep Banik	USI	subhadeep.banik@usi.ch

Document history

Document version #	Date	Change
0.0	13/03/2025	Definition of structure and initial draft.
0.1	10/04/2025	First complete release available.
0.2	11/04/2025	Draft available for internal review.
0.3	15/04/2025	Internal reviews received.
0.4	28/04/2025	Document ready for submission

Document data

Keywords	MYRTUS DPE, SoA Analysis, Preliminary Advancements, Example Flows
Editor Address data	Name: Jeronimo Castrillon Partner: TUD Address: Helmholtzstr. 18, 01069, Dresden (DE) Phone: +49 (0)351 463 42716 Email: jeronimo.castrillon@tu-dresden.de



Table of Contents

1 Executive Summary	5
2 Introduction	6
2.1 Overview of the Design and Programming Environment	6
2.2 Modifications in the DPE	9
2.3 Structure of the Document	9
2.4 Related Documents	9
3 Continuum Modeling, Simulation, and Analysis	11
3.1 Overview of the OASIS TOSCA standard	11
3.2 Modelio TOSCA Designer	13
3.3 DynAA	14
3.4 FREVO	16
3.5 COSMOS	17
3.6 Example Flows	20
3.6.1 Gains w.r.t status quo	22
4 Model to Implementation	25
4.1 Overview	25
4.2 Modelio Attack-Defense Tree (ADT) Designer	26
4.3 Countermeasure-Synthesizer	27
4.4 Third-party tools: DSL/ML Frameworks	28
4.5 Example Flows	29
4.5.1 Gains w.r.t status quo	31
5 Node-Level Optimization and Deployment	34
5.1 Overview	34
5.2 MLIR-based compiler	36
5.2.1 dfg-mlir	36
5.2.2 vitis-mlir	36
5.2.3 cgra-mlir	37
5.3 Mocasim	38
5.4 MDC	39
5.5 CGRA mapper	40
5.6 Third-party tools	40
5.6.1 IREE Importer	40
5.6.2 Polygeist	40
5.6.3 Vitis Design Tools	41
5.7 Example Flows	41
5.7.1 Example for CNN compilation flow with dfg-mlir	41



5.7.2 Example flow for adaptive CNNs with MDC	43
5.7.3 Gains w.r.t status quo	44
6 Towards End-to-End Flows	46
7 Conclusion and Future Work	48
8 References	50



1 Executive Summary

The third pillar of the MYRTUS project corresponds to the MYRTUS Design and Programming Environment (DPE), featuring interoperable support for cross layer modelling, threat analysis, design space exploration (DSE), modelling, components synthesis, and code generation. This deliverable describes the status of the tools that compose the MYRTUS DPE and the evolution of the DPE design as a whole. The individual tools are described in the context of the three steps of the DPE, namely, (1) tools for continuum modelling, simulation and analysis, (2) tools to transition from model to implementation, and (3) tools for node-level optimization and deployment. A focus of this deliverable is in describing the tools that have been implemented in the first phase of the project and how they advance the state of the art in their respective fields. We also report on initial interfaces in between the tools and demonstrate existing flows within the steps, resulting from initial integration efforts carried out so far.

The MYRTUS DPE as a whole has undergone significant improvements compared to its status at proposal writing. The improvements consist in a refinement of the roles of individual tools and a definition of the data exchange interfaces among them. In Step 1 ([Section 3](#)), the mobility use case was used as driver to steer the development of modelling tools (e.g., Modelio Tosca Designer now supports 80% of the Tosca standard), where Tosca-compliant templates, inputs, output, interfaces, and export mechanisms have been defined. Modelio was also extended to security threat modeling. In Step 2 ([Section 4](#)), a methodology for automatic countermeasure synthesis has been devised, and the role and interfaces to third-party tools has been refined. In Step 3 ([Section 5](#)), the analysis of use cases led to requirements on languages, intermediate representations, and node-level optimization. This, in turn, led to advances in the dialects of the MLIR-based interoperability framework and refinement in the compilation flows to interface with high-level synthesis for FPGA targets and optimization for CGRA targets. Bilateral and trilateral connections between the tools are demonstrated in each step, with the vision of the connection in between the steps and to other work packages described in [Section 6](#). The focus of the second phase of the project will be in finalizing the connections within and outside the DPE, thus materializing the integration with the first and second pillars of the MYRTUS project.

Table 1-1: D7.1 description from the Grant Agreement

D7.1 - MYRTUS DPE v1 - M16

MYRTUS tools and their extensions are described with their individual SoA analysis and preliminary advancements. The three different steps of MYRTUS DPE are detailed and partially integrated/accessed.

Task Involved:

T7.1(M3-M18) - Continuum Modelling, Simulation and Analysis [SOFT(L), LAKE, TNO]

T7.2(M3-M18) - Model to Implementation [USI(L), SOFT, TUD]

T7.3(M3-M18) - Node Level Optimization and Deployment [TUD (L), UPM, UNISS, UNICA, ABI]



2 Introduction

2.1 Overview of the Design and Programming Environment

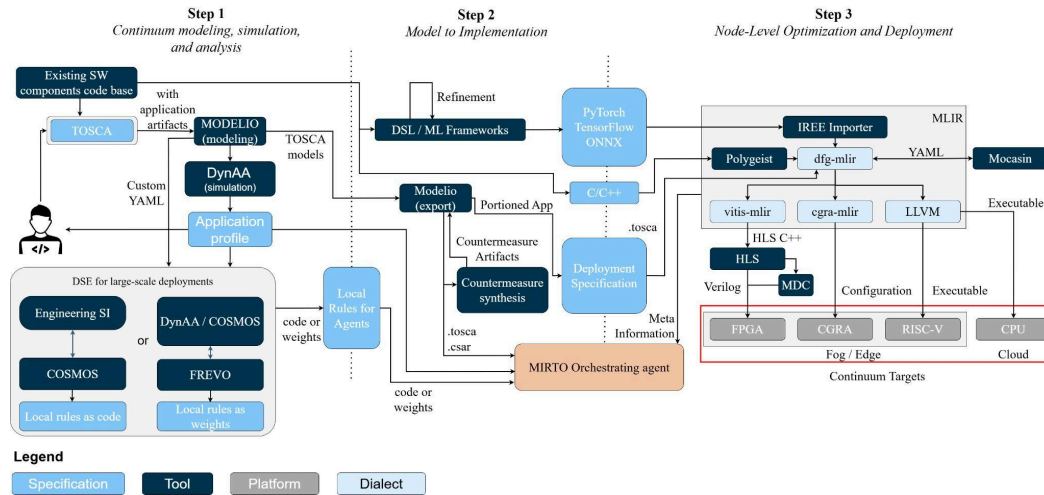


Figure 2.1: DPE Big Picture

The MYRTUS Design and Programming Environment (DPE) provides an integrated tool set for seamless and interoperable application deployment across a heterogeneous computing continuum. It enables the generation of deployment specifications, including all necessary executables and configuration files, while exporting meta-information on non-functional properties to support runtime decision-making by the MIRTO Cognitive Engine. The DPE leverages open-source tools and formats to describe and exchange applications and meta-information thereof, i.e., TOSCA¹ and MLIR².

The DPE is structured into three key steps, as in [Figure 2.1](#). The first step, **continuum modeling, simulation and analysis**, enables early-stage application design and evaluation. As described in [Section 3](#), this step leverages several tools, most prominently Modelio TOSCA Designer, to model the functional architecture of the application, provides support for security threats modeling and specifies application policies and constraints (such as the expected end-to-end latency). Workload deployment and management in TOSCA-compliant environments will be provided by Modelio TOSCA Designer, and tools such as FREVO, COSMOS and DynAA aid in high-level exploration of the deployment, including KPI estimations, and interfaces with the MIRTO Cognitive Engine. The second step, **model to implementation**, covers the step from going from abstract models into concrete implementations. As discussed in [Section 4](#), this refers to generating deployment artifacts or pointing to existing implementations of software components of the application. In this step, we also synthesize countermeasures for the identified security threats, and deploy the countermeasures as software components. The “portioned app” in the figure represents key kernels (e.g., Digital Signal Processing (DSP) kernel or ML components) that ought to be further optimized by step

¹ <https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html>

² <https://mlir.llvm.org>



3 of the DPE. Interoperability mechanisms allow for implementations to be derived from external Design Specific Languages (DSLs) and/or ML frameworks. Key program code is then passed to Step 3 for compilation, optimization, and, in the case of FPGA-based computing components, also for accelerator synthesis. This step exports a component-level view of the application to the MIRTO Cognitive Engine (defining the interface from design time to runtime, aka from the MYRTUS Pillar 3 to Pillar 2) and synthesizes the swarm agents to be included in the MIRTO Manager of the Cognitive Engine from the local rules. Finally, countermeasures are generated to protect against the threats specified in step 1. The third step, **node-level optimization and deployment**, optimizes and deploys key computational kernels for heterogeneous target hardware for cross-platform support. As discussed in [Section 5](#), this step results in the executables and bitstreams for running and/or configuring the different computing components. Key to this step is an interoperability framework based on MLIR, which allows for i) importing third-party codes (from DSLs or ML models), ii) having access to third-party tools (like polyhedral compilers for optimization purposes), and iii) compiling code for different targets (as reconfigurable accelerators, CPUs, or customizable RISC-V cores). For the optimization, DSE tools such as Mocasin [[Menard-2021](#)] are used.

The DPE provides key support in the following areas:

1. **Security and Privacy:** These aspects are integrated from the early stages of development. Modelio Attack-Defense Tree (ADT) Designer allows the system architect to model the security threats of the system, by means of ADTs of the system, enabling the automatic synthesis of countermeasure software components to address security and privacy concerns.
2. **Orchestration and AI:** Both are facilitated through the initial deployment specification, established at design time, and by the local rules synthesized for swarm agents to guide adaptive decision-making.
3. **Data Management:** The DPE's node-level optimization and deployment step enables efficient data processing across heterogeneous devices, with a particular focus on computation over distributed data.

These capabilities reflect the mapping between the MYRTUS reference infrastructure and the EU-CEI building blocks, a process conducted within WP1 with support from WP7. Indeed, MYRTUS, besides mapping all the EU-CEI building blocks (as already detailed in D1.1), aims also at feeding/contributing to EU-CEI. The EU-CEI reference architecture is focussed on the architecture functionalities to be offered when dealing with continuum, and does not address the problem of turning applications into executable implementations. This is non-trivial for architectures that rely on heterogeneous families of CPUs, and it becomes progressively more challenging as HW accelerators are introduced in the architecture. As the MYRTUS-compliant continuum infrastructures comprise a heterogeneity of computing nodes, the need for a DPE with **cross-platform support capabilities** emerged as a fundamental EU-CEI Building Block (BB) to enable the use of the continuum architecture.

Compared to the current status-quo in software development for the computing continuum, the DPE in [Figure 2.1](#) represents a unique attempt to provide end-to-end tooling support with extensible and interoperable interfaces. Today, software development is characterized by



fragmented tool chains (e.g., for modeling only or for high-level synthesis (HLS) for isolated software components). The DPE, instead, integrates several tools via standard interfaces and standards-based data exchange formats, i.e. OASIS TOSCA to drastically reduce the amount of manual and error-prone efforts. While the structure of the DPE is complex in itself, it represents an improvement with respect to current practices. The individual tools that conform the DPE represent and advance the current state of the art in their respective domains. For instance, (1) the use of Modelio TOSCA designer coupled with high-level support for design space exploration greatly reduces the effort in manual configuration and trial-and-error deployments which lead to high inefficiency in systems design, (2) the specification of threats and synthesis of countermeasure automates the process of guarding against security flaws in a seamless manner, and (3) we leverage and extend the state of the art in programming language and compiler support via MLIR to target heterogeneous and bleeding edge hardware targets in the continuum. For details about how the individual tools advance the state of the art is provided in [Section 3.6.1](#), [Section 4.5.1](#), and [Section 5.7.1](#), for each of the steps in the DPE.

Nevertheless, we base the work of the DPE on industry- and academia-backed frameworks. Particularly, the TOSCA standard and the MLIR framework are central to the DPE. The MLIR compiler infrastructure is governed by a nominated team,³ maintained by industry and academia, and, as a part of the LLVM project, it follows the LLVM Foundation governance rules⁴. On top of that, as demonstrated by numerous industry and research projects⁵, in MYRTUS, the MLIR modularity allows us to develop an end-to-end compilation flow leveraging the LLVM and MLIR infrastructure. Thus, enabling us to focus on the support of the MYRTUS hardware platforms, development of novel optimizations, and on the interface with other tools within the DPE. This modularity increases the overall robustness of the DPE given the possibility of quickly adapting any component by, for example, modifying a single MLIR dialect instead of replacing a monolithic compiler.

Similar reasoning holds for the TOSCA standard, in which the standard's robust foundation is built upon the collaborative efforts of industry leaders and leading academic institutions. Developed under the auspices of the OASIS consortium, TOSCA benefits from the participation of key European players like Siemens, SAP, and Ericsson, who contribute to its evolution and drive its adoption in real-world deployments. This industry support is complemented by significant contributions from academic research, with universities across Europe exploring TOSCA's potential in areas like cloud orchestration, service management, and network automation. This strong backing from both industry and academia ensures TOSCA remains a relevant, adaptable, and vendor-neutral solution for modeling and managing cross-platform cloud applications and services. Furthermore, adopting TOSCA guarantees a level of robustness to the development chain of cloud-fog-edge systems. As an open standard, TOSCA enables cross-platform interoperability and portability, meaning that if one component in the

³ MLIR Governance: <https://mlir.llvm.org/governance/>

⁴ LLVM Governance Rules: <https://github.com/llvm/llvm-www/blob/main/proposals/LP0004-project-governance.md>

⁵ Users of MLIR: <https://mlir.llvm.org/users/>



toolchain is missing or needs replacement, it can be substituted with another TOSCA-compliant alternative, ensuring continuity and minimizing disruption.

2.2 Modifications in the DPE

Below, we list the modifications in the Design and Programming Environment with respect to [Deliverable D1.1](#).

Step 1:

1. TOSCA system and application topologies, described in Modelio, can be almost directly translated to simulation schemas of DynAA (see [Section 3.3](#)). This opens an opportunity for users to quickly evaluate or profile their application description in terms of its temporal behavior, resource usage, etc.
2. The new DSE (Design-Space Exploration) block: Step 1 contains a set of tools for DSE, which leverage swarm intelligence for design-space exploration (DSE) of the most optimal workload distribution among compute nodes. They are appropriate in the case of large-scale deployments of cloud-fog-edge architectures, where the number of nodes exceeds 50. These tools, presented in the next section, produce as output a set of local rules, either in the form of python code describing the workload assignment rules or as a set of artificial neural network weights. In any of both cases, this output will be exploited in the MIRTO Cognitive Orchestrating Agent (See [Deliverable D5.1](#)) in the workload manager for effective orchestration.

Step 2:

1. The new “Refinement” arrow over “DSP/ML Frameworks” ([Figure 2.1](#)) indicates that, under the Model to Implementation step, it is possible for the user to adapt the DSP/ML task for a particular target or considering some eventual platform restriction (e.g, quantizing an ML model for deployment on an integer-only target). Such refinements are the responsibility of the user and are implemented in the application’s source framework. One may also note that the original application code is input by the user under Step 1. See [Section 4.4](#) for more details.

Step 3

1. A new MLIR dialect for supporting the HLS code generation was included (see [Figure 2.1](#)). The new dialect represents an additional compilation flow targeting FPGA execution for generic applications (that can be represented as dataflow graphs). [Subsection 5.2.2](#) presents this new dialect in detail.

2.3 Structure of the Document

The remainder of this document is structured as follows. Sections [3](#), [4](#), and [5](#) present the three steps of the Design and Programming Environment. Each section gives an overview of its tools and their integration, presents the modifications with respect to previous deliverables, demonstrates an example flow showcasing the step usage, and introduces the step’s contributions over the state-of-the-art. [Section 6](#) presents an end-to-end flow with an example, focusing on the integration of the three DPE steps. Finally, [Section 7](#) concludes this deliverable and points to the next steps of this working package.

2.4 Related Documents



D1.1 – UC Needs, requirements, and KPIs v1

This document sets the foundation for WP7, where the initial and incremental drafts of the DPE architecture have been reported along with requirements, technical specifications, and KPIs for the DPE.

D3.1 — MYRTUS Reference Infrastructure v1

This document provides the description of the MYRTUS Project Referenced Architecture spread across Edge-Fog-Cloud that was modeled within the first year. This includes the hardware accelerator flavors for the edge nodes, which will require the tools and frameworks from the third step of the MYRTUS DPE to deploy applications.

D5.1 – MIRTO Cognitive Engine v1

This deliverable defined the MIRTO cognitive engine architectures. The DPE is responsible for providing the CSAR package with initial indication for the deployment, part in defining the software components and hardware accelerators. Moreover, it also provides support for the runtime adaptation defining the rules for the swarm-based Workload Manager and also some meta-information to be exploited at runtime to make more informed decisions.



3 Continuum Modeling, Simulation, and Analysis

Step 1 focuses on the initial phases of designing and evaluating a distributed application within a cloud-fog-edge continuum. This stage involves creating a comprehensive representation of the application and its deployment environment, followed by simulation and analysis to understand its behavior and performance. [Figure 3.1](#) shows an overview of this step.

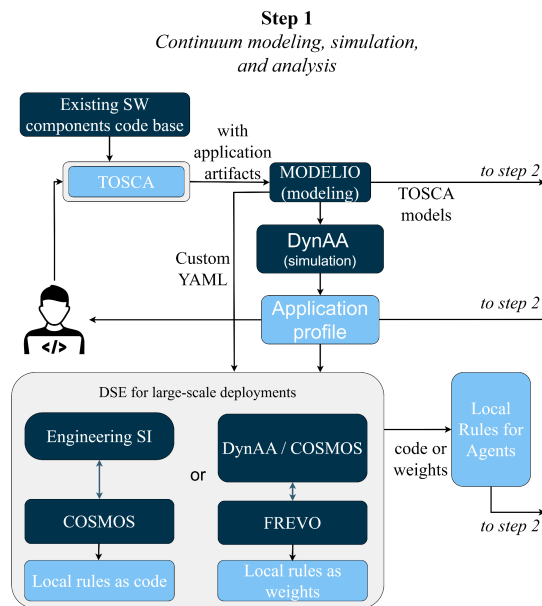


Figure 3.1: Step 1 overview

This section is structured as follows. First, we provide an overview of the OASIS TOSCA standard for cloud-fog-edge topology modeling, followed by the presentation of TOSCA Designer, the tool we are developing to create models describing the system topologies. Then, we present the tools for simulation and analysis, which are used for different goals of DSE. We close the section with an example flow explaining how these tools integrate together, and an analysis of the gains with respect to the current state of practice (*status quo*) in cloud-fog-edge modeling.

3.1 Overview of the OASIS TOSCA standard

TOSCA (Topology and Orchestration Specification for Cloud Applications): This standard language is used to define the application's topology. This includes a high-level, black-box view of the software components, their dependencies, and deployment requirements, including placement requirements e.g. which computing nodes should be used for each software component instance.

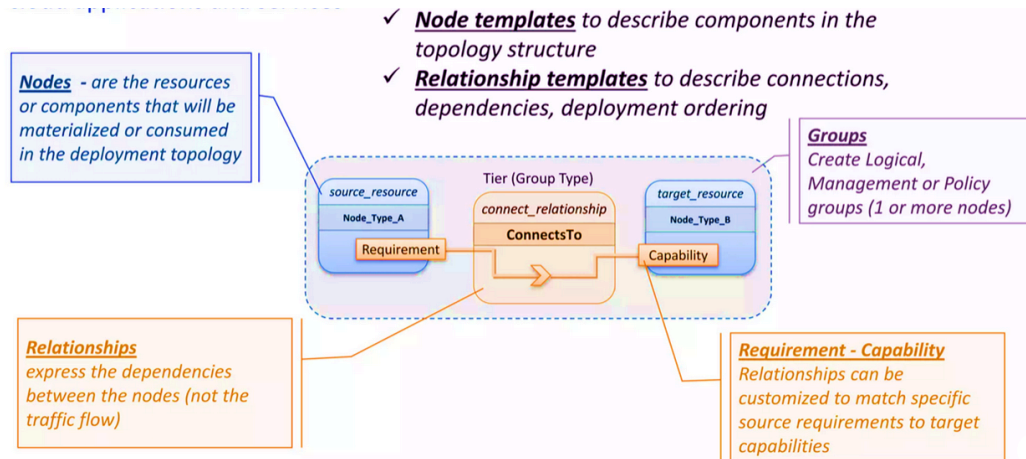


Figure 3.2: OASIS TOSCA overview. Source: OASIS TOSCA

Figure 3.2 illustrates the core concepts of OASIS TOSCA by depicting how application topologies are constructed. Nodes, representing individual resources or components, are defined using node templates, forming the fundamental building blocks of the deployment. Relationships, expressed through relationship templates, capture the dependencies between these nodes, focusing on structural connections rather than traffic flow, and are tailored through requirements and capabilities to match specific source needs with target provisions. Groups further organize these nodes into logical sets for management and policy application, enabling the creation of tiered structures and facilitating complex deployment scenarios.

One fundamental part of the standard, the TOSCA standard library provides a foundational set of reusable node types, essential for streamlining the modeling of cloud applications. This library offers basic components like `tosca.nodes.Compute` for virtual machines, `tosca.nodes.Database` for database systems, and `tosca.nodes.WebApplication` for web applications, among others. These pre-defined node types encapsulate common functionalities and properties, reducing the need to define them from scratch and fostering consistency across TOSCA models. For instance, using `tosca.nodes.Compute` simplifies the definition of a virtual machine by providing standardized properties for CPU, memory, and disk, while `tosca.nodes.Database` offers pre-defined attributes for database type and connection parameters.

While the TOSCA standard library provides essential node types for common cross-platform cloud deployments, such as `tosca.nodes.Compute` for virtual machines, it is often necessary to define custom node types to accurately model specialized hardware. For instance, when incorporating Reconfigurable Accelerators into a cloud or edge infrastructure, a new node type, such as `tosca.nodes.Accelerator`, would be defined. This custom type would typically inherit from `tosca.nodes.Compute` to retain standard properties like `cpu`, `memory`, and `disk`, while also including specific properties of the reconfigurable Accelerator, such as the `flavour` (i.e., MDC, ARTiCo3, CGRA), `size` (also dependent on the flavour, such as number of slots in MDC, number of supported threads in MDC, or size of



CGRAs), memory type, and supported programming languages. This practice of defining problem-specific node types is standard in TOSCA modeling and allows for precise descriptions of the underlying infrastructure.

DSE within a cross-platform cloud-fog-edge continuum can be significantly enhanced by leveraging TOSCA models as a starting point. The system architect can utilize the application's topology and resource requirements defined in TOSCA, the DSE tools (FREVO, COSMOS) in our DPE can explore various workload placement configurations across the distributed infrastructure. This process involves evaluating different mapping strategies of application components to available computing resources, considering factors such as latency, bandwidth, and resource utilization. Furthermore, the opportunity to integrate application profiling using DynAA simulation engine during this exploration allows for a more refined analysis. By capturing runtime performance metrics and resource consumption patterns, the exploration can be guided by real-world application behavior, leading to the identification of optimal deployment configurations that balance performance, cost, and resource efficiency in a cross-platform setting.

In the following sections, we describe the tools involved in step 1 of the DPE.

3.2 Modelio TOSCA Designer

The TOSCA Designer, an extension of the Modelio modeling platform, provides a user-friendly environment for creating and managing TOSCA models, crucial for defining and deploying applications in cloud-fog-edge environments. It streamlines the development process by offering intuitive tools for building service templates, topology templates, policies, and constraints. The export functionality supports the export of TOSCA models in two standard formats: 1) .CSAR (Cloud Service Archive): Facilitates the creation of cloud service blueprints, enabling portability and interoperability across different cloud platforms. 2) .TOSCA: Allows for the export of individual TOSCA files for node types and topology templates, promoting modularity and reusability.

The TOSCA Designer has been successfully integrated into the MYRTUS project, demonstrating its practical application in real-world scenarios. Integration tests have been performed with two key examples: 1) Face Recognition Application. A basic face recognition application was modeled and deployed, showcasing the tool's ability to handle complex application deployments. The application involves a web application that analyzes a webcam video stream and performs face recognition. 2) Mobility Use Case: An early modelization of the MYRTUS mobility use case was created, complete with a cloud-fog-edge template. This example includes examples of deployment policies, highlighting the tool's ability to define constraints for distributed applications.

It is crucial to understand that Modelio is a modeling tool, not an orchestration or deployment tool. Modelio allows users to visually model the application's architecture and generate the necessary TOSCA artifacts (like .tosca or .csar files) that will be used by the external orchestration engine, the MIRTO cognitive engine agent, for the actual deployment (see [Deliverable D5.1](#)).



We have published a release for Modelio TOSCA Designer (v0.5), available at <https://github.com/Modelio-R-D/MYRTUSDesigner/>.

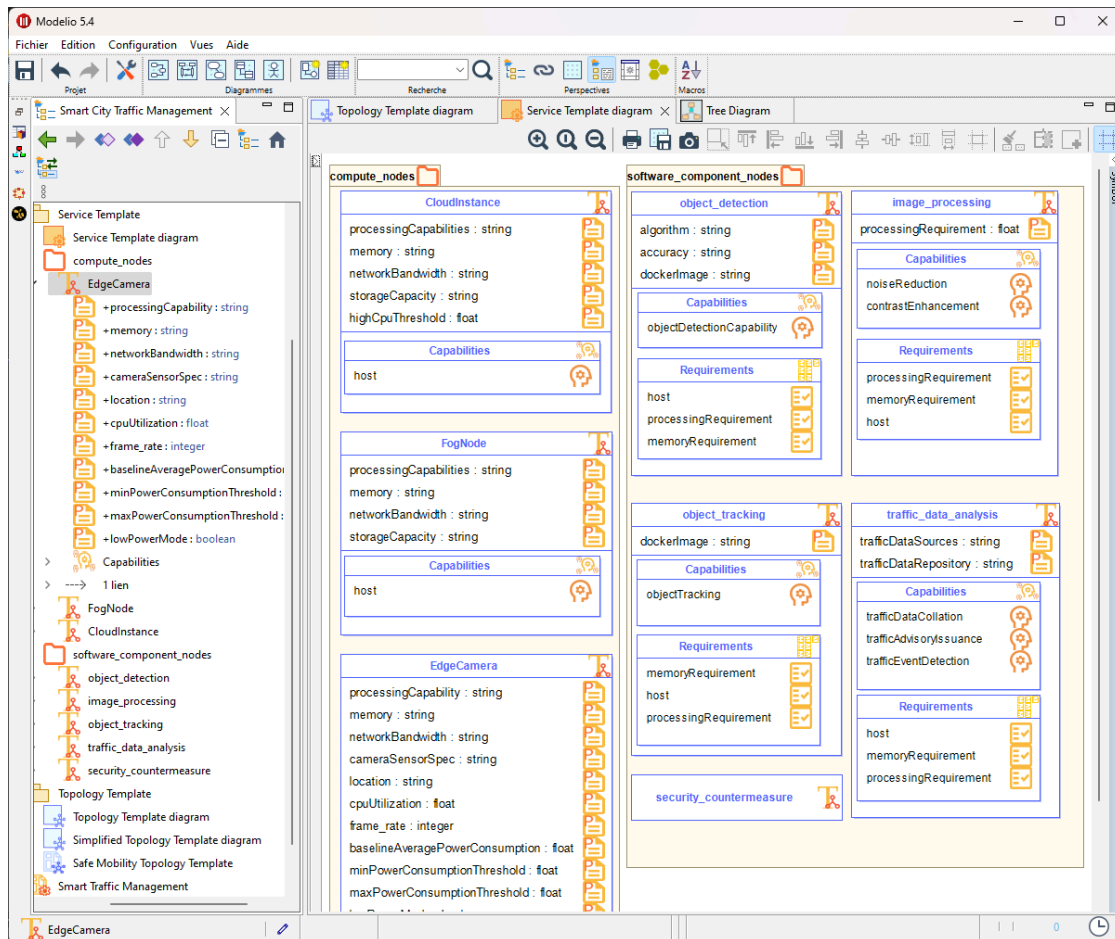


Figure 3.3: Modelio TOSCA Designer showing the mobility use case node types

Figure 3.3 presents the Modelio TOSCA Designer interface showcasing the design of a Smart City Traffic Management system using TOSCA, aligned with the MYRTUS mobility use case, featuring a service topology with interconnected node types like “EdgeCamera”, “FogNode”, and “CloudInstance”, along with nested specific software components such as “ObjectDetection” and “ImageProcessing.” It depicts relationships between these computing nodes, the software components like “ObjectDetection” and “ImageProcessing”, and the definition of policies and constraints for resource management and performance.

It is also worth noticing that the entire TOSCA standard library is included in Modelio TOSCA Designer, allowing users to reuse the standard types therein in the definition of their service topologies.

3.3 DynAA



In the MYRTUS project, the core orchestration task within the MIRTO cognitive engine involves finding a (quasi-)optimal mapping of software components onto continuum computing nodes. This problem is inherently combinatorial, and its objective function (or “fitness” measure) is typically not expressed in analytical form. Instead, it must be evaluated through simulation or measurement. This applies to solvers used in the DPE tools (e.g., FREVO) and in the Distributed Constraint Optimization Problem (DCOP) workload manager within the MIRTO agents. Consequently, these solvers must integrate with simulation tools, such as DynAA [\[Filho-2016\]](#) or COSMOS [\[Varzonova-2025\]](#) (discussed in [Section 3.5](#)).

DynAA is a fast, low-memory-footprint discrete-event simulation engine. Users can leverage DynAA to simulate computing systems with behavioral models representing thousands of independent machines, software components, and network connections. During simulation, these models interact to “emulate” resource usage and power consumption over time. By analyzing simulation traces, users can visualize and estimate key metrics, such as CPU and memory usage, network activity, and interactions between software components.

DynAA was originally developed prior to the MYRTUS project. Within MYRTUS, we have been enhancing the engine with the following features:

1. **Domain Layer:**
We extended DynAA with a domain-specific layer that models continuum components, including computing machines (with CPUs, memory, etc.), network connections, and software components.
2. **Simulation Schema:**
We defined a data format (schema) to describe and configure simulation setups. The schema specifies how to create and parameterize components, define their interconnections (e.g., network topology), describe the temporal behavior of software modules, and map software modules to specific execution environments.
3. **Front-End Integration:**
We developed a front-end that transforms the simulation description (as defined in the schema) into a functional simulation model, leveraging the enhanced domain model.

With these features in place, DynAA can integrate with various tools in the DPE suite or in the MYRTUS cognitive engine. The tool combinations envisioned within the DPE are illustrated in [Figure 2.1](#).

System and application topologies defined using the Modelio tool (via TOSCA) can be simulated in DynAA to generate estimated application profiles. TOSCA descriptions can be transformed almost directly into the DynAA simulation schema. Additionally, the evolutionary tool FREVO (discussed in the next [Section 3.4](#)) can leverage DynAA to simulate and evaluate feasible system configurations, such as swarm setups, in order to design effective swarm organization rules.



Finally, DynAA will also be combined with modules in the MYRTUS runtime engine. In particular, it will function as the online simulation engine for the DCOP workload manager within the MIRTO agent, enabling efficient evaluation of potential application deployments.

3.4 FREVO

The FREVO framework uses evolutionary optimization to tackle challenges in decision-making and hardware-software co-design. FREVO's modular, component-based design makes it adaptable to a variety of applications. Its components are organized into categories for simulation, candidate representation, optimization methods, and algorithm ranking. The system also features an intuitive Graphical User Interface (GUI) that simplifies the process of configuring sessions, selecting components, and running optimization procedures.

FREVO is implemented in a strictly component-based architecture, organizing the evolutionary design process into discrete, interchangeable components. This design allows for easy replacement and independent development of individual components. Consequently, users can readily evaluate different configurations to identify the most suitable one for a specific problem. Each component addresses a particular aspect of the evolutionary approach. The “problem” component defines the specific characteristics of a computing node, the environment, and the fitness function. The “representation” component dictates how the agent is represented within the system. The “optimization” component defines the method used to discover the optimal candidate representation. Lastly, the “ranking” component specifies how candidate representations are evaluated and ranked based on their performance.

FREVO is implemented in Java, embracing object-oriented programming principles for component encapsulation. Each component is defined by an abstract class, standardizing interfaces between parts such as agent representation, simulation, and the optimization algorithm. This ensures seamless integration of new components with existing ones, expanding the overall system's capabilities. A built-in component generator guides the creation of new components, aiding developers in generating the necessary code framework within the existing class hierarchy. FREVO is released under an open-source license to foster the sharing of research insights and technical solutions. The source code is freely available at <https://sourceforge.net/projects/frevo/>.

FREVO tackles the problem that evolutionary methods often hinge on evaluating a “fitness” or “cost” metric for each potential solution. This typically involves running experiments where swarm members interact within a simulated environment mirroring their real-world operating conditions, collaboratively tackling representative tasks. However, these fitness scores are usually determined through post-experiment analysis, which then necessitates reinforcement learning techniques capable of managing delayed rewards.

The FREVO GUI, shown in [Figure 3.4](#), provides a step-by-step guide through the configuration process, prompting users to select a component for each stage of the evolutionary process. A single component configuration is referred to as a FREVO session. These sessions, along with the results of the optimization process, can be exported, saved, and imported for later use.

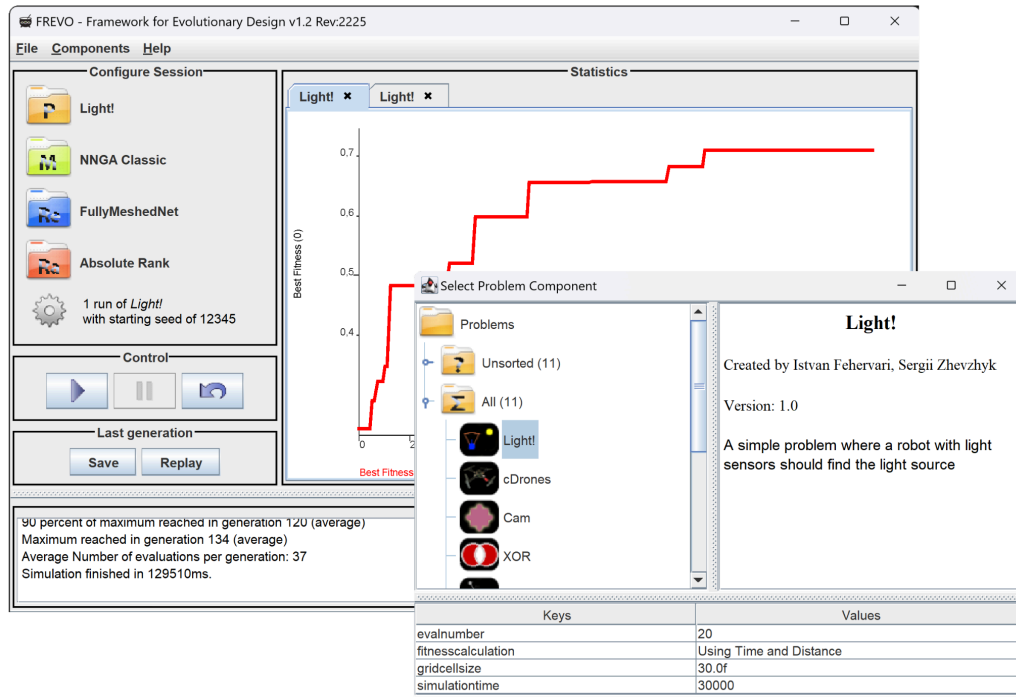


Figure 3.4: Screenshot of the FREVO GUI showing the evolution of an example problem.

The core task involves implementing the candidate representation through simulation. This simulation can occur directly within FREVO or by calling an external simulator. Finally, an appropriate performance metric is implemented to calculate the fitness value resulting from the simulation run. Exactly this part presents the contribution in MYRTUS: the implementation of the interface between FREVO and an external simulation environment, where we chose COSMOS (see [Section 3.5](#)).

The problem definition is contained in the COSMOS simulation environment. The code responsible for agent behavior and data flow is based on the MESA framework⁶, a Python framework for agent based modeling, and is written as an external program. An IPC (Inter-Process Communication) interface between FREVO and simulation is responsible for the exchange of current candidate parameters and EFC (Evolutionary Fuzzy Controller) model state-related statistics. Therefore, we have a two-way communication to enable candidate recreation at COSMOS's side and candidate evaluation at FREVO's side. In general, [Figure 3.5](#) below gives a good overview on the workflow between COSMOS and FREVO.

3.5 COSMOS

In the MYRTUS project, we implemented a novel simulation framework called COSMOS⁷ (Continuum Optimization for Swarm-based Multi-tier Orchestration System). COSMOS was

⁶ MESA Framework. <https://mesa.readthedocs.io/stable/index.html>

⁷ COSMOS. <https://github.com/Incomprehensible/COSMOS>



developed to provide a runtime environment for implementing and evaluating agent-based scheduling algorithms within a self-organizing context. Implemented in Python and built upon MESA, a multi-agent orchestration library, COSMOS facilitates the exploration of swarm intelligence approaches for workload management.

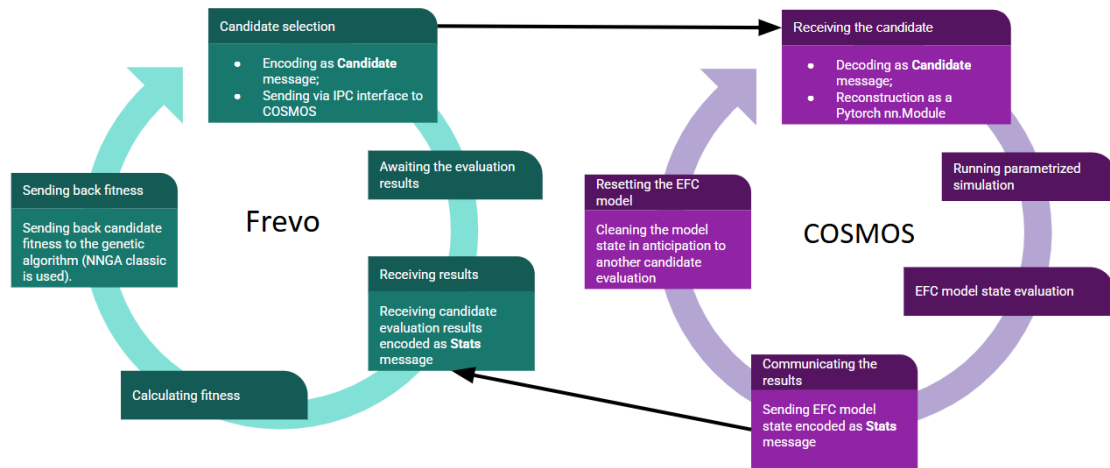


Figure 3.5: overview on the workflow between COSMOS and FREVO.

COSMOS envisions the edge-fog-cloud continuum as a dynamic landscape where Swarming Resource Agents offer their computing resources to process incoming Service Chain Components, known in this case as pods. These pods arrive with varying and diverse resource requirements. Each agent, belonging to either the Edge, Fog, or Cloud layer, provides a specific allocation of CPU power and memory, determined by its unique profile. At the same time, the model is a dense-connected network graph consisting of a number of Edge, Fog, and Cloud devices. Pods (i.e., demand agents) enter this network dynamically with randomized arrival times, controlled arrival density, and varying resource demands.

The framework facilitates the generation of diverse network topologies, enabling the adjustment of load conditions and agent distributions.

At the highest level of the framework's hierarchy is the Simulation class. This class manages the system's configuration, controls the runtime flow, handles model initialization, creates and distributes pods, and collects data. The Simulation class utilizes the Factory design pattern to create pods. The Model class constructs the network graph, composed of nodes representing resource agent locations and edges connecting the three main levels. It assigns resource agents to these locations and defines communication costs as edge weights, leveraging the NetworkX Python library⁸. The pod behavior algorithm is selected based on configuration settings and activated using the Strategy design pattern. The BehaviorProfile class creates the PodBehavior strategy and assigns it to the Pod class. The PodBehavior strategy class then initializes the specific strategy corresponding to the configured scheduling algorithm. In our experiments, we observed results produced by the ACO (Ant Colony Optimization) and

⁸ <https://networkx.org/>



HormoneAlgorithm classes, respectively. The block diagram illustrating the framework's architecture is presented in [Figure 3.6](#).

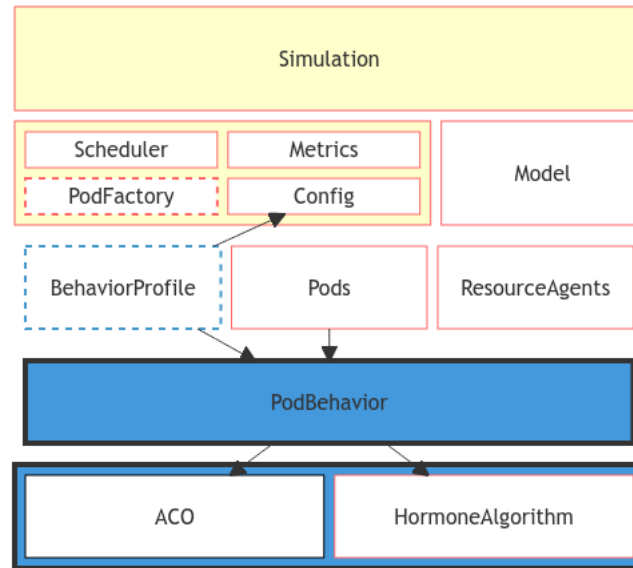


Figure 3.6: Block diagram illustrating the COSMOS architecture

The simulation framework offers a high degree of flexibility through adjustable parameters. The key parameters used in our experiment are listed below in [Listing 3.1](#).

```

1  sim_steps: 2000
2  pods_per_step: num_pods/sim_steps
3  num_edge_devices: 10
4  num_fog_nodes: 5
5  num_cloud_servers: 2
6  behaviour_type: 'hba' and 'aco'
7  randomized_pod_interval: True
8  mu: 0.9

```

Listing 3.1. COSMOS configuration parameters

where

- num_pods: The total number of pods included in the simulation.
- pods_per_step: The count of pods that arrive at each step of the simulation.
- sim_steps: The cumulative number of steps for which the simulation is executed.
- num_{edge/fog/cloud}_devices: Defines the topology of the network graph, specifying the number of devices at each level.
- behaviour_type: Specifies the pod behavior profile, dictating how pods act within the simulation.
- randomized_pod_interval: A boolean value; if set to True, pods will arrive at random intervals rather than at fixed intervals.
- mu: The parameter μ , where μ is a real number between 0 and 1 (exclusive), defining the rate, or frequency, of pod arrivals throughout the simulation.

In short, COSMOS simulates agent-based scheduling in edge-fog-cloud environments, allowing for flexible parameter adjustments to evaluate swarm intelligence algorithms.



3.6 Example Flows

In this section, we imagine a data flow example for the MYRTUS mobility use case, using the tools and concepts from Step 1 of the Design-Space Exploration (DPE) process.

1. **Initial Modeling:** The city planners, traffic engineers, and application developers collaborate to define the requirements of the mobility use case. They specify the functionalities, performance expectations, and resource constraints. Using Modelio TOSCA Designer, they begin to visually model the system's topology template, as shown in [Figure 3.7](#). They create TOSCA node templates for the worker nodes, representing the edge, fog, and cloud computing nodes of the system. They define the relationships between these nodes using the `connectsTo` relationship from the TOSCA Standard library, indicating the data flow and dependencies. They define the software components, for example ObjectDetection, which will be deployed on the edge camera, and TrafficDataAnalysis, which will be deployed on the fog node. The initial TOSCA model is saved as a .tosca file.

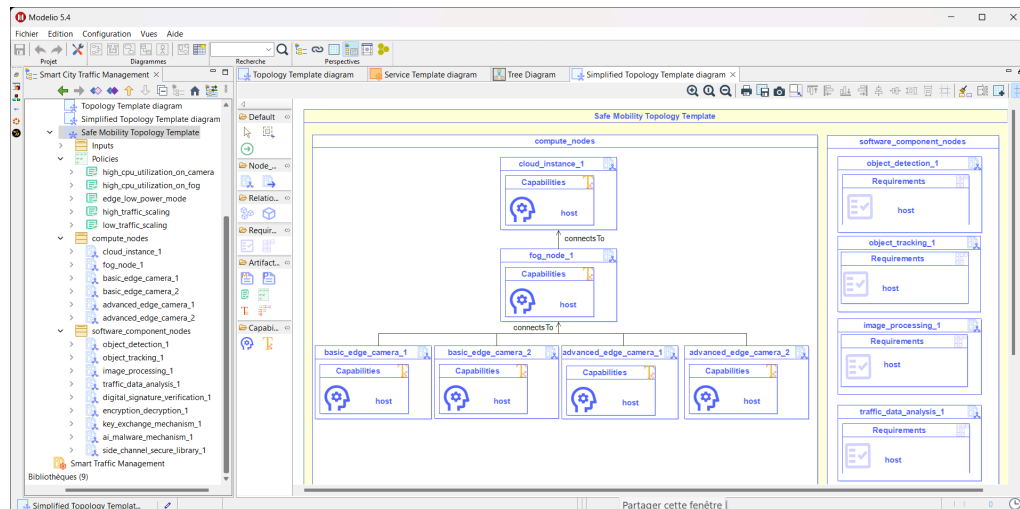


Figure 3.7: TOSCA Designer screenshot showing the modeling of the topology template

2. **Policy and Constraint Definition:** The team defines policies to ensure efficient resource utilization and performance, such as “Edge Resource Efficiency” and “Network Bandwidth Management.” These policies are associated with the relevant node templates. They also specify constraints to regulate resource usage, latency, and power consumption, such as “Max utilization of computational resources (CPU, GPU) at edge” and “Latency object detection”. These policies and constraints are added to the TOSCA model within Modelio TOSCA Designer. [Figure 3.8](#) shows the edition of one of such policies.

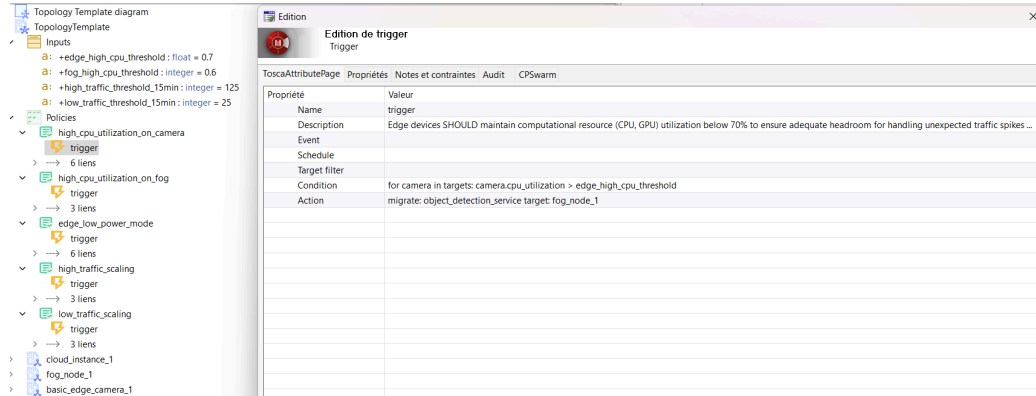


Figure 3.8: Edition of TOSCA Policies in TOSCA Designer

3. **Simulation and Analysis Setup:** The team decides to use DynAA to simulate the system's behavior. They configure DynAA to read the generated TOSCA model. They define the simulation parameters, such as the number of cameras, the frequency of data collection, and the expected traffic patterns. DynAA executes the simulation, generating data on resource utilization, latency, and power consumption. The simulation data is collected and stored for analysis. To make the simulation more realistic, they can either use FREVO or swarm intelligence algorithms to generate local rules as code for the allocation of software components in computing nodes, defining how the workload is distributed among the computing nodes. [Listing 3.2](#) shows the parameters for configuring a simulation. The results of the COSMOS simulation will also be used to evaluate the local rules for workload placement.

```
num_pods: 20
pods_per_step: 3
sim_steps: 20
num_{edge/fog/cloud}_devices: 6
behaviour_type: 'hba' and 'aco'
```

Listing 3.2: Simulation parameters for the example flow

4. **Analysis and Refinement:** The team analyzes the simulation data, identifying potential bottlenecks and areas for improvement. They use the insights gained from the simulation to refine the TOSCA model, adjust policies and constraints, and modify the local rules for agents. They may also decide to perform application profiling to gather more detailed information about the resource consumption of individual pods. This iterative process of modeling, simulation, and analysis continues until the team is satisfied with the system's performance and behavior.
5. **Exporting Artifacts:** Once the model is finalized, they export the TOSCA model as a .csar file, which will be used in subsequent steps for cross-platform deployment in the target computing nodes. They also export individual node templates and other artifacts as .tosca files for reusability. The local rules for workload placement are also exported from the design-space exploration tool to the runtime orchestrator.



This data flow example illustrates how the tools and concepts from Step 1 are used to design, simulate, and analyze a cloud-fog-edge architecture, using TOSCA as the common language, leading to the creation of a robust and efficient deployment blueprint.

The focus in the first phase of the project has been in enabling points 1-3 in the above example flow. The end-to-end integration to enable point 4 will be the focus of the second phase. As for point 5, an initial version of the export of .CSAR files is already available. This implementation is being validated in a collaboration with WP5-6, since the MIRTO agent takes the .CSAR files as primary artifact for deployment.

3.6.1 Gains w.r.t status quo

Traditionally, deploying complex cross-platform applications like the mobility use case involved manual configuration and trial-and-error, leading to inefficiencies and potential bottlenecks. The initial modelling step within this section revolutionizes that by utilizing the Modelio TOSCA Designer to define the application's architecture in a structured TOSCA file, enabling location-aware deployment and simulation-driven tuning with tools like DynAA or COSMOS. This allows for iterative optimization *before* deployment, exporting a comprehensive .csar file, and generating initial agent rules, effectively shifting from a reactive, error-prone status quo to a proactive, optimized deployment process.

Modelio TOSCA Designer positioning. With respect to the state of the art and practice in the area of TOSCA modeling, Modelio TOSCA Designer presents a compelling offering, particularly due to its unique integration with the other modeling capabilities of Modelio. The integration with UML and SysML within the Modelio platform allows architects to directly translate system designs into TOSCA models; this helps ensure consistency and reduces manual translation errors. Regarding deployment flows, the synergy with Modelio's BPMN capabilities enables the modeling of these workflows alongside the infrastructure topology, providing a holistic view. Finally, the unique integration with Modelio ADT Designer (introduced in [section 4.2](#)) provides a significant advantage for achieving the security-by-design requirement in the cloud-fog-edge continuum. This multi-viewpoint facilitates the modeling of cross-layer concerns. While commercial tools like Tricentis TOSCA⁹ may offer a wide range of features for TOSCA modeling (i.e. an AI copilot), and open-source tools like Eclipse Winery¹⁰, OCClware TOSCA Studio¹¹ and OpenTOSCA¹² are focused on producing artifacts for their own runtime environments, Modelio TOSCA Designer focus on end-to-end system modeling and standards-based compatibility with external tools for simulation and optimization, such as it is the case with the design-space exploration tools in the MYRTUS DPE (DynAA, FREVO and COSMOS), thanks to the capability of producing a specification-compliant .csar file describing the cloud-fog-edge system topology, which will enable cross-platform deployment by TOSCA-compliant runtime engines.

⁹ Tricentis Tosca. <https://www.tricentis.com/products/automate-continuous-testing-tosca>

¹⁰ Eclipse Winery. <https://projects.eclipse.org/projects/automotive.winery>

¹¹ OCClware TOSCA studio. <https://github.com/occiware/TOSCA-Studio>

¹² OpenTOSCA Ecosystem. <https://www.opentosca.org/>



DYNAA positioning. As part of this project, DynAA is being enhanced with an extension library that introduces models capturing the behavior of compute nodes and the execution of software components. This new library provides a detailed description of nodes and network resources across all layers, expanding DynAA’s ability to simulate large, distributed applications spanning cloud, edge, and fog environments. These enhancements will showcase DynAA’s efficiency and flexibility in evaluating large-scale systems and supporting time-critical, runtime applications. This progress also aligns with the initiative to open-source DynAA and foster a vibrant, engaged community around it.

FREVO positioning. Evolutionary Algorithm Frameworks for Multi-Agent and Swarm Systems leverage specialized tools to optimize complex interactions. DEAP (Distributed Evolutionary Algorithms in Python) provides a flexible Python-based environment for implementing genetic algorithms, particle swarm optimization, and other evolutionary strategies, with applications in traffic flow and swarm robotics [[Rainville-2012](#), [Krishna-2024](#)]. HeuristicLab offers a C# framework for heuristic and evolutionary algorithm design, featuring advanced visualization and cloud parallelization, ideal for prototyping optimization strategies in multi-agent contexts [[Wagner-2005](#)]. Additionally, the MCACEA (Multiple Coordinated Agents Coevolution Evolutionary Algorithm) framework employs separate evolutionary algorithms for each agent, focusing on both individual and cooperative objectives, making it suitable for optimizing multi-agent systems where coordination is essential [[Yu-2011](#)].

Evolutionary algorithms optimize swarm robotics, enhancing UAV formation adaptability [[Stolfi-2022](#)] and outperforming traditional methods via Egret Swarm Optimization (ESOA) [[Chen-2022](#)]. Research explores communication topologies in simulations like Webots [[Türkler-2022](#)], while neuro-evolution studies highlight reality gap challenges in real-robot applications [[Hasselmann-2021](#)].

FREVO (FRamework for EVolutionary design) [[Sobe-2012](#)] applies self-organization principles to engineer adaptive systems by evolving agent behaviors through genetic algorithms. It separates inputs into three components: problem evaluation, controller representation (e.g., neural networks, finite state machines), and optimization method (e.g., NNGA, GASpecies). The framework includes a GUI for simulating evolutionary processes and validating outcomes. FREVO supports diverse applications like cooperative robotics and smart grid optimization, demonstrated in a case study where evolved neural networks managed dynamic energy markets. Its modular design allows seamless component exchange, enabling scalable solutions for complex agent-based systems [[Fehervari-2010](#), [Elmenreich-2011](#), [Schranz-2025](#)].

In the MYRTUS project, the integration of FREVO with COSMOS represents a critical advancement in designing adaptive agent-based systems for the edge-fog-cloud continuum. FREVO’s evolutionary algorithms, which optimize local agent rules through genetic manipulation, require a robust simulation environment to evaluate candidate solutions iteratively. By developing a dedicated interface to COSMOS—a simulation framework modeling resource dynamics across edge, fog, and cloud layers—the project enables seamless evaluation of evolved behaviors in realistic scenarios. This integration allows FREVO to leverage COSMOS’s swarm intelligence simulations, where agents (e.g., microservices, edge



devices) interact within a dynamic resource environment, testing parameters such as neural network-based decision-making (behaviour_type: 'mlp'), pod arrival rates (μ : 0.5), and multi-layer resource allocations (num_edge_devices: 35, num_fog_nodes: 10). The interface supports evolutionary training loops, where FREVO generates candidate solutions (e.g., neural networks, finite state machines) and COSMOS evaluates their performance in tasks like workload orchestration, ensuring optimized swarm behaviors emerge through selective pressure. This synergy enhances MYRTUS's 360° vision by enabling data-driven, self-organized resource management strategies, validated through scalable and modular simulations.

COSMOS positioning. Agent-Based Modeling (ABM) is ideal for swarm algorithms in decentralized systems, using autonomous agents with local interactions to produce emergent behaviors [Umlauf-2022]. It excels in medium-scale systems (dozens to thousands of agents) and models neighbor-based communication and spatial constraints essential for swarms [Wilensky-2015, Schranz-2021].

In distributed computing, particularly across the edge-fog-cloud continuum, various platforms support resource allocation and scheduling. COSMOS uniquely integrates swarm intelligence with modular design, while other platforms offer distinct strengths and focuses, each balancing autonomy, coordination efficiency, and infrastructure complexity.

- ENIGMA is suited for multi-tier resource management and provides detailed evaluation metrics for scheduling strategies, supporting large-scale simulations with customizable configurations [Zyskind-2015].
- iFogSima, a popular toolkit for fog computing environments, boasts extensive documentation and a strong emphasis on energy consumption and latency analysis [Gupta-2017].
- CloudSim, a mature and widely-used framework for cloud computing simulations, benefits from a large user base and community support, offering high extensibility for cloud-specific research [Calheiros-2011].
- EdgeCloudSim, an extension of CloudSim tailored for edge computing scenarios, provides features for mobility modeling and network latency analysis in edge-cloud hybrid environments [Sonmez-2018].

COSMOS differentiates itself through swarm intelligence integration and edge-fog-cloud continuum modeling, employing modular Factory/Strategy patterns for unified API integration and code reuse. It enables decentralized scheduling and real-time simulation, catering to adaptive systems research in distributed environments.



4 Model to Implementation

4.1 Overview

This section focuses on the Step 2 of the Design and Programming Environment, that is dedicated to translating a model of the application into the implementation. To do so, it takes the models and the profiles of the applications that are produced in Step 1 and, after refining them and extending them (for instance with security functionalities), it generates the source code that will be optimized and deployed in Step 3. [Figure 4.1](#) shows an overview of step 2.

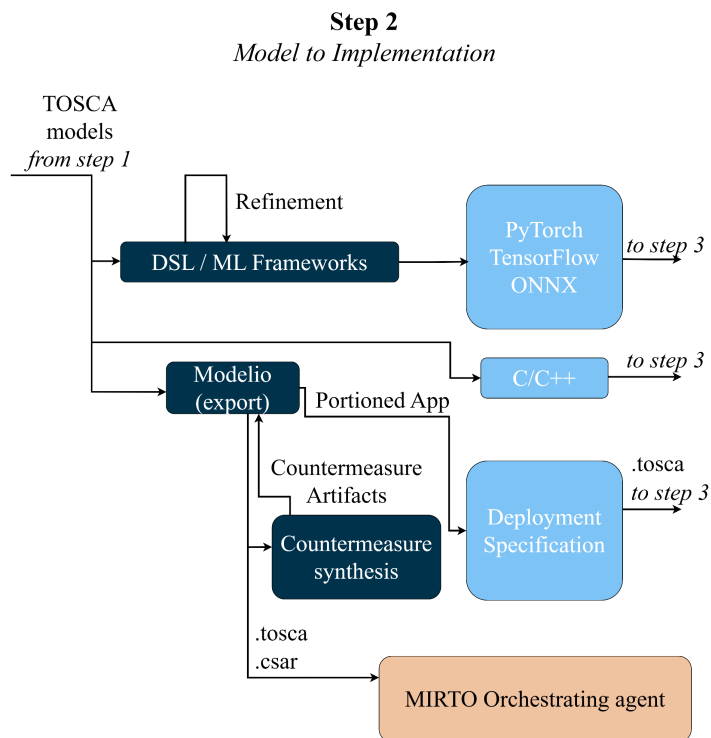


Figure 4.1: Step 2 overview

This section is structured as follows. At the beginning, the main components in [Figure 4.1](#) are introduced and presented. We start with Modelio and its extension, the Attack-Defense Tree (ADT) Designer, used for modeling the threat model within each application. We continue with the countermeasure synthesizer, that extends the application with countermeasures to counteract the identified threats. We conclude the introduction of the tools presenting the integration to DSL/ML Frameworks to program the software components of the applications. We then present an example of flow, focusing on the synthesis of countermeasures part, since it is the one developed from scratch in MYRTUS. We show how countermeasures are added to the application and how the security functionalities are included in the application topology. We conclude the section positioning the activities of this task within the state of the art and the current best practice on the modeling to implementation.



4.2 Modelio Attack-Defense Tree (ADT) Designer

In this section we focus on the description of the extension of Modelio that provides system architects with the facility to model threats, that we called Modelio ADT Designer. Modelio ADT Designer is an Open Source Modelling Tool conceived for designing ADTs. It is deployed as a module in the Modelio modelling environment.

ADT Designer allows users to design attack trees diagrams showing how an asset or a target might be attacked. These diagrams are intended for security specialists for modelling the attacks that occur on complex systems, such as cloud-fog-edge systems, and describe the events that lead to the attack in the form of a tree in which the attack is represented by the root element which is related with “OR” and “AND” conditions to the children that represent the sequence of events that lead to the root attack.

This module offers an ergonomic environment for designers and contains an additional set of features that allow users to configure security attributes for the attacks, such as the severity and the likelihood of the attack. It allows users to attach counter-measures to their attacks and to detect encountered attacks. Moreover, it allows referencing other trees, importing and exporting attack trees.

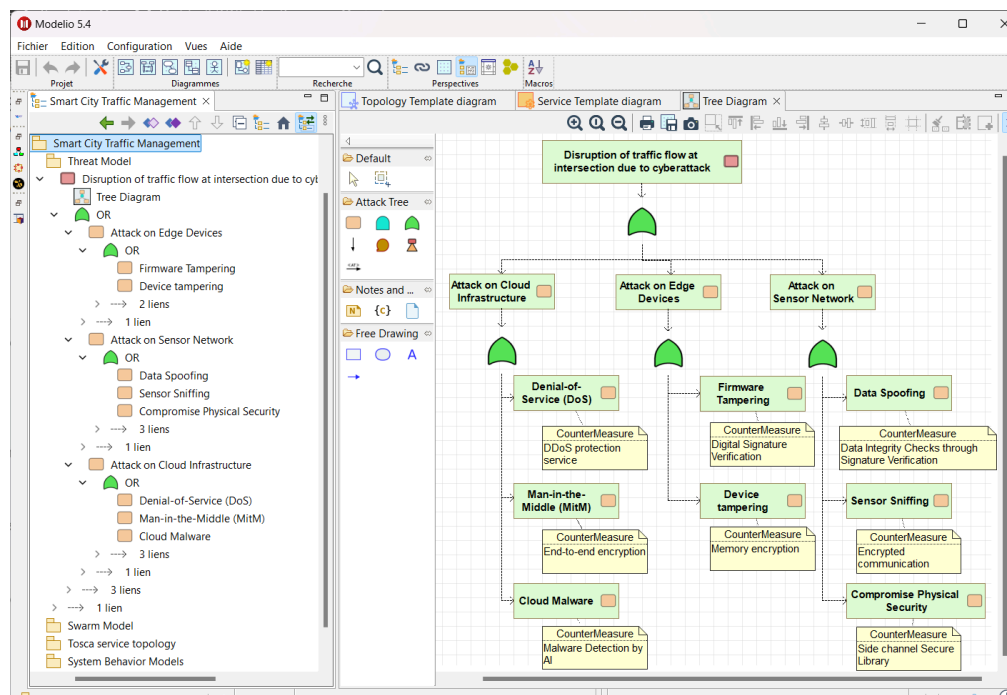


Figure 4.2: Modelio ADT Designer

Figure 4.2 depicts a screenshot of Modelio ADT Designer, showcasing a hierarchical threat model for the MYRTUS mobility use case. The central canvas displays the ADT, starting with a root node representing “Disruption of traffic flow at intersection due to cyberattack”, branching into attack vectors like “Attack on Cloud Infrastructure”, “Attack on Edge Devices”,



and “Attack on Sensor Network”. Each attack node is further detailed with specific attack methods, such as “Denial-of-Service (DoS)” and “Firmware Tampering”, and is associated with corresponding countermeasures, indicated by the yellow attached notes, like “DDoS protection service” and “Digital Signature Verification”, illustrating the mitigation strategies for each identified threat. It is worth noticing that these countermeasures will correspond to software component nodes that make part of the service topology. The yellow boxes detailing the countermeasures will be used by the countermeasure synthesizer to select from a library and instantiate the specific code of the countermeasure.

4.3 Countermeasure-Synthesizer

The countermeasure synthesizer is a tool designed completely from scratch during the MYRTUS project. In a nutshell, it takes a generic request of including a countermeasure in an application, and selects from a library a specific countermeasure that is instantiated into the application to mitigate a potential security threat. At high level, the Countermeasure-synthesizer thus requires two inputs: a library of specific countermeasures and their characteristics, and a request of countermeasure, and as output, gives an instantiation of a specific countermeasure (or a set of specific countermeasures) within the application.

Practically, the library of specific countermeasure is a table including the name of the countermeasure, a pointer to the source code implementing it, and the information about the countermeasure that will be used by the countermeasure synthesizer, such as the level of security (high, medium, low) and the performance of the countermeasure.

The synthesis process, which is later demonstrated in [Figure 4.3](#), starts with a request to instantiate a countermeasure. Such a request is specified in the Modelio ADT Designer, and can be generic (e.g. end-to-end encryption) or specific (e.g. end-to-end encryption with the AES 128 algorithm). These requests are specified in the ADT designed in Modelio and exported from there. In these initial phases of the development of the countermeasure synthesizer, the ADT is visually analyzed and the relevant information is stored in a text file that acts as input to the countermeasure synthesizer. In the next phases of the project, this step will be also automated extracting the relevant countermeasure request from the ADT directly by parsing the .csar file produced by Modelio.

The actual countermeasure synthesizer is currently provided as a docker container. It takes the inputs described above and it generates one or several docker containers each of them implementing one countermeasure. At the moment, each countermeasure specified in the ADT is implemented as a separate docker container that provides the underlying functionality of the defense measure. Depending on the exact functionality to be implemented, the container will include the needed executable or source code. In the synthesis step, the application is augmented with all the needed routines to allow for dynamically changing the countermeasures where needed. This adaptation will be managed by the MYRTUS Security Manager (presented in Deliverable D5.1). Because of this, the countermeasure synthesizer and the MYRTUS Security Manager will instantiate the execution of countermeasure in a similar way, via parametric command lines issued to the specific instances of a docker. Reporting here for clarity the example presented in Deliverable D5.1, if the underlying



functionality inferred by the ADT is encryption: the functionality can be accessed by running the following command:

```
docker run encryption -s [0/1/2] -d datafile -k keyfile -power [0/1/2] -tp [0/1/2]
```

This will execute the encryption on the datafile specified after the -d delimiter with the keyfile after the -k option. The executable presents three options 0/1/2 for the -s (security level), -power (power consumed) and -tp (throughput). For example, the argument 0/1/2 for the security level automatically invokes AES 128/192/256 as the underlying encryption primitives. An argument 0 for the -power option selects an executable which consumes low power etc. The command line to run the executable implementing the countermeasures is returned back to Modelio, as indicated in [Figure 2.1](#). Using docker containers adds considerable flexibility to the synthesis step. It makes it easy to change any particular module in the countermeasure artifacts, and enables the aforementioned dynamic adaptation of security primitives by the MYRTUS Security Manager. At the moment, each docker container implementing countermeasures is created from scratch by a) taking the software code of each countermeasure from a library and b) compiling the docker container so that it essentially serves as the executable of the countermeasure.

4.4 Third-party tools: DSL/ML Frameworks

Third-party tools are used to code the behavior of software components (those described in the high-level model of the application in Step 1, see in [Section 3](#)). To this purpose, the DPE supports code from Machine Learning (ML) frameworks and from Digital Signal Processing (DSP) applications. As can be seen in [Figure 2.1](#), the “DSL/ML Frameworks” box under Step 2 (Model to Implementation) offers an input to the DPE’s Step 3, enabling easy and optimized deployment of DSP/ML applications. Particularly, user-defined DSL/ML applications (i.e., defined in [Figure 2.1](#) Step 1 through “Existing SW components code base”), can be defined in terms of, for example, an ML Framework like PyTorch¹³. Besides, before compilation, DSP/ML components can go under a series of refinement steps (see arrow over “DSL/ML Frameworks” box). These refinements can either prepare the DSP/ML applications for our compilation flow (planned support of NumPy and other ML frameworks) or even add further optimizations to adhere to high-level requirements derived from the KPI evaluation in Step 1. Given the application-specific and data-dependent behaviors of such refinements, refinements are implemented as pre-compilation tasks in the source framework. This is a common development practice that could serve as a blueprint to interface other DSLs to the DPE in the future.

As an example, assume a DSP application written in Messner¹⁴, a NumPy-like DSL. When targeting an integer-only target (e.g., a RISC-V core without floating-point unit), it is required to convert the application to an integer-only version. To that end, tasks like stability evaluation [[Kester-2003](#)] can be performed in the refinement step, before exporting an integer version of

¹³ PyTorch. <https://pytorch.org/>

¹⁴ <https://github.com/everest-h2020/messner/tree/impl/execution>



the application to Step 3. These refinement tasks are implemented *by the user* in its framework of choice (the same used in the application code base, NumPy in this case). Other possible refinements are compression optimizations of Deep Neural Networks (DNNs), notably quantization and pruning [Deng-2020]. They may be applied due to target-specific restrictions (e.g., fitting the model to the target available memory) or to further improve the DNN execution (e.g., reducing bandwidth requirements with quantized and pruned DNNs). It is important to note that refinements are, by default, the responsibility of the user. In case of modern DSL compilers, the user can refer to a performance engineer that can steer the optimizations and refinements using meta-programming capabilities of the DSL compiler.

In the scope of ML acceleration, hardware-software co-design is crucial to achieve high performance and to fit into resource-constraint edge devices. For such a goal, methods like quantization are fundamental to refine an initial model depending on the target hardware for deployment. **QKeras**¹⁵ is an extension of the Keras deep learning framework, designed to enable **quantization-aware training (QAT)**. It provides layers and operations that simulate fixed-point arithmetic during training, which helps developers create neural networks optimized for resource-constrained devices such as FPGAs. QKeras is particularly valuable in **edge AI** scenarios where power efficiency and memory footprint are critical. It supports mixed-precision models and works seamlessly with TensorFlow/Keras workflows, making it suitable for integration into custom toolchains. This latter aspect is further enhanced by the possibility to export models in QONNX that provides quantization support for models in the **ONNX** format. Note that MLIR has an ONNX dialect and that the IREE importer currently supports ONNX.

4.5 Example Flows

In this section, we present as an example, the flow followed to synthesize a countermeasure. As previously described, some of the steps where different tools have to be integrated are currently done manually. These steps will be automated in the second phase of the project.

Any underlying functionality is accessible through command line arguments. There is a top level security entity that accepts all the characteristics required from the security component as arguments. For example if the user needs to hash a given datafile, a request has to be forwarded to this entity with the datafile, security level and other performance characteristics as inputs. The entity passes the request to the corresponding docker container which deals with hashing, and the result is forwarded back to the entity and then to the address through which the request originated. The docker container has the required source code and executables to synthesize exact countermeasure scenarios. [Figure 4.3](#) shows an example flow. The steps are summarized as follows:

¹⁵ <https://github.com/google/qkeras>

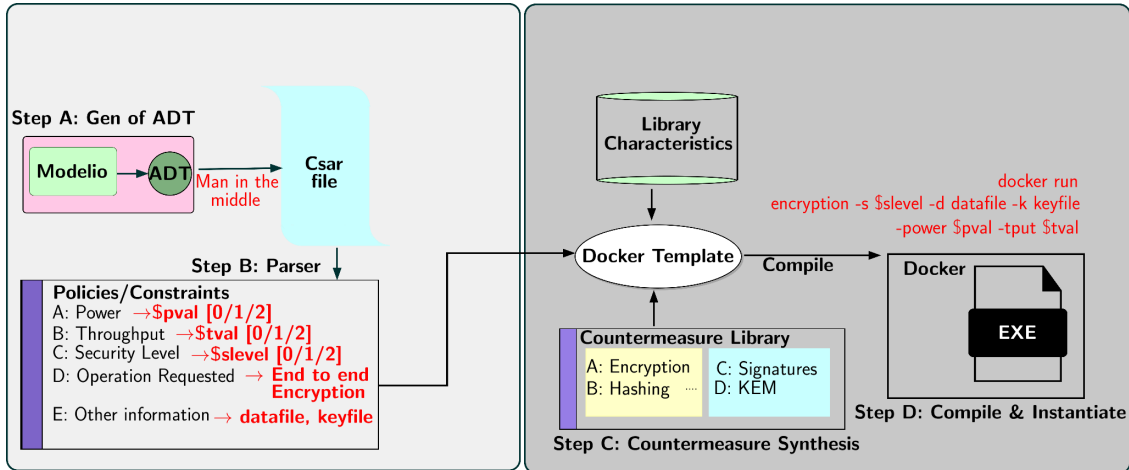


Figure 4.3 Example flow for countermeasure synthesis

- a) **Generation of the ADT:** The first step of the flow for implementing countermeasures is the definition of the ADT for the application. In the ADT, the threats to which the application is exposed are identified and annotated. Together with the threats, countermeasures addressing and mitigating these threats are also reported in the ADT. This step is done in Modelio. [Figure 4.1](#) reports as example the ADT of the mobility use case, where we can see that, among the requirements, there is “Man in the middle” as possible threats that will be mitigated by the “End-to-end Encryption” countermeasure. The ADT, including the specification of the countermeasure, is saved in the .csar file, which is ready for being parsed.
- b) **Model Parser:** The second step of the process is the parsing of the .csar file to extract the information needed to synthesize the countermeasure. In detail, this step parses the csar file produced by Modelio which contains the ADT. It extracts the countermeasure request from the ADT. The relevant information that it should be able to extract are a) the security functionality requested (encryption, hashing, ...), b) the security level of the operation (Low, medium or high), c) if needed, other extra functional requirements (whether power level should be low, any requirements on throughput), and d) any other relevant information to be passed on to the module performing the operation (this could be datafile, keyfile required to do encryption etc). In the above example of “Man in the middle”, the information to be extracted is “End-to-End encryption”, which means that the parser will pass to the synthesizer only the request to instantiate an encryption algorithm.
- c) **Countermeasure synthesis:** based on the information extracted from the parser, the synthesizer searches in the library of countermeasures and selects a specific countermeasure that satisfies the request extracted from the .csar file. In case specific extra-functional requirements are specified in the .csar file, the synthesizer will attempt to select a countermeasure that fulfills all of them. When no extra functional requirements are specified, the synthesizer selects the countermeasure offering the required security functionality that is characterized by good performance in all the



target platforms. The synthesizer returns the selected countermeasure, the command line to execute it, and, where relevant, a message about the meeting of the extra-functional requirements. In the running example for this section, where there was as a request just the presence of an encryption algorithm, the synthesis of the countermeasure will select to use, as countermeasure, a software implementation of the AES 128 bit algorithm, corresponding to the low level security with medium performance and medium power consumption.

- d) **Countermeasure instantiation and compiling of the Docker container:** This step starts from the output of the countermeasure synthesis step, that consists of the selected countermeasure. After selecting the countermeasure, the code needed to execute the countermeasure should be instantiated. At the moment, the code is instantiated within a Docker container. The docker container for instantiating the countermeasure is created starting from a template, that is then extended with the source code of the countermeasure. The source code of the countermeasure is taken from the library of countermeasure and is placed in the docker. The docker and all the software in it are then compiled. Following the example in this section, where the synthesized security primitives is AES 128 with medium performance, the docker code instantiated will be executed with the command “**encrypt -s 0 -power 1 -tput 1**” is meant to invoke AES-128 (corresponding to the lowest security level). Additionally, the power consumption is to be kept medium and a reasonable throughput is requested. Note that the requirements might contradict each other. This is not the case in this example but consider for instance a requirement of high throughput and low power. Usually high throughput needs high power consumption. The synthesizer resolves this contradiction and prioritizes throughput over power.

4.5.1 Gains w.r.t status quo

Modelio ADT designer. In the field of security threat modeling, various methodologies and tools present varying advantages. The STRIDE methodology¹⁶, popularized by Microsoft, is distinguished by its ability to identify threats from the initial design phases, particularly within software development lifecycles; however, an explanation of defense mechanisms is often lacking. The Microsoft Threat Modeling Tool¹⁷, which implements the STRIDE methodology, offers an interface for visualizing potential threats. However, its primary focus on software does not fully address the complexities inherent in distributed architectures such as cloud-fog-edge. OWASP Threat Dragon¹⁸, for its part, constitutes an open-source alternative focused on web applications and considerations relating to confidentiality. Although adaptable, its application may require extensions to accurately capture the hardware-software interactions prevalent in cloud-fog-edge environments. In this state of the art, the positioning of Modelio ADT Designer is distinguished by the use of ADTs, allowing for explicit modeling of

¹⁶ Microsoft. The STRIDE Threat Model.

[https://learn.microsoft.com/en-us/previous-versions/commerce-server/ee823878\(v=cs.20\)](https://learn.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20))

¹⁷ Microsoft Threat Modeling tool.

<https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool>

¹⁸ OWASP Threat Dragon. <https://owasp.org/www-project-threat-dragon/>



both attack vectors and countermeasures. In addition, its integration within the Modelio ecosystem, and more particularly with Modelio TOSCA Designer, offers an advantage by enabling a close coupling of security considerations with the design and deployment of cloud-fog-edge systems. This integration proves particularly relevant for projects such as MYRTUS, where security imperatives must be integrated from the early stages of the system development life cycle. It is therefore distinguished by the clarity of its representations of threats and defense countermeasures and their alignment with cloud-fog-edge topology design with Modelio TOSCA designer.

Countermeasure-Synthesizer positioning. Automatic synthesis and applications of security countermeasure is an active research topic. Among the efforts in this direction, certainly the one of automatic protection against side channel attack is one of the most relevant ones [Tiri-2004, Popp-2005, Bayrak-2011]. These approaches are however at a lower level compared to the one we are developing in the countermeasure synthesizer. They, in fact, either take as input an unprotected software routine and, by means of a compiler, produce a side channel resistant executable, or take as input an unprotected HDL implementation of a circuit and produce a protected hardware netlist. The countermeasure synthesizer operates at a higher level, starting from an ADT produced in Modelio and automatically instantiates the cryptographic functionalities to counteract a threat identified in the model.

The countermeasure synthesizer is to be seen as a variant between a circuit compiler like Synopsys DC-Shell¹⁹ and the GNU C compiler²⁰, which converts code into an instruction set for the underlying CPU. A circuit compiler converts source code written in a hardware description language to a set of gates in a cell library. It is possible to request the compiler to design the circuit to be low in area or having very low end to end latency etc. Similarly, a typical code compiler, converts a program written in a high level language into a series of instructions. The synthesizer converts a request of countermeasure into an instantiation of a countermeasure selected from a pool of countermeasures belonging to a library. The countermeasure synthesizer carries out the selection based on parameters such as power consumption, energy, throughput and security level, providing automation at an abstraction level that a circuit or a program compiler can not reach. It draws this intelligence from a) extensive prior benchmarking of implementation styles both in software and hardware and b) circuit level understanding of trade-offs between power/energy/throughput. To our understanding, this kind of tool would be one of a kind and unique.

Also, another novelty of our approach, is that the countermeasure synthesizer aims to create a subsystem implementing a cryptographic countermeasure that is suitable for the computing continuum computational paradigm. To achieve that, we are currently relying on Docker. Docker, in fact, integrates all system level dependencies under one entity. While the construction of the docker container currently includes manual steps, in the future we intend to automate also this process. Despite using Docker as target technology, it is important to

¹⁹ Synopsys Design Compiler

<https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html>

²⁰ GCC, the GNU Compiler Collection, <https://gcc.gnu.org/>



underline however, that the approach proposed is sufficiently generic to be adapted to other virtualized environments.



5 Node-Level Optimization and Deployment

The third step in the MYRTUS DPE is in charge of the final deployment of key tasks (which map to software components or part of them) in the application, while providing the right abstraction layer for avoiding vendor lock-in issues. These tasks, or kernels (e.g., selected within the application at previous DPE steps), are compiled in a single MLIR-based compilation flow. Particularly, this compiler integrates new and existing MLIR *dialects*²¹ for (i) supporting MYRTUS cross-platform deployment, (ii) performing kernel- and task-level optimizations, (iii) insert adaptation mechanisms, and (iv) communicating with external frameworks and supporting MYRTUS Cognitive Engine with feedback from the tasks.

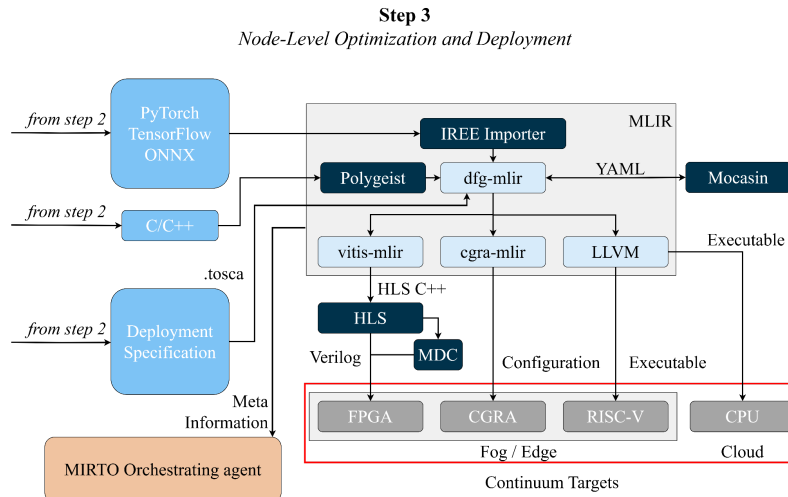


Figure 5.1 Step 3 Overview

In the following sections, we provide an overview of Step 3, which is shown in [Figure 5.1](#), (along with the interface to other steps and the modifications with respect to previous deliverables — [Section 5.1](#)), and present our custom MLIR dialects implementing our compiler ([Section 5.2](#)). Next, we detail the frameworks used in the DPE, namely, Mocasin used for DSE ([Section 5.3](#)), MDC for execution of workloads on reconfigurable platforms ([Section 5.4](#)), and the CGRA Mapper for generation of CGRA configurations ([Section 5.5](#)). In [Section 5.6](#) we define the third-party tools that will be leveraged in Step 3. Finally, in [Section 5.7](#), we showcase Step 3 compilation flow with a use-case driven application, and point out the gains with respect to status quo deployment flows.

5.1 Overview

This step generates the executables and bitstreams required to execute and configure the various computing nodes. Multiple application variants can be provided to the MIRTO Cognitive Engine to support runtime adaptability. This is achieved by exporting

²¹ In MLIR, dialects are namespaces holding specific operations, types, and transformations.



meta-information about these variants, detailing resource types and system-level trade-offs, as explored in [Korol-2023, Smejkal-2024].

To ensure interoperability of multiple applications and targets, the MYRTUS DPE extends a common MLIR framework originally proposed in the EVEREST project [Pilato-2024]. This framework will enable:

- Importing third-party code from either Domain-Specific Languages (DSLs) [Katol-2018, Susungi-2018] (as already supported in EVEREST) or ML models (e.g, PyTorch, ONNX [Manca-2024]).
- Integrating third-party tools, such as polyhedral compilers, for optimization.
- Compiling for different targets, from reconfigurable architectures [Soldavini-2023] to CPUs including customizable RISC-V cores (as initially covered in EVEREST) and for the MYRTUS targets (MDC, CGRA, and graph-base FPGA designs).

Besides, the Mocasin²² DSE tool is extended to (i) support Coarse-Grain Reconfigurable Architectures (CGRA), and (ii) to support modelling and simulation of the reactor dataflow Model of Computation (MoC).

To enable interoperability in the MLIR-based framework utilized in this step, several key dialects have been selected:

- dfg-mlir²³ for modeling applications as dataflows. This dialect will perform a series of dataflow optimizations, as well as bridge the frontend MLIR dialects (e.g., for linear algebra) to the supported targets through its lowerings.
- cgra-mlir for generating CGRA configurations. This dialect will identify and map basic blocks in the application to the CGRA. This dialect will also make use of the Mocasin tool for exploring different CGRA mappings at compile time.
- Tel [Rink-2019] (extended with a Numpy-like front end Messner) for numerical kernels, supporting custom data types via the base2-mlir dialect [Friebe-2023].
- reactor (currently, under definition) for enabling a MLIR-based reactive adaptive execution in MYRTUS.

Besides the key dialects, existing MLIR tools will be leveraged to support multiple inputs (e.g, torch-mlir, Polygeist [Moses-2021]) and enable targeting (dfg-mlir and vitis-mlir, see Subsections 5.2.1 and 5.2.2):

- FPGAs, generating IPs via Vitis²⁴ High-Level Synthesis (HLS) tool, which feeds into MDC for runtime reconfigurable accelerators.
- CPUs/GPUs, compiling through LLVM IR.

The deployment specification will be transferred from Modelio to dfg-mlir in TOSCA format (YAML implementation). The remaining application components will be compiled to standard compilers, ensuring seamless interoperability with the accelerated sections.

²² <https://github.com/tud-ccc/mocasin>

²³ <https://github.com/Feliix42/dfg-mlir/tree/dev-myrtus>

²⁴ <https://www.amd.com/de/products/software/adaptive-socs-and-fpgas/vitis.html>



5.2 MLIR-based compiler

MLIR is a promising framework for constructing reusable and extensible compiler infrastructure. It aims to tackle software fragmentation, enhance compilation for diverse hardware systems, considerably lower the expenses associated with developing domain-specific compilers, and facilitate the integration of existing compilers.

In this section, we introduce several MLIR dialects that represent key intermediate representations and transformations, bridging the gap between high-level modeling and low-level hardware-specific optimizations and accelerator generation.

5.2.1 dfg-mlir

The dfg-mlir dialect was created as part of the EVEREST project to enable the modeling of different types of Dataflow Graphs (DFG). With a set of lowerings and custom operations for defining the graph's nodes and channels, dfg-mlir facilitates implementing such DFGs (e.g., by offering implicit mechanisms for pulling and pushing data between nodes), while supporting code transformations to multiple targets. For MYRTUS, dfg-mlir was extended to operate on tensors and enable operations such as convolution (e.g., for supporting neural networks).

Another extension taking place for dfg-mlir regards the support for FPGA execution. Lowerings from dfg-mlir are designed for targeting different hardware platforms, enabling parallel execution. The CPU backend, for instance, lowers to OpenMP. FPGA execution, in the original dfg-mlir dialect, was provided via lowerings using the CIRCT²⁵ project, which provides not only a way to perform HLS using MLIR framework, but also various tool flows such as simulation, tracing, and placement generation. However, using CIRCT HLS path for generating Verilog code has a limitation of lacking crucial optimizations.

For instance, one of the most important optimizations for an FPGA hardware design, instruction-level pipelining, is not implemented along the HLS path within CIRCT, causing performance degradation when compared to more mature solutions like Vitis from AMD. Additionally, there is no native support in CIRCT for floating point numbers, which limits the expressiveness of the dfg-mlir dialect. In this context, we have replaced the original dfg-mlir FPGA code generation with a new dialect (called vitis-mlir) that generates HLS code from a MLIR representation.

5.2.2 vitis-mlir

The vitis-mlir dialect provides an IR specifically designed to capture the semantics and coding style used in Vitis HLS tool. This dialect encapsulates key constructs that facilitate the representation of hardware-oriented programming paradigms and supports the application of compiler directives, commonly known as HLS pragmas. To achieve this, vitis-mlir defines appropriate lowering and translation passes, converting operations from standard MLIR

²⁵ <https://circt.llvm.org>



dialects such as `arith`²⁶ (arithmetic operations) and `memref`²⁷ (memory reference operations) into Vitis-compliant HLS C++ code. Within the MYRTUS DPE, the `vitis-mlir` dialect plays a crucial bridging role by enabling the translation from the higher-level, dataflow-centric representation provided by `dfg-mlir` into synthesizable code suitable for FPGA implementation via Vitis tools.

It is noteworthy that `vitis-mlir` is not limited solely to generating HLS code adapted for Vitis HLS tool. Within the MYRTUS project, we use an AMD Xilinx FPGA board at the edge layer, so that the Vitis HLS code generation was naturally the first choice. However, though simple extensions, which are the HLS pragmas, `vitis-mlir` can support other HLS tools such as Intel Quartus Prime tooling. Additionally, open-source HLS tool Bambu [Ferrandi-2021] can be used as backend as well, since the Vitis HLS pragmas are compatible.

5.2.3 `cgra-mlir`

The `cgra-mlir` dialect provides a software infrastructure to support the automatic acceleration of computationally intensive sections of code of a given application on a general-purpose CGRA accelerator. The proposed compilation infrastructure is based on MLIR to deploy general-purpose application code (in C/C++).

It encapsulates CGRA-specific operations, abstractions, and execution semantics. This dialect supports hardware-aware optimizations by defining operations such as load/store, arithmetic computations, and control primitives. It models dynamic reconfiguration capabilities, represents computations mapped to individual Processing Elements (PEs), and includes constructs for configuring the CGRA's interconnections. The work carried out in this project is organized into three steps that interact closely: DFG extraction, hardware/software partitioning, and CGRA hardware configuration.

- DFG extraction — In order to start a CGRA-specific compilation path from C/C++, the DFG must be extracted from the intermediate representation obtained with Polygeist. First, a vectorization transformation is used to provide vector operations from scalar code, leveraging the inherent parallelism of CGRAs. Then, through a custom pass, a DFG is obtained from the MLIR representation so that it can be directly converted to the `dfg-mlir` dialect. This extraction delivers precise representations of computations for efficient scheduling and mapping to the CGRA processing elements. Nodes in the DFG represent computations, while edges denote data dependencies.
- Hardware/software partitioning — Hardware/software partitioning determines which parts of an application should be executed on the CGRA fabric (hardware) and which ones on the host CPU (software). This partitioning is key to achieving an optimal utilization of resources, good performance, and energy efficiency. At this point, after the transformations in the `dfg-mlir` dialect, an identification of basic operations to be executed on the accelerator is carried out. With pattern matching, existing operations

²⁶ <https://mlir.llvm.org/docs/Dialects/ArithOps/>

²⁷ <https://mlir.llvm.org/docs/Dialects/MemRef/>



are transformed in order to become suitable for the deployment. This may be fusing two very simple operations (such as a multiplication and addition that is being accumulated) or converting high-level operations into loops and more basic operations. With the use of Mocasin, a profiling that analyzes energy consumption, memory access patterns, and latency can be done automatically. This selects the candidate code to be accelerated on the CGRA and is annotated using the `cgra-mlir` dialect for the final compilation steps regarding the program offloading onto the CGRA system. This partitioning method ensures transparency to the user. After lowering to an intermediate representation, custom passes are then used to generate hardware-specific intermediate representations that involve tailoring code for the CGRA as well as optimized code for the CPU, dividing the DFG into these two sub-graphs.

- CGRA hardware configuration — The graph structure of the subdivided DFG is used directly in the mapping. However, the synchronization between the host CPU and the CGRA accelerator must still be done. This is implemented through the extension of the instruction set to synchronize the execution and to manage data transfers between the CPU and the CGRA, with the use of the MLIR dialect. An example of a multiply-accumulate operation written in C and transformed with `cgra-mlir` is shown in [Figure 5.2](#).

```
1  module {
2      func.func @mac(%arg0: memref<?xi32>, %arg1: memref<?xi32>) -> i32 {
3          %c0_i32 = arith.constant 0 : i32
4          "cgra.deploy" {
5              %0 = affine.for %arg2 = 0 to 100 iter_args(%arg3 = %c0_i32) -> (i32) {
6                  %1 = affine.load %arg0[%arg2] : memref<100xi32>
7                  %2 = affine.load %arg1[%arg2] : memref<100xi32>
8                  %3 = "cgra.mac"(%1, %2) : (i32, i32) -> i32
9                  affine.yield %3 : i32
10             }
11         }
12         return %0 : i32
13     }
14 }
```

Figure 5.2 cgra-mlir dialect example

5.3 Mocasin

Mocasin is an open-source rapid prototyping framework for simulating and exploring the mapping of software applications onto heterogeneous Multi-Processor System-on-Chip (MPSoC) platforms. It is implemented in Python and designed as a research tool to help developers and researchers quickly prototype and evaluate new algorithms for task mapping, scheduling, and resource allocation on multicore systems. Unlike many existing MPSoC design tools that target specific use cases or models, Mocasin aims to be a flexible and generic environment that abstracts and integrates approaches from various other tools. In essence, its purpose is to provide a unified playground for experimenting with different MoCs and mapping strategies without having to build a custom simulator from scratch.



Mocasin has following key features and capabilities:

1. Support for different MoCs: Mocasin can represent applications using different MoCs (e.g., task graphs, Khan Process Network (KPN), Synchronous Data Flow (SDF), etc.), rather than being limited to one.
2. Extensible platform and application modeling: The framework provides built-in data structures to model applications (as directed graphs of tasks and communication edges), target platforms (processing elements, memory, interconnect, etc.), and mappings between them. Users can freely define custom hardware platforms, specifying features like clusters of cores, different types of processing elements, and on-chip interconnect (e.g., a Network-on-Chip), using provided templates.
3. Multiple mapping strategies: Mocasin comes with numerous mapping algorithms and strategies implemented out-of-box. These include static allocation, heuristic algorithms, genetic algorithms, and user-provided mapping methods.
4. DSE, simulation and tracing: Mocasin provides an adjustable high-level simulator to estimate performance metrics for a given application-to-platform mapping, which can be obtained from the built-in DSE. The framework can automatically explore different task-to-processor assignments and scheduling options to find Pareto-optimal solutions under multiple objectives (i.e. execution time and energy consumption). While performing simulations, Mocasin will generate execution traces that detail the timeline of task executions and communications.

5.4 MDC

The Multi-Dataflow Composer (MDC)²⁸ is a framework designed to facilitate the development and management of **coarse-grained reconfigurable (CGR) accelerators**. It enables the creation of multi-dataflow architectures, allowing dynamic reconfiguration of hardware accelerators to support multiple execution modes. MDC plays a key role in optimizing edge computing by supporting runtime reconfigurability and resource sharing.

MDC supports **dataflow-based** reconfigurable computing, where execution is expressed in terms of data dependencies rather than sequential control flow. This paradigm enables efficient **multitasking** and **multithreading** on FPGA MPSoCs. MDC supports **multitasking execution** by dynamically reconfiguring accelerators to accommodate multiple tasks within the same hardware. This is achieved through **resource sharing**, where computational units common to multiple tasks are shared across different configurations. In addition to multitasking, MDC enables **multithreading execution** by supporting the concurrent execution of multiple dataflow threads within a single accelerator. This is achieved through fine-grained resource sharing, where processing elements are dynamically assigned to different threads based on availability and execution demands.

By leveraging a **streaming architecture**, MDC is also well-suited for AI acceleration at the edge, enabling efficient processing of machine learning workloads with minimal latency. The newly

²⁸ <https://github.com/mdc-suite/mdc>



developed QONNX2MDC frontend²⁹ allows the automatic derivation of neural networks accelerators starting from arbitrary quantized models expressed in QONNX. To do that, the complete toolchain leverages Vitis HLS and MDC for the hardware generation. The combination of the custom approximate models of QONNX, custom data type supported by Vitis HLS and the runtime reconfigurability support of MDC allows obtaining runtime adaptive inference engines for neural networks that can adapt at runtime their precision and, consequently, power consumption.

5.5 CGRA mapper

As it was explained in [Section 5.2.3](#), the DFG is subdivided into the part to be deployed on the CGRA and the code to be executed on the RISC-V core.

For the hardware acceleration, so far, the mapping is done with a simple integer linear programming optimization that takes into account the DFG structure directly, assigning a PE to each of the DFG nodes. It will be connected with Mocasin.

Once the mapping is successfully generated, the information from the DFG subdivision is retrieved for the generation of the final binaries through a unified process. The section of the sub-graph which is not targeted for the CGRA is lowered to a RISC-V binary via LLVM. During execution, the main program handles CGRA configuration and synchronization of mapped operations using custom ISA extensions.

5.6 Third-party tools

5.6.1 IREE Importer

The IREE importer is a component of the Intermediate Representation Execution Environment (IREE³⁰) compiler toolchain that brings models from popular machine learning frameworks into MLIR (Multi-Level Intermediate Representation). Its primary purpose is to bridge the gap between ML frameworks (like PyTorch, TensorFlow, ONNX) and IREE's MLIR-based compiler, enabling a unified workflow for optimization and deployment. In practice, each framework's model (e.g. a PyTorch `nn.Module`, a TensorFlow `SavedModel`, or an ONNX `.onnx` file) is exported to a standard format and then imported into IREE's MLIR using dedicated tools. This conversion produces an MLIR representation of the model, which will then be fed into DPE's compiler for further optimization and code generation.

5.6.2 Polygeist

Within the MLIR ecosystem, Polygeist serves as a crucial bridge that enables advanced optimizations on C/C++ and OpenMP programs. By translating these programs into MLIR with rich structural information, Polygeist allows the use of MLIR's powerful optimization passes and multi-level transformation pipeline on traditional software code. High-level optimizations,

²⁹ <https://github.com/mdc-suite/gonnx2mdc>

³⁰ <https://iree.dev>



such as loop nest optimizations, data layout transformations, and automated parallelization, become easier because the code is represented in a high-level MLIR dialect (SCF/Affine) rather than opaque machine-level IR. For OpenMP-parallel code, Polygeist is extended [Thangamani-2024] to keep the parallel structure explicit, which means MLIR passes can analyze parallel loops or even introduce parallelism where appropriate (e.g., turning a sequential affine loop into a parallel loop if it has no dependencies). This is particularly important for performance tuning on modern architectures, as it allows exploiting multicore and vector units more effectively. Polygeist helps general-purpose programs leverage MLIR's domain-specific optimizations and vice versa. It essentially broadens MLIR's applicability beyond machine learning or DSLs, to include legacy and systems programming code, which can now be optimized with the same infrastructure.

5.6.3 Vitis Design Tools

AMD's Vitis design tool suite supports FPGA and MPSoC development, covering low-level Register Transfer Level (RTL) design (Vivado), HLS design with C/C++ (Vitis HLS) and embedded system design (Vitis). AMD also provides PYNQ³¹, which is a Python-based alternative to Vitis.

- Vivado is the primary FPGA design and implementation tool, enabling HDL-based development, synthesis, place-and-route, and verification. It supports RTL design with Hardware Description Language (HDL), such as Verilog and VHDL, along with IP integration and timing analysis.
- Vitis HLS allows FPGA hardware to be designed using C/C++ instead of HDL. It automatically converts high-level algorithms into optimized HDL logic with multiple pre-defined compiler directives (known as pragmas), enabling faster prototyping with features like C-based simulation, pipelining, and parallelization for performance optimization.
- PYNQ is an open-source framework that lets developers use Python and Jupyter Notebooks to control FPGAs, making hardware acceleration accessible without deep FPGA and embedded system design expertise. It is widely used for embedded applications on ZYNQ SoCs and Kria SOMs, which is the hardware at the edge node in the computing continuum.

5.7 Example Flows

In this section, we present two of the supported flows (for FPGA execution) in the DPE's step 3 (see Figure 2.1). Let us assume a convolutional neural network (CNN) application (such as the one covered in the MYRTUS mobility use case) as our driving example.

5.7.1 Example for CNN compilation flow with dfg-mlir

Given that deployment specification and partitioning (produced by Modelio at the Model to Implementation step) are available, the kernels that will go through the DPE compilation and optimization are known to the MLIR compiler.

³¹ <https://www.pynq.io>



```
func.func @cnn(%input: tensor<1x8x8x3xf32>,
               %w1: tensor<16x3x3x3xf32>,
               %b1: tensor<16xf32>,
               %w2: tensor<32x3x3x16xf32>,
               %b2: tensor<32xf32>) -> tensor<1x8x8x32xf32> {
  // First layer
  %input_zp1 = "tosa.const"() <{values = dense<0.0> : tensor<1xf32>}> : () -> tensor<1xf32>
  %weight_zp1 = "tosa.const"() <{values = dense<0.0> : tensor<1xf32>}> : () -> tensor<1xf32>
  %conv1 = tosa.conv2d %input, %w1, %b1, %input_zp1, %weight_zp1 {
    acc_type = f32,
    pad = array<i64: 1, 1, 1, 1>,
    stride = array<i64: 1, 1>,
    dilation = array<i64: 1, 1>
  } : (tensor<1x8x8x3xf32>, tensor<16x3x3x3xf32>, tensor<16xf32>, tensor<1xf32>, tensor<1xf32>)
    -> tensor<1x8x8x16xf32>
  %zero1 = "tosa.const"() <{values = dense<0.0> : tensor<1x8x8x16xf32>}> : () -> tensor<1x8x8x16xf32>
  %relu1 = tosa.maximum %conv1, %zero1 : (tensor<1x8x8x16xf32>, tensor<1x8x8x16xf32>) -> tensor<1x8x8x16xf32>

  // Second layer
  // Omitted similar code, the result is %relu2
  return %relu2 : tensor<1x8x8x32xf32>
}
```

Figure 5.3 2-layer CNN in MLIR

When imported to MLIR, these kernels are represented by one of the DPE's frontend dialects. In this case, let us assume that the partitioning done in step 2 of the DPE consists of a 2-layer CNN kernel, within which each layer has a 2D convolution and a ReLU activation. Then, the imported MLIR looks like [Figure 5.3](#).

<pre>dfg.process @layer1 inputs(%input: tensor<1x8x8x3xf32>, %weight: tensor<16x3x3x3xf32>, %bias: tensor<16xf32>) outputs(%output: tensor<1x8x8x16xf32>) { %in = dfg.pull %input : tensor<1x8x8x3xf32> %w = dfg.pull %weight: tensor<16x3x3x3xf32> %b = dfg.pull %bias: tensor<16xf32> // layer1 content, result in %relu dfg.push (%relu) %output : tensor<1x8x8x16xf32> } // Similar layer2, omit implementation dfg.process @layer2 // Same inputs and outputs as the original func.func dfg.region @cnn inputs(...) outputs(...) { // Define channels to connect layer nodes %channel0:2 = dfg.channel() : tensor_type // Instantiate layers with channel ports // Omit implementation dfg.instantiate @layer1 dfg.instantiate @layer2 } }</pre>	<pre>// Helper function vitis.func @mem2stream_f32() vitis.func @stream2mem_f32() // CNN nodes (omit implementation) vitis.func @layer1(%input: !vitis.stream<f32>, %weight: !vitis.stream<f32>, %bias: !vitis.stream<f32>, %output: !vitis.stream<f32>) { // vitis loop to pull data from input ports // vitis arith/array ops for the calculation // vitis loop to push data to output port } vitis.func @layer2() // Top function (omit implementation) vitis.func @cnn() { vitis.call @layer1() vitis.call @layer2() }</pre>
---	--

(a)

(b)

Figure 5.4 2-layer CNN in dfg-mlir and vitis-mlir

The next step in the flow is the dfg-mlir that will translate this kernel to a DFG graph with the appropriate interface (i.e., input/output buffers and channels connecting nodes). The output of dfg-mlir is presented in [Figure 5.4a](#). Since, the compilation target in this example is the FPGA, lowerings in dfg-mlir will translate the code in [Figure 5.4a](#) to a vitis-mlir representation ([Figure 5.4b](#)). At this point in the flow, the code can be mapped to an HLS representation, which is done by lowerings available in vitis-mlir.

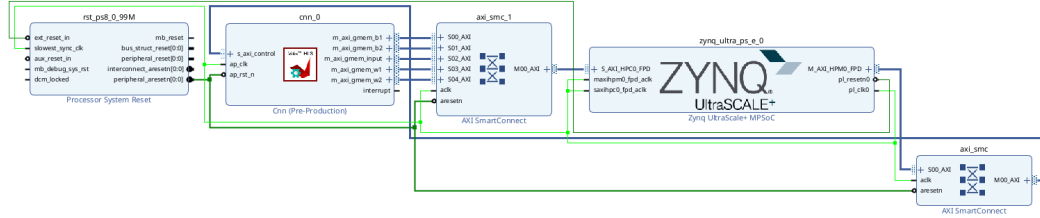


Figure 5.5 Final Block Design of 2-layer CNN

Finally, this code can be fed to a third-party tool (Vitis HLS) to be synthesized to an HLS IP, which will be integrated to an FPGA accelerator in Vivado using the generated scripts along with the code. [Figure 5.5](#) shows a block diagram of the generated kernel.

5.7.2 Example flow for adaptive CNNs with MDC

The CNN and its quantization levels can be modeled through the QKeras API as shown in [Figure 5.6](#).

```
x = x_in = Input(mnist_input_shape[1:])

x = (QConv2D(32, (3,3), padding='same', kernel_quantizer= quantized_bits(bits=8, integer=4, alpha=1),
  bias_quantizer= quantized_bits(bits=8, integer=4, alpha=1)))(x)
x = (QActivation(quantized_relu(bits=16, integer=8, use_sigmoid=0, negative_slope=0.0), name="act_1"))(x)
x = (MaxPool2D(pool_size=(2,2)))(x)
x = (QConv2D(32, (3,3), padding='same', kernel_quantizer= quantized_bits(bits=8, integer=4, alpha=1),
  bias_quantizer= quantized_bits(bits=8, integer=4, alpha=1)))(x)
x = (QActivation(quantized_relu(bits=16, integer=8, use_sigmoid=0, negative_slope=0.0), name="act_2"))(x)
x = (MaxPool2D(pool_size=(2,2)))(x)
x = (Flatten())(x)
x = (Dropout(0.5))(x)
x = (QDense((10), kernel_quantizer= quantized_bits(bits=8, integer=4, alpha=1),
  bias_quantizer= quantized_bits(bits=8, integer=4, alpha=1)))(x) # num_classes = 10
x = (Activation(activation='sigmoid', name='out_activation'))(x)

model = Model(inputs=x_in, outputs=x)
```

Figure 5.6 QKeras description of a quantized CNN to be ported on an FPGA with the MDC toolchain

The developer can specify multiple models with different levels of quantization and train them with quantization-aware training. Once defined and trained, the models can be exported to the standard QONNX format through the qonnx utility function.

From the qonnx input files, the newly developed QONNX2MDC frontend generates all the files needed by MDC and VITIS HLS. These are:

- *Network topology (.xdf)*: definition of the dataflow graphs to be merged by MDC Frontend;
- *Actors interface (.cal)*: definition the actors considered by MDC;
- *Actors definition (.cpp)*: implementation of the layers of the CNN to be synthesized by Vitis HLS. The resulting hardware is then passed to the MDC Backed to generate the reconfigurable accelerator;
- *Automation scripts (.tcl)*: automation scripts for the Vitis HLS synthesis;

With this design flow, it is possible to derive runtime adaptive inference engines for CNNs. Indeed, QONNX supports custom quantization levels. Each quantization level corresponds to a



different working point, considering the accuracy VS energy trade-off. Using more bits for representing activations and weights leads to a higher accuracy, but also to higher resource consumptions. This is combined with the capability of MDC of merging working points to generate a runtime reconfigurable accelerator. The result is the possibility of adapting the accuracy and power consumption of the inference engine at runtime, depending on the application's needs.

A complete example of this design flow is available in the QONNX2MDC repo³².

5.7.3 Gains w.r.t status quo

With this example flow, it also becomes clearer the contributions of the MYRTUS DPE with respect to the status quo approaches for optimizing and deploying applications on the continuum.

On one hand, we have most state-of-the-art MLIR-based HLS frameworks [[Wang-2021](#), [Ciambra-2022](#), [Ye-2022](#), [Zhang-2024](#), [Basalama-2025](#)] proposing compilation flows that assume that inputs are in affine representations (i.e., leaving to the user the responsibility of *lowering* the application down to this abstraction level). Contrarily, the Node-Level Optimization and Deployment step improves over the state-of-the-art tools by enabling an easier entry to MLIR (and, thus, to the MYRTUS supported backends). This is achieved by supporting inputs represented in higher levels of abstraction, such as the `tosa` and `linalg` dialects. For example, conversions to these dialects are covered and maintained by most of the ML frameworks³³, allowing users to compile complex tasks like DSP and ML models to HLS FPGA modules with a single compiler.

On the other hand, state-of-the-art domain-specific frameworks, most prominent for ML applications (e.g., the MLIR-based IREE³⁴ and the non-MLIR TVM³⁵), provide deployment and optimization mechanisms for CPU, GPU, and accelerators. With respect to these state-of-the-art tools, the main improvements of the MYRTUS DPE are two-fold. First, in a single compiler, the DPE supports a wider range of targets (from FPGA to CGRA and CPU). Second, the DPE supports external optimization tools for a cloud-to-edge deployment. This is done by: offering an interface with application-level design tools (in case of MYRTUS, the [continuum modeling](#) and [implementation](#) steps) through TOSCA files; and, by providing meta information on the compiled task so that external orchestrators can optimize the application across multiple execution nodes.

From the perspective of each tool, we have the following positioning w.r.t the state-of-the-art:

Compiler Positioning: Regarding the compiler activities covered in this deliverable (dkg-mlir and vitis-mlir), the proposed MLIR-based compiler advances the state-of-the-art of HLS code generation by enabling inputs and optimizations from a higher abstraction level (as discussed

³² <https://github.com/mdc-suite/qonnx2mdc/tree/main/examples>

³³ <https://github.com/llvm/torch-mlir> and <https://github.com/onnx/onnx-mlir>.

³⁴ <https://iree.dev/>.

³⁵ <https://tvm.apache.org/>.



above). First, this eases the adoption of the compiler, given that it provides an easier entry point for applications like ML. Second, it allows graph-level optimizations to take place prior to the lower level HLS code generation. On top of that, we plan to extend this approach to enable parallel streaming nodes based on the semantics of the input application, providing a more direct path from algorithm representation to optimized hardware implementation.

Mocasin Positioning: Mocasin, contrarily to other existing modeling tools for exploring and researching, MoC focuses on the application properties that are relevant for rapid performance estimation, shortening the design and evaluation time of hardware platforms and applications. For MYRTUS, Mocasin will be (a) extended to accept inputs from external sources by creating a new frontend (i.e., integrated to MLIR, advancing the interoperability of state-of-the-art modeling tools); and, (b) integrated and extended to support DSE at compile-time of CGRAs.

MDC Positioning: State-of-the-art tools like HLS4ML³⁶ and FINN³⁷ efficiently generate FPGA-based streaming CNN accelerators tailored to specific models but lack runtime adaptivity. The proposed flow, by exploiting the coarse-grained runtime reconfiguration capabilities of MDC and integrating with the custom quantization offered by QKeras introduces runtime adaptability, enabling dynamic execution of multiple NN profiles on reconfigurable hardware.

CGRA Mapper Positioning: alternative solutions from the state-of-the-art targeting CGRA mapping, like CGRA-ME [Ragheb-2024] or Morpher,³⁸ also trigger the mapping stage starting from intermediate representations obtained after running a front-end compiler stage. However, all of these solutions rely on the LLVM framework for the compilation and intermediate representation, as opposed to the MYRTUS proposal, which relies on MLIR to enable seamless interconnection with additional optimization mechanisms (e.g., polyhedral transformations). Although these optimizations have been successfully applied to software [Moses-2021], in MYRTUS we will extend them to target CGRA-based hardware accelerators.

³⁶ <https://github.com/fastmachinelearning/hls4ml>

³⁷ <https://github.com/Xilinx/finn?tab=readme-ov-file>

³⁸ <https://github.com/ecolab-nus/morpher>



6 Towards End-to-End Flows

In the first phase of the project, we focused on tool development and defining interfaces within the three steps of the DPE. This has been carried out as planned originally, as described in Sections 3-5. As an outlook for the next phase, this section provides a conceptual end-to-end flow through the DPE, from the Continuum Modeling step to the Model Implementation and Node-Level Optimization steps. To this end, we take the MYRTUS mobility use case as our guide example.

Step 1. As seen in [Section 3.6](#), application developers and system architects can start by defining, in a TOSCA file, the application architecture in the Modelio TOSCA Designer. For the mobility use case, this means defining TOSCA nodes representing the edge (camera), fog, and cloud computing nodes. Besides the connection between nodes, users can also specify the location (i.e., computing node) where each Service Chain Component will be deployed. Most importantly, in this initial DSE step, the system can be simulated using either DynAA or COSMOS tools (that can ingest the TOSCA description). The output of these simulations is a set of data points on resource utilization, latency, and power dissipation that can be exploited by the users to tune the application (e.g., fix eventual application bottlenecks by editing the TOSCA file). The application architecture can then be exported as .csar file to the second DPE step. This .csar file contains the specification of all computing nodes and all software components. In the mobility use case, the .csar file contains not only the orchestration of nodes, but also actual references of the self-contained software components to be deployed. Additionally, this step outputs the local rules for agents that will feed the MIRTO Agent.

Step 2. The second DPE step will read the produced .csar file and parse the file to get the necessary countermeasures. The countermeasure synthesis component suggests a set of appropriate security countermeasures, each one in the form of a self-contained software component, to be included in the TOSCA service topology under construction.

The user then proceeds to add the suggested software components to the TOSCA model of the service topology, in TOSCA Designer. With the attacks and countermeasures modeled (updated TOSCA file), the countermeasures are implemented with a set of precompiled executables or source codes (countermeasure artifacts) to be included in the TOSCA file for later deployment.

For ML and DSP components, this step offers an additional feature for refining these components. This refinement will act on the components alone without changing the TOSCA definition. As explained in [Section 4.4](#), this refinement can, for instance, apply a change of numeric format (e.g., quantization) of an ML model. Refinements applied in this step target the optimizations and the targets supported in the third step of the DPE.

An important task of the DPE second step is the partitioning of the application (selecting which software components will be fed to the third step). This partitioning is done by tagging the software components that need optimization in the .csar (TOSCA) file. For example, an ML model that was refined in this step can be selected for further optimization and compilation in the third step. Therefore, the output of step 2 is the components selected for node-level



optimization along with the initial deployment specification (e.g, Execution target for the components).

Step 3. The last step of the DPE accepts the C/C++ and ML/DSP components that were previously selected for optimization. In this step, it is generated optimized executables (or bitmaps, in case of FPGA execution) for deployment at the edge. The step is implemented by an MLIR-based compiler that, based on the target and requirements defined in the deployment specification, will perform a set of optimizations. For example, for components targeting FPGA execution, the compiler in Step 3 will be able to generate either a dataflow graph implementation of the component in HLS. Or, in the case of a DNN, leverage tools like MDC for DNN-specific FPGA accelerators optimization. On top of that, step 3 can generate a set of operating points with a diverse set of performance-power profiles, for example, that are fed to the MIRTO agent that can exploit them to adapt the application. In the mobility use case, the component selected for optimization, a DNN, can be compiled with many different FPGA implementations (e.g., by exploring different HLS pragmas in dfg-mlir) that will return in different power dissipation and performance levels.



7 Conclusion and Future Work

In this deliverable, we have provided an overview of the current status of the diverse tools that comprise the MYRTUS DPE. We put emphasis on the interfaces within the three steps of the DPE and sketched example flows within the steps and across the steps (see [Section 6](#)). The achievements in Step 1 ([Section 3](#)) concretely addresses the objectives T3.OO.a, T3.OO.b, and T3.OO.e, the achievements in Step 2 ([Section 4](#)) addresses objectives T3.OO.a, and T3.OO.b, while Step 3 ([Section 5](#)) addresses objectives T3.OO.a, T3.OO.c, T3.OO.d, and T3.OO.e. In terms of results, the work carried out so far is aligned with the planned program. More concretely:

- R15 Modelio: Preview release of TOSCA Designer v0.1 available. It currently supports 80% of TOSCA specification elements. The initial validation plan consists of 13 integration tests defined in two use cases.
- R16 DynAA: Transformation of topology system descriptions into DynAA models and demonstration of system simulations using DynAA.
- R18 FREVO: Implementation of one interface to FREVO and first results in the design of swarm behavior with this tooling loop.
- R19 Mocasin: 3 out of the following 5 features are now supported: modelling changing topology, modelling time-triggered actions, modelling input from external sources for runtime adaptation, modelling CGRA and platform-side reconfiguration, calibrated models for simple kernels with error below 20%.
- R20 Lingua Franca: We decided not to use Lingua Franca as the programming language as a result of the requirement analysis phase (see [Deliverable D1.1](#)). However, we do model elements of the reactor model in MLIR. Today, this includes acyclic reaction graphs, single timer and logical actions.
- R21 Code-to-CGRA compiler: Proper application mapping onto the CGRA for the manually mapped code sections of all application benchmarks, and measurable performance improvements when compared against the reference software baseline running on the same host processor.
- R22 MDC: A standalone version of MDC is available with multi-threading support partially integrated. Approximate computing is supported through the new QONNX2MDC frontend for approximate adaptive CNN inference. The standalone version and the new frontend are demonstrated in a publicly available tutorial.
- R23 Interoperability layer: As shown in the example flows, interoperability of 2–3 tools has been demonstrated.

Apart from planned results, two new results have been added to the project as a reaction. These are:

- R24 COSMOS (see [Section 3.5](#)): Simulation framework with 2 baseline algorithms (random, greedy) and one swarm algorithm (ant-based algorithm) running and ready for evaluation.



- R25 Countermeasure Synthesizer (see [Section 4.3](#)): Inputs and outputs and functionalities of the synthesizer have been defined. Generation of test script to instantiate the countermeasure completed. Parsing of constraints will be ready.

An analysis of the KPIs indicates that WP7 is well on track with respect to the original plan. More concretely:

- KPI3.1 — At least 2 MYRTUS tools and 2 external ones are made MLIR-interoperable: As discussed in this deliverable (see [Section 5.7](#)), 3 internal tools have been made compatibility with MLIR (dfg application model, MDC and CGRA tooling) and interfacing to the iree external tool has been implemented.
- KPI3.2 — Transparent support for at least 2 different HW-based computing devices using common entry points: Support provided for MDC, CGRA and RISC-V computing devices.
- KPI3.2 — 10X productivity improvement using MYRTUS DPE vs fragmented toolchains. To really assess this KPI, we need to have more complete flows within the DPE. However, it is worth mentioning that the integration with commercial tools (e.g. vitis) and automatic insertion of HLS pragmas is known to provide a high productivity boost to software developers with little or no expertise with FPGA devices.
- KPI3.4 — At least: 1) 10-20% additional energy efficiency and processing/communication latency reduction, and 2) 10% quicker adaptation, with respect to those achievable by using MIRTO engine alone. This KPI also requires a more complete DPE. With respect to the first point, recent evidence demonstrates 5-10% gains in energy efficiency via approximate computing (see [Section 5.7](#)).



8 References

- [Basalama-2025] Basalama, S. and J. Cong (2025). “Stream-HLS: Towards Automatic Dataflow Acceleration”. Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 103–114.
- [Bayrak-2011] A. G. Bayrak, F. Regazzoni, P. Brisk, F. Standaert, P. lenne. “A first step towards automatic application of power analysis countermeasures”. Proceedings of the 48th Design Automation Conference, p. 230–235.
- [Calheiros-2011] Calheiros, R. N., R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya (2011). “CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms”. Software: Practice and Experience 41.1, pp. 23–50. doi: 10.1002/spe.995.
- [Chen-2022] Chen, Z., A. Francis, S. Li, B. Liao, D. Xiao, T. Ha, J. Li, L. Ding, and X. Cao (2022). “Egret Swarm Optimization Algorithm: An Evolutionary Computation Approach for Model Free Optimization”. Biomimetics 7.4, p. 144. doi: 10.3390/biomimetics7040144.
- [Ciambra-2022] Ciambra, P. F. (2022). “laRa: dataflow programming with the MLIR framework= laRa: programação dataflow com a ferramenta MLIR”. PhD thesis. [sn].
- [Deng-2020] Deng, L., G. Li, S. Han, L. Shi, and Y. Xie (2020). “Model compression and hardware acceleration for neural networks: A comprehensive survey”. Proceedings of the IEEE 108.4, pp. 485–532.
- [Elmenreich-2011] Elmenreich, W. and I. Fehérvári (2011). “Evolving self-organizing cellular automata based on neural network genotypes”. LNCS 6557, pp. 16–25.
- [Fehérvári-2010] Fehérvári, I. and W. Elmenreich (2010). “Evolving neural network controllers for a team of self-organizing robots”. Journal of Robotics.
- [Ferrandi-2021] Ferrandi, F., V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo (2021). “Bambu: an open-source research framework for the high-level synthesis of complex applications”. 2021 58th ACM/IEEE Design Automation Conference (DAC). IEEE, pp. 1327–1330.
- [Filho-2016] Filho, J. O., T. Vogel, and J. de Gier (2016). Runtime Services and Tooling for Reconfiguration. Springer.
- [Friebe-2023] Friebe, K. F., J. Bi, and J. Castrillon (2023). “BASE2: An IR for binary numeral types”. Proceedings of the 13th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies, pp. 19–26.
- [Gupta-2017] Gupta, H., A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya (2017). “iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments”. Software: Practice and Experience 47.9, pp. 1275–1296. doi: 10.1002/spe.2509.
- [Hasselmann-2021] Hasselmann, K., A. Ligot, J. Ruddick, N. Bredeche, A. Winfield, and A. Christensen (2021). “Empirical assessment and comparison of neuro-evolutionary methods for the automatic off-line design of robot swarms”. Nat Commun 12, p. 4345. doi: 10.1038/s41467-021-24642-3.
- [Karol-2018] Karol, S., T. Nett, J. Castrillon, and I. F. Sbalzarini (2018). “A domain-specific language and editor for parallel particle methods”. ACM Transactions on Mathematical Software (TOMS) 44.3, pp. 1–32.
- [Kester-2003] Kester, W. (2003). Mixed-signal and DSP design techniques. Newnes.



- [Korol-2023]** Korol, G., M. G. Jordan, M. B. Rutzig, J. Castrillon, and A. C. S. Beck (2023). "Design Space Exploration for CNN Offloading to FPGAs at the Edge". 2023 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). IEEE, pp. 1–6.
- [Krishna-2024]** Krishna, S. A., D. Narayana, D. Prasad, and R. S. Kumar (2024). "A Comparative Study on using Genetic Algorithm with DEAP framework on Different Optimization Problems". 7.
- [Manca-2024]** Manca, F., F. Ratto, and F. Palumbo (2024). "Onnx-to-hardware design flow for adaptive neural-network inference on fpgas". International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation. Springer, pp. 85–96.
- [Menard-2021]** Menard, C., A. Goens, G. Hempel, R. Khasanov, J. Robledo, F. Tewelett, and J. Castrillon (2021). "Mocasin—rapid prototyping of rapid prototyping tools: A framework for exploring new approaches in mapping software to heterogeneous multi-cores". Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings, pp. 66–73.
- [Moses-2021]** Moses, W. S., L. Chelini, R. Zhao, and O. Zinenko (2021). "Polygeist: Raising C to polyhedral MLIR". 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE, pp. 45–59.
- [Pilato-2024]** Pilato, C., S. Banik, J. Beránek, F. Brocheton, J. Castrillon, R. Cevasco, R. Cmar, S. Curzel, F. Ferrandi, K. F. Friebe (2024). "A system development kit for big data applications on fpga-based clusters: The everest approach". 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, pp. 1–6.
- [Popp-2005]** T. Popp, S. Mangard. "Masked Dual-Rail Pre-charge Logic: DPA-Resistance Without Routing Constraints". Cryptographic Hardware and Embedded Systems – CHES 2005. Lecture Notes in Computer Science ((LNCS, volume 3659)) p. 172–186.
- [Ragheb-2024]** O. Ragheb et al., "CGRA-ME 2.0: A Research Framework for Next-Generation CGRA Architectures and CAD," 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 642-649, doi: 10.1109/IPDPSW63119.2024.00124.
- [Rainville-2012]** De Rainville, F.- M., R. Islam, G. Trédan, E. Granger, and D. Courtois (2012). "DEAP: A python framework for evolutionary algorithms".
- [Rink-2019]** Rink, N. A. and J. Castrillon (2019). "TelL: a type-safe imperative Tensor Intermediate Language". Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, pp. 57–68.
- [Schrantz-2021]** Schrantz, M., M. Umlauf, and W. Elmenreich (2021). "Bottom-up Job Shop Scheduling with Swarm Intelligence in Large Production Plants." SIMULTECH, pp. 327–334.
- [Schrantz-2025]** Schrantz, M., A. Farshad, and W. Elmenreich (2025). Engineering Swarms of Cyber-Physical Systems. CRC Press.
- [Smejkal-2024]** Smejkal, T., R. Khasanov, J. Castrillon, and H. Härtig (2024). "E-Mapper: Energy-Efficient Resource Allocation for Traditional Operating Systems on Heterogeneous Processors". arXiv preprint arXiv:2406.18980.
- [Sobe-2012]** Sobe, A., I. Fehérvári, and W. Elmenreich (2012). "FREVO: A tool for evolving and evaluating self-organizing systems".
- [Soldavini-2023]** Soldavini, S., K. Friebe, M. Tibaldi, G. Hempel, J. Castrillon, and C. Pilato (2023). "Automatic creation of high-bandwidth memory architectures from domain-specific languages: The case of computational fluid dynamics". ACM Transactions on Reconfigurable Technology and Systems 16.2, pp. 1–34.



- [**Sonmez-2018**] Sonmez, C., A. Ozgovde, and C. Ersoy (2018). "EdgeCloudSim: An environment for performance evaluation of edge computing systems". *Transactions on Emerging Telecommunications Technologies* 29.11, e3493. doi: 10.1002/ett.3493.
- [**Stolfi-2022**] Stolfi, D. and G. Danoy (2022). "An Evolutionary Algorithm to Optimise a Distributed UAV Swarm Formation System". *Appl. Sci.* 12.20, p. 10218. doi: 10.3390/app122010218.
- [**Susungi-2018**] Susungi, A., N. A. Rink, A. Cohen, J. Castrillon, and C. Tadonki (2018). "Meta-programming for cross-domain tensor optimizations". *ACM SIGPLAN Notices* 53.9, pp. 79–92.
- [**Thangamani-2024**] Thangamani, A., V. Loechner, and S. Genaud (2024). "Extending Polygeist to Generate OpenMP SIMD and GPU MLIR Code". 30th International European Conference on Parallel and Distributed Computing-PhD Symposium.
- [**Tiri-2004**] K. Tiri, I. Verbauwhede. "A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation", *Proceedings Design, Automation and Test in Europe Conference and Exhibition, Vol 1*, p. 246-251.
- [**Türkler-2022**] Türkler, L., T. Akkan, and L. Akkan (2022). "Usage of Evolutionary Algorithms in Swarm Robotics and Design Problems". *Sensors* 22.12, p. 4437. doi: 10.3390/s22124437.
- [**Umlauf-2022**] Umlauf, M., M. Schranz, and W. Elmenreich (2022). "Simulation of Swarm Intelligence for Flexible Job-Shop Scheduling with SwarmFabSim: Case Studies with Artificial Hormones and an Ant Algorithm". *Proceedings of the International Conference on Simulation and Modeling Methodologies, Technologies and Applications*. Springer, pp. 133–155.
- [**Varzonova-2025**] Varzonova, N. and M. Schranz (2025). "COSMOS: A Simulation Framework for Swarm-Based Orchestration in the Edge-Fog-Cloud Continuum". *Proceedings of the International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*.
- [**Wagner-2005**] Wagner, S. and M. Affenzeller (2005). "HeuristicLab: A Generic and Extensible Optimization Environment". doi: 10.1007/3-211-27389-1_130.
- [**Wang-2021**] Wang, J., L. Guo, and J. Cong (2021). "AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA". *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 93–104.
- [**Wilensky-2015**] Wilensky, U. and W. Rand (2015). *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo*. MIT Press.
- [**Ye-2022**] Ye, H., H. Jun, H. Jeong, S. Neuendorffer, and D. Chen (2022). "ScaleHLS: a scalable high-level synthesis framework with multi-level transformations and optimizations". *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 1355–1358.
- [**Yu-2011**] Yu, X., S. Dong, and H. Wang (2011). "A method of COA based on multi-agent co-evolutionary algorithm". 8005.
- [**Zhang-2024**] Zhang, W., J. Zhao, G. Shen, Q. Chen, C. Chen, and M. Guo (2024). "An Optimizing Framework on MLIR for Efficient FPGA-based Accelerator Generation". *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, pp. 75–90.
- [**Zyskind-2015**] Zyskind, G., O. Nathan, and A. Pentland (2015). "Enigma: Decentralized Computation Platform with Guaranteed Privacy". *arXiv preprint arXiv:1506.03471*. url: <https://arxiv.org/abs/1506.03471>.