Software Engineering Institute
Carnegie Mellon University

TwinOps: Digital Twins Meets DevOps

Jérôme Hugues
Joe Yankel
John Hudak
Anton Hristozov

March 2022

TECHNICAL REPORT
CMU/SEI-2022-TR-001
DOI: 10.1184/R1/19184915

Software Solutions Division

http://www.sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100

DM22-0259

Table of Contents

List of Figures

Executive Summary

This technical report summarizes the findings of the TwinOps project, a one-year project executed during FY20.

TwinOps researched the engineering of cyber-physical systems (CPSs). CPSs exhibit multiple engineering, validation and verification (V&V), and testing challenges. In this project, we aimed at reducing the time to deliver a software-intensive system by improving engineering and testing activities.

TwinOps explored the interplay between three core technologies:

- Model-based engineering (MBE): An engineering approach that relies on models as first-class abstractions of a system to support engineering activities.
- DevOps: An organizational effort to support continuous delivery of software through a better coupling between development and operations activities.
- Digital twins: An infrastructure to support system monitoring and diagnosis in real time to enable continuous system improvement.

By the conclusion of FY20, the SEI achieved the following outcomes through its research on TwinOps:

- The SEI delivered a ModDevOps example that extends DevOps by incorporating MBE and V&V capabilities. We demonstrated how MBE model processing capabilities enable rapid system prototyping.
- The SEI created an enhanced analysis and testing process for systems architects who build software-intensive CPSs by defining the TwinOps process.
- The SEI used TwinOps to build on ModDevOps and digital twins to collect data on a system at runtime and compares it to other engineering artifacts: model simulation and analysis. This comparison enables rapid system diagnosis.

Both ModDevOps and TwinOps processes are documented as SysML models. These models support a full definition of the process. Hence, these processes can be reviewed and adapted to other project needs.

Two case studies using a combination of modeling languages (SysML, Architecture Analysis and Design Language [AADL], and Modelica [in addition to C programming]) illustrate these two contributions. We used Azure IoT and GitLab forge as supporting DevOps infrastructure.

Abstract

This report summarizes the contributions of the TwinOps project, a one-year project funded by the Software Engineering Institute and executed during FY20. The contributions of this research are twofold. First, it introduced ModDevOps as an innovative approach to bridging model-based engineering and software engineering using DevOps concepts and code generation from models. ModDevOps smooths the transition from model-level verification and validation (V&V) to software production. Second, the research developed TwinOps, a specific ModDevOps pipeline that equips system engineers with new analysis capabilities through the careful combinations of model artifacts as they are built.

1 Summary

In this chapter, we provide a summary of the TwinOps project, its context, and its key results.

1.1 Context and Objectives of the TwinOps Project

The increased complexity of cyber-physical systems (CPSs) is causing a wide range of undefined behaviors. Harmful issues, such as imprecise component characterization (in the functional, timing, or safety viewpoints), or emergent system behaviors, such as system deadlocks or erratic behaviors, often emerge during testing or after system deployment. These unwanted behaviors cause significant and costly rework and delay system delivery.

DevOps for software systems, and digital twins for CPSs, have recently emerged as two interesting technologies for improving CPS engineering. Our initial research objective was to ease the deployment of simulations or instrumented CPS through DevOps, which uses continuous integration, deployment, and real-time monitoring of the whole system, while leveraging the digital twins concept to review the software-centric view of monitored data with actual physical parameters. Yet, the connection between DevOps and digital twins relies on specific engineering artifacts and processes. The TwinOps project aimed to define them.

The SEI MBE team advocates for model-based technologies. We claim that models can address some of these concerns. To do so, we want to combine multiple classes of models:

- *systems engineering models* to capture system requirements, interfaces, and the system's functional decomposition into subsystems
- *simulation models* to evaluate the system general behavior
- *engineering models* for the system's design, and then model transformations towards analytical models (e.g., for model checking, performance evaluation) and, finally, code generation.

We note these classes of models are usually considered in isolation and developed concurrently. We claim that these models can be combined in a uniform process to improve the whole engineering process.

In this report, we first define *ModDevOps*, an extension of DevOps that leverages model-based-engineering approaches to improve system continuous integration/continuous delivery (CI/CD). ModDevOps is defined as a generic process.

TwinOps is a specialization of ModDevOps aimed at CPS engineering, verification and validation (V&V), and deployment. TwinOps combines DevOps practice and model-based code generation practice to facilitate system deployments for multiple targets to build the simulation testbench, validation platform, and digital twins of a cyber-physical system.

The TwinOps project builds on the foundations of DevOps and digital twins infrastructures. *DevOps* is a software development process that relies on an iterative workflow combining the development and operation of software, from coding to deployment on a monitored runtime infrastructure, either simulated or on an actual platform. DevOps is supported by a collection of cloud technologies to encourage automation in building, testing, and deployment activities. *Digital*

twins combine the simulation of engineering models (using the Functional Mock-up Interface [FMI] standard) with run-time system monitoring to analyze deployed CPS [Blochwitz 2012]. The concept of digital twins has been used in various domains to improve system quality (e.g., manufacturing and automotive). Digital twins rely heavily on Internet of Things (IoT) and cloud-based technologies to collect and propagate data.

1.2 Delivered Contributions

TwinOps aims at expediting the testing phase of system development by generating most of the software that can be deduced from a system's architectural model, and its testbench, in a single unified process. System architecture virtual integration (SAVI) studies demonstrated that a significant number of errors are discovered and mitigated during the integration and acceptance phases of a project. This phenomenon is reflected in an increase in the required testing and/or retesting effort [Feiler 2009].

By relying extensively on code generation and linking it to a test bench or a digital twin, TwinOps reduces the number of faults leaking through to later phases of CPS engineering at design time, so engineers will have more confidence in the system under construction, and also at runtime, to allow the instrumentation to improve system verifiability.

In the following chapters, we present the following contributions:

- Definition and example of a *ModDevOps* process (see Section 3): We define ModDevOps, an extension DevOps that incorporates MBE engineering and V&V capabilities. We demonstrate how MBE model processing capabilities enable rapid system prototyping.
- Definition of *TwinOps* process (see Sections 4 and 5): TwinOps builds on ModDevOps and digital twins to collect data on a system at runtime and compare it to other engineering artifacts: model simulation and analysis. This comparison enables rapid system diagnosis. We detail the TwinOps process as follows:
  - In Section 4, we introduce an instantiation of ModDevOps to engineer and deploy a CPS using a DevOps pipeline, and we illustrate this process with a case study that highlights the key steps of ModDevOps to provide a "systems engineer view" of the process.
  - In Section 5, we present this case study in more depth. We introduce all the models built and the specific configuration of the CI/CD pipeline we implemented.

We evaluate the contributions of TwinOps first by the capability to synthesize required code to deploy systems, their simulations, and their corresponding digital twin; and second, by the capability to collect runtime execution traces.

2 ModDevOps: Coupling Model-Based Engineering and DevOps

In this section, we first introduce the main contribution of the TwinOps project: *ModDevOps*, a coupling of model-based engineering and DevOps.

In Section 2.1, we review relevant technologies to the research. Section 2.2 presents model-based systems and software engineering. In Section. 2.3, we introduce ModDevOps as a high-level concept.

2.1 Technology Overview

The TwinOps line of work addresses the general context of *model-based systems engineering* and *model-based software engineering* applied to CPSs. We consider systems engineering different from software engineering, because each of these disciplines has different objectives and processes.[1]

"Model" is a term used in many different settings. So is the term "system." There are models trained for artificial intelligence (AI), models built for systems engineering (model-based systems engineering [MBSE] such as SysML), models built for architecting a system and assessing its non-functional properties (e.g., Architecture Analysis & Design Language [AADL]), models of algorithms, and many more. Many of these models support aspects of the development process, while very few are deployed in the operations phase of the system.

In this section, we analyze various topics related to models and the modeling process. These concepts will allow us to define ModDevOps.

2.1.1 DevOps

DevOps has been codified as a set of practices that combines software development (Dev) and IT operations (Ops). These practices have been combined as a process that aims to shorten the systems development lifecycle and provide continuous delivery with high software quality [Wikipedia 2020]. DevOps is defined more precisely as a "collaborative and multidisciplinary effort within an organization to automate continuous delivery of new software versions while guaranteeing their correctness and reliability" [Leite 2019].

The DevOps processes uses automation to expedite specific steps, such as software building, testing, or deployment. Beyond the human organization, DevOps focuses primarily on automation to discharge engineers from error-prone tasks (so that they can focus on core activities, such as feature update or debugging) and to monitor the system execution to debug or update the software.

2.1.2 Perspectives On Modeling

The specification, design, and V&V of CPSs rely on a common set of modeling capabilities:

[1] Fairley discusses this dichotomy in great detail [Fairley 2019].

- *Modeling capabilities that lay out the foundations of the system, its components, interfaces, and behaviors*. Following Rauzy's thesis on the foundations of MBSE, we acknowledge Rauzy's first thesis that "The diversity of models is irreducible" [Rauzy 2019]. One needs to combine models with heterogeneous notations and semantics to cover all of the system's execution scenarios. In our research, we restrict modeling to the capability to capture the semantics of the system in some form.
- *Analysis capabilities to infer properties from the system's model*. Analysis capabilities are inherently linked to the model itself.
- *The set of analyses that can be executed (the questions that can be answered) depends on the model itself*. As Marvin L. Minsky noted: "To an observer B, an object A* is a model of an object A to the extent that B can use A* to answer questions that interest him about A. The model relation is inherently ternary. Any attempt to suppress the role of the intentions of the investigator B leads to circular definitions or to ambiguities about 'essential features' and the like" [Minksy 1965].
- *Synthesis capabilities to transform a model into another formalism*. These capabilities can be an analytical model used for analysis or source code that can be used for simulating the system or deployed on the target. Typical examples of the latter are AADL-to-code or Simulink-to-code.

These capabilities have been defined and led to the definition of many model-based initiatives, such as Object Management Group (OMG) Unified Modeling Language (OMG UML), OMG SysML, Society of Automotive Engineers (SAE) AADL, Simulink, Safety-Critical Application Development Environment (SCADE), Modelica, and International Telecommunication Union Specification and Description Language (ITU SDL). Each language supports its own set of modeling objectives, analysis support, and synthesis capabilities.

2.1.3 Modeling Cyber-Physical Systems

As stated by Minsky, models and analysis are linked. In addition, the collection of models reveals other intrinsic properties [Minksy 1965]:

- As expressed in systems, CPS engineering relies on model-based systems engineering to capture systems requirements, functions, and stakeholders. These models are capturing relationships between elements and delegating to other models more precise details (e.g., behaviors and performance). They are pragmatic models per Rauzy's definition.
- As expressed in cyber and physical entities, CPS engineering is also better described by a collection of formal models, each of which supports different analyses. This is Rauzy's first thesis.

During the execution of the TwinOps project, we retained the following taxonomy of modeling languages, based on their primary concern area:

- At a system-level: OMG SysML [OMG 2019]
- At a software architectural level: SAE AADL [SAE 2017]
- At a software level: Matlab Simulink, ANSYS SCADE, and traditional programming languages such as C, Ada, etc.
- Other domains: physics (Modelica), cyber-physical modeling (Ptolemy), etc.

In addition, Rauzy's second thesis states, "There is an epistemic gap between pragmatic and formal models." Because they have different natures and purposes, deriving a formal model from a pragmatic one requires an engineering process that cannot generally be automated. In Rauzy's definition, a formal model is a model with precise semantics from which one can derive analysis results automatically.

To revisit the previous list: SysML and C are pragmatic models, they either do not fully specify system internals, like SysML, or are subject to interpretation like C and its undefined behaviors. On the other hand, the SAE AADL, Matlab Simulink, and others might be considered formal models. From these formal models, one can derive an automated process to perform analysis, generate code, or run simulations.

2.1.4 Models and Processes

The previous considerations call for a careful definition of model-supported engineering processes. It has been established that it is necessary to have multiple modeling notations and that they must be combined. In some cases, models are amenable to some automation, such as analysis or simulation.

In this section, we review two relevant Department of Defense (DoD) initiatives: digital thread and digital engineering.

2.1.4.1 Digital Thread

In their paper "Untangling the Digital Thread: The Challenge and Promise of Model-Based Engineering in Defense Acquisition," Timothy D. West and Art Pyster introduce the digital thread concept as "a framework for merging the conceptual models of the system (the traditional focus of MBSE) with the discipline-specific engineering models of various system elements" [West 2015]. This vision supports data interchange between the various design stages of an aircraft to consolidate the acquisition process within the U.S. Air Force.

The concept of a digital thread relies on simulation and high-performance computing to support the engineering of full aircraft. The key relevant domains are physical, such as mechanics, electrical, and manufacturing. Yet, digital thread is also relevant for software-intensive systems, considering software engineering and its delivery.

2.1.4.2 Digital Engineering

The DoD defines digital engineering as "an integrated digital approach that uses authoritative sources of system data and models as a continuum across disciplines to support lifecycle activities from concept through the disposal" [DoD 2018]. It builds on the concept of "an end-to-end enterprise. This will enable the use of models throughout the lifecycle to digitally represent the system of interest (i.e., a system of systems, systems, processes, equipment, products, parts) in the virtual world."

Digital engineering is defined as a strategy rather than as a process. As mentioned in the final words of his foreword to the *Department of Defense Digital Engineering Strategy*, Michael D. Griffen, then Undersecretary of Defense for Research and Engineering, noted, "This strategy describes the 'what' necessary to foster the use of digital engineering practices. Those implementing

the practices must develop the ‘how’—the implementation steps necessary to apply digital engineering in each enterprise” [DoD 2018].

Hence, digital engineering must be adapted to unique project needs. The Architecture Centric Virtual Integration Process (ACVIP) proposes a declination of the digital engineering strategy for the avionics system built on top of AADL [Boydston 2019]. Other declinations are being designed across the DoD.

In addition, it is significant that both digital thread and digital engineering insist on defining an Agile development process with automation at its core. This resonates with DevOps practices.

2.1.5 Conclusion

This quick survey of relevant technology highlighted some key aspects of modeling. First and foremost among them is the need for a diverse range of models and the need to combine models in efficient ways. As of the time of writing of this report, there is no unified definition of digital engineering, digital thread, or ACVIP.

Through its definition of ModDevOps, the TwinOps project is contributing to this general reflection on how to deliver systems faster and with increased confidence.

2.2 MBS2E Overview

Under model-based systems and software engineering (MBS2E), systems are defined as a collection of models and source code artifacts. Their combination covers all steps of the engineering cycle, from high-level requirements to the delivery of the source code.

An overview of the interplay between abstract activities and the corresponding supporting notation is shown in Figure 1.[2] It covers only the embedded software side of a CPS and the languages selected in the previous section. The SysML activity diagram formalism is used to capture the general process attached to it.

[2] The diagram was created using Papyrus, an SysML editor that is part of the Eclipse ecosystem. https://www.eclipse.org/papyrus/

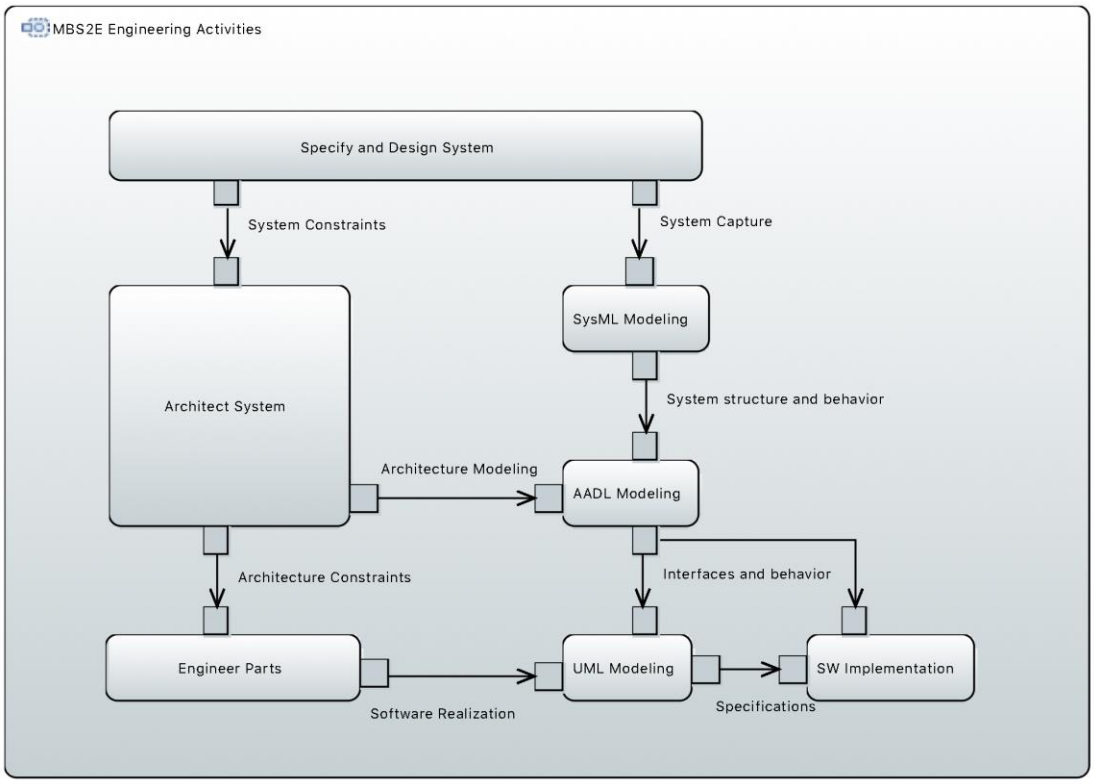

Figure 1: MBS2E Overview Created Using Papyrus

1. First, the system is specified and designed. A first high-level model of the system is captured using SysML, along with a set of system constraints that will serve as requirements for the next step. At this stage, the model of the system is informative and covers its requirements, high-level breakdown structure, and a high-level description of each component interface and behavior.
2. Second, the system's architecture is defined. It is derived from the SysML representation and associated constraints. AADL allows for a more precise definition of the system architecture as a collection of components that captures regular software or hardware behaviors (thread, device, processor, etc.) and can precisely address the runtime aspects of the system.
3. Finally, the system parts can be engineered. From the AADL model definition, we can derive the software low-level requirements (e.g., the subprogram interfaces to be implemented). Then, these parts can be generated automatically from Simulink or other models, or engineers may select UML to capture more precisely the model of the software to be implemented first or implement it directly in their language of choice.

Note that each modeling technology provides automated processes to perform model analysis or code synthesis. The following lists the accepted role of each formalism:

- SysML provides some capabilities to perform trade-off analysis and semantics checks [Leserf 2019]. SysML supports requirements engineering and will provide a mechanism to assess that all requirements are traced to system constituents.
- AADL provides more capabilities to support performance, safety, or security analysis. In addition, one can define project-specific analysis to assess some structural properties, such as

conformance to some modeling guidelines [Delange 2016]. This specificity allows designers to assess most of the non-functional properties and ensure partial functional correctness.

- UML supports semantics checks to validate that the model is sound, just like compilers. The benefits of a modeling approach pertain to the ability to master the complexity of software functions.
- Programming languages now propose a variety of tools to assert software correctness based on SMT solvers, such as ACSL for the C language or SPARK2014 for Ada.

These formalisms can be embedded into larger system engineering processes or included in a software production environment.

2.3 ModDevOps, a Primer

In the previous sections, we introduced various considerations on models and how they can be interconnected. In this section, we define ModDevOps, an extension of DevOps that incorporates models.

We noted earlier that one desirable feature of models, beyond their advanced analysis capabilities, is their ability to accelerate system delivery. This consideration is echoed by the DevOps approach for software. DevOps has been codified as a set of practices that combines software development (Dev) and IT operations (Ops). It aims to shorten the systems development lifecycle and provide continuous delivery with high software quality [Wikipedia 2020]. DevOps relies on the idea of CI/CD and infrastructure-as-code as central pillars.

We introduce ModDevOps as an extension of DevOps, embracing model-based systems and software engineering.

2.3.1 ModDevOps Definition

Model-based engineering relies on models as first-class abstractions of a system under study [Rodrigues da Silva 2015]. In the NASA paper "Survey on Model-Based Software Engineering and Auto-Generated Code," the authors show how automated code generation in the engineering of embedded software both increased confidence in produced software and accelerated delivery [Goseva-Popstojanova 2016]. Yet, this is usually a one-way process in which debugging generated software and informing model updates pose challenges.

The U.S. Air Force proposed a definition of DevOps that insists on the whole system lifecycle [Air Force 2022]. We extend this definition and define ModDevOps as follows, with our additions in bold:

> ***Mod****DevOps is a* ***systems****/software* ***co-****engineering culture and practice that aims to unify* ***model-based systems engineering (Mod),*** *software development (Dev), and software operation (Ops). The main characteristic of the* ***Mod****DevOps movement is to strongly* ***advocate abstraction****, automation, and monitoring at all steps of system construction, from integration, testing, and releasing to deployment and infrastructure management.*

ModDevOps is built on the premise that model-based engineering is the natural complement to software engineering. By providing machine-processable models, one can increase automation to improve system V&V and to generate code, whether it is application code or infrastructure code.

2.3.2 ModDevOps "Infinity Loop"

ModDevOps extends DevOps by refining how specific steps can be supported by model-based techniques. ModDevOps refines the typical DevOps "infinite loop" steps as shown in Figure 2.

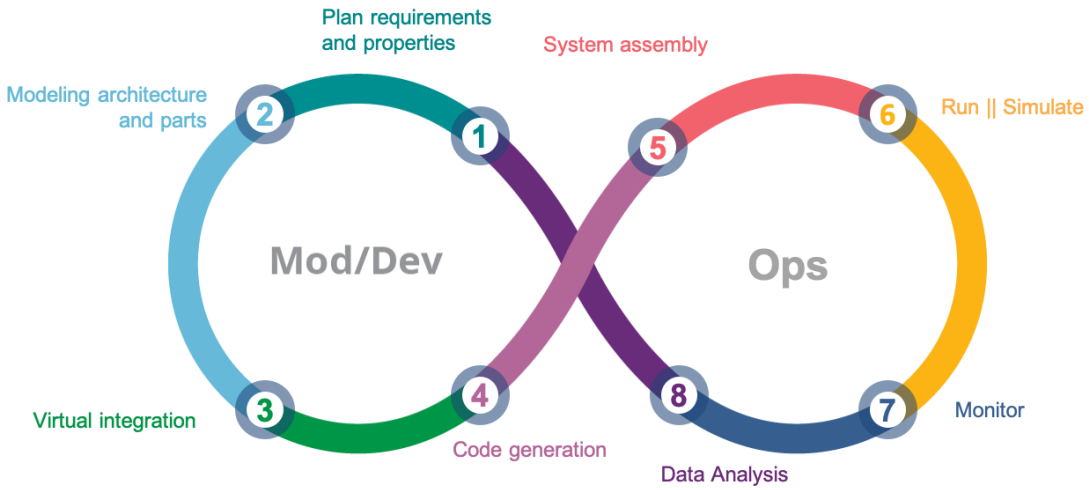

Figure 2: ModDevOps Infinity Loop

Modeling encompasses physical, architectural, and software modeling aspects and source code definition. Source code is the ultimate machine-processable model of the function to be implemented. This step encompasses the following activities of ModDevOps:

1. Planning, requirements definition, and properties (identification/definition): Define the systems engineering models of the system, along with a validation plan.
2. Modeling architecture and parts: Refine the models and define domain-specific models to cover the various parts. Models address specific concerns captured in the previous phases (e.g., need to model the environment, or control, or the architecture of an embedded system).
3. Virtual Integration: Define the interaction points between these models (e.g., how the realization of an architecture executes specific functions or associated engineering models and the environment model).

Test bench/system realization is an automated software factory that builds the various artifacts: simulation code, executables.

4. *Code generation* produces code from models with multiple objectives: generating functional and middleware code to run on the target; generating simulation elements. Also, glue code is generated to (1) monitor properties, such as resource consumption or data exchange, and (2) detect specific execution patterns.
5. *Software Assembly* combines the various pieces to build multiple targets.

"Ops" deploys and executes the generated software.

6. *Monitor* collects data.
7. *Data Analysis* confronts the various data for accuracy and consistency.

8. *Analysis* will inform updates to the system requirements, properties, and updates to the system design.

As defined, ModDevOps extends DevOps with MBE. The ModDevOps processes appear mostly during the Dev phase, aggregating engineering artifacts.

2.3.3 ModDevOps ⊄ Dev(*)Ops

Recently, the U.S. Air Force wrote a note on the risk of stacking terms between (Dev) and (Ops) [Tanner 2022]. Starting with security (Sec), the author mentions that many other concerns could be stacked.

We agree that (Dev) must embed the right engineering principles to deliver expected software and its associated attributes, such as security, performance, and reliability. Therefore, DevOps should be self-sufficient.

Yet, this claim provides an incomplete assessment of the engineering issues at stake. Model-based techniques have demonstrated their added value in defining proper abstractions to conduct early analysis and virtual system integration. Modeling encompasses activities well beyond software development. They touch all engineering domains, including systems engineering, mechanical engineering, and electrical engineering. This characteristic calls for a larger view of a DevOps-like process that embraces modeling activities as an integral part of the engineering of software-intensive systems. This point will be refined in Section 4, in which we illustrate the benefits of a ModDevOps approach for the engineering of CPSs.

2.4 ModDevOps Defined as SysML model

In this section, we propose a comprehensive definition of ModDevOps. To do so, we use SysML to document the key stakeholders and components of ModDevOps.[3]

Note: In the following section, models reference *TwinOps*. This is a reference to the name of the project itself.

[3] All models have been built using Cameo Enterprise Architecture 19.0.

2.4.1 ModDevOps Use Cases

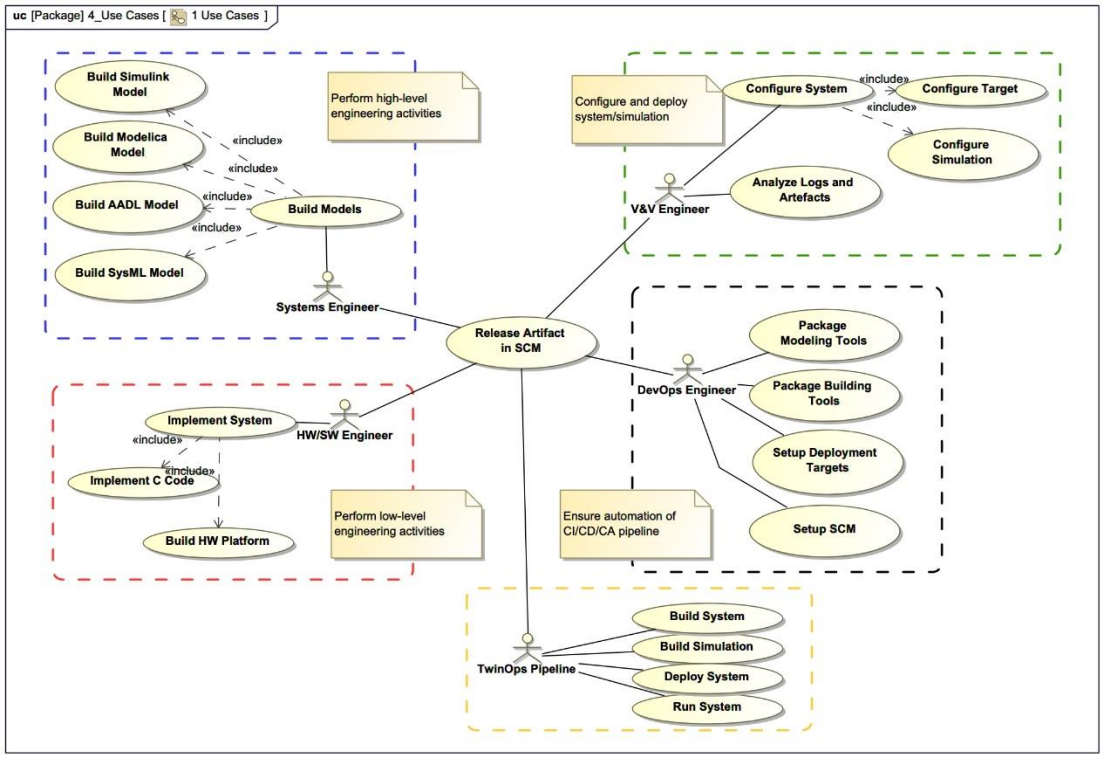

Figure 3: ModDevOps Use Cases

The ModDevOps process has five stakeholders. The following overview highlights their respective roles and associated use cases (see Figure 3 for details).

- *Systems engineers* capture the high-level aspects of the systems as a collection of models. These models represent hardware and software architecture, component and system simulations, and data models. The associated use cases cover all modeling activities.
- *Hardware/software engineers* implement the system by writing C code or designing and/or building the hardware platform. Depending on the project, software engineers may also perform precise Simulink or AADL modeling activities with the objective of performing code generation.
- *V&V engineers* are responsible for the V&V of the system: They develop testing approaches, perform analysis, collect execution logs, and generate reports.
- *DevOps engineers* provide and maintain the infrastructure-as-code DevOps platform used to federate all activities.

These stakeholders use a common repository to manage artifacts. These artifacts are processed in a DevOps pipeline. To some extent, this DevOps pipeline facilitator acts as a fifth team member to support other stakeholders.

- The *DevOps pipeline* automates various tasks, such as building the system, execution of tests, or deployment.

2.4.2 ModDevOps Blocks

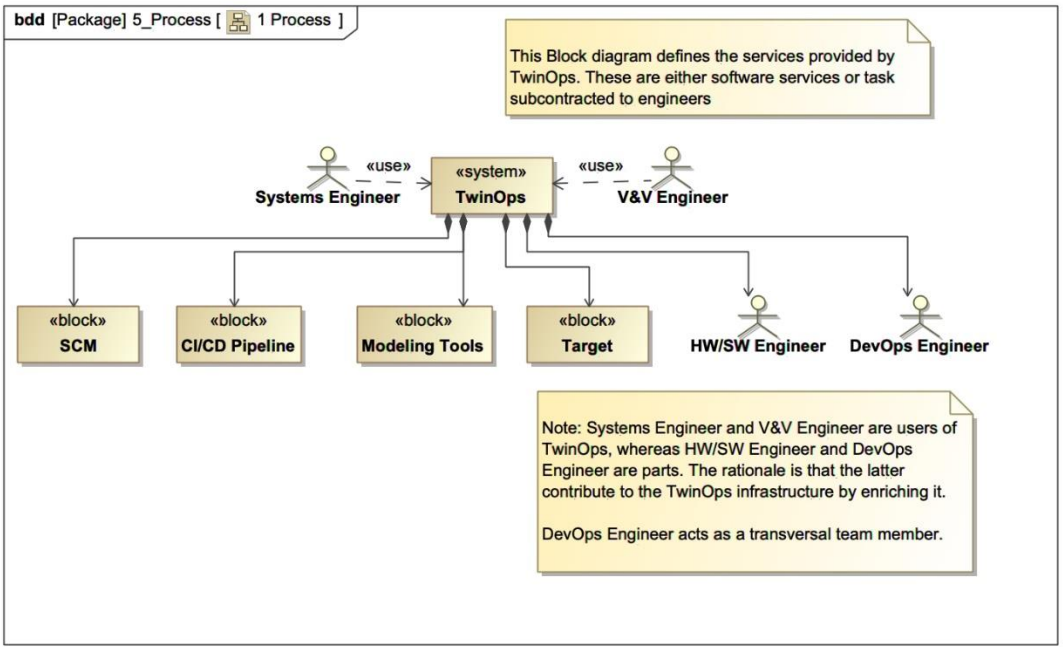

Figure 4: ModDevOps Blocks

We can now review the ModDevOps blocks and how they interact with the defined stakeholders. This is shown in Figure 4.

ModDevOps is made of four blocks and two supporting stakeholders:

- *SCM* is the source configuration management. Usually, it is a Git repository supporting CI/CD capabilities. It is usually deployed in a cloud-based infrastructure. In our case studies, we use a GitLab instance hosted on Amazon Web Services (AWS).
- *CI/CD* is the infrastructure supporting continuous integration/continuous deployment. It is also a Cloud-based node, or alternatively executed directly on the target running a GitLab runner.
- *Modeling Tools* is the collection of modeling tools used for building models.
- *Target* is the deployment target, such as an x86/64 computer or Raspberry Pi board.

Formally speaking, the Hardware/software engineer and the DevOps engineer "belong" to the ModDevOps process: They provide services to the user of the process. The systems engineer and V&V engineer are the users of the process. They will interact with its parts to build, integrate, and deploy a system.

2.4.3 ModDevOps Activities

The block diagrams from the previous section provide a static view of the process; namely, its parts and stakeholders. In this section, we provide a definition of the dynamic of ModDevOps.

First, each block has a collection of activities it can support. These activities capture the services it supports:

- SCM: Support SCM configuration
- CI/CD pipeline: Configure CI/CD pipeline and Trigger CI/CD pipeline
- Modeling tools: Verify Models
- Target: Setup Target and Run Target

Stakeholders will be in charge of other activities, such as “build model” or “store/fetch elements from SCM.” This mapping is straightforward.

These atomic actions are the building blocks of the ModDevOps process. To do so, we use a SysML activity diagram to mention how activities (attached to each block) are sequenced, as shown in Figure 5. First, the SCM is configured. Next, the target and the CI/CD pipeline are configured. Modeling tools are packaged as containers and used to build models. Models can eventually be verified prior to implementing the system. When the models and code are ready, the CI/CD platform is triggered, leading to the integration of the software elements and their deployment on the target. Upon successful deployment, the system starts its execution. Logs are collected and analyzed. This leads to a new modeling phase that will repeat the whole process.

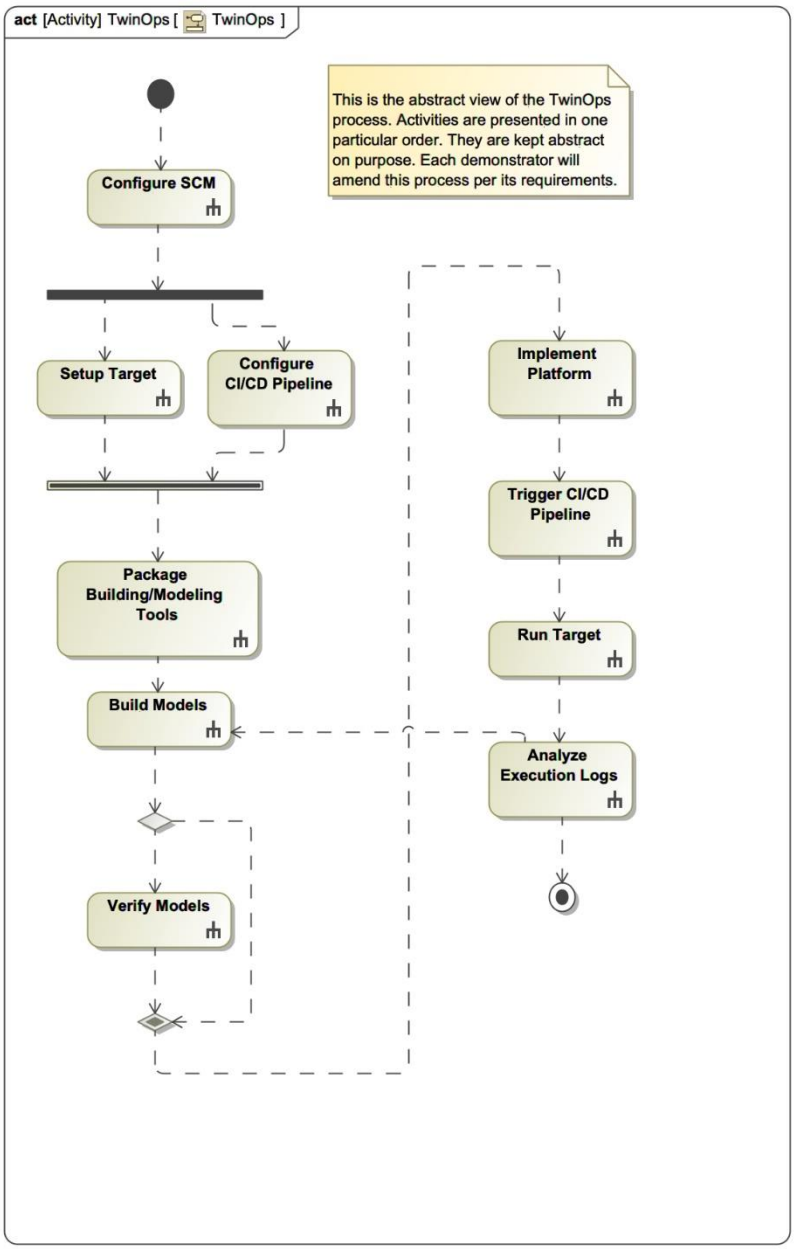

Figure 5 ModDevOps Activities

Finally, we can use a SysML traceability matrix to relate all activities from the previous activity diagram to elements from the initial use case we presented as shown in Figure 6. This matrix guarantees that all use cases have been properly covered by the definition of the ModDevOps process.

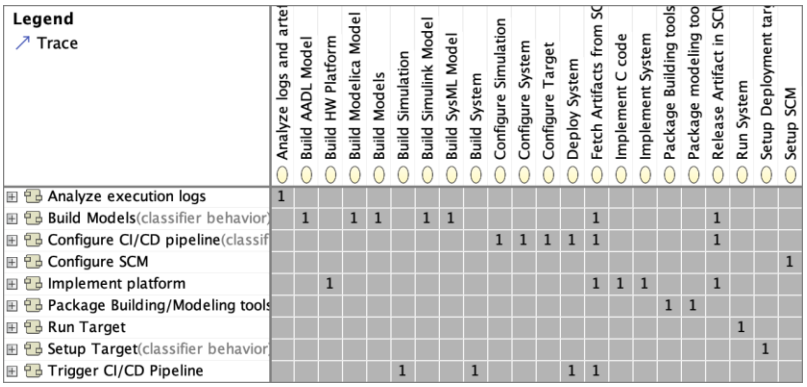

Figure 6: ModDevOps Activities Traced to Its Use Cases

2.5 Conclusion

In this section, we provided a full definition of ModDevOps. ModDevOps extends DevOps by recognizing the role of modeling activities in the engineering of systems, especially cyber-physical systems. ModDevOps was gradually defined: first informally, then formally using SysML. This formalization allows us to better characterize the various steps of the process.

As we have defined it, ModDevOps remains a generic process that can be tailored. In the following sections, we illustrate various instances of ModDevOps to support the engineering of CPSs.

3 TwinOps Defined: ModDevOps for CPS

In this chapter, we give a concrete realization of ModDevOps in a cyber-physical setting. We define the first part of *TwinOps*, a ModDevOps process geared towards the engineering of CPSs. We concentrate on the ModDev part of the process.

3.1 TwinOps Introduction

The coupling of model-based techniques and DevOps is an emerging research topic. Contemporary to the definition of the TwinOps research agenda, Combemale et al. presented a roadmap for model-based DevOps for CPS [Combemale 2019]. The authors propose a collection of challenges to be addressed to better couple MBE and DevOps. They list research challenges rather than actual solutions for adapting DevOps to the engineering of cyber-physical systems. Following a typical DevOps cycle, they list two main challenges that apply to the first half of the process, Dev-to-Ops:

1. *Integration of model-driven techniques to DevOps*: The authors propose code generation from models to be integrated to a DevOps pipeline. Code generation is triggered by model updates and subsequent deployment for running the code on a simulation platform or the actual system. They also call for specific languages for defining the corresponding CI/CD pipeline.
2. *Integration of heterogeneous artefacts*: The authors highlight the need for semantics interoperability across different modeling techniques, where a computer-aided design (CAD) model will be used to test a controller implemented using another modeling technology.

TwinOps provides a solution to these two challenges by leveraging well-established model processing capabilities. TwinOps builds on one core idea: using engineering models from other domains (mechanics, electronics, etc.) to validate software-intensive systems against faithful representations of the environment. TwinOps extends ModDevOps and builds on two central technologies:

- ModDevOps: rapid fielding of software capabilities with confidence, using models as inputs. We leverage code generation from models in addition to typical DevOps pipeline definition.
- Digital twins: through code generation, one can generate a digital mock-up of a system, fully synchronized with the actual system.

By combining digital twins and DevOps to engineer CPS, the TwinOps process aims to show the conformance of a system to its high-level objectives (e.g., system requirements). The TwinOps process incorporates increasingly refined models of the system, its environment, and the mission description and objectives, down to the final deployment. This is made possible by the integration of model-based assets through AADL models, extensive code generation, and parallel execution of the CPS.

In the following section, we provide a hands-on introduction to TwinOps through a case study: an IoT sensor processing unit known as “SensorProcessing.”

3.2 The SensorProcessing Demonstrator

Let us assume we want to build a monitoring system for a building. The system will monitor and collect environmental conditions to ensure the proper operation of an air conditioning system. The system participates in a digital twin of the building. We elicit the following requirements:

- *R1:* The system shall monitor the humidity and temperature in multiple points of a building every 10 minutes during office hours or every 30 minutes thereafter.
- *R2:* The system shall store humidity, temperature, and timestamp data in a central repository.
- *R3:* The system shall detect and report any error in the reported data, such as out-of-range values or a sudden surge in values.
- *R4:* The system shall monitor its health status and report issues.

From these considerations, an industrial survey shows that a platform built on the Azure IoT Cloud platform for data management, and a Raspberry Pi platform with a BME280 sensor device, could deliver the expected functionalities [Microsoft 2021]. The Azure IoT framework associated with a Raspberry Pi board supports the implementation of a digital twin of the building to monitor and control its temperature.

Several open-source projects propose a full description of such a system down to software implementation of sensor reading, but they do not cover the error detection and logging/reporting. In the following sections, we illustrate how model-based engineering and ModDevOps could be combined to support the definition and engineering of this system. We also illustrate how combining MBE code generation and CI/CD techniques allows for the automated deployment of a solution.

3.3 ModDevOps Applied to SensorProcessing: Models

In the following sections, we illustrate how ModDevOps, which was introduced in Section 2.3, can be used to design, analyze, and then deploy the SensorProcessing system. For each step, we provide an overview of the provided solution. The implementation details are discussed in Section 4.

In the first step, "Plan requirements and properties," we define the requirements of the system, their decomposition as subfunctions, and use case scenarios attached to it.

3.3.1 TwinOps Solution #1: Use Containers for Delivering Modeling Environments

According to the DevOps philosophy, the first concern is to ensure all team members use the same baseline for the modeling environment. We propose to use docker containers to build a reproducible modeling environment [Boettiger 2014]. The "TwinOps DevOps engineer" role defined a container with the Eclipse baseline for modeling environment, comprised of Papyrus SysML 1.6, OSATE AADL toolchain, and Modelica Development Tooling (MDT) tools. Their use scenarios are detailed below.

Using this environment, the first set of models of the system can be captured: a collection of OMG SysML 1.6 diagrams that capture the high-level requirements of the system, use cases, and first-level system decomposition [OMG 2019].

In the second step, “Modeling architecture and parts,” these models are refined as AADL models as shown in Figure 7. AADL models capture the embedded system architecture as a collection of processors, buses, devices, and software attached to it. Our choice for AADL has been dictated by the classes of analyses and code generation capabilities.

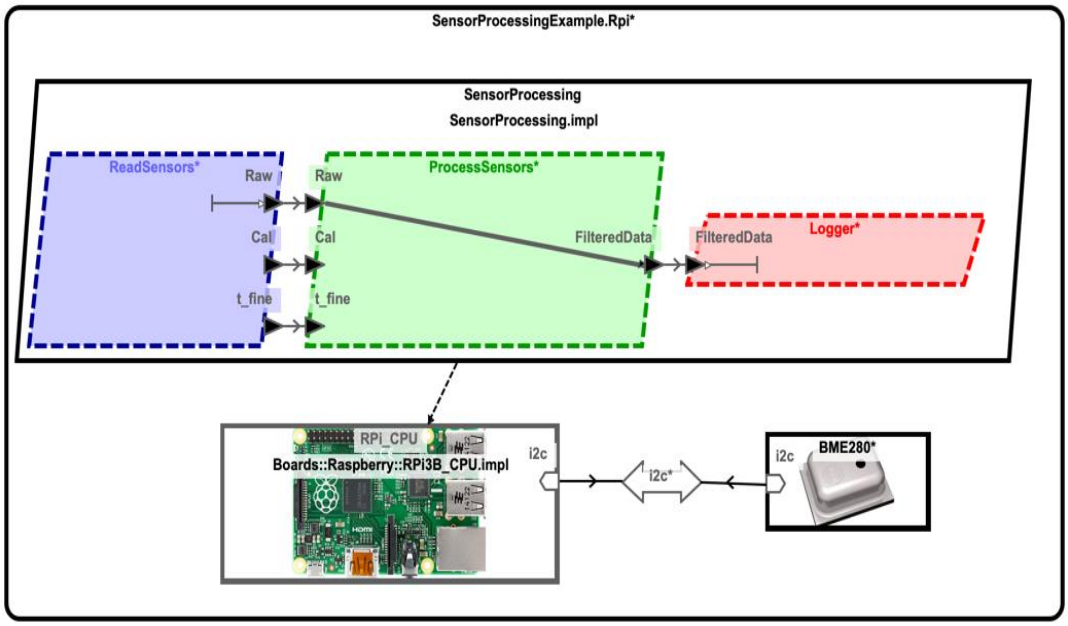

Figure 7: SensorProcessing / AADL Model

3.3.2 TwinOps Solution #2: Perform Virtual Integration from Models

We leveraged the Architecture-Led Incremental System Assurance (ALISA) Domain-Specific Language (DSL) [Delange 2016] to refine requirements into verifiable items attached to target metrics. An ALISA verification plan binds requirements to verification methods to be executed, usually a verification plug-in, and reports on any discrepancy. This virtual integration ensures that the model, as currently engineered, can be integrated on the target platform and meet stated performance metrics.

The AADL model combined with an ALISA verification plan supports the evaluation of some key metrics, such as the number of messages processed per unit of times and energy consumption. An ALISA verification plan can be executed from within Eclipse or integrated as tests in a regular test suite environment such as JUnit.

It is important to note that both AADL and ALISA are amenable to continuous integration using the Civis tool by Adventium Labs [Smith 2018]. Using Civis, a designer may run an ALISA verification plan and report on performance indicators or other metrics.

3.3.3 TwinOps Lessons Learned

TwinOps builds on a model-based CI/CD pipeline that contains both models and a reproducible modeling environment. Model-level analysis and evaluation of some metrics are performed, and discrepancies can lead to model refactoring. These steps result in the modeling pipeline (see Figure 8). Initially, SysML and AADL modeling steps are performed, then ALISA verification may

either detect an error or continue to the next step. SysML covers use cases definition and requirements capture, whereas AADL covers architecture modeling activities.

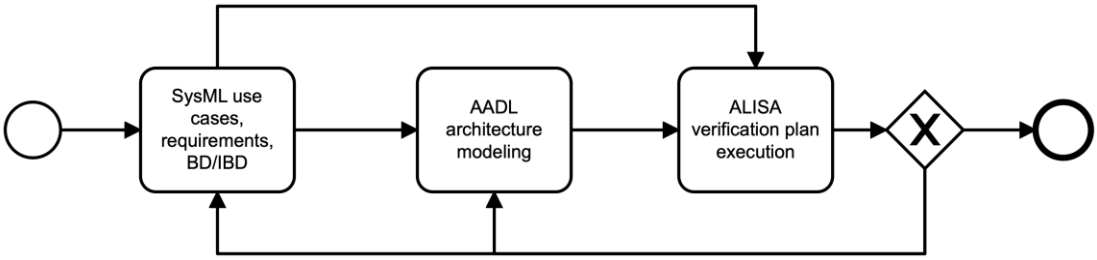

Figure 8: Modeling Pipeline for SensorProcessing

This pipeline forms the first level of ModDevOps, closing the loop at model-level.

3.4 ModDevOps Applied to SensorProcessing: Implementation

This second step addresses dual objectives: support V&V activities and deliver the final system. One limit in the previous MBE CI/CD pipeline is that not all properties may be assessed at model-levels. Figure 9 illustrates some contributors to issues that can only be evaluated at runtime: timing budgets for end-to-end flows (highlighted flow in yellow) may not be respected by the implementation or communication bus, devices may experience some bias at runtime (in blue) that must be detected and mitigated, or loss of the connection to the logging facility (in orange).

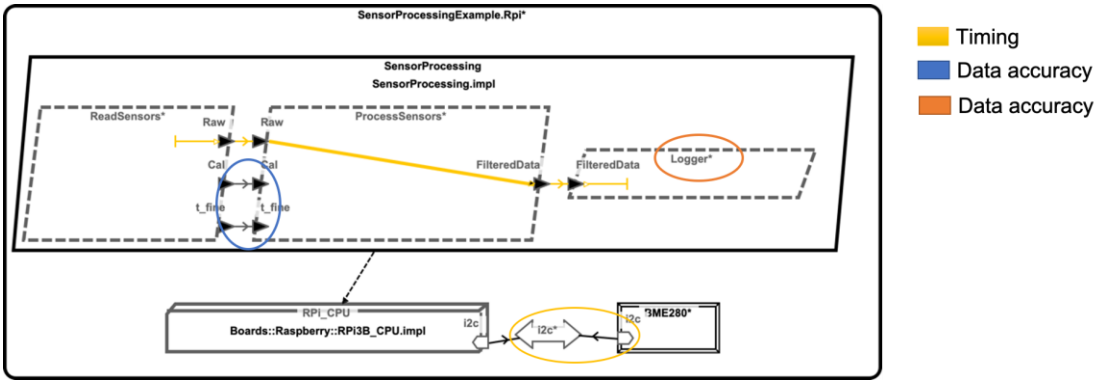

Figure 9: Runtime Monitoring Points

In this second step, the system is implemented and enriched it with monitoring probes. Existing models are leveraged to perform extensive code generation from the architectural model description (Figure 10). First, software probes are implemented. Probes either validate input data or measure the execution time of functions. Second, we implement the core logic of our application. Then, we use the Ocarina AADL code generator to generate code [Lasnier 2009]. These three source code elements are combined to produce the final binary.

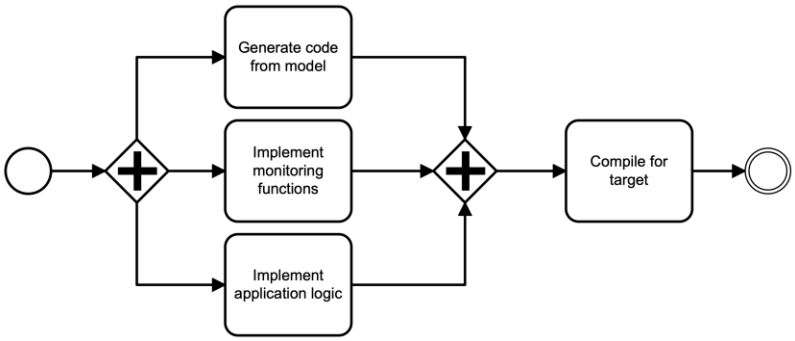

Figure 10: ModDevOps Code Generation Pipeline

3.4.1 TwinOps Solution #3: Multiple Targets Code Generation

From an AADL model, Ocarina generates minimal middleware that supports the execution of the model (tasks, communication buffers, ports, etc.). The targeted language can be C (running on a variety of real-time operating systems [RTOS] or the Portable Operating System Interface [POSIX]), Ada, or formal languages for simulation and model-checking, such as LNT (a member of the Language Of Temporal Ordering Specification [LOTOS] family of formal description techniques) [Mkaouar 2020]. This multiplicity in targets allows for diverse means to evaluate the system as follows:

- LNT supports executing functional C code embedded in a formal model of the system and state-space exploration for safety or liveness properties.
- C allows for direct execution on the target using devices drivers or a mock-up of the device implemented as a Functional Mockup Unit [Hugues 2018].

This process allows us to build three different targets:

1. *The LNT target* enables model-checking capabilities, weaving an abstract model of the environment and the execution platform with actual functional code. This process allows for a systematic evaluation of the functional side of the system but may be limited to some platform-specific aspects: error in sensors, timing issues, etc.
2. *The C/FMI target* with device mockups leverages the FMI standard to build a simulated environment using a Modelica model to capture the physical environment. For our sensor demo, we used a first-principles model of the sensor device and the generation of temperature and pressure from a meteorological simulation. Using FMI allows us to define specific use scenarios by adjusting physical variables while evaluating the actual execution on the target.
3. *The C/Azure target*, with execution on the target platform, allows for the execution of the system and its monitoring. We generated specific monitoring probes to collect all data, which is then sent to an Azure IoT digital twin of the system. The digital twin is a representation of the system in terms of its state properties, telemetry events, commands, components, and relationships. This provides a data stream that can be queried and analyzed.

3.4.2 TwinOps Solution #4: Integration as a DevOps CI/CD Pipeline

In the previous sections, we presented a mapping of modeling, model transformation, and code generation activities to a notional DevOps pipeline. We integrated these steps in a CI/CD pipeline using the GitLab platform. This pipeline supports all steps that could be automated: model transformation or code generation, compilation, testing activities, containerization, and deployment on targets.

Figure 11: Deployment Pipeline

To facilitate deployment, a docker container is built that hosts the binary along with its dependencies. This container is then stored in a container registry that provides versioning and future accessibility. This process supports a reproducible runtime environment across multiple targets. The final step in our pipeline is the deployment of the container on the target. We leveraged the Azure IoT capability to send a request to all targets to deploy and run the latest released version from the container registry.

The current configuration of the GitLab pipeline involves a manual process. Future work will consider linking the GitLab configuration to a model that configures the CI/CD pipeline and the set of deployment targets in a uniform way.

3.4.3 TwinOps Solution #5: System Analytics

All targets are ultimately combined to improve the system through data analytics: The LNT or C/FMI targets use data collected from the C/Azure targets to replay specific execution traces. Since all targets share the same code base, they provide representations of the same system at various levels of fidelity.

Finally, the same data can lead to model improvements. For instance, timing traces can be compared to theoretical time budgets used for latency or scheduling analyses, and sensor biases can lead to a different mitigation policy, for instance, to force specific recalibration. Hence, such a comparison between execution traces and the initial model can inform updates of the system to improve its accuracy.

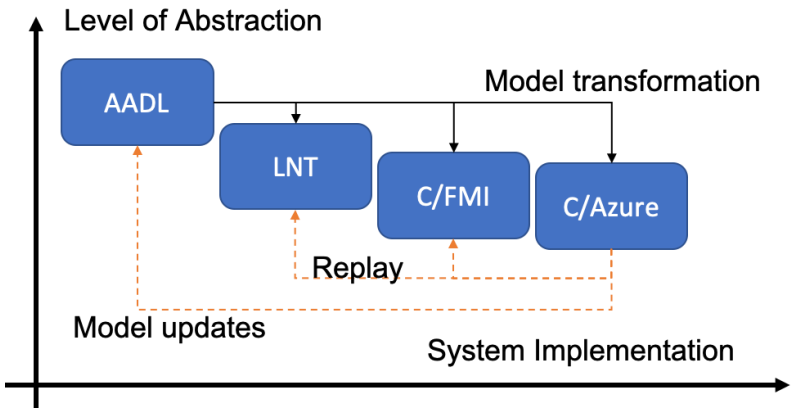

Figure 12: Feedback from (Ops) to (Mod/Dev)

Model transformation and code generation, combined with the automated integration of monitoring probes, support the feedback loop prescribed by DevOps philosophy: the capability to monitor the system at "Ops-time" to inform updates during "Dev-time" as shown in Figure 12.

3.5 Conclusion

In this chapter, we provided an overview of TwinOps, a declination of ModDevOps tailored for the engineering of CPS. We provided a high-level description, insisting on the key steps of the process and how they are articulated. In the next chapter, we will provide a more detailed review of the SensorProcessing example.

4 TwinOps: The SensorProcessing IoT Demo

In this chapter, we review the SensorProcessing IoT demo in more detail.

Note: Our objective is to demonstrate how to combine models using ModDevOps/TwinOps. Therefore, each model provides a basic solution to engineering problems. We took the decision to use the project effort on defining ModDevOps rather than doing unitary modeling activities.

4.1 Step 1: Defining the Modeling Process

In the previous chapter, we introduced the SensorProcessing IoT demo as follows:

Let us assume we want to build a monitoring system for a building. The system will monitor and collect environmental conditions to ensure the proper operation of an air conditioning system. The system participates in a digital twin of the building. We elicit the following requirements:

- *R1:* The system shall monitor the humidity and temperature in multiple points of a building every 10 minutes during office hours or every 30 minutes thereafter.
- *R2:* The system shall gather all data in a central repository.
- *R3:* The system shall detect and report any error in the reported data, such as out-of-range values or a sudden surge in values.
- *R4:* The system shall monitor its health status and report issues.

From these considerations, an industrial survey shows that a platform built on the Azure IoT Cloud platform for data management and a Raspberry Pi platform with a BME280 sensor device could deliver the expected functionalities [Microsoft 2021]. The Azure IoT framework associated with a Raspberry Pi board supports building a digital twin of the building to control its temperature.

The system under consideration is a cyber-physical system doubled with an IoT system. This dual nature is calling for a specific implementation path that demonstrates that both the supporting architecture and the software artifacts meet all requirements.

We plan to use the following technologies:

- OMG SysML: to capture the high-level model of the system: its requirements, its breakdown structure, and use case scenarios
- SAE AADL: to capture the architecture of the embedded platform supporting the execution of the system
- C language: to implement the embedded system on top of the Linux operating system
- Azure IoT middleware and cloud platform: to automate the deployment of the system and the collection of data
- Modelica: to build a mock-up of the environment and test the behavior of the system

In the following sections, we first introduce each modeling stage, then show how we implemented a ModDevOps pipeline to support automated generation and deployment of the system on target and as a simulation.

4.2 Step 2: System Model / SysML

Let us first recall the objectives of the SysML modeling activity:

1. Support the elicitation of the system requirements, an initial breakdown structure, and components.
2. Perform the initial allocation of functions to hardware/software elements.

From these requirements, one can derive requirements on the embedded platform itself and a first architecture that shows how the parts contribute to the system realization.

As we mentioned, a survey shows that a platform built on the Azure IoT Cloud platform for data management, and a Raspberry Pi platform with a BME280 sensor device, could deliver the expected functionalities. This is captured in the following block diagram.

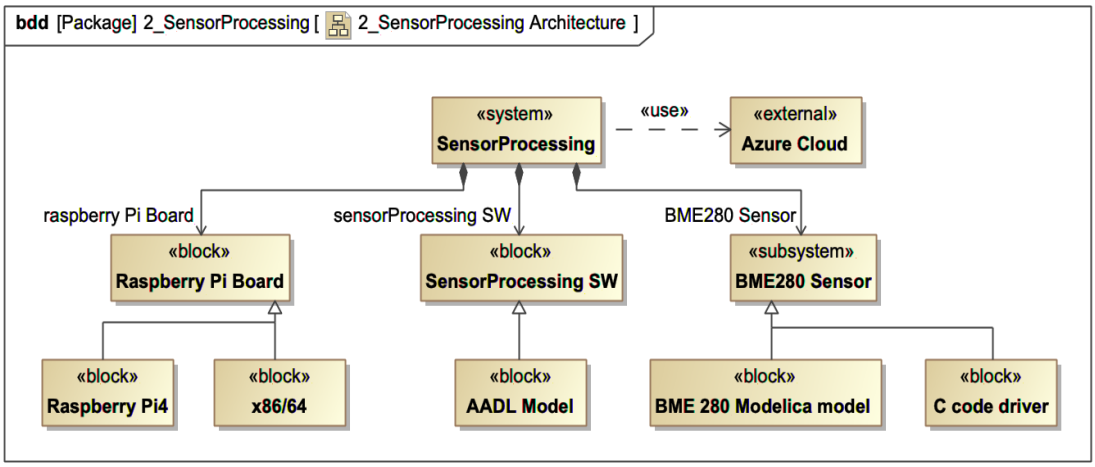

Figure 13: SensorProcessing SysML Architecture Breakdown

Additional SysML modeling steps have been performed to define system requirements and how they are allocated to this architecture. These are discussed at the end of this chapter.

4.3 Step 3: Embedded Software and Hardware Mode / AADL

Through AADL, we aim to expand the SysML modeling activity and do the following:

1. Model the system requirements and the system architecture and its subcomponents, their interfaces, configuration parameters, etc.
2. Evaluate the system performance: latency analysis, schedulability analysis.
3. Synthesize the system.

4.3.1 Setting Up the Modeling Environment

Setting up a modeling environment, which involves the installation of software and plug-ins (and their proper configuration), can be perceived as a tedious process.

Following DevOps principles, we first built a docker container that hosts all required modeling software. In this occurrence, a complete installation of the OSATE AADL toolchain and companion plugins for schedulability analysis.

4.3.2 Modeling the System Requirements Using ALISA

Verification activities connect requirements to model elements. This process uses the ALISA toolset, which is part of the OSATE AADL toolset.[4] ALISA combines requirements (captured to ReqSpec) architecture models, verification techniques, and assurance case traceability [Delange 2016]. For each requirement, a claim must be implemented that verifies it. Verification methods can be existing AADL verification plug-ins, user-defined methods using Resolute, or manual review methods of generated reports, such as a fault tree [Gacek 2014].

Note: In the following, we performed a "vertical" modeling of the system: We only captured minimal concepts at each level to demonstrate how they complement each other. A larger case study would have more elements at each stage.

First, we define the system goals. This is a high-level description of the set of goals the system must fulfill. We restricted it to performance objectives.

```
stakeholder goals Goals_SensorProcessing_Stakeholder
    for SensorProcessing::SensorProcessing.impl [
     goal sensorprocessing_performance [
        category Metrics.Performance
        description "The system meets expected performance"
        stakeholder CPS_Roles.Engineer
    ]
]
```

From this objective on system performance, we selected timing performance. This objective translates into a requirement on end-to-end processing time (or latency) from sampling to sending the data to the central repository. This process is captured in the model below: one high-level requirement that mandates the maximum time to process a sample.

```
system requirements Reqs_SensorProcessing for
SensorProcessing::SensorProcessing.impl [
    description "High-level requirements for the SensorProcessing demo,
software part"
    see goals Goals_SensorProcessing_Stakeholder requirement LatencyCheck :
        "Sensor data processing response time is less than 1 second"
            for SensorDataProcessing [
        category Metrics.Performance
        see goal
Goals_SensorProcessing_Stakeholder.sensorprocessing_performance
    ]
]
```

[4] This section corresponds to the content of the `alisa` folder.

Then, ALISA allows one to define assurance cases and plans that indicate for each requirement how they are verified. In this case, we use the OSATE Latency Analysis feature presented in "Flow Latency Analysis with the Architecture Analysis and Design Language (AADL)" [Feiler 2008]. This analysis is controlled by a verification plan that indicates how the requirement `LatencyCheck` is verified. On this occurrence, it is by executing the corresponding analysis plugin.

```
verification plan VerificationPlan_SensorProcessing for
Reqs_SensorProcessing [
    claim LatencyCheck [
        activities
        responsetime : Plugins.EndToEndFlowLatencyAnalysis ( )
    ]
]
```

4.3.3 Modeling the System Architecture Using AADL

In parallel to the ALISA requirement capture, the architecture of the system is captured using AADL.[5] Figure 14 illustrates the model we built.

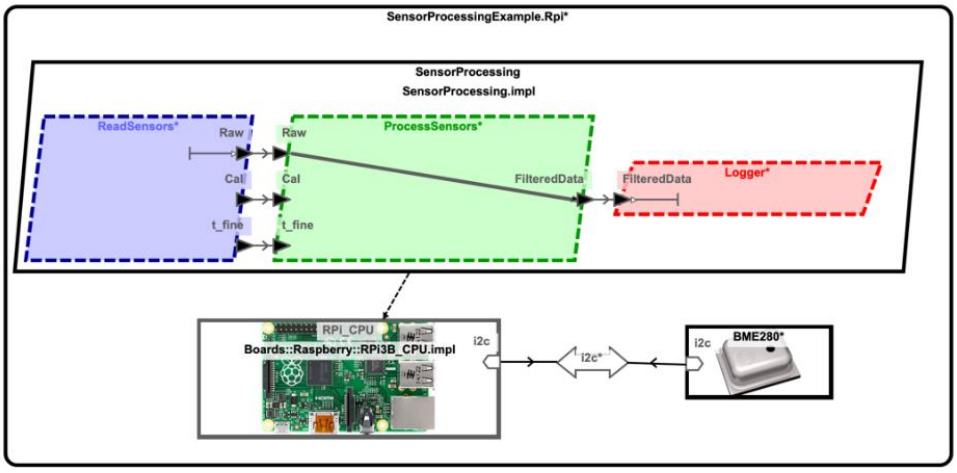

Figure 14: SensorProcessing / AADL Model

This model is built around typical AADL concepts:

- A `system` component acts as the boundary of the system we design. Its subcomponents host all parts.
- A `processor` and `device` connected through a `bus` capture the hardware platform consisting of a Raspberry Pi computer and a sensor.
- A `process` hosts the software part of the system.

Note: We dedicated most of the modeling effort to the software itself to capture the functions to be executed, the interfaces, and the connection to software implementation.

[5] This section corresponds to the content of the `aadl` folder.

We connect AADL model elements such as data types or subprograms to existing source code using the Data modeling annex to describe data types [SAE 2019], and the Code generation annex [SAE 2015] to link C code to AADL models.

```
   --- BME280_Initialize: initialize the device
   subprogram BME280_Initialize
   properties
      Source_Language => (C);
      Source_Name => "bme280_initialize_entrypoint";
      Source_Text => ("../c/aadl_bme280.c", "../c/bme280.c");
   end BME280_Initialize;

   --- Calibration data (internal opaque type)
   data BME280_Calib_Data
   -- This type is wrapped in types.h to its original definition through a C
typedef.
   properties
      Source_Language => (C);
      Type_Source_Name => "bme280_calib_data";
      Source_Text => ("../c/bme280");
   end BME280_Calib_Data;
```

4.4 Step 4: IoT Concerns and Implementation of C Functions

This step is concerned with the implementation of C functions that will support the execution of the system. Recall that code generated from AADL covers threads, communication, etc.

The user should implement the logic of the application itself: device drivers, data processing, and storage.

Each part is implemented as C functions, with the following considerations:

- Original device drivers for the sensors are used. These drivers are wrapped into utility functions that provide the relevant data structures.
- Data processing is a basic step that turns data into time-stamped artifacts.
- Data storage is implemented using the Azure IoT middleware. AzureIoT provides cloud connectivity and data storage capabilities. We will elaborate on this part later in this chapter.

4.5 Step 5: ModDevOps: A Model-Level CI/CD Pipeline

4.5.1 Model-Only CI/CD Pipeline

From the combination of AADL models and ALISA verification plans, one may contemplate building a CI/CD pipeline to check the model. This has been investigated by Adventium Labs [Smith 2018]. Using the Civis tool, a designer may run an ALISA verification plan and report on other metrics, such as performance indicators. We consider this activity as mature and did not investigate using this tool during the execution of the TwinOps project.

Instead, we run the ALISA verification plan (as shown in Figure 15) and confirm the system is feasible.

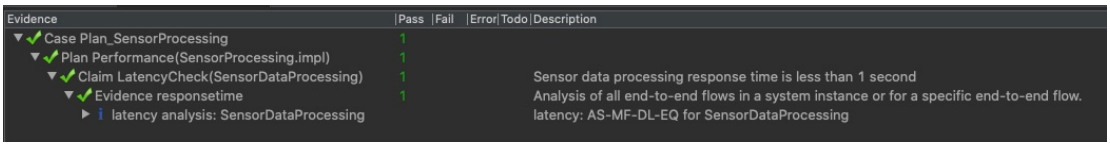

Figure 15: SensorProcessing: Execution of ALISA Verification Plan

4.5.2 Model-to-Code-to-Target CI/CD

We investigated how to go further and complement model-level CI/CD with a model-to-code pipeline that expands the envelope of CI/CD automation. Leveraging code generation from AADL using the Ocarina toolset [Lasnier 2009], we defined a pipeline that combines AADL models and C code artifacts and produces a binary for a specific target.

This pipeline is made of three stages:

- A code generation stage transforms AADL models into C compilation units. This stage uses the Ocarina AADL code generator.
- A compilation stage combines all C compilation units and compile a binary for a specific target. This stage uses a regular C compiler and the required libraries and headers (e.g., for concurrency and communication). The outcome of this stage is a docker container stored in one shared docker registry.
- A deployment stage deploys the binary to its execution platform. We use the Azure IoT platform to trigger the deployment on the target platform. On each deployment node, an Azure IoT node waits for a trigger message and pulls the latest container from the registry.

This pipeline is executed as an orchestrated CI/CD pipeline by a GitLab instance hosted in AWS.

Note: All configuration scripts, docker containers, and build scripts are bundled with this demonstration.

4.5.3 Conclusion

With this demonstration, we illustrated one possible ModDevOps pipeline. This pipeline relies on a set of modeling tools for the early stages. This part cannot be automated and relies on particular project guidance. Then, when all models were complete, we illustrated how a full CI/CD pipeline could be created to trigger model transformation, produce code, compile it, and deploy it.

Hence, ModDevOps is a particular instance of a DevOps process with additional verifications performed on models and code generated from. These additional steps increase the confidence in the software that is ultimately deployed. Such early verification capabilities have been evaluated in the context of another study by the SEI [Hansson 2018]. Automation, in particular code generation from AADL, reduces the gap between models and software by automating coding steps [Lasnier 2009].

4.6 Step 4 Revisited: Modeling the Environment / Modelica

To capture the environment of the system, we use Modelica [Fritzson 2011]. Modelica is a language that allows modeling of complex physics-based systems as mathematical models. Modelica

is defined as an object-oriented, equation-based programming language to model all types of physics phenomena (e.g., mechanics, thermodynamics, and electromagnetism).

4.6.1 About Modelica

The Modelica language is a nonproprietary, object-oriented, equation-based language to model complex physical systems that combine mechanics, electrics, hydraulics, and other types of physical systems. It is developed by the Modelica Association.

A Modelica modeling environment supports model editing, compiling, and model translation towards C to later perform a simulation of the model. In the following case study, we use the open-source OpenModelica environment, compiler, and translator.

Modelica is a modeling language as much as a programming language. It supports an acausal way of describing a system (i.e., as a set of equations). A Modelica system translator will compile the model into a causal imperative program to be executed.

Modelica is based on four idioms: connectors, variables, equations (including derivative operators), and connections. In the following section, we list only relevant elements for this study. More details can be found in the book *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach, 2nd ed*. [Fritzson 2015].

A **model** is the entry point of a Modelica component. **Parameters** denote constant values used by the system (e.g., its mass). The **equation** part of a system comprises the equations that control the dynamics of the system. The following example illustrates a basic model of a resistor.

```
model Resistor
 Pin r1, r2;
 parameter Real Resistance = 1000;
equation
 0 = r1.i + r2.i;
 0 = r1.v - r2.v - Resistance * r1.i;
end Resistor;
```

4.6.2 Setting Up the Modelica IDE and Tools

We propose a docker container that embeds one installation of the OpenModelica toolset, similar to the one provided for OSATE. OpenModelica allows for model edition, simulation, and code generation targeting the FMI.

4.6.3 Modeling the SensorProcessing environment

We used Modelica to provide a basic model of the environment illustrated in Figure 16. This model has two internal sources that set the temperature and pressure of a system.

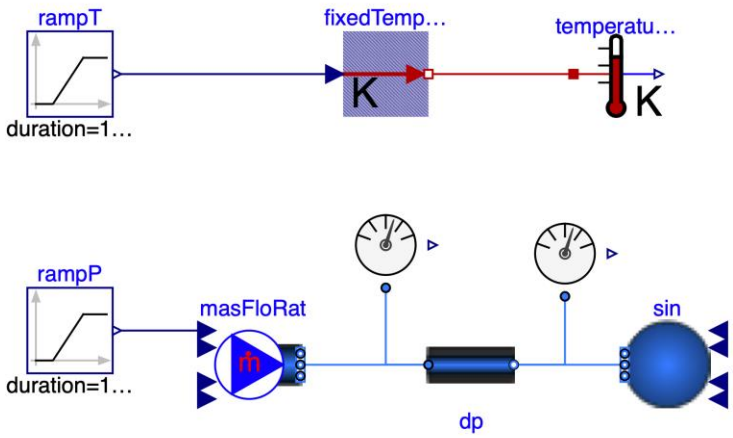

Figure 16: SensorProcessing: Model of the Environment

The model has no input. Internal sources control the temperature and pressure of an ideal room. Two output parameters report measured values for the pressure and temperature parameters. These values can also be displayed during the simulation.

Note: For simplicity, we captured a very basic system with no variations in the physical parameters. Our objective in this exercise is to demonstrate model interoperability. A more representative physical system is introduced in the next section.

4.7 Step 5 Revisited: TwinOps: A ModDevOps Specialization for CPS Simulation

In this section, we revisit this demo with the objective of building a simulator rather than the actual system. The simulator would execute a model of the environment that interacts with the software elements. To do so, we leverage the FMI standard to connect models and perform co-simulation.

4.7.1 About the FMI Standard

Modelica provides a large set of simulation capabilities. In this project, we were interested in the capability to export a Modelica model to FMI.

FMI is the main result of the MODELISAR ITEA 2 European project [Blochwitz 2011]. FMI was first designed to improve vehicles embedded systems modeling and simulation. FMI usage and research investigations now spread over a variety of domains and industries.

FMI defines an interface to be implemented by an executable called a Functional Mockup Unit (FMU). The FMI functions are used (called) by a simulation environment to create one or more instances of the FMU and to simulate them, typically in combination with other models. An FMU may either have its own solver (FMI for Co-Simulation) or requires the simulation environment to perform numerical integration (FMI for Model Exchange).

Each model has an FMU. An FMU is a compressed archive that contains a binary file (actually a dynamic library) that embeds a simulator or the model and an XML file that describes the model contents/properties (its associated model variables). Both FMI for Model-Exchange and Co-Simulation support the design and execution of

- discrete-time systems (e.g., describing a sampled-data controller)
- continuous-time systems (e.g., describing continuous behaviors with DAE)
- a combination of the systems above (e.g., describing hybrid system)

In the following demo, we considered only the Co-Simulation case. It enables interoperability across heterogeneous models. Co-simulation is a technique used for the simulation of coupled models. A coupled model is a model that describes a system as a network of (logically or physically) coupled (or connected) components. In the coupled model formalism, the connections between subsystems are represented with connectors, or mathematical equalities. Formally, a coupled model may be represented as a graph structure.

For non-causal and continuous models, the graph is undirected. For causal models, the graph is directed. A coupled model is valid if the type and causalities of connected ports are compatible [Gomes 2017].

4.7.2 Coupling FMI and AADL

FMI 2.0 for Co-Simulation supports the connection of causal models only. The data exchange between subsystems is performed at discrete communication points. In the interval between two communication points, the subsystems are solved independently by their respective solvers. Primary algorithms control exchanges of data between the subsystems and the synchronization between secondary algorithms. We have previously demonstrated how to leverage AADL to import FMUs as AADL components and use AADL semantics to define this primary algorithm [Hugues 2018]. We rely on the following assumptions.

Actual sensors and actuators are connected to a CPS to provide data and means of actuation. The CPS interacts with those sensors and actuators at a precise time. A Modelica model produces sensor data as output or reacts to actuators' inputs, whereas an AADL model captures the processing chain from sensor data inputs to actuators. The AADL model also specifies when data should flow. The times of these interactions provide the basis to define communication points so that the AADL and Modelica simulation engines can run in parallel.

Hence, we use AADL in two different but complementary ways. First, as a model of a CPS; and second, as a master algorithm to couple an architectural description and other models transformed into an FMU. This approach allows bringing new modeling formalisms to AADL: Modelica, Simulink, or SCADE Suite.

For each FMU, we generate an AADL model that captures the interface of the component, and we support C code to interact with this FMU. This component can then be integrated directly. This model is then integrated into a variant of the model we used for code generation; first as a variant of the device in `BME280_FMU`; and then fully integrated into the `SensorProcessingExample_FMU.x86` system. The latter combines the functional processing with the simulation of the environment provided by Modelica as a binary that can be executed on x86 platforms.

5 Conclusion: TwinOps: A ModDevOps Pipeline for CPS

In the previous sections, we presented our modeling process comprising several modeling and model transformation steps. These steps are combined through a ModDevOps pipeline (as shown in Figure 17). Per construction, all interactions between stages are interactions through the project SCM repository that stores artifacts.

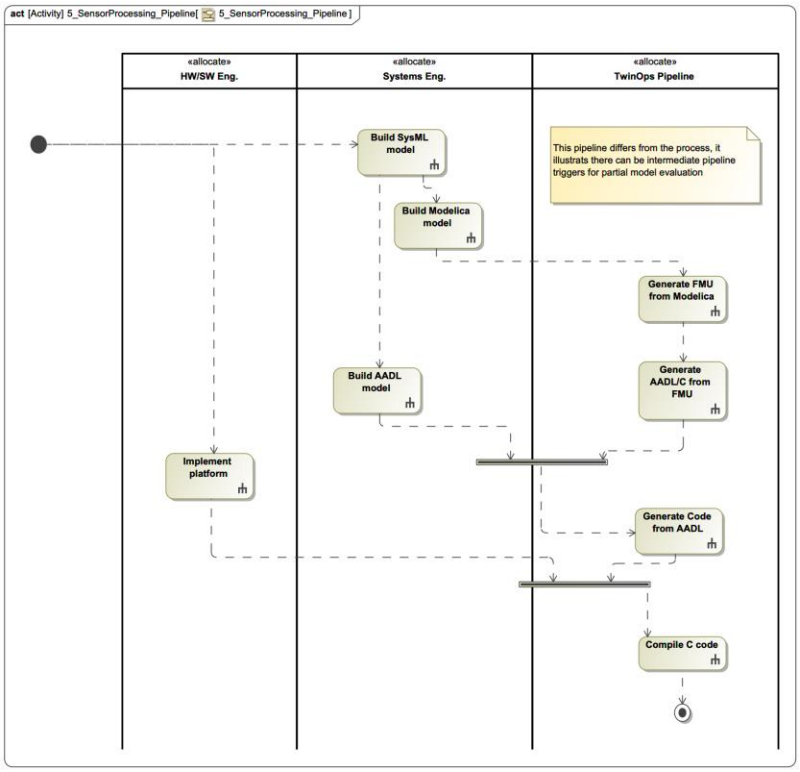

Figure 17: ModDevOps Pipeline

We have organized a traditional CI/CD pipeline that will consider any update at the model or code level to produce a binary that will be deployed on the final platform. This combination of model-based systems and software engineering and DevOps, which we call *ModDevOps*, provides a seamless composition of analyses at model level, model simulation, and deployment on the final targets. It leverages the complete ecosystem of model-based tools to bring increased automation.

We have demonstrated one instance of ModDevOps on the SensorProcessing demo, an IoT system. It leverages SysML, AADL, Modelica, and the C language. First, we illustrated how one can combine SysML and AADL for the development of the embedded component of the system. Next, we showed how code generation and the Azure IoT platform enables a model-to-code pipeline. Finally, we demonstrated the capability to a model-to-simulation pipeline and achieved co-simulation of the AADL and Modelica models.

The capability to generate a simulation of a system from AADL and Modelica, and the actual system from AADL and C, enables a digital twin capability. The AADL models used in both cases

only differ in their interfaces to the sensors and actuators: C for the actual system and Modelica for the simulated one. All other components are shared. This means that the simulation of the system could be faithfully compared to the actual system running, and that we produced an actual digital twin of a CPS.

References

URLs are valid as of the publication date of this document.

[Air Force 2022]
United States Air Force. DevOps. United States Air Force. January 11, 2022 [accessed]. https://software.af.mil/training/devops/

[Blochwitz 2011]
Blochwitz, T. et al. The functional mockup interface for tool independent exchange of simulation models. Pages 105–114. In *Proceedings of the 8th International Modelica Conference*. Dresden, Germany. March 2011. https://ep.liu.se/ecp/063/013/ecp11063013.pdf

[Blochwitz 2012]
Blochwitz, T. et al. Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. Pages 173–184. In *9th International Modelica Conference.* Munich. Germany. November 2012. https://ep.liu.se/ecp/076/017/ecp12076017.pdf

[Boettiger 2014]
Boettiger, C. An introduction to docker for reproducible research, with examples from the r environment. *ACM SIGOPS Operating Systems Review*. Volume. 49. Number 1. January 25. Pages 71-79. https://dl.acm.org/doi/pdf/10.1145/2723872.2723882

[Boydston 2019]
Boydston, A.; Feiler, P.; Vestal, S.; & Lewis, B. Architecture Centric Virtual Integration Process (ACVIP): A Key Component of the DoD Digital Engineering Strategy. In *Proceedings of the 22nd Annual Systems and Mission Engineering Conference*. Tampa, Florida. October 2019. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=634965

[Combemale 2019]
Combemale, B. & Wimmer, W. Towards a Model-Based DevOps for Cyber-Physical Systems. Pages 84-89. In *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment - Second International Workshop, DEVOPS 2019* (revised selected papers, 2019.Volume 12055). Château de Villebrumier, France. May 2019. DOI: 10.1007/978-3-030-39306-9_6

[Delange 2014]
Delange, J.; Feiler, P.; Gulch, D.; & Hudak, J. *AADL Fault Modeling and Analysis Within an ARP4761 Safety Assessment*. CMU/SEI-2014-TR-020. Software Engineering Institute. Carnegie Mellon University. 2014. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=311884

[Delange 2016]
Delange, J.; Feiler, P.; & Neil, E. Incremental Life Cycle Assurance of Safety-Critical Systems. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. Toulouse, France. January 2016. https://hal.archives-ouvertes.fr/hal-01289468

[DoD 2018]
Office of the Deputy Assistant Secretary of Defense for Systems Engineering. Digital Engineering Strategy. *Department Of Defense*. 2018. https://ac.cto.mil/digital_engineering/

[Fairley 2019]
Fairley, R. E. *Systems Engineering of Software-Enabled Systems*. John Wiley & Sons, Inc. 2019. ISBN-13 978-1119535010.

[Feiler 2008]
Feiler, P. H. & Hansson, J. *Flow Latency Analysis with the Architecture Analysis and Design Language (AADL)*. CMU/SEI-2007-TN-010. Software Engineering Institute. Carnegie Mellon University. January 2008. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=8229

[Feiler 2009]
Feiler, P. H.; Hansson, J.; de Niz, D.; & Wrage, L. *System Architecture Virtual Integration: An Industrial Case Study*. CMU/SEI-2009-TR-017. Software Engineering Institute. Carnegie Mellon University. November 2009. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=9145

[Feiler 2016]
Feiler, P.; Delange, J.; Gluch, D.; & McGregor, J. D. *Architecture-Led Safety Process*. CMU/SEI-2016-TR-012. Software Engineering Institute. Carnegie Mellon University. December 2016. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=484826

[Fritzson 2011]
Fritzson, P. *Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica*. Wiley-IEEE Press. 2011. ISBN 978-1-118-01068-6.

[Fritzson 2015]
Fritzson, P. Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach, 2nd edition. Wiley-IEEE Press. 2015. ISBN: 978-1-118-85912-4.

[Gacek 2014]
Gacek, A.; Backes, J.; Cofer, D.; Slind, K.; & Whalen, M. Resolute: An Assurance Case Language for Architecture Models. Pages 19-28. In *Proceedings of the 2014 ACM Sigada Annual Conference On High Integrity Language Technology*. Portland, Oregon. October 2014. https://dl.acm.org/doi/10.1145/2663171.2663177

[Gomes 2017]
Gomes, C.; Thule, C.; Broman, D.; Larsen, P.G.; & Vangheluwe, H. *Co-simulation: State of the art*. *CoRR*. February 17, 2017. http://arxiv.org/abs/1702.00686

[Goseva-Popstojanova 2016]
Goseva-Popstojanova, K.; Kahsai, T.; Knudson, M.; Kyanko, T.; Nkwocha, N.; & Schumann, J. *Survey on Model-Based Software Engineering and Auto-Generated Code*. NASA/TM–2016–219443. National Aeronautics and Space Administration. October 2016. https://ti.arc.nasa.gov/publications/36691/download/

[Hansson 2018]
Hansson, J.; Helton, S.; & Feiler, P. ROI *Analysis of the System Architecture Virtual Integration Initiative*. Software Engineering Institute. Carnegie Mellon University. CMU/SEI-2018-TR-002. April 2018. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=517157

[Hugues 2018]
Hugues, J.; Gauthier, J.; & Faudou, R. Integrating AADL and FMI to Extend Virtual Integration Capability. In *Proceedings of 9th European Congress on Embedded Real Time Software and Systems (ERTSS) 2018.* Toulouse, France. January 2018. https://dblp.org/rec/bib/journals/corr/abs-1802-05620

[Lasnier 2009]
Lasnier, G.; Zalila, B.; Pautet, L.; & Hugues, J. Ocarina: An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications. Pages 237-250. In *Proceedings of Reliable Software Technologies—Ada-Europe 2009, 14th Ada-Europe International Conference.* Brest, France. June 2009. DOI: 10.1007/978-3-642-01924-1_17

[Leite 2019]
Leite, L.; Rocha, C.; Kon, F.; Milojicic, D.; & Meirelles, P. A Survey of DevOps Concepts and Challenges. *ACM Computing Surveys.* Volume 52. Number 6. November 2019. DOI: 10.1145/3359981

[Leserf 2019]
Leserf, P.; de Saqui-Sannes, P.; & Hugues, J. Trade-off Analysis for SysML Models Using Decision Points and CSPS. *Software and Systems Modeling*. Volume 18. Number 6. 2019. Pages 3265–3281. DOI: 10.1007/s10270-019-00717-0

[Microsoft 2021]
Microsoft. Connect Raspberry Pi to Azure IoT Hub (C). *Microsoft*. June 20, 2021. https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-raspberry-pi-kit-c-get-started

[Minksy 1965]
Minksy, Marvin L. Matter, Mind and Models. Pages 45-49. In *Proceedings of the International Federation of Information Processing Congress 1965, Volume 1*. New York, New York. May 1965. https://web.media.mit.edu/~minsky/papers/MatterMindModels.html

[Mkaouar 2020]
Mkaouar, H.; Zalila, B.; Hugues, J.; & Jmaiel, M. A Formal Approach to AADL Model-Based Software Engineering. *International Journal on Software Tools for Technology Transfer*. Volume 22. Number 2. April 2020. Pages 219–247. DOI: 10.1007/s10009-019-00513-7.

[OMG 2019]
OMG. *OMG Systems Modeling Language (OMG SysML) Version 1.6*. OMG. December 2019. https://www.omg.org/spec/SysML/1.6/About-SysML/

[Rauzy 2019]
Rauzy, A.B. & Haskins, C. Foundations for Model-Based Systems Engineering and Model-Based Safety Assessment. *Systems Engineering.* Volume 22. Number 2. March 2019. Pages 146–155. DOI: 10.1002/sys.21469

[Rodrigues da Silva 2015]
Rodrigues da Silva, A. Model-Driven Engineering: A Survey Supported by the Unified Conceptual Model. *Computer Languages, Systems & Structures*. Volume 43. October 2015. Pages 139–155. DOI: https://doi.org/10.1016/j.cl.2015.06.001

[SAE 2015]
SAE International. Aerospace Standard AS5506/1A. SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: ARINC653 Annex, Annex C: Code Generation Annex, Annex E: Error Model Annex SAE International, Standard AS5506/1A. *SAE International*. September 3, 2015. https://saemobilus.sae.org/content/as5506/1a

[SAE 2017]
SAE International. Aerospace Standard AS5506C. Architecture Analysis & Design Language v2.2. *SAE International*. January 2017. https://saemobilus.sae.org/content/as5506c

[SAE 2019]
SAE International. Aerospace Standard AS5506/2. SAE Architecture Analysis and Design Language (AADL) Annex Volume 2: Annex B: Data Modeling Annex, Annex D: Behavior Model Annex, Annex F: ARINC653. *SAE International*. February 18, 2019. DOI: 10.4271/AS5506/2

[SEI 2017]
Software Engineering Institute. AADL and OSATE: A Tool Kit to Support Model-Based Engineering [SEI fact sheet]. *Software Engineering Institute*. March 2017. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=495278

[Smith 2018]
Smith, T.; Whillock, R.; Edman, R.; Lewis, B.; & Vestal, S. *Lessons Learned in Inter-Organization Virtual Integration*. Presented at the Aerospace Systems and Technology Conference. October 2018. DOI: 10.4271/2018-01-1944

[Tanner 2022]
Tanner, Michael. DevStar. *United States Air Force*. January 7, 2022 [accessed]. https://software.af.mil/dsop/dsop-devstar/

[West 2015]
West, T.D. & Pyster, A. Untangling the Digital Thread: The Challenge and Promise of Model-Based Engineering in Defense Acquisition. *INSIGHT*. Volume 18. Number 2. August 2015. Pages 45–55. DOI: 10.1002/inst.12022

[Wikipedia 2020]
Wikipedia. DevOps. *Wikipedia*. January 11, 2022 [accessed]. https://en.wikipedia.org/wiki/DevOps

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	March 2022	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
TwinOps: Digital Twins Meets DevOps	FA8721-05-C-0003

6. AUTHOR(S)
Jérôme Hugues
Joe Yankel
John Hudak
Anton Hristozov

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116	n/a

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

This report summarizes the contributions of the TwinOps project, a one-year project funded by the Software Engineering Institute and executed during FY20. The contributions of this research are twofold. First, it introduced ModDevOps as an innovative approach to bridging model-based engineering and software engineering using DevOps concepts and code generation from models. ModDevOps smooths the transition from model-level verification and validation (V&V) to software production. Second, the research developed TwinOps, a specific ModDevOps pipeline that equips system engineers with new analysis capabilities through the careful combinations of model artifacts as they are built.

14. SUBJECT TERMS	15. NUMBER OF PAGES
model-based engineering, digital twins, DevOps, DevSecOps, system modeling, MBSE, MBE	45

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102