# Model-driven engineering in digital signal processing*

Shirin Hussein[1,*], Sobhan Y. Tehrani[1] and Hossein Beheshti[2]

[1]University College London (UCL), Gower St, London WC1E 6BT
[2]Iran University of Science and Technology (IUST), Tehran, Iran

**Abstract**

This paper introduces a model-driven engineering (MDE) approach for automating MATLAB and Simulink model development in field-programmable gate array (FPGA) based digital signal processing (DSP). To describe system parameters, a domain-specific language (DSL) called YAML is used, which is then processed by a Python script to dynamically produce MATLAB scripts and Simulink models. This automation decreases development time while also ensuring model consistency. The proposed approach is tested by automatically translating a YAML-defined DSP model to MATLAB and Simulink representations. We also discuss how hardware description language (HDL) coder can extend this procedure to FPGA implementation by producing Verilog/VHDL code.

**Keywords**

Model-driven engineering, digital signal processing, domain-specific language, MATLAB, field-programmable gate array

## 1. Introduction

Model-driven engineering (MDE) provides a structured approach to creating complex systems by defining high-level abstractions that can be automatically converted into executable models. This paradigm is commonly used to simplify design specifications, increase reusability, and reduce errors through automated code development [1, 2, 3]. By separating the abstract model definition from the underlying implementation details, MDE improves design efficiency, verification, and maintainability across various engineering disciplines, including embedded systems and cyber-physical systems.

MDE serves as an essential approach for streamlining the development process in digital signal processing (DSP), bridging the gap between hardware implementation and algorithm design [4, 5].

Conventional DSP techniques require manual coding, frequent debugging, and several refining cycles. Using these methodologies can lead to slower development, inconsistencies, and difficulty maintaining correctness, especially as system complexity increases [6]. Advanced DSP applications, such as adaptive filtering, multirate processing, and real-time calculation, require precise optimization that manual design alone may not achieve. MDE enables faster prototyping, thorough verification, and more effective DSP system implementation by using automation and high-level modeling. Although automated model transformations handle the underlying code creation, users can concentrate on improving their algorithms rather than directly dealing with low-level implementation concerns. This method guarantees a seamless transfer from software-based DSP models to field-programmable gate array (FPGA) based hardware, enabling optimal code creation to satisfy processing demands in real time [6]. In addition to increasing efficiency, MDE improves system scalability and model reusability, facilitating the adaptation of DSP architectures to changing requirements without requiring a complete component redesign.

As highlighted in [1], model transformations and automation tools can improve accuracy and scalability and also reduce manual effort. Model-based design techniques, coupled with automatic code

*Joint Proceedings of the STAF 2025 Workshops: OCL, OOPSLE, LLM4SE, ICMM, AgileMDE, AI4DPS, and TTC. Koblenz, Germany, June 10–13, 2025.*

*Corresponding author.
†These authors contributed equally.
shirin.s.hussein@ucl.ac.uk (S. Hussein); sobhan.tehrani@ucl.ac.uk (S. Y. Tehrani); beheshti_h@alumni.iust.ac.ir (H. Beheshti)
0000-0003-0521-4801 (S. Hussein); 0000-0003-4417-0477 (S. Y. Tehrani)

generation, provide a structured method for translating high-level DSP descriptions into optimized implementations.

Domain-specific languages (DSLs) have become an effective tool for expressing system specifications at a higher abstraction level in order to further improve automation. Designers can improve model consistency and reduce the reliance on manual coding by using a DSL to capture system parameters and processing components in an organized way [7]. Additionally, this abstraction makes automated code generation easier, especially for FPGA deployment and hardware acceleration, where system specifications must be effectively converted into hardware description language (HDL) code for synthesis.

In this work, a YAML-based DSL is employed to automate the creation of MATLAB and Simulink models using an MDE-based methodology. YAML (YAML Ain't Markup Language) is a human-friendly data serialization language designed for easy use with modern programming languages [8]. Users can define signal characteristics, processing stages, and system limitations in a structured way using the suggested approach. Following that, a Python script converts this specification into Simulink models and MATLAB scripts, eliminating the need for manual model building and guaranteeing consistency and effectiveness in the creation of DSP systems. Additionally, the created models can be converted to HDL for FPGA implementation by utilizing MATLAB's HDL Coder. The effectiveness of using YAML in MDE contexts for model management and transformation has been previously demonstrated in [9].

YAML has gained popularity as a choice for organizing configuration data due to its straightforward syntax, readability for humans, and user-friendly nature. Provides essential elements such as mappings, sequences, and scalar values, making it ideal for representing data in a hierarchical format. The features of YAML have contributed to its popularity in areas such as cloud infrastructure management, DevOps practices, and API development [10, 11, 12]. Furthermore, YAML has shown its value in software engineering research, especially in MDE, by facilitating expressive and lightweight domain-specific modeling and aiding in automation of tasks related to code generation and model transformation [9].

This research enhances the Agile MDE paradigm by facilitating quick, iterative development of DSP systems through automation and elevated abstractions. The implementation of a YAML-based DSL enables developers to quickly adjust system specifications, while the automated creation of MATLAB and Simulink models fosters rapid development cycles and constant prototyping. These features resonate with Agile principles, including adaptability, rapid feedback, and ongoing model refinement.

The structure of the paper is as follows: Section 2 reviews related work in MDE for DSP and FPGA-based systems. Sections 3 and 4 detail the proposed methodology, including DSL design, automated model generation and, its applicability to FPGA deployment. Section 5 presents an evaluation of the approach. Finally, Section 6 concludes with key findings and future research directions.

## 2. Related Work

MDE has become a preferred approach in areas such as software development, embedded systems, and high-integrity systems. What makes it so appealing is its ability to reduce repetitive work by automating parts of the design process, especially code generation. Existing research explores how model-to-model and model-to-code transformations can help streamline system development [13]. This kind of automation plays an important role in DSP workflows, where it can reduce the amount of manual effort needed and make it easier to adapt designs as requirements change. In many DSP projects, MDE has been used to keep development consistent across different phases and to speed things up [14, 15, 16, 1].

However, applying MDE to the design of DSP systems, particularly when working with FPGA-based platforms, remains challenging due to hardware-specific constraints and the limited capabilities of current model synthesis approaches [17, 18, 19]. Numerous current tools depend on rigid configurations, providing minimal versatility for model personalization or alteration modifications [20]. Furthermore, incorporating MDE into hardware design processes presents distinct challenges that are not found in software-focused scenarios, particularly when aiming for FPGA architecture [21].

There has been growing evidence supporting the value of MDE in areas like embedded systems and sensing technologies. One example can be seen in work that used Simulink to model sensing and actuation subsystems, showing how MDE approaches can simplify the design process through automated code generation [22]. This kind of simplification is closely related to our own efforts in automating the development of DSP systems.

In a separate case, a framework was introduced for embedded system design that leverages model transformations to automate design generation [23]. By doing so, it not only eases the exploration of different design options but also cuts down on overall development complexity. These findings further emphasize the strength of structured, model-driven workflows, especially when applied to DSP-based and FPGA-based systems. Being able to describe DSP processes through high-level models is especially useful when bridging the gap between algorithm development and actual hardware implementation.

Some studies have explored the role of DSLs in the design of DSP systems. Feldspar, a DSL designed for DSP algorithms, employs a functional programming approach to generate optimized C code for execution, demonstrating the potential for high-level abstraction in DSP workflows [24]. Similarly, a survey of FPGA-based DSLs highlights how high-level modeling techniques streamline the transition from software-defined models to efficient hardware implementations, reducing development complexity and improving adaptability across different platforms [6].

Despite these advancements, many current workflows still depend heavily on manual setup when it comes to configuring models and generating executable code. Many existing tools adhere to predefined toolchains, making it hard to adapt quickly to different DSP architectures. When targeting FPGA deployment for DSP systems, users usually choose one of the two options: writing everything manually in Verilog or VHDL or using MATLAB's System Generator (SysGen) together with Simulink.

In the first case, using Verilog or VHDL, DSP functions are written directly in a hardware description language. This gives users complete control over how FPGA resources are used and allows them to fine-tune performance. But the downside is clear: it is slow, prone to mistakes, and requires deep hardware knowledge. For users who are more comfortable working at a high level, the transition to HDL presents a significant challenge, and it slows down rapid prototyping.

On the flip side, SysGen offers a more visual design environment by integrating with Simulink. It lets users build DSP systems using block diagrams, which can be a lot more intuitive. However, it is not without its own challenges. Designers still have to manually pick blocks, adjust parameters, and configure the toolchain. In addition, SysGen does not really offer a clean way to abstract or generalize designs, which makes iterating on different DSP setups harder than it should.

In this work, we build on previous research by introducing a YAML-based DSL for defining DSP systems in a more structured way. This DSL addresses some of the pain points in both traditional HDL workflows and block-based environments like SysGen. Using Python scripts, we automate the creation of MATLAB and Simulink models, cutting down on repetitive tasks and improving reusability across various DSP setups. We also show how this process connects smoothly to FPGA deployment using HDL Coder, effectively bridging the gap between high-level modeling and low-level implementation. The result is a more flexible workflow that allows users to define, simulate, and deploy DSP systems without being constrained by hardware-specific requirements or inflexible toolchains.

## 3. Methodology

This section presents the proposed DSL-driven approach to automate MATLAB and Simulink model generation and FPGA deployment. The technique presents a standardized workflow for converting high-level DSP system specifications into executable models, assuring consistency and automation throughout the design process.

As shown in Figure 1, the process includes four key stages:

- Model Definition
- DSL Parsing & Code Generation

- Simulink Model Generation
- Extension to Hardware Implementation

Each stage contributes significantly to the automation of DSP system modeling, the elimination of manual configuration, and the ability to generate hardware seamlessly. The proposed approach follows a structured workflow that transforms high-level DSP system specifications into executable models, ensuring consistency and automation throughout the design process. As illustrated in Figure 1, the process begins with the definition of the model, where the user defines the DSP system using a DSL based on YAML. This includes specifying signal characteristics such as type, frequency, and amplitude, along with processing components such as filters and sampling rates. The next step, DSL parsing and MATLAB script generation, involves a Python-based automation script that reads the YAML file, extracts key parameters, and generates the corresponding MATLAB code, eliminating the need for manual scripting and reducing inconsistencies. Once the MATLAB script is generated, the process moves to Simulink model generation, where the system is dynamically constructed in Simulink based on the YAML input. This includes the automatic addition of signal sources, processing blocks, and visualization tools, allowing for a fully automated system representation. Finally, the generated Simulink model can be extended to hardware implementation through HDL code generation. By integrating FPGA-compatible blocks, MATLAB's HDL coder can be used to generate synthesizable VHDL/Verilog code, making the model suitable for deployment on FPGA platforms. This workflow provides a flexible and scalable approach to DSP system design, reducing manual effort while ensuring efficient code generation. The following sections detail each transformation step in the methodology.

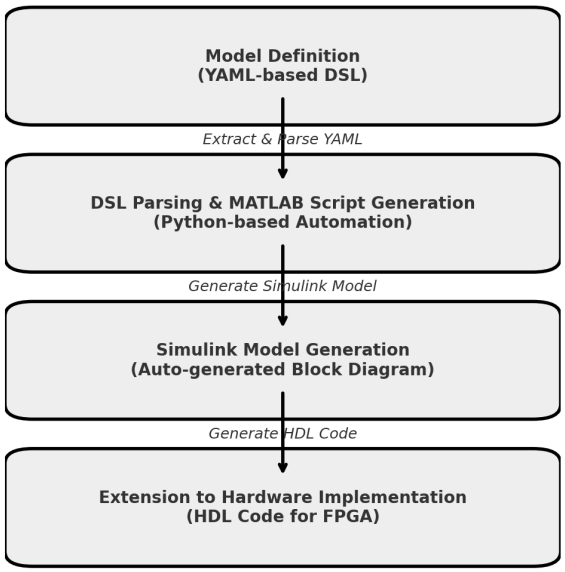


Figure 1: Process overview diagram.

## 3.1. Model Definition

The YAML based DSL defines DSP system components in a systematic and intuitive manner, speeding the specification of signal sources, filtering processes, and system parameters. This approach separates DSP design from low-level implementation concerns through the use of a high-level abstraction, making it more flexible, scalable, and easy to alter.

During this model definition stage, users define crucial DSP attributes such as signal type, frequency, amplitude, phase, and sampling rate. Furthermore, processing components such as filters can be set with

appropriate parameters such as cutoff frequency and filter type. An example of a YAML specification is shown in Figure 2.

```yaml
signal:
  type: sine
  frequency: 1e6        # 1 MHz
  amplitude: 1.0        # 1V
  phase: 45             # 45 degrees

processing:
  filter: lowpass   # type of filter
  cutoff: 200e3     # 200 kHz
  sampling_rate: 10e6 # 10 MHz
```

Figure 2: Sine signal and low-pass filter configuration.

This structured method helps separate the high-level design of DSP systems from the actual implementation in MATLAB and Simulink. By using YAML to provide DSP settings, developers may easily change system specs without having to make repeated changes to the underlying code, which speeds up prototyping and increases flexibility in general. It also helps keep things consistent across different configurations, which reduces the chance of errors and makes it easier to adapt designs to various hardware platforms.

In addition, the YAML file acts as a machine-readable blueprint that automated scripts can use to generate executable models. This setup not only speeds things up, but it also ensures that any changes made to the DSP design are automatically reflected throughout the development pipeline from early simulation stages all the way to hardware deployment.

### 3.2. DSL Processing and Code Generation

A Python-based automation tool that parses the YAML file and dynamically generates MATLAB scripts and Simulink models helps to convert high-level DSP requirements to executable models. This automated technique maintains consistency, eliminates manual effort, and allows quick prototyping of DSP systems by converting abstract specifications into functional implementations.

The script pulls key information from the YAML file, such as signal properties (frequency, amplitude, phase) and processing parameters such as filter type, cutoff frequency, and sampling rate. It then converts these data into MATLAB code for signal generation, filtering operations, and the construction of corresponding Simulink models. Figure 3 presents a simplified example of how the script converts YAML input into functional MATLAB code.

### 3.3. Simulink Model Generation

After the Python script is executed, it generates a MATLAB script that is subsequently run by MATLAB. This process results in the dynamic creation of a Simulink model. The following command is used:

```
run('generated_simulink.m')
```

Simulink block diagrams are automatically generated based on YAML defined parameters. This technique eliminates the need for manual block selection and configuration, guaranteeing that the model complies with the user-defined DSP specifications. The created model contains a sine wave block, which serves as the signal source and is configured with the desired frequency, amplitude, and phase values. The signal is then passed through a low-pass filter block, which performs the filtering operation at the specified cutoff frequency. Finally, the model includes a scope block, which visualizes the processed signal and allows study and verification of the generated output. Figure 4 illustrates the autogenerated Simulink model based on the YAML input.

```
import yaml

# Loading YAML DSL file
with open("signal_processing.yml", "r") as file:
    config = yaml.safe_load(file)

# Extracting parameters from YAML
frequency = float(config["signal"]["frequency"])
amplitude = float(config["signal"]["amplitude"])
phase = float(config["signal"]["phase"])
cutoff = float(config["processing"]["cutoff"])
sampling_rate = float(config["processing"]["sampling_rate"])

# Ensuring stopband frequency is greater than the passband frequency
stopband = cutoff * 1.5

# Block path
lowpass_filter_path = 'dspfdesign/Lowpass Filter'

# Generating MATLAB Simulink script
simulink_code = f"""
% Creating and opening a new Simulink Model
new_system('GeneratedModel');
open_system('GeneratedModel');

% Adding Sine Wave Block
add_block('simulink/Sources/Sine Wave', 'GeneratedModel/SineWave', ...
    'Frequency', '{frequency}', 'Amplitude', '{amplitude}', 'Phase', '{
        phase}', ...
    'SampleTime', '1/{sampling_rate}');

% Adding Lowpass Filter Block
add_block('{lowpass_filter_path}', 'GeneratedModel/LowpassFilter');

% Setting correct filter parameters
set_param('GeneratedModel/LowpassFilter', 'PassbandFrequency', '{cutoff}');
set_param('GeneratedModel/LowpassFilter', 'StopbandFrequency', '{stopband}'
    );
set_param('GeneratedModel/LowpassFilter', 'SampleRate', '{sampling_rate}');

% Adding Spectrum Analyzer Block
add_block('audiosinks/Spectrum Analyzer', 'GeneratedModel/SpectrumAnalyzer'
    );

% Connecting blocks
add_line('GeneratedModel', 'SineWave/1', 'LowpassFilter/1');
add_line('GeneratedModel', 'LowpassFilter/1', 'SpectrumAnalyzer/1');

% Saving and Running the Model
save_system('GeneratedModel');
sim('GeneratedModel');

% Opening the Spectrum Analyzer to view frequency response
open_system('GeneratedModel/SpectrumAnalyzer');
"""
# Saving MATLAB script for Simulink
with open("generated_simulink.m", "w") as simulink_file:
    simulink_file.write(simulink_code)
print(" Simulink model script generated successfully!")
```

Figure 3: Python script for generating a MATLAB filter simulation.

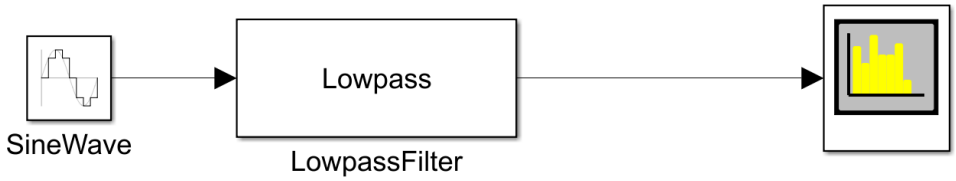


Figure 4: Example of autogenerated Simulink model from DSL input.

## 4. Extending to FPGA Implementation

The Simulink models generated can be extended for FPGA implementation using the MATLAB HDL coder. By configuring the Simulink model with HDL-compatible blocks, the system can be synthesized into Verilog or VHDL code. The following MATLAB command is used to generate HDL from a Simulink block:

```
makehdl('GeneratedModel')
```

This step enables the seamless transition from high-level DSP design to hardware realization, bridging the gap between software-defined models and FPGA deployment.

The automation of MATLAB script generation and Simulink model construction provides numerous benefits that improve the efficiency and accuracy of DSP system development. One of the key advantages is consistency and accuracy. By removing manual scripting, the system ensures that the created models adhere properly to user-defined parameters, reducing the possibility of errors and inconsistencies. This ensures that each created model retains the desired design characteristics, allowing for more consistent and predictable results. Rapid prototyping is yet another important benefit. Instead of requiring substantial manual adjustments in MATLAB or Simulink, developers may quickly test various DSP setups by just altering the YAML file. Scalability is a further significant component of this approach. More complicated DSP workflows, such as multirate processing, adaptive filtering, and FPGA-based implementations, can be quickly expanded to using the YAML-based DSL. Without having to rewrite low-level code, users may conveniently add new processing stages, alter current configurations, or incorporate more signal processing components by organizing DSP system definitions at a high level. Additionally, this approach provides smooth hardware integration. Efficient deployment on FPGA-based platforms is made achievable by the direct extension of the created Simulink model to HDL production. By bridging the gap between real-time hardware implementations and software-defined DSP models, this approach makes it feasible to generate efficient code for high-performance embedded systems.

## 5. Evaluation and Comparison

To determine the efficiency of the proposed DSL-driven MATLAB/Simulink automation, we compare it to typical Verilog-based DSP simulations. Although Verilog is widely used for cycle-accurate simulation and FPGA synthesis, it poses difficulties in development, debugging, and automation. In contrast, MATLAB/Simulink provides a higher-level modeling environment that interacts well with DSP procedures but may impose a significant processing burden. Table 1 compares Verilog-based simulation and MATLAB/Simulink modeling in the context of DSP system development [25, 26, 27, 28, 29].

According to the comparison Table 1, our solution expands on MATLAB/Simulink's modeling capabilities while addressing its limitations in manual configuration and automation. Because MATLAB offers a structured environment for DSP algorithm development and verification, we use its interaction with Simulink to automate model building and simulation. Unlike manual block-based design, our

approach uses a DSL-driven automation framework in which DSP system parameters are defined in YAML and automatically translated into MATLAB scripts and Simulink models. This maintains consistency, eliminates manual work, and speeds up the transition from algorithm design to FPGA implementation via MATLAB's HDL Coder for hardware synthesis. In a standard approach, changing Simulink parameters, for example, the filter cutoff or signal frequency, requires opening the model manually, finding the required blocks, and changing their settings via the graphical user interface. In particular, when used over several models or iterations, this procedure can be repetitious and prone to errors. The suggested MDE method enables such changes to be implemented with only one line change in the YAML configuration. After that, the system automatically regenerates the Simulink model and the MATLAB script, ensuring that all updates are performed uniformly and without the need for human intervention. This speeds up iterative development, improves reliability, and streamlines the modification process.

**Table 1**
Verilog Simulation vs. MATLAB/Simulink-Based DSP Modeling

| **Aspect** | **Verilog Simulation** | **MATLAB/Simulink Modeling** |
|---|---|---|
| **Development Process** | Requires manual coding of DSP operations and testbenches. | Uses block-based modeling, reducing coding effort. |
| **Simulation Approach** | Cycle-accurate, low-level signal representation. | System-level modeling with solver-based execution. |
| **Debugging & Analysis** | Uses waveform viewers (e.g., ModelSim, Vivado). | Built-in visualization (Scopes, Spectrum Analyzer). |
| **Modification Effort** | Modifications require HDL code changes. | Block parameters can be adjusted more easily. |
| **FPGA Readiness** | Directly synthesizable but requires optimization. | Requires conversion to HDL using HDL Coder. |
| **Common Use Cases** | Low-level hardware validation and FPGA prototyping. | Algorithm development, high-level DSP design. |

## 6. Conclusion

In this work, we introduced an MDE approach aimed at automating the creation of DSP models using a YAML-based DSL. By combining this with Python-driven automation, we showed how MATLAB scripts and Simulink models can be dynamically generated, significantly reducing manual effort and promoting consistency across designs. This approach helps close the gap between high-level DSP design and hardware implementation, making it easier to go from abstract model definitions to HDL code that is ready for FPGA deployment using MATLAB's HDL Coder.

We set out to tackle some of the challenges that come with traditional DSP design methods. Writing everything manually in Verilog or VHDL can offer precise control over hardware, but it is time consuming and demands deep hardware knowledge. On the other hand, tools like MATLAB's SysGen give users a more visual interface but still involve a lot of hands-on configuration, which can slow things down, especially when dealing with larger or more complex systems. What we are proposing streamlines this entire process by automating model generation and encouraging code reuse, which makes development faster and easier to scale.

Future work will focus on extending the framework to handle more advanced DSP use cases, such as multi-rate signal processing and adaptive filtering. We are also exploring ways to optimize how hardware resources are allocated, and we may experiment with alternative ways to define system behavior beyond YAML. Finally, testing the framework on real FPGA hardware will give us a clearer

picture of how well it performs in practical scenarios.

## Declaration on Generative AI

The authors declare that no generative artificial intelligence tools were used to generate text, figures, code, data, analyses, or reviews for this submission.

## References


[1] A. R. Da Silva, Model-driven engineering: A survey supported by the unified conceptual model, *Comput. Lang. Syst. Struct.*, vol. 43, Elsevier, 2015, pp. 139–155.

[2] V. Cortellessa, D. Di Pompeo, R. Eramo, and M. Tucci, A model-driven approach for continuous performance engineering in microservice-based systems, *J. Syst. Softw.*, vol. 183, Elsevier, 2022, p. 111084.

[3] G. Sebastián, J. A. Gallud, and R. Tesoriero, Code Generation Using Model Driven Architecture: A Systematic Mapping Study, *Journal of Computer Languages*, vol. 56, pp. 100935, Elsevier, 2020.

[4] L. Li, C. Sau, T. Fanni, J. Li, T. Viitanen, F. Christophe, F. Palumbo, L. Raffo, H. Huttunen, J. Takala, et al., An integrated hardware/software design methodology for signal processing systems, *J. Syst. Archit.*, vol. 93, Elsevier, 2019, pp. 1–19.

[5] D. Akdur, V. Garousi, and O. Demirörs, A Survey on Modeling and Model-Driven Engineering Practices in the Embedded Software Industry, *Journal of Systems Architecture*, vol. 91, pp. 62–82, Elsevier, 2018.

[6] N. Kapre, S. Bayliss, Survey of domain-specific languages for FPGA computing, in: 2016 26th International Conference on Field Programmable Logic and Applications (FPL), IEEE, 2016, pp. 1–12.

[7] K. Panayiotou, C. Doumanidis, E. Tsardoulias, and A. L. Symeonidis, SmAuto: A domain-specific language for application development in smart environments, *Pervasive Mob. Comput.*, vol. 101, Elsevier, 2024, p. 101931.

[8] C. Evans, O. Ben-Kiki, and I. döt Net, *YAML Ain't Markup Language (YAML™) Version 1.2*, Mar. 2017. [Online]. Available: https://yaml.org/spec/1.2/spec.html

[9] I. Predoaia, D. Kolovos, A. Garcia-Dominguez, M. Lenk, W. Ebel, and J. Burkl, Towards processing YAML documents with model management languages, *Proc. ACM/IEEE 27th Int. Conf. Model Driven Eng. Lang. Syst. (MODELS Companion)*, pp. 970–979, 2024.

[10] D. Calcaterra, V. Cartelli, G. Di Modica, and O. Tomarchio, A Framework for the Orchestration and Provision of Cloud Services Based on TOSCA and BPMN, *Proc. Int. Conf. on Cloud Computing and Services Science (CLOSER)*, pp. 262–285, Springer, 2018.

[11] A. Deljouyi and R. Ramsin, MDD4REST: Model-Driven Methodology for Developing RESTful Web Services, *Proc. 10th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pp. 93–104, INSTICC, SciTePress, 2022.

[12] B. Piedade, J. P. Dias, and F. F. Correia, An Empirical Study on Visual Programming Docker Compose Configurations, *Proc. 23rd ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS-C)*, pp. 1–10, ACM/IEEE, 2020.

[13] L. Burgueño, D. Di Ruscio, H. Sahraoui, and M. Wimmer, Automation in Model-Driven Engineering: A look back, and ahead, ACM Trans. Softw. Eng. Methodol., ACM, New York, NY, 2025.

[14] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, Morgan & Claypool Publishers, 2017.

[15] V. Cortellessa, D. Di Pompeo, R. Eramo, and M. Tucci, A Model-Driven Approach for Continuous Performance Engineering in Microservice-Based Systems, *Journal of Systems and Software*, vol. 183, pp. 111084, Elsevier, 2022.

[16] D. Di Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi, and M. Wimmer, Low-Code Develop-

ment and Model-Driven Engineering: Two Sides of the Same Coin?, *Software and Systems Modeling*, vol. 21, no. 2, pp. 437–446, Springer, 2022.

[17] R. Dalbouchi, C. Trabelsi, M. Elhajji, and A. Zitouni, A Model-Driven Platform for Dynamic Partially Reconfigurable Architectures: A Case Study of a Watermarking System, *Micromachines*,

[18] O. Diouri, A. Gaga, H. Ouanan, S. Senhaji, S. Faquir, and M. O. Jamil, Comparison Study of Hardware Architectures Performance Between FPGA and DSP Processors for Implementing Digital Signal Processing Algorithms: Application of FIR Digital Filter, *Results in Engineering*, vol. 16, pp. 100639, Elsevier, 2022.
vol. 14, no. 2, pp. 481, MDPI, 2023.

[19] L. Jóźwiak, N. Nedjah, and M. Figueroa, Modern Development Methods and Tools for Embedded Reconfigurable Systems: A Survey, *Integration*, vol. 43, no. 1, pp. 1–33, Elsevier, 2010.

[20] G. Singh, M. Alser, D. S. Cali, D. Diamantopoulos, J. Gómez-Luna, H. Corporaal, and O. Mutlu, FPGA-based near-memory acceleration of modern data-intensive applications, IEEE Micro, vol. 41, no. 4, IEEE, 2021, pp. 39–48.

[21] Q. Sun, T. Chen, S. Liu, J. Chen, H. Yu, and B. Yu, Correlated multi-objective multi-fidelity optimization for HLS directives design, ACM Trans. Des. Autom. Electron. Syst. (TODAES), vol. 27, no. 4, ACM, New York, NY, 2022, pp. 1–27.

[22] F. S. Gonçalves and L. B. Becker, Model driven engineering approach to design sensing and actuation subsystems, in: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), IEEE, 2016, pp. 1–8.

[23] F. A. M. do Nascimento, M. F. S. Oliveira, and F. R. Wagner, A model-driven engineering framework for embedded systems design, Innov. Syst. Softw. Eng., vol. 8, Springer, 2012, pp. 19–33.

[24] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, A. Vajda, Feldspar: A domain specific language for digital signal processing algorithms, in: Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), IEEE, 2010, pp. 169–178.

[25] G. Zheng, S. P. Mohanty, and E. Kougianos, Design and modeling of a continuous-time delta-sigma modulator for biopotential signal acquisition: Simulink vs. Verilog-AMS perspective, *Proc. 3rd Int. Conf. Comput., Commun. Netw. Technol. (ICCCNT'12)*, IEEE, 2012, pp. 1–6.

[26] C. Bäck, Evaluation of high-level synthesis tools for generation of Verilog code from MATLAB-based environments, 2020.

[27] X. Tang, E. Giacomin, G. De Micheli, and P.-E. Gaillardon, FPGA-SPICE: A simulation-based architecture evaluation framework for FPGAs, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 3, pp. 637–650, IEEE, 2018.

[28] R. Saralegui, A. Sanchez, and Á. de Castro, Efficient hardware-in-the-loop models using automatic code generation with MATLAB/Simulink, *Electronics*, vol. 12, no. 13, pp. 2786, MDPI, 2023.

[29] Y. Gu, K. Kintali, and E. Cigan, Model-based design using Simulink, HDL Coder, and DSP Builder for Intel FPGAs, *Matlab Inc. White Paper*, 2014.