Classification and Mapping of Model Elements for Designing Runtime Reconfigurable Systems

DENIS S. LOUBACH1, (Member, IEEE), RICARDO BONNA2, GEORGE UNGUREANU3, INGO SANDER3, (Member, IEEE), INGEMAR SÖDERQUIST4, (Senior Member, IEEE)

1Department of Computer Systems, Computer Science Division, Aeronautics Institute of Technology – ITA, São José dos Campos, SP, Brazil 12228-900, email: dloubach@ita.br
2Advanced Computing, Control & Embedded Systems Lab., University of Campinas – UNICAMP, Campinas, SP, 13083-860 Brazil, email: rbonna@fem.unicamp.br
3School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, Kista, SE-164 40, Sweden, email: {ugeorge, ingo}@kth.se
4Saab AB, Linköping, Sweden, email: ingemar.soderquist@saabgroup.com

Corresponding author: Denis S. Loubach (email: dloubach@ita.br).

This research work is supported by the Regular Research Grant #2019/27327-6 from São Paulo Research Foundation (FAPESP). This work publication was supported by the project “Real-Time Electronics & Computers for Highly Computing Intensive Real-Time Systems”.

Abstract

Embedded systems are ubiquitous and control many critical functions in society. A fairly new type of embedded system has emerged with the advent of partial reconfiguration, i.e. runtime reconfigurable systems. They are attracting interest in many different applications. Such a system is capable of reconﬁguring itself at the hardware level and without the need to halt the application’s execution. While modeling and implementing these systems is far from a trivial task, there is currently a lack of systematic approaches to tackle this issue. In other words, there is no unanimously agreed upon modeling paradigm that can capture adaptive behaviors at the highest level of abstraction, especially when regarding the design entry, namely, the initial high-level application and platform models. Given this, our paper proposes two domain ontologies for application and virtual platform models used to derive a classiﬁcation system and to provide a set of rules on how the different model elements are allowed to be composed together. The application behavior is captured through a formal model of computation which dictates the semantics of execution, concurrency, and synchronization. The main contribution of this paper is to combine suitable formal models of computation, a functional modeling language, and two domain ontologies to create a systematic design flow from an abstract executable application model into a virtual implementation model based on a runtime reconfigurable architecture (virtual platform model) using well-deﬁned mapping rules. We demonstrate the applicability, generality, and potential of the proposed model element classiﬁcation system and mapping rules by applying them to representative and complete examples: an encoder/decoder system and an avionics attitude estimation system. Both cases yield a virtual implementation model from an abstract application model.

Index Terms

Embedded systems, Runtime reconfiguration, Models of computation (MoC), Domain ontology, Mapping rules.

I. INTRODUCTION

Nowadays, embedded systems are ubiquitous and control many vital and safety-critical domains, such as transportation, communication, and medical devices. Modeling and correctly implementing these systems is not an easy task.

In embedded systems, formal models have for a long time aided in managing the complexity of emergent behaviors arising from combining heterogeneous components [1]. The role of models, in general, is to capture the essential aspects of a system’s behavior at a high level as abstract, well-behaved, and mathematical representations. A formalism gives value to a model from the perspective of a computer-aided design flow, thus equipping it with tools and methods for analysis, and rules for evaluation and veriﬁcation. In this context, formal models become specifications, or entry points in a design flow enabling rigorous, traceable an
amendable path to system implementations.

Yet, there is a lack of systematic approaches for the design of runtime reconfigurable systems (RTR), which can guarantee a possible and correct implementation, especially when regarding the design entry, namely, the initial high-level models for system specification.

RTR systems are embedded systems capable of recon-figuring part of their hardware without the need to stop the execution of the application implemented on it. Such a system is gaining interest for many different applications due to its adaptive capability that could be explored to reduce power consumption, increase performance [2] and provide fault tolerance [3]. Needless to say, a systematic design methodology for RTR systems is essential for the correct implementation of such systems. Serpanos et al. [4] state important research challenges, including the need for novel tools to enable the embedded systems’ design at different abstraction levels and also considering platform definitions and runtime reconfigurability aspects.

In view of this, the scientific community needs to consider new strategies for the design of RTR embedded systems. Other researches take for example software and hardware codesign [5], [6], testing and debugging [7], or even concepts analyzed as separated pieces. Also, most of them do not tackle software and hardware reconfiguration as formal aspects in the design flow, even when examining RTR [8].

We address these questions through the combination of ontologies and formal models of computation. In this sense, our paper proposes two domain ontologies: an application-domain ontology and a platform-domain ontology. They are used to derive a classification system and to provide directions on how the different model elements are allowed to be composed together. On the other hand, a well-defined semantics is provided by a formal model of computation (MoC) [1] that dictates execution, concurrency, and synchronization. Considering the design of runtime reconfigurable embedded systems, we propose a flow taking into account (i) an application model, (ii) a virtual platform model, and (iii) a set of mapping rules developed in the paper.

The application model is described based on the theory of models of computation and admitting reconfigurable behavior. The scalable platform model represents the abstraction of the underlying software and heterogeneous hardware, i.e. it constitutes a virtual platform. When applying a set of well-defined mapping rules on these models, the result is a correct virtual implementation model able to be further compiled into software or synthesized into hardware. Compilation and synthesis are not within the scope of the paper, much rather as potentially design steps enabled by virtual implementations.

The models are expressed as functional programs (i.e. hosted on a functional programming language), where functions are used as first-class citizens. This means that a host language provides both a semantic foundation and the evaluation mechanisms necessary to pass functions as values. Hence functional programming enables a convenient environment to describe adaptive models of computation and demonstrate the applicability of our classification system and mapping rules.

Using two different representative examples, we apply our proposal to two runtime reconfigurable embedded systems to demonstrate the applicability, generality, and scalability of our approach.

As main contributions of this paper, we combine a set of suitable formal models of computation, a functional modeling language, and two domain ontologies to create a systematic design flow from an abstract executable system model into a virtual implementation based on a runtime reconfigurable architecture, a virtual platform, and using well-defined mapping rules.

The main contribution points are summarized as follows:
1) an application-domain ontology used to derive a classification system and composition criteria to model an application (Sections III-A1 and III-A5);
2) a platform-domain ontology used to derive a classification system and composition criteria to model a virtual platform (Section III-B1);
3) a generalization of the runtime reconfiguration concept by using different MoCs, i.e. timed and untimed. We formulate the requirements for a MoC to be able to describe reconfigurable behaviors and we give examples of suitable MoCs found in the literature (Section III-A4); and
4) a set of well-defined mapping rules that when applied to the high-level application and platform models, leads to a feasible virtual implementation model (Section III-C).

II. RELATED WORKS

For over two decades [9], commercial FPGAs can perform a partial reconfiguration. However, the usage of runtime hardware reconfiguration has been mostly underutilized. Nguyen et al. [2] argue that the lack of use cases, demonstrating the advantage of runtime reconfiguration (RTR) over traditional static reconfiguration, may be the root cause. For this matter, the authors of [2] present a quantitative demonstration of the benefits of such an approach for embedded vision applications. They mention aspects like power consumption and performance.

Fortunately, in recent years, runtime reconfiguration is being revisited, and many different applications are exploring the advantages of such a feature. In [3], runtime partial hardware reconfiguration is used in the embedded control system of a multirotor enabling different rotor configurations to be switched in runtime. The authors point out that runtime reconfiguration will make the multirotor more fault-tolerant and adaptive. In [10], the usage of runtime reconfiguration is explored to increase the security of implemented cryptographic algorithms against side-channel attacks.

Even with the increasing interest in runtime reconfiguration capabilities, most applications are still designed using ad-hoc methods due to the lack of a well-structured flow.
for the design of general RTR applications. Here, we intend to advance the state-of-the-art regarding the design of RTR embedded systems by using the concepts of ontologies, models of computation, and mapping rules. Our goals address several of the challenges posed in [11], such as the system’s possibility to adapt itself in runtime and the use of formal methods to enable correct operation guarantees.

A model-based design is introduced in [12], although they not use runtime reconfiguration. The authors claim for an approach for rapid development of embedded systems. Their solution is based on STMicroelectronics and Mathworks related tools where they created a Simulink blockset. Our proposal here is to use as much as possible of free and even open-source frameworks and tools mainly in the specification, modeling, and simulation phases. In the case of low-level automatic code generation, one can take advantage of commercial tools’ engines, such as the Simulink embedded coder mentioned in [12].

In [13], the authors show a formal model and a corresponding design methodology applicable for the context of a heterogeneous system on chip. However, they address only the specification of access permissions and information flow requirements for embedded systems in an abstract manner and provide configuration codes at the end. In the present paper, we conceptualize a design flow taking into account executable functional specifications, based on a suitable model of computation, along with functional blocks of a virtual platform. Next, we derive a feasible virtual implementation by combining the functional specifications and blocks by a set of well-defined mapping rules.

Khalgui et al. [14] propose a design methodology applicable to reconfigurable discrete-event systems. They use a profile called R-UML, which is a UML-extension. Then, in a further step, the method counts on a transformation to consider formalisms, e.g., Petri nets, and timed automata. In [15], the authors combine semi-formal and formal methods and also use model transformations. Our proposal already starts with formal models of computation.

According to [16], an ontology is an “explicit organization of knowledge” structured by concepts and the relation between them. When used in a specific domain, this enables formalization of information. In this sense, we developed two domain ontologies: application and platform, as well as a set of mapping rules connecting both.

A method to infer semantic properties using lattice-based ontologies supported by manual annotations is presented in [17]. In the present paper, we introduce ontologies used to derive a classification system that allows for efficient mapping of application model elements to platform functional blocks.

Embedded systems design is intrinsically domain-specific. Given this, the work [18] proposed a formal foundation to evaluate if two different domain-specific modeling languages or metamodels are equivalent.

The authors of [19] defined hardware and software co-design as a possibility to integrate the hardware and software design techniques in a framework and introduced a formal co-design methodology applicable to embedded systems specifications. Here, we present a set of well-defined mapping rules to enable the elements of an application model to be systematically mapped to the functional blocks in a platform model, leading to a possible and feasible implementation model.

Self-adaptation is one of the most relevant strategies to cope with the increasing complexity of software systems [20]. As claimed by those authors, this complexity demands the adoption of formal methods in systems design. In the present research, the application model is based on formal models of computation, which confers execution, concurrency, and synchronization semantics.

III. DESIGN FLOW CONCEPTUALIZATION

Our proposed classification system and set of mapping rules, supported by formally analyzable application models, enable a correct-by-construction approach leading to a feasible virtual implementation model, as illustrated in Fig. 1. It is especially relevant mainly for safety-critical systems, where correctness is fundamental.

![Design flow conceptualization illustrating the classification and mapping](image.png)

Figure 1: Design flow conceptualization illustrating the classification and mapping. The classification of both (i) application model elements and (ii) virtual platform model functional blocks serve as inputs to the mapping rules, thus resulting in a possible and feasible virtual implementation model.

Here, we suggest a design flow starting from the application model (Section III-A) and the virtual platform model (Section III-B), followed by an application of the mapping rules (Section III-C) to yield the virtual implementation model as the result.

We envision our proposal as paving the way towards a correct-by-construction system design flow [21], where each design decision is accounted for and can be justified by the need to both satisfy the required application properties and to meet the design criteria. This motivates founding our approach on formal models of computation which grant
unambiguous semantics for an application and systematizing the classification systems and mapping rules to have a virtual platform.

The system design starts with an executable specification, based on a sound formal model of computation. The application model $\Lambda$ formally specifies the system under development. This model is described as a network of concurrent processes communicating through signals on a MoC-based framework. In turn, this framework describes runtime reconfigurable behavior in an abstract way, hiding specific details about the implementing technology from the designer perspective. Therefore, the designer can focus on developing a functional executable specification without restrictions imposed by technology. Fig. 2 illustrates this concept.

In this sense, a control actor is responsible for sending functions to a regular actor. In turn, this actor processes inputs into outputs by using the received function.

The virtual platform model $P$ presents itself as an abstraction of a heterogeneous architecture. It captures the main concepts of a wide range of physical hardware platforms and configurations into functional basic blocks and hides device-specific information. This can be thought of as a template that can be implemented on a variety of physical technologies.

The application and platform models provide information about the functional aspects of the system being designed. This is possible due to the orthogonalization of design concerns this approach relies on. The application and platform information is combined by a set of well-defined rules that takes into account the functional aspects of the application together with the functional aspects of the platform resulting in a virtual implementation model $I$.

The modeling language uses a small yet powerful set of modeling elements, which have a correspondence in the implementation domain. This enables the mapping rules.

The ontologies are used to restrict both the application domain and the implementation domain to the set of necessary components, so that efficient mapping rules can be derived.

A. APPLICATION MODEL

A model is considered as an abstraction of an element that can be a physical system or even another model. Abstraction comprises a way of choosing which level of information or aspects to consider when modeling a given system [22].

1) Application-Domain Ontology

Classification and composition rules along with semantic information (as depicted in Fig. 1) contribute to a clear and unambiguous understanding of a model. In view of this, we developed an application-domain ontology used to attribute a classification to each element of our application model and to define how the elements are allowed to be composed. The semantics of the application model $\Lambda$ is finally given by the MoC used to describe the application, according to the characteristics of the application, e.g. execution, concurrency and synchronization (detailed in Section III-A2).

As illustrated in Fig. 3, the application-domain ontology has three higher abstract entities: procedure, path, and value.

The higher abstract entity procedure performs a computation that can be either a controller procedure or an executor procedure. The computation performed by the controller procedure is about deciding and informing a variable executor procedure which function (here we consider a function as common data, allowing them to be exchanged between procedures and stored in memory) it has to use to perform its computation. On the other hand, a fixed executor procedure has its unchangeable function defined at design time.

Another higher abstract entity is the path. Paths serve as unidirectional channels between procedures, so that they can communicate with each other and exchange entities from the value class. There is also the control path to enable the transport of functions from a controller procedure to a variable executor procedure. And finally, there is the data path, used to transport info values. Data paths are classified as either homogeneous data path or hybrid data path. Homogeneous data paths are used as a communication channel involving two entities executor procedure (e.g. variable or fixed), and the hybrid data path is used when an executor procedure communicates to a control procedure.

The relation “is part of” means that the higher abstract entities procedure, path, and value, together compose the application. The relation “is a” stands for an entity generalization as it moves upwards in the ontology hierarchy.

The realizable entities (leaves in the tree representation, Fig. 3) in the application-domain ontology are the controller procedure, fixed executor procedure, variable executor procedure, control path, homogeneous data path, hybrid data path, info value and function value. These entities are interpreted as classes, as described in Section III-A5.
2) Functional Modeling

The application model $\Lambda$ is structured as a network of concurrent processes communicating through signals, having their formal foundation in the theory of models of computation.

Models of computation (MoC) are classes of behaviors dictating the semantics of execution, concurrency, and synchronization in heterogeneous cyber-physical systems. They are founded in the tagged signal model (TSM), which is a denotational framework introduced in [1], [23], as a common meta-model for describing properties of concurrent systems as sets of possible behaviors. From a structural point of view, concurrent systems are defined as networks of processes communicating through signals.

In this context, signals are ordered sets of events characterized by a tag system, which describes the causality between events and can model time, precedence relationships, synchronization points, and other key properties. Events are the elementary information units exchanged among processes and are composed of a tag $t \in T$ and a value $v \in V$.

Processes can be regarded as sets of possible behaviors over signals or relations between multiple signals. In other words, processes are basic structures that encapsulate computation and execute with the semantics dictated by a specified tag system, i.e., a MoC. Processes are allowed to have an internal state. When the behavior of a process is determinate and can be uniquely defined by a function from inputs to outputs, such a process is said to be functional. Models possessing only functional processes are named functional models.

A suitable formal framework to host our approach is ForSyDe (Formal System Design). It is a transformational design methodology based on MoCs and the functional programming paradigm (FPP). ForSyDe targets the modeling of heterogeneous embedded systems by providing a common framework for the use of specific MoCs for different design goals [24].

A key idea of ForSyDe is to use processes constructors to create processes. Exploiting the FPP, ForSyDe defines process constructors as higher-order functions that take combinational functions and values as arguments to produce processes. The process constructors implement the semantics of the specific MoC, thus enabling the orthogonalization between communication and computation [24]. In this way, the behavior of a system given by a specific MoC can be captured using a finite set of process constructors. ForSyDe defines processes from a functional programming perspective as functions over the (history of) signals.

Signals, in ForSyDe, are the only means of communication and synchronization between processes, i.e., processes communicate to each other by writing to and by reading from signals.

a: Synchronous Dataflow

The synchronous dataflow (SDF) MoC, introduced by Lee and Messerschmitt [25], is a paradigm for untimed computation in which execution is dictated by a partial order between events. An SDF application can be expressed as a directed graph, where vertices are called actors and represents the computations or functions to be executed, and edges are directed signals which represent data channels. The partial ordering between events is satisfied by firing the actors according to fixed production and consumption rates. In other words, vertices execute whenever there is sufficient data in the input signals according to the consumption rates, and yield a number of data tokens at the outputs according to the production rates. In this case, an actor without input signals can fire at any time. Similarly, it is also possible to have sink-node actors, i.e., without any output.

Fig. 4 illustrates an SDF graph actor named $\text{actor}_\nu$. In our application model concept, the index $\nu = \{1, 2, \ldots, V\}$.
stands for the actor number, and $\mathcal{V} \in \mathbb{N}^+ = \{1, 2, 3, \ldots\}$ is the number of existing vertices/actors in the graph. The actor itself represents a computation.

Let’s assume that it is possible to change the actor’s computation in runtime. In this case, the actor is a function placeholder, that is, an actor that has defined but not fixed behavior over time, as detailed in Section III-A4. We base our assumption on the fact that the partial ordering constraints between events dictated by an SDF graph can be satisfied regardless of the functions executed by its actors.

![Figure 4: Synchronous dataflow actor and directed signals.](image)

Still, in Fig. 4, there are two explicit incoming signals and another two explicit outgoing signals as in an SDF actor. Each actor $\nu$ can have $A \in \mathbb{N} = \{0, 1, 2, \ldots\}$ incoming signals, and $B \in \mathbb{N}^+$ outgoing signals. It means that each actor can receive data from different inputs and produces data into different outputs. When fired, i.e., executing the function it holds, the actor $\nu$ consumes $c_{\nu,\alpha}$ samples from incoming signal $i_{\nu,\alpha}$ where $\alpha \in \{1, 2, \ldots, A\}$, and produces $p_{\nu,\beta}$ samples into the outgoing signal $o_{\nu,\beta}$ where $\beta \in \{1, 2, \ldots, B\}$. The parameters $c_{\nu,\alpha}$ and $p_{\nu,\beta}$ are the consumer and producer rates, respectively. It is also possible to have sink-node actors, i.e., without any output.

b: Scenario-Aware Dataflow

The scenario-aware dataflow (SADF) MoC is an extension of SDF to address dynamic behavior [26]. SADF has two types of actors: kernels and detectors. Kernels are data processing actors whose configuration, i.e., production and consumption rates and function, is defined by input control tokens. Every time a kernel fires, it consumes a single control token from the control port, setting the configuration of the firing. Then, the firing proceeds in the same manner as in an SDF actor. Detectors are the actors responsible for producing control tokens for all kernels connected to them via control channels. The functional behavior of a detector is dictated by a finite state machine (FSM) and every state of the FSM represents a detector’s scenario [27]. The token consumption rates of the detector are constant, however, the token production is a function of the current state. The FSM behavior of the detector ensures that the model is analyzable for properties such as consistency and absence of deadlocks.

c: Synchronous

The synchronous (SY) model, from the timed family of MoCs, is characterized by the perfect synchrony hypothesis in which neither computation nor communication takes time. The synchronous MoC splits the time domain into abstract time slots where the evaluation cycle of processes occur.

Two signals are synchronous if all events in one signal are synchronous to the events in the other signal and vice-versa [1].

Following this hypothesis, the timing behavior of the model is simply defined by the arriving of input events considering the system handles input samples in zero time and waits until the next input event arrives [28].

3) Runtime Reconfiguration

An adaptive system can actively change its configuration based on an internal state or as a response to changes on the external environment. Adaptiveness can be enabled by runtime reconfiguration. For a specific application and a specific time frame, the reconfigurable devices’ spatial structure is changed to comply with a given objective [29], [30]. Reconfigurable computing aims at both high performance and flexibility [31].

We distinguish between two possible reconfiguration techniques: full and partial. A device supports full reconfiguration when it considers its whole reconfigurable area as just one monolithic block. In that case, the hardware configuration can be changed at the cost of stopping the execution of the whole reconfigurable area. Partial reconfiguration, on the other hand, allows the reconfigurable area to be divided into one or more independent partitions or slots that can be reconfigured, i.e., changed, without affecting each other.

Runtime reconfiguration is the possibility to change a reconfigurable device’s functionality while that device has already started executing and without stopping it.

4) Generalization of the Runtime Reconfiguration Concept

We formulate the requirements for a MoC to be able to formally describe runtime reconfigurable behaviors. Our application model $\Lambda$ is supported by this generalization of the runtime reconfiguration concept.

The basic principle is the use of functions as events [32]. Once a model of computation or a language supports first-class functions, and consequently, higher-order functions, modeling adaptivity by defining runtime reconfigurable processes which operate on signals carrying functions, becomes transparent. In other words, the key to expressing runtime reconfigurability at the highest level of abstraction in a transparent manner is the capability of the modeling language (or MoC) to support first-class functions, i.e., functions that can be treated as variables.

In this sense we define the concepts of meta-behavior and function placeholder as follows.

**Definition 1 (Meta-behavior)** A meta-behavior is a process’ functional behavior that is defined by a function application function, generically defined by the apply function as follows:

$$\text{apply} : (V^m \rightarrow V^n) \times V^m \rightarrow V^n$$

$$\text{apply}(f, x_1, \ldots, x_m) = f(x_1, \ldots, x_m)$$

(1)
The *apply* function is a higher-order function that takes a function and a number of arguments and, as a result, applies the function to the remaining arguments. Each argument \( x_i \) can belong to a different \( V_i \subseteq V \).

**Definition 2 (Function Placeholder - FPH)** A function placeholder is any functional process possessing a meta-behavior as its defined behavior. As an implication, one of its input ports receives functions, which are applied to the rest of the input events to compute the outputs. A functional process is a computing element such as an actor:

\[
\Phi_{\nu} = FPH_{\nu \in \{i_1, \ldots, i_{\alpha}\}} \circ \Phi_{\nu} = P\Phi_{\nu}
\]

![Figure 5: Function placeholder notation.](Image)

The concept of function placeholder is exhibited in Fig. 5. The process \( \Phi_{\nu} \) receives functions in the input signal \( sf_{\nu} \) to apply them to its remainder inputs and then generates the outputs, as in (2) and (3).

\[
(\alpha_1, \ldots, \alpha_\beta) = \Phi_{\nu}(sf_{\nu}, (i_1, \ldots, i_\alpha)) \tag{2}
\]

\[
(o_1[\delta], \ldots, o_\beta[\delta]) = apply(sf_{\nu}[\delta], i_1[\delta], \ldots, i_\alpha[\delta]) \tag{3}
\]

For synchronous timed models (e.g., SY MoC), the variable \( \delta \) represents the clock tick \( k \), while inputs \( i_n \) where \( n \in \{1, \ldots, \alpha\} \), and outputs \( o_m \) where \( m \in \{1, \ldots, \beta\} \), are scalar events.

For dataflow untimed models (e.g., SDF and SADF), the variable \( \delta \) represents the actor firing \( k_\nu \), while inputs \( i_n \) where \( n \in \{1, \ldots, \alpha\} \), and outputs \( o_m \) where \( m \in \{1, \ldots, \beta\} \), are vectors of events with size given by the token rates.

Given this, the application model elements of the class Procedure.Executor.Variable follows the FPH definition (Definition 2), independent of the model of computation used for the modeling.

5) Classification System

The application-domain ontology introduced in Section III-A1 is used to derive and build the classification system related to the application part, as depicted in Fig. 1. Considering the semantics of the application model \( \Lambda \) and the concepts of the application-domain ontology, Table 1 shows the application model elements and their corresponding classifications. Notice that the application model elements, showed in the left column of Table 1, are always the same, independent of the MoC used.

The mapping rules step uses this resulting classification system, as presented, for the application side. In other words, once one designs a given system supported by the MoC-based application model \( \Lambda \), the model element class is given by our defined classification as is in Table 1.

Table 1: Application model elements and corresponding classes.

| # | Application Element | Model Element | Class |
|---|---------------------|---------------|-------|
| 1 | Control actor       | Procedure.Controller |       |
| 2 | Actor receiving a signal containing function values | Procedure.Executor.Variable |       |
| 3 | Actor ONLY receiving signals containing info values | Procedure.Executor.Fixed |       |
| 4 | Directed signals from a control actor to an actor(*) | Path.Control |       |
| 5 | Directed signals from an actor to an actor | Path.Data.Homogeneous |       |
| 6 | Directed signals from an actor to a control actor | Path.Data.Hybrid |       |
| 7 | Regular data values carried through data paths | Value.Info |       |
| 8 | Functions \((f_1, \nu)\) carried through control paths | Value.Function |       |

(*) #4: In other words, directed control signals from a control actor to a regular actor.

**B. VIRTUAL PLATFORM MODEL**

As a scalable virtual platform model \( P \), we consider the use of the runtime reconfigurable (RTR) approach when abstracting architectural details through functional blocks. The scalability aspect is related to the platform’s ability to receive different functionalities over time enabled by both hardware and software reconfiguration schemes.

1) Platform-Domain Ontology

In the same sense of the application-ontology, we developed a platform-domain ontology used to attribute classification and composition directions to each functional block of our virtual platform model \( P \), as shown in Fig. 1.

The platform-domain ontology has three higher abstract entities: static, adaptable and pool, as Fig. 6 illustrates.

The entity *static* means fixed underlying hardware architecture in which the software needs to adapt to the hardware. On the other hand, the entity *adaptable* stands for runtime reconfigurable circuitry where an algorithm can have an optimized hardware architecture to run on in terms of performance and power consumption. In this case, different basic logic gates are combined to form a circuit, i.e. reconfigurable hardware.

The entity *pool* represents a memory pool where dissimilar and ready-to-load and execute software-based and hardware-based configurations are stored in. Configurations, can be loaded and selected at runtime.

Notice that *static* and *adaptable* have almost the same entities relating to them. When an entity is adaptable it has the suffix \( \chi \), e.g. interface\( \chi \). The \( \chi \) indicates that an entity also supports hardware runtime reconfiguration, both full and partial. This gives flexibility to the proposed ontology, i.e. it is possible to have different combinations on the number of instances from these two entities static and adaptable when describing a platform.

The entity *interface* is divided into data interface and control interface. Interfaces are in charge of sending and
As in the application-domain ontology, the *realizable entities* in the platform-domain ontology are the leaves in the tree representation (Fig. 6), namely the data interface static, control interface static, performer static, manager control unit static, dispatcher control unit static, data interface adaptable, control interface adaptable, once performer adaptable, mode performer adaptable, function performer adaptable, manager control unit adaptable, dispatcher control unit adaptable, and configuration pool.

These entities are interpreted as *classes*, and are used to build a classification system (Section III-B3), in the same way as the application-domain ontology.

2) Virtual Platform Functional Blocks

The virtual platform architecture model is build up from basic functional blocks: *worker, steward, and handler*. These functional blocks can be attached to a *programmable device* (ProgDev), and a *reconfigurable device* (ReconDev). ProgDev stands for fixed hardware architecture devices, such as the central processing unit (CPU), and the graphics processing unit (GPU). On the other hand, ReconDev represents the runtime reconfigurable devices, such as modern field-programmable gate array (FPGA) chips, supporting full and partial reconfiguration.
The introduced platform-domain ontology allows for various possible combinations to describe a virtual platform. It is possible to have a platform with functional blocks following definitions from:

1) just entity static;
2) just entity variable; and
3) both entities static and variable, i.e. heterogeneous.

The functional block worker (single-line box in the following figures) represents a block doing a software-based, when attached to ProgDev, or even a hardware-based computation, when attached to ReconDev. The functional block handler (double-line box in the figures) is responsible for deciding which function configuration, from a configuration repository, a functional block worker has to perform.

a: Fixed Computation Cases

Fig. 7a describes a virtual platform architecture model with two workers sharing data through a bus. In this case, they are attached to a programmable device, meaning their computations are fixed. On the other hand, Fig. 7b describes another possibility where workers are attached to a reconfigurable device executing a configuration assigned to them in design time and loaded into them once by full reconfiguration. In this case, they will not change in runtime, since there is no functional block such as the handler or steward in the scenario.

![Platform's functional blocks attached to a ProgDev with fixed computation workers.](image)

![Platform's functional blocks attached to a ReconDev with fixed (loaded during runtime) computation since they do not have handlers or stewards related to them.](image)

Figure 7: Platform’s functional blocks illustrating two workers communication through a data bus, in a fixed computation case.

The underscore, i.e. index $i \in 1 \ldots n$, in the worker indicates that a device supports $n$ different functional blocks, including worker, handler and steward. The steward is represented with a dotted-line box in the following figures.

b: Variable Computation Cases: Mode Reconfiguration

As illustrated in Figs. 8a and 8b, the block handler decides which function configuration the block worker has to perform. This decision relies on a user-defined algorithm which can be a static schedule. By the presence of a handler and a steward, this is a variable computation scenario.

Figs. 8a and 8b also depict the case where the handler commands the functional block steward so that it can point to the address of a function to a worker execute it. In this way, the steward controls the worker functions selection through a control bus. This approach is known as mode reconfiguration [33], detailed in Fig. 8c, where the functions are selected via $sf_1$.

Considering a variable computation case based on mode reconfiguration, all possible functions, software- or hardware-based configurations, must be initially loaded into the device’s memory, either a programmable device or reconfigurable device. Finally, just one function must be chosen at a time from $1 \ldots i$ possibilities using the signal $sf_1$ to switch to a function (i.e. configuration) and select its output (Fig. 8c). Note that worker and handler can communicate with each other via a data bus.

In Fig. 8c, the switch connects the input interfaces $i_1, i_2 \ldots i_{1,n}$ to the switched function (i.e. configuration), and the select connects the selected function output to the output interface $o_1$. Mode reconfiguration is possible in both programmable device and reconfigurable device, as illustrated.

c: Variable Computation Cases: Function Reconfiguration

Up to this point, we took into consideration homogeneous platform architecture models, either following only entity static or only entity adaptable.

Now, Fig. 9a illustrates a heterogeneous model where the worker is attached to a reconfigurable device. In this particular case, the steward reconfigures the worker with a new function (i.e. new configuration) instead of just pointing out the function address. This is known as function reconfiguration scheme (Fig. 9c). Here, just one function, i.e. just one configuration, is fetched at runtime from the configuration repository, and loaded using partial reconfiguration into a worker to be executed.

Fig. 9b describes the function reconfiguration scheme possibility with both the handler and steward attached to a reconfigurable device, besides the worker itself. In this case, it is again a homogeneous platform architecture model, however following the variable computation case based on function reconfiguration.

In Fig. 9c, a configuration is fetched from the pool based on the signal $sf_1$. The input interfaces $i_1, i_2 \ldots i_{1,n}$, and the the output interface $o_1$ are initially connected to the worker, as it figures as a placeholder, according to Definition 2.

3) Classification System

The platform-domain ontology introduced in Section III-B1 is used to derive and build the classification system related to the platform part, as also depicted in Fig. 1. Considering the proposed meaning of the virtual platform model $P$ and the concepts of the platform-domain ontology, Table 2 shows the virtual platform model’s functional blocks along...
(a) Platform’s functional blocks attached to a programmable device, based on mode reconfiguration (switch/select scheme). This option can be implemented on both ProgDev or ReconDev, here is shown the ProgDev with a software-based implementation.

(b) Platform’s functional blocks attached to a reconfigurable device, based on mode reconfiguration (switch/select scheme). This option can be implemented on both ProgDev or ReconDev, here is shown the ReconDev with a hardware-based implementation.

(c) Mode reconfiguration based on the switch/select scheme.

Figure 8: Mode reconfiguration considering the configurations repository to indicate the address of the desired configuration so that the worker is able to execute it.

with their corresponding classifications. It follows the same approach as in the application model classification system (Section III-A5).

The mapping rules step uses this resulting classification system, as presented, for the platform side. In other words, once one designs a given system supported by the virtual platform model $P$, the functional block class is given by our defined classification as is in Table 2.

(b) Platform’s functional blocks attached to a reconfigurable device, based on function reconfiguration (partial reconfiguration scheme). This is a homogeneous implementation option where all functional blocks are attached to ReconDev.

(c) Function reconfiguration scheme exploring runtime partial hardware reconfiguration.

Figure 9: Function reconfiguration considering the configurations repository to indicate the desired configuration so that the worker is loaded with and executes it. This is possible by the hardware partial reconfiguration technique present in modern FPGA-based devices.

C. MAPPING RULES

The general mapping rule $m$ is given by (4), where one class of elements in the application model $\Lambda$ (summarized in Table 1) maps to one class of functional blocks in the virtual platform model $P$ (summarized in Table 2). Thus, the mapping result is equivalent to the virtual implementation
model $I$, i.e. a feasible MoC-based high-level implementation.

$$m : (\Lambda \rightarrow P) \equiv I \quad (4)$$

As previously mentioned, we use ontologies to restrict both the application domain and the implementation domain to the set of needed components. Given this, we derive correct and efficient mapping rules $m_{i=1...7}$. Note that our mapping rules are general enough to be applied to any computer system since we consider its minimum basic parts, namely processing, communication, and storage. It follows the von Neumann machine [34] in terms of minimum computer parts but advances in software and hardware architectures as it considers runtime reconfiguration on its central characteristics.

The defined rules are stated in Table 3. The column label “$\rightarrow$" denotes the constraint used to define the rule application.

Notice that some rules give more than one option to map the application model element classes to the functional block classes in the virtual platform model. As mentioned before, our platform-domain ontology allows for different possible combinations to describe a virtual platform following definitions from just entity static, just entity variable, or both entities forming a heterogeneous virtual platform. In this sense, the mapping rules follow the constraint shown in (5).

$$m_{i=1...6} \text{ maps } \Lambda \text{ to } \begin{cases} \text{Entity Static } S, & \text{when targeting only programmable device} \\ \text{Entity Adaptable } A, & \text{when targeting only reconfigurable device} \\ \text{Mixed } M, & \text{when targeting heterogeneous device} \end{cases} \quad (5)$$

Another constraint is defined when targeting a heterogeneous device, depending on the function’s configuration, which can be software- or hardware-based, as described in (6). When targeting a reconfigurable or heterogeneous device, we assume the function configuration will be hardware-based, as it represents the most customized solution in terms of power consumption and performance.

$$m_{i=7} \text{ maps } \Lambda \text{ to } \begin{cases} \text{software-based configuration,} \\ \text{when } m_{i=1...6} \text{ maps } \Lambda \text{ to } S \end{cases} \quad \begin{cases} \text{hardware-based configuration,} \\ \text{when } m_{i=1...6} \text{ maps } \Lambda \text{ to } (A \lor M) \end{cases} \quad (6)$$

According to (6), when a software-based configuration is required, rules 1 to 6 states the mapping to the entity static $S$. Conversely, when hardware-based is the requirement, rules 1 to 6 states the mapping to either entity adaptable $A$ or a mixed case $M$. This case is also present in Table 3.

There is a special case for the class Path when considering the use of heterogeneous devices. This general path precedence rule supersedes other path rules as described next.

a: General Path Precedence Rule

When a class either regarding the Path.Control or the Path.Data.Homogeneous/Hybrid connects two application model element classes that in turn will be mapped to two distinct platform model functional block classes, e.g. Static.Interface and Adaptable.InterfaceX, the class Path will be mapped partially to the class Static.Interface and also partially to Adaptable.InterfaceX.

This is the case illustrated in Fig. 9a, where there is a “data bus” interconnecting the handler and the worker, and a “control bus” interconnecting the steward and the worker.

Next, the rules are described.

1) Procedure.Controller Rule

The mapping of elements from the class controller to the either the class manager or managerX also creates a new functional block in the virtual platform model from the classes dispatcher or dispatcherX, respectively.

The decision between entities static $S$ or adaptable $A$ follows the constraint as defined in (5). In other
words, the class **Procedure.Controller** will be mapped to **Static.ControlUnit.Manager** and will create another functional block from class **Static.ControlUnit.Dispatcher** when having entity static S or heterogeneous devices in place (i.e. mixed M).

2) **Procedure.Executor.Variable Rule**
The mapping of elements from class variable to classes performer or mode or function. These three options follow the constraint as defined in (5). On the other hand, the decision between mode or function follows the constraint (6).

3) **Procedure.Executor.Fixed Rule**
The mapping of elements from class fixed to classes performer or once follows (5), which depends on targeting only programmable, only reconfigurable or heterogeneous devices. In other words, the **Procedure.Executor.Fixed** class will be mapped to **Adaptable.PerformerX.Once** when having entity adaptable A or heterogeneous devices in place (i.e. M).

4) **Path.Control Rule**
The mapping of elements from class control to either class InterfaceX.ControlX or Interface.Control follows (5).

5) **Path.Data.Homogeneous Rule**
The mapping of elements from class homogeneous to either class Interface.DataX or InterfaceX.Data follows (5).

6) **Path.Data.Hybrid Rule**
The mapping of elements from class hybrid to either class Interface.Data or InterfaceX.DataX follows (5).

7) **Value.Function Rule**
Functions are mapped to software- or hardware-based configurations, depending on the constraint given by (6).

### IV. ILLUSTRATIVE EXAMPLES
Next, we introduce two examples to demonstrate the applicability of our proposal. The first (Section IV-A) is an encoder and decoder system following a tutorial style. The second example (Section IV-B) is an avionics attitude estimation system. That one shows a complete system modeling and how our approach manages both complexity and scalability through abstraction.

#### A. ENCODER AND DECODER SYSTEM
We adapted and implemented a model introduced in [32] as a representative and qualitative tutorial example to demonstrate the applicability of the proposed classification system and mapping rules in our design flow.

This model presents components such as cipher and decipher which can perform arithmetic operations. Fig. 10 illustrates our runtime reconfigurable embedded application model AExample based on the synchronous MoC.

The application model example shows two runtime reconfigurable processes based on the function placeholder definition (Section III-A4): cipher and decipher.

The signal \( s_{input}[k] \) is encoded with an encoding function \( s_{encF}[k] \) given by \( f_{j}[k](s_{key}[k]) \), which is the cipherGen process output signal. Following this hierarchical processes network, the encoded signal \( s_{enc}[k] \) is taken by the decipher process together with the decoding function \( s_{decF}[k] \) given in the signal \( f_{j-1}[k](s_{key}[k]) \). Finally, the decipher produces the decoded signal \( s_{output}[k] \). The input signal \( s_{input}[k] \) must be equal to the output signal \( s_{output}[k] \).

---

**Table 3: Mapping rules summary.**

| i # | Application Model Element’s Class          | \( m_{binary} \) as per (5) | Virtual Platform Model Functional Block’s Class          |
|-----|-------------------------------------------|------------------------------|----------------------------------------------------------|
| 1   | Procedure.Controller                      | \( S \lor M \)               | Static.ControlUnit.Manager and creates a Static.ControlUnit.Dispatcher |
|     |                                           |                              | Adaptable.ControlunitX.ManagerX and creates an Adaptable.ControlunitX.DispatcherX |
| 2   | Procedure.Executor.Variable               | S                            | Static.Performer                                         |
|     |                                           | A                            | Adaptable.PerformerX.Mode                                |
|     |                                           | M                            | Adaptable.PerformerX.Function                            |
| 3   | Procedure.Executor.Fixed                  | S                            | Static.Performer                                         |
|     |                                           | (A \lor M)                   | Adaptable.PerformerX.Once                                |
| 4   | Path.Control                              | S                            | Static.Interface.Control                                 |
|     |                                           | (A \lor M)                   | Adaptable.InterfaceX.ControlX                           |
| 5   | Path.Data.Homogeneous                     | S                            | Static.Interface.Data                                    |
|     |                                           | (A \lor M)                   | Adaptable.InterfaceX.DataX                              |
| 6   | Path.Data.Hybrid                          | S                            | Static.Interface.Data                                    |
|     |                                           | (A \lor M)                   | Adaptable.InterfaceX.DataX                              |
| 7   | Value.Function                            | Pool.Configuration, as defined in (6) |                                                          |
The signal $s_{key}$ comprises the keys used in the encoding and decoding processes. Signal $s_f$ contains the encoding functions $f$, while $s_f^{-1}$ contains the decoding functions $f^{-1}$, which stands for the inverse of encoding function.

The signal $s_{encF}$ comprises the encoding functions along with the respective keys. Signal $s_{input}$ is a plain-text to be encoded. The signal $s_{enc}$ comprises the encoded data, and the signal $s_{output}$ the decoded data. (7) to (10) define the processes in this example model.

$$cipherGen : s_{encF}[k] = s_f[k](s_{key}[k])$$  \hspace{1cm} (7)
$$cipher : s_{enc}[k] = apply(s_{encF}[k], s_{input}[k])$$  \hspace{1cm} (8)
$$decipherGen : s_{decF}[k] = s_{f}^{-1}[k](s_{key}[k])$$  \hspace{1cm} (9)
$$decipher : s_{output}[k] = apply(s_{decF}[k], s_{enc}[k])$$  \hspace{1cm} (10)

Both cipher and decipher are runtime reconfigurable processes, class Procedure.Executor.Variable according to the application-domain ontology (Table 1), since they receive a function that truly changes their behavior, i.e., the data processing function changes over time $k$.

A runtime reconfigurable process may have different behavior according to the function it is currently executing. Notice that cipherGen and decipherGen could also be considered as runtime reconfigurable processes since they combine regular data input with a function to create a new function as output. However, in this case, their behavior is not changed. Therefore, cipherGen and decipherGen belong to the class Procedure.Controller because they are indeed sending functions to another actors.

The application model functional code implemented in Haskell/ForSyDe using the synchronous MoC is shown in Listing 1. This is also available in [35]. Listing 1 exposes the annotated classes of the model elements according to application-domain ontology as in Table 1.

**Listing 1: Synchronous MoC Haskell/ForSyDe code.**

```
module ApplicationModel where
import ForSyDe.Shallow

-- ::class Path.Data.Hybrid (Application-domain Ontology)
```

--- regular data input signal arbitrary definitions
s_key = signal [1, 4, 6, 1, 1]
--- ::class Path.Data.Homogeneous (Application-domain Ontology)
--- regular data input signal arbitrary definitions
s_input = signal [256, 512, 1024, 2048, -512]
--- ::class Value.Function (Application-domain Ontology)
--- functions definition
fsub x y = y - x
fadd x y = x + y
--- ::class Path.Data.Hybrid (Application-domain Ontology)
--- function signals arbitrary definition (ie static scheduling)
f_f = signal [(fadd), (fsub), (fadd), (fadd)]
f_f_inv = signal [(fsub), (fadd), (fsub), (fadd)]
--- function placeholder (FPH) implementation for synchronous (SY) MoC
--- runtime reconfigurable process constructor definition
apply = ($) 

```
fhbSY = comb2SY apply
```

--- ::class Procedure.Controller (Application-domain Ontology)
--- control processes/vertex definition
cipherGen s_f s_key = comb2SY ($) s_f s_key
decipherGen s_f_inv s_key = comb2SY ($) s_f_inv s_key
--- ::class Procedure.Executor.Variable (Application-domain Ontology)
--- reconfigurable processes/vertex definitions following
--- function placeholder (FPH) definition
cipher s_encF s_input = fhbSY s_encF s_input
decipher s_decF s_enc = fhbSY s_decF s_enc
--- hierarchical process network definition
lambdaExample s_key s_input = (s_enc, s_output)
where -- s_encF :: Path.Data.Homogeneous
a_encF = cipherGen s_f s_key
-- s_decF :: Path/Data.Homogeneous
a_decF = decipherGen s_f_inv s_key
-- a_enc :: Path.Data.Homogeneous
a_enc = cipher s_encF a_input
-- a_output :: Path.Data.Homogeneous
a_output = decipher s_decF a_enc
--- use the following for testing this application model
--- illustrative example in GHC
```

```
(257, 508, 1030, 2047, -511), (256, 512, 1024, 2048, -512)
```

---
The synchronous function placeholder $f_{\text{phS}}$ is defined with the comb2SY ForSyDe process constructor using, as behavior, the single argument function application function $\$ \$ from Haskell, which is equivalent to the apply function in (1) with $m = n = 1$.

The runtime reconfigurable processes (class Procedure.Executor.Variable) $\text{cipher}$ and $\text{decipher}$ receive the reconfiguration signals $s_{\text{encF}}$ and $s_{\text{decF}}$ (class Path.Control). In this example, the reconfiguration decision logic is based on a static schedule policy, i.e., user-defined algorithm to give the sequence, namely, $\text{fadd}$, $\text{fsub}$, $\text{fadd}$, $\text{fsub}$, $\text{fadd}$ (class Value.Function).

Notice also that the elements $\text{cipher}$ and $\text{decipher}$ from Listing 1 [35] follows the function placeholder’s definition.

1) Classes of Application Model Elements
Table 4 shows the element classes of the application model example, as shown in Fig. 10, following the application-domain ontology.

Table 4: Application model example: elements and classes.

| Application Model Element | Class                  |
|---------------------------|------------------------|
| $\text{cipherGen}$, $\text{decipherGen}$ | Procedure.Controller   |
| $\text{cipher}$, $\text{decipher}$ | Procedure.Executor.Variable |
| $s_{\text{encF}}$, $s_{\text{decF}}$ | Path.Control           |
| $s_{\text{input}}$, $s_{\text{enc}}$, $s_{\text{output}}$ | Path.Data.Homogeneous |
| $s_{f}$, $s_{\text{key}}$, $s_{f-1}$ | Path.Data.Hybrid      |
| $\text{fadd}$, $\text{fsub}$ | Value.Function         |
| regular data (i.e. numbers) | Value.Info             |

2) Utilization of the Mapping Rules
After modeling the application based on the synchronous MoC, we apply the defined mapping rules to obtain a possible and feasible virtual implementation model.

There exist two constraints (5) and (6) that must be specified in order to apply the rules. To show a comprehensive and representative case, we set the first constraint to use a heterogeneous device. In this case, the second constraint, functions implementations must be hardware-based.

Table 5 shows the mapping rules summary according to the constraints as in (11) and (12).

\[ m_{i=\{1...6\}} \text{ maps } \Lambda_{\text{Example}} \text{ to } \text{Mixed}_M \]  \hspace{1cm} (11)

\[ m_{i=7} \text{ maps } \Lambda_{\text{Example}} \text{ to hardware-based implementation} \]  \hspace{1cm} (12)

3) Model Elements and Functional Blocks
According to the application of the mapping rules, Table 6 shows the correspondence from the elements of the application model example to the functional blocks in the resulting virtual implementation model example $I_{\text{Example}}$.

Table 5: Mapping rules result, according to (11) and (12).

| Application Model Element | Implementation Model Functional Block |
|---------------------------|--------------------------------------|
| $\text{cipherGen}$, $\text{decipherGen}$ | Handler attached to ProgDev and also creates a Steward attached to ProgDev |
| $\text{cipher}$, $\text{decipher}$ | Worker attached to ReconDev (hardware-based implementation) |
| $s_{\text{encF}}$, $s_{\text{decF}}$ | Control Bus mixed attached to ProgDev and ReconDev |
| $s_{\text{input}}$, $s_{\text{enc}}$, $s_{\text{output}}$ | Data Bus attached to ReconDev |
| $s_{f}$, $s_{\text{key}}$, $s_{f-1}$ | Data Bus attached to ProgDev |
| $\text{fadd}$, $\text{fsub}$ | Function from Configuration Repository |

Fig. 11, and the virtual implementation model $I_{\text{Example}}$ is illustrated in Fig. 12.

Figure 11: Partitioned application model showing the separation into ProgDev and ReconDev, in the application model perspective.

The signal $s_{\text{encF}}$ (class Path.Control) is one example of the general path precedence rule stated in Section III-C. This signal connects two application model classes (Procedure.Controller and Procedure.Executor.Variable) which are being mapped to two different platform model functional block classes (Static.ControlUnit.Manager/Dispatcher and Adaptable.PerformerX.Function), as shown in
Table 5. In this specific case, the signal, i.e. path, is partially implemented in a programmable device and partially in the reconfigurable device, as illustrated in Figs. 11 and 12. The same applies to signal $s_{decF}$.

### B. AVIONICS ATTITUDE ESTIMATION SYSTEM

The integrated modular avionics (IMA) architecture became a de-facto standard in the aeronautical industry and it represents the state-of-the-art with respect to avionics platforms. It aims to host multiple avionics applications, considering different criticality levels, on the same platform instance [36]. The second generation of IMA (IMA-2G) is discussed in [37], where the authors present some future requirements and challenges for it, including reconfiguration capabilities. However, the reconfiguration addressed there is basically in terms of software. Our design can advance the concept by addressing also hardware reconfiguration possibilities (as in Section III-A4).

Singh et al. [38] assert that conventional testing methodologies for software can be expensive and ineffective. Thus, they use formal verification techniques such as model checking to verify embedded software inside the context of modern avionics systems. There is a guideline in the aeronautical industry for the use of formal methods, DO-333 “Formal Methods: DO-178C Supplement” [39]. As previously mentioned, our proposal starts with formal models of computation which leads to a correct-by-construction system design flow [21] for RTR embedded systems with both software and hardware.

Given this context, the second example we introduce is an avionics attitude estimation system (AAES) with triple modular redundancy (TMR) and hardware runtime reconfiguration. This one shows a complete system modeling and how the proposed approach manages both complexity and scalability through abstraction.

The AAES is responsible for getting data from three different inertial measurement units (IMU) to compute the airplane’s attitude (roll angle $\phi$, pitch angle $\theta$, yaw angle $\psi$) along with linear velocity and linear position.

The computations are performed in terms of quaternions to avoid the gimbal lock phenomenon (also known as Euler angle singularity), i.e. pitch angle near to $\pm 90$ degrees. After that, Euler angles are obtained. The aircraft equations of motion used in our application model are supported by [40].

The final estimated attitude, i.e. voted attitude, is a result of a voting process concerning redundant parts, i.e. angular rates, and linear accelerations measurements. Should any computation is considered wrong, that data is assumed as failed or invalid. Besides, a hardware reconfiguration can take place to reconfigure the faulty processor as a parallel mitigation action.

Here, the AAES is ultimately responsible for supplying the voted attitude, namely angular positions, linear velocity, and linear position to the flight control computer (FCC) system. FCC runs the aircraft control laws that actuate the flight control surfaces. According to DO-178 [41], a failure in FCC is considered catastrophic, namely the most critical level. As a consequence, AAES is also critical (i.e. catastrophic) since it provides input navigation data to FCC.

On the other hand, AAES can also feed the in-flight entertainment (IFE) system with navigation data to be shown in moving map passenger’s screen. In this case, an IFE failure is considered minor, according to DO-178.

Our AAES model providing data to both FCC and IFE systems is a typical IMA scenario. However, here we also introduce runtime partial hardware reconfiguration.

Next, we follow our design flow considering: 1) the AAES application modeling; 2) the obtention of the classes concerning AAES application model elements, as defined by the application-domain ontology; 3) the utilization of defined mapping rules; 4) the obtention of the correlation concerning the model elements and functional blocks, as defined by the platform-domain ontology; and 5) finally getting the resulting virtual implementation model.

1) AAES Application Model

Table 7 shows the signals used in the AAES application model.
Figure 13: AAES application model illustrating the signals and processes as a network of concurrent processes. This is the core functionality of the AAES. Here the \( \int \text{quat} \) is short for the quaternion integration, \( \text{quat2euler} \) stands for the transformation from quaternion to Euler angles, \( \text{ins} \) is short for transformation to the inertial frame, \( \int \text{Tustin} \) is the Tustin integration, and \( \text{aBodyZNz} \) is the computation for the load factor.

Table 7: AAES Application Model Signals and Descriptions.

| Signal            | Description                                                                 |
|-------------------|-----------------------------------------------------------------------------|
| \( \dot{\omega}_i \) | Angular rates in the axes \( x, y, z \) provided by gyroscopes within IMU \( i = \{0, 1, 2\} \) |
| \( \overline{\omega}_i \) | Tuple of \( \dot{\omega}_x, \dot{\omega}_y, \dot{\omega}_z \) from IMU number \( i \in \{0, 1, 2\} \) |
| \( a_{ix}, a_{iy}, a_{iz} \) | Linear acceleration in the axes \( x, y, z \) provided by accelerometers within IMU \( i = \{0, 1, 2\} \) |
| \( \overline{a}_i \) | Tuple of \( a_{ix}, a_{iy}, a_{iz} \) from IMU number \( i \in \{0, 1, 2\} \) |
| \( \overline{\omega}_v[k] \) | Tuple of voted \( (\dot{\omega}_{vx}, \dot{\omega}_{vy}, \dot{\omega}_{vz}) \), in the time instant \( k \), from one of the IMU number \( i \in \{0, 1, 2\} \) |
| \( \overline{a}_v[k] \) | Tuple of voted \( (a_{ix}, a_{iy}, a_{iz}) \), in the time instant \( k \), from one of the IMU number \( i \in \{0, 1, 2\} \) |
| \( q[k] \) | Estimated attitude quaternion \( q_0 + q_1 \hat{a} + q_2 \hat{b} + q_3 \hat{c} \) in the time instant \( k \), just after the quaternion integration of \( \overline{\omega}_v[k] \) |
| \( (\phi[k], \theta[k], \psi[k]) \) | Roll, pitch and yaw Euler angles after the transformation from quaternion to Euler in the time instant \( k \) |
| \( \overline{\alpha}[k] \) | Estimated linear acceleration in the time instant \( k \), considering the inertial frame |
| \( \overline{v}[k] \) | Estimated linear velocity in the time instant \( k \) just after Tustin (trapezoidal) integration of \( \overline{\alpha}[k] \) |
| \( \overline{p}[k] \) | Estimated linear position in the time instant \( k \) just after Tustin (trapezoidal) integration of \( \overline{v}[k] \) |
| \( Nz[k] \) | Load factor |

Fig. 13 depicts the core of the AAES application model by using the synchronous MoC. It considers the input of three different IMU to form a triple modular redundancy. IMU is a micro-electro-mechanical system (MEMS) composed basically of gyroscopes and accelerometers. The core of AAES presents two voters: one dedicated to gyroscopes’ data, as in (13); and the other to accelerometers’ data, as in (14). The voted angular rates (i.e. gyro scope data) enter the quaternion integration, as defined in (15), and then are next transformed to Euler angles, as in (19) to (21).

As illustrated in the bottom of Fig. 13, the voted linear accelerations (i.e. accelerometer data) and the estimated attitude quaternion enter the transformation to the inertial frame reference, as in (16). Finally, the estimated linear acceleration is integrated once resulting in the estimated linear velocity, and twice to produce the estimated linear position, as in (17) and (18), respectively. The load factor is computed as in (22).

\[
\begin{align*}
\overline{\omega}_v[k] &= \text{voter}(\overline{\omega}_0[k], \overline{\omega}_1[k], \overline{\omega}_2[k], \overline{\omega}_v[k-1]) \\
\overline{a}_v[k] &= \text{voter}(\overline{a}_0[k], \overline{a}_1[k], \overline{a}_2[k], \overline{a}_v[k-1]) \\
q[k] &= q[k-1] \exp(T/2 \overline{\omega}_v[k]) \\
\overline{\alpha}[k] &= -g \overline{z} + q[k] \overline{a}_v[k] \overline{q}[k] \\
\overline{v}[k] &= (\overline{\alpha}[k] + \overline{a}_v[k-1])T/2 + \overline{\alpha}[k-1] \\
\overline{p}[k] &= (\overline{v}[k] + \overline{v}[k-1])T/2 + \overline{v}[k-1] \\
\phi[k] &= \text{atan2}(2(q_xq_y - q_yq_z), 1 - 2(q_x^2 + q_y^2)) \\
\theta[k] &= \text{asinf}(2(q_xq_y + q_yq_z)) \\
\psi[k] &= \text{atan2}(2(q_xq_z + q_zq_x), 1 - 2(q_y^2 + q_z^2)) \\
Nz[k] &= \overline{z}^T \overline{a}_v[k]/g 
\end{align*}
\]
The function \( \text{voter}(x_0, x_1, x_2, x_1) \) should return:
- The average \( x_{m3} \) between vectors \( x_0, x_1 \) and \( x_2 \) if the distance between \( x_{m3} \) and each vector is no bigger than a chosen threshold; else
- The average \( x_{m1} \) between the two vectors that are closer to \( x_{m3} \) if the distance between these two vectors is no bigger than twice the threshold; else
- The average \( x_{m1} \) between \( x_1 \) and its closest vector.

Equations using both quaternions and vectors, such as (15) and (16) consider a vector as being a quaternion with zero real part. As such, the quaternion exponential of (15) is defined as in (23).

\[
\exp\left( \frac{T}{2} \tilde{\omega}^v \right) = \cos \left( \frac{T}{2} ||\tilde{\omega}^v|| \right) + \frac{\tilde{\omega}^v}{||\tilde{\omega}^v||} \sin \left( \frac{T}{2} ||\tilde{\omega}^v|| \right)
\]  

(23)

Note that equations (13) to (15), (17) and (18) are dependent on values of \( \tilde{\omega}^w, \tilde{a}_w, q, \tilde{d}_w, \tilde{\nu}^w \) and \( \tilde{p}^w \) from previous evaluation cycles, namely \( k-1 \). These variables are considered states of the system, and after their computation, they must be stored in a memory to be used in the next evaluation cycle. The AAES core must have a memory with enough size to support the storage of the states between different evaluation cycles.

The AAES core is formally described as an actor that receives functions in one control input port, that is, an actor with the runtime reconfiguration property. To model this, we consider the runtime reconfigurable processor (RTRP) model based on [42]. The RTRP is modeled as a function placeholder with configuration and data memories as shown in Fig. 14.

![Figure 14: Runtime reconfigurable processor model.](image)

Next, we detail the RTRP.

1) The RTRP behavior is dictated by a pair of functions \( f \) and \( g \), defining the synchronous discrete state machine

\[
x[k+1] = f(x[k], u[k])
\]  

(24)

\[
y[k] = g(x[k], u[k])
\]  

(25)

with \( x \) the state vector, \( u \) the input vector and \( y \) the output vector;

2) A control signal \( s_f[k] \) is used to input either a pair \((f, g)\), or an absent event represented by the symbol \( \bot \). Receiving a pair \((f, g)\) represents a reconfiguration request;

3) In a reconfiguration request, the RTRP goes to a reconfiguration state in which the inputted pair \((f, g)\) is stored in a configuration memory. This state lasts for \( r \) cycles, representing the reconfiguration time defined by the designer. While the RTRP is being reconfigured, it outputs \( \bot \) for both \( y[k] \) and \( x[k+1] \);

4) When \( \bot \) is inputted via the control port and when the RTRP is not in the reconfiguration state, the output \( y[k] \) and the next state \( x[k+1] \) are computed as in Item 1, with \( f \) and \( g \) the functions stored in the configuration memory. When \( k \) is the first cycle after the reconfiguration state is finished, the current state \( x[k] \) is equal the inputted initial state \( x_0[k] \); and

5) Optionally, the RTRP outputs the computed states for the next cycle \( x[k+1] \), in case it is necessary to either verify the computed states or to store the states in a external memory.

To model an AAES core in an RTRP, all that is needed is to write (13) to (22) as a pair \((f, g)\), considering the state vector \( x \) composed of \( \tilde{\omega}^w, \tilde{a}_w, q, \tilde{d}_w, \tilde{\nu}^w \) and \( \tilde{p}^w \), the input vector \( u \) composed of \( \tilde{\omega}^a \) and \( \tilde{a}_w \), with \( i \in \{0,1,2\} \), and the output vector \( y \) composed of \( \phi, \theta, \psi, \tilde{d}_w, \tilde{\nu}^w \) and \( N_z \). Listing 2 shows the functions \( f \) and \( g \) that define the AAES core functionality. The full code is available in [43].

Listing 2: AAES Synchronous MoC Haskell/ForSyDe code snippet.

```haskell
-- ::class Value.Function (Application-domain Ontology)
-- f (function definition)
f :: Float -- ^ Sampling period
-> Float -- ^ threshold for the voter of w
-> Float -- ^ threshold for the voter of a
-> AbstExt [Quaternion Float] -- ^ Current state x[k] = (w0, a0, w1, a1, w2, a2)
-> AbstExt [ImuVal, ImuVal, ImuVal] -- ^ Outputs from the three IMUs as pairs (w, a)
-> AbstExt [Quaternion Float] -- ^ Next state x[k+1]
f ts ths_w ths_a (Prst [wv', av', q', a', v', p']) (Prst (w0', a0'), (w1', a1'), (w2', a2'))
= Prst (w0, a0, w1, a1, w2, a2)
where (w0, w1, w2) = (from3tupleQ w0', from3tupleQ w1', from3tupleQ w2')
(a0, a1, a2) = (from3tupleQ a0', from3tupleQ a1', from3tupleQ a2')
wv = voter ths_w w0 w1 w2 wv'
av = voter ths_a a0 a1 a2 av'
q = q' + expQ(mulQR wv (ts/2))
a = fromTuple (0, 0, 9.81) + rotQ q av
v = mulQR (a + a') (ts/2) + v'
p = mulQR (v + v') (ts/2) + p'
f = -- ^ threshold for the voter of w

-- ::class Value.Function (Application-domain Ontology)
-- voter helper (function definition)
voter :: (RealFloat a, Ord a) => a -> Quaternion a -> Quaternion a
-> Quaternion a
-> Quaternion a
-> Quaternion a
-> Quaternion a
-> Quaternion a
voter ths_q0 q1 q2 qx
= qmax < ths_q = qm
| normQ (head q11 - last q11) < 2 * ths = divQR (head q11 + last q11) 2
| otherwise = if head qnxl = last qnxl then divVR (head q11 + qnxl) 2 else divQR (last q11 + qnxl) 2
where q1 = [q0, q1, q2]
qm = divQR (q0 + q2) 3
qnxl = [normQ (q0 - qm), normQ (q1 - qm), normQ (q2 - qm)]
qmax = maximum qnxl
qmaxPos = fromJust (elemIndex qmax qnxl)
qnxl = take qmaxPos qnxl ++ drop (qmaxPos + 1) qnxl
```

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/
Fig. 15 illustrates the complete AAES application model. See that the actors corei, with i ∈ {0, 1, 2, . . . , n}, abstract the complexity of the AAES. The actor corei function is detailed in Fig. 13 and described as in (13) to (22). This scheme allows having a defined number of active actors besides a given number of spare actors, i.e., actors with no function or void function placeholders. A voting mechanism named cores voter compare the outputs of the three active core actors and, in case of a mismatch between one active core and the other two, such core actor is flagged as faulty to the control actor named cores control. Then a new actor core (a spare one) is reconfigured in runtime to take the place of the faulty one under the command of cores control.

2) Classes of the AAES Application Model Elements

Table 8 shows the element classes of the AAES application model, as illustrated in Fig. 15, following the defined application-domain ontology.

3) AAES Mapping Rules Utilization

After modeling the AAES application based on the SY MoC, we apply the defined mapping rules to obtain a possible and virtual functional model of the complete AAES.

In the same way as in the previous example, there exist two constraints (5) and (6) that must be specified before applying the rules. To demonstrate a real scenario, we set the first constraint to use a heterogeneous device. In this case, for the second constraint, functions implementations must be hardware-based.

| Application Model Element | Class          |
|----------------------------|----------------|
| cores control              | Procedure.Controller |
| corei, i ∈ {0, 1, 2, . . . , n} | Procedure.Executor.Variable |
| cores voter, fcc, ife     | Procedure.Executor.Fixed |
| scorei, i ∈ {0, 1, 2, . . . , n} | Path.Control |
| ocores | Path.Data.Homogeneous |
| ocore0, ocore1, ocore2, ocores | Value.Action |
| ototed | Value.Info |
| ccore, ccin, ccout         | Path.Data.Hybrid |

Table 9 shows the mapping rules summary according to the constraints as in (26) and (27).

| i | Application Model Element Class | Platform Block Class |
|---|---------------------------------|----------------------|
| 1 | Pr.Controller                   | S.ControlUnit.Manager and creates a |
| 2 | Pr.Executor.Variable            | A.PerformerX.Function |
| 3 | Pr.Executor.Fixed               | A.PerformerX.Once    |
| 4 | Pa.Control                      | A.InterfaceX.ControlX |
| 5 | Pa.Data.Homogeneous             | A.InterfaceX.DataX   |
| 6 | Pa.Data.Hybrid                  | A.InterfaceX.DataX   |
| 7 | Value.Function                  | Pool.Implementation (hardware-based) |

4) Model Elements and Functional Blocks

According to the application of the mapping rules, Table 10 shows the correspondence from the elements of the AAES application model example to the functional blocks in the resulting virtual implementation model, namely AAES.

| Application Model Element | Implementation Model Functional Block |
|---------------------------|---------------------------------------|
| cores control             | Handler attached to ProgDev and also creates a S.Steward attached to ReconDev |
| corei, i ∈ {0, 1, 2, . . . , n} | Worker attached to ReconDev (hardware-based implementation) |
| scorei, i ∈ {0, 1, 2, . . . , n} | Control Bus mixed attached to ProgDev and ReconDev |
| ocore0, ocore1, ocore2, ocores | Data Bus mixed attached to ProgDev and ReconDev |
| ototed, ccore, ccin, ccout | Data Bus attached to ProgDev |

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/
Figure 15: AAES complete application model illustrating the signals and processes as a network of concurrent processes. Here, we handle the complexity of the system by abstracting the AAES core function into actors receiving a signal containing function values, namely core_i with 0 ≤ i ≤ n. Finally, the FCC and IFE are shown as receivers of the voted data.

5) AAES Resulting Virtual Implementation Model
Finally, after the application of the mapping rules, the corresponding view of the partitioned application model is shown in Fig. 16, and the virtual implementation model of AAES is illustrated in Fig. 17.

V. DISCUSSION

a: Generalization of the Runtime Reconfiguration Concept
We formulated the requirements so that a model of computation can describe a semantic notion of runtime reconfiguration. This is granted by the principle of functions as events and the functional programming paradigm. The function placeholder definition (Section III-A4) can be used in an application model to describe a runtime reconfigurable process. These processes can have their functions changed through full or partial reconfiguration in the virtual implementation model.

Notice that this concept is general, and therefore applicable to different models of computation. In other words, the runtime reconfiguration concept as defined here is orthogonal to the models of computation.

b: Domain Ontologies
The development of two domain ontologies enabled the definition of a classification system. The first one (Section III-A1) defines the classes for the elements in the application model. The second one (Section III-B1) defines the classes for the functional blocks in the virtual platform model.

These ontologies are general enough to serve as a classification system for most models of computation and reconfigurable architectures and platforms.

c: A Set of Mapping Rules
A set of unambiguous and well-defined mapping rules was developed along with two constraints (Section III-C). By systematically applying these rules to the high-level...
Figure 16: Partitioned application model showing the separation into ProgDev and ReconDev, in the application model perspective.

Figure 17: Virtual implementation model instance AAES resulting from the application of the mapping rules and considering the two constraints (26) and (27).
application $\Lambda$ and virtual platform $P$ models, our approach leads to a feasible virtual implementation model $I$, as stated in (4).

Those two constraints were added to allow for different possible organizations in terms of architectures concerning heterogeneous systems including hard processors and reconfigurable logic. They also account for a possible trade-off between power consumption and performance.

d: Representative Examples

In the encoder/decoder system example, we show an application being modeled by using the synchronous model of computation. After the constraints fixing, the mapping rules were employed for this application instance. Finally, we applied the defined rules and the result was a feasible virtual implementation model.

In the second example, we address a much complex scenario involving safety-critical systems, i.e. an avionics attitude estimation system in the integrated modular avionics context. By that modeling, we show how to model a larger system (i.e. scalability) and handle complexity within abstraction layers. In that case, the functions present in the AAES core (as in Fig. 13) are carried as a single function in the perspective of the AAES complete application model (as in Fig. 15).

Although not covered by the example applications, other models of computation such as dataflows, e.g. synchronous dataflow or scenario-aware dataflow, are also encompassed in the function placeholder Definition 2. Therefore, these MoC have equivalent mapping rules as the ones shown for the synchronous MoC, making our design flow general enough to model other systems from different areas.

e: Guidelines to a Platform-based Design: The Next Step

The proposed methodology relies on a high-level of abstraction and formal models of computation. The design starts with the application being modeled based on a given MoC according to the application’s domain, for instance, synchronous reactive or dataflow.

Considering the model simulation still in the specification phase, i.e. an executable specification, the formalism of MoCs can be captured by a wide range of computer programming languages from different programming paradigms such as object-oriented programming (OOP), imperative, and functional, either interpreted or compiled.

When following the concepts of OOP [44], there is the object (i.e. an instance of an OOP-class) with data and methods. Thus, the application model element $actor$ (application-domain ontology’s classes $Procedure.Controller$ and $Procedure.Executor.Fixed$) can be implemented by using an object. Notice that there may be a limitation towards equivalence in OOP for $Procedure.Executor.Variable$ considering its semantics. Object polymorphism is a possibility that most approximates, however it does not cover the essence of $Procedure.Executor.Variable$. The element $signal$ (classes $Path.Control$, $Path.Data.Homogeneous$, $Path.Data.Hybrid$) can be captured through relationships like dependency and association, for instance. The element $function$ (class $Value.Function$) can be implemented by using an objects’ method. In this specific case, both unified modeling language (UML) 2 and systems modeling language (SysML) can aid in the modeling activity [17], [45].

When using imperative programming languages, like Python or C++, the application model element $actor$ (application-domain ontology’s classes $Procedure.Controller$, $Procedure.Executor.Variable$, $Procedure.Executor.Fixed$) can be implemented by using a task. The element $signal$ (classes $Path.Control$, $Path.Data.Homogeneous$, $Path.Data.Hybrid$) can be implemented with a message queue or a pipe, for example. The element $function$ (class $Value.Function$) can be implemented by using a software function along with the function pointer technique. These guidelines apply when the programming language does not support higher-order functions. However, functions are not pure in the imperative paradigm since they are likely to produce side-effects.

We argue that the semantics of a MoC, as described in Section III-A2, is better preserved when using the functional paradigm. In this sense, the model element $actor$ is implemented as a process, and the element $signal$/communication paths as a $signal$ from the functional language. $Functions$ are simply regular data carried by signals, as the functional paradigm supports higher-order functions, and consequently the function placeholder definition (Section III-A4).

In this case, a physical implementation of our virtual implementation model is based on a CPU, GPU, or any other fixed hardware platform following the platform-domain ontology’s classes from entity static. Table 11 shows in the first five rows a possible mapping guideline from classes related to the platform-domain ontology to a physical platform based on only fixed processing units. This states feasible and concrete possibilities.

Conversely, provided physical hardware supporting runtime reconfiguration is in hand, the mapping from the application model to the virtual platform model relies on the platform-domain ontology’s classes from entity adaptable.

In this way, classes Adaptable.InterfaceX.DataX and Adaptable.InterfaceX.ControlX can be physically implemented (e.g. concrete) with wires and buses in a modern FPGA-based device. Functional blocks such as the worker (classes Adaptable.PerformerX.Once, Adaptable.PerformerX.Mode, and Adaptable.PerformerX.Function) can be implemented in a hardware description language (HDL) and become a runtime reconfigurable partition inside the FPGA area. Blocks like the handler (classes Adaptable.ControlUnitX.ManagerX and Adaptable.ControlUnitX.DispatcherX) can also be implemented in HDL, as illustrated in Fig. 9b, or even in software, as shown in Fig. 9a, where the handler follows classes Static.ControlUnit.Manager and Static.ControlUnit.Dispatcher. Table 11 also shows these cases.
Our approach enhances the following qualitative key performance indicators (KPIs) for the physical implementation. The expression “RTL/HDL reconfigurable part, hardware partition” means that it is possible to implement a hardware-based function (i.e., circuit) using various possibilities including RTL abstraction and HDL code. In turn, that hardware-based function can be placed into a location allowing for runtime full and partial reconfiguration, as required.

| # | Class of Platform-domain Ontology                          | Physical HW Platform (Fixed) (examples of possibilities)                                                                 | Physical HW Platform (Reconfigurable/FPGA) (examples of possibilities)                                                                 |
|---|-------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| 1 | Static.Interface.Data                                       | Message Queue, Pipe                                                                                                         | —                                                                                                                                   |
| 2 | Static.Interface.Control                                    | Message Queue, Pipe                                                                                                         | —                                                                                                                                   |
| 3 | Static.Performer                                            | SW Task, SW Function, OOP Object                                                                                             | —                                                                                                                                   |
| 4 | Static.ControlUnit.Manager                                  | SW Task, SW Function, OOP Object                                                                                             | —                                                                                                                                   |
| 5 | Static.ControlUnit.Dispatcher                               | SW Task, SW Function, OOP Object                                                                                             | —                                                                                                                                   |
| 6 | Adaptable.InterfaceX.DataX                                 | —                                                                                                                          | Register-transfer level (RTL)/HDL Wire, Bus                                                                                         |
| 7 | Adaptable.InterfaceX.ControlX                              | —                                                                                                                          | RTL/HDL Wire, Bus                                                                                                                   |
| 8 | Adaptable.PerformerX.Once                                   | —                                                                                                                          | RTL/HDL reconfigurable part, hardware partition                                                                                     |
| 9 | Adaptable.PerformerX.Mode                                  | —                                                                                                                          | RTL/HDL reconfigurable part, hardware partition                                                                                     |
| 10 | Adaptable.PerformerX.Mode                                 | —                                                                                                                          | RTL/HDL reconfigurable part, hardware partition                                                                                     |
| 11 | Adaptable.ControlUnitX.Manager                             | —                                                                                                                          | RTL/HDL reconfigurable part, hardware partition                                                                                     |
| 12 | Adaptable.ControlUnitX.DispatcherX                         | —                                                                                                                          | RTL/HDL reconfigurable part, hardware partition                                                                                     |
| 13 | Pool.Configuration                                         | RAM, Flash Memory                                                                                                          | RAM, Flash Memory                                                                                                                   |

where \( \text{fm} \) is short for final mapping.

Notice that our virtual implementation model is sufficiently flexible to allow several possibilities when it comes to physical implementations in different hardware architectures. In this sense, we can say the high-level abstraction allows our models to be platform-agnostic, they do not depend on any specific hardware technology, but at the same time, they can be concretely (e.g., physically) implemented on a variety of commercial off-the-shelf platforms.

The final mapping from the virtual platform to a physical platform follows the expression (29), and Table 11 gives the mapping guidelines to the final implementation.

\[
m : (\Lambda \rightarrow P) \equiv I \quad (28)
\]

\[
\text{fm} : I \rightarrow \text{Physical Implementation} \quad (29)
\]

where \( \text{fm} \) is short for final mapping.

### f: Key Performance Indicators

Our approach enhances the following qualitative key performance indicators (KPI):

1) **hardware area reusability** – assuming that not all system’s functions must be available during all system’s operation, and enabled by the generalization of the runtime reconfiguration concept (Section III-A4), the same hardware can be used in a mutually exclusive time fashion. In turn, it also contributes to decreasing single-event upset (SEU) due to less hardware area exposition. For example, SEU can be faced during flights by electronic devices present in commercial airplanes;

2) **reduced testing needs** – this is due to the presence of formalism already in the specification project step, and convenient way to simulate, e.g., formal models of computation contribute to correct-by-construction;

3) **taming of complexity** – the most complex system can take profit from our approach since we use different levels of abstraction to focus on the applications’ functions (i.e., application model) and platform functional blocks (e.g., virtual platform model) hiding away implementation details that will be taken into account on further design steps; and

4) **clear separation of concerns** – we make a clear separation between the application model from the platform model and enable to map of different virtual platforms to real physical platforms.

### VI. CONCLUSION

This paper introduced a classification system derived from application- and platform-domain ontologies. The ontologies were used to restrict both the application domain and the implementation domain to the set of necessary components, so that correct and efficient mapping rules were created.

We suggested a design methodology considering an application model that admits runtime reconfigurable behavior, described by formal models of computation, and also a scalable virtual platform model, described by functional blocks. When applying a set of well-defined mapping rules to these models, the result is a feasible virtual implementation model. Functional programming is also used here as an elegant and convenient way to simulate, i.e., execute, the application model already in the design entry stage.

The novelty introduced in this research was on combining suitable formal models of computation, a functional modeling language, and two domain ontologies to create a systematic system design flow from an abstract executable system model into a virtual implementation targeting a runtime reconfigurable architecture using well-defined mapping rules.

We demonstrated the potential, applicability, and scalability of our classification system and mapping rules in two runtime reconfigurable embedded system examples: a tutorial-oriented encoder/decoder system, and a larger and more complex system, i.e., an avionics attitude estimation.
system. As future work, we intend to automate the application of the mapping rules based on the high-level abstraction application model and the classification systems, besides the final mapping from the virtual implementation model to the physical platform implementation.

ACKNOWLEDGMENT

The authors would like to thank the project “Real-Time Electronics & Computers for Highly Computing Intensive Real-Time Systems” (RTEC) at ITA for the funding support to publish this research work. The RTEC project involves ITA/Brazil, KTH Royal Institute of Technology/Sweden, LiU Linköping University/Sweden, and Saab AB/Sweden.

The research results presented here are part of the Regular Research Grant #2019/27327-6 from São Paulo Research Foundation (FAPESP).

References

[1] E. Lee and A. Sangiovanni-Vincentelli, “A framework for comparing models of computation,” Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 17, no. 12, pp. 1217–1229, Dec. 1998.
[2] M. Nguyen, R. Tamburo, S. Narasimhan, and J. C. Hoe, “Quantifying the Benefits of Dynamic Partial Reconfiguration for Embedded Vision Applications,” in 29th International Conference on Field Programmable Logic and Applications (FPL). IEEE, Sep. 2019.
[3] N. Thomas, A. Felder, and C. Bobda, “Adaptive controller using runtime partial hardware reconfiguration for unmanned aerial vehicles (UAVs),” in International Conference on ReConFigurable Computing and FPGAs (ReConFig). IEEE, Dec. 2015.
[4] D. Serpanos, G. Ferrari, G. Nikolakopoulos, J. Perez, M. Tauber, and S. Van Baelen, “Embedded Artificial Intelligence: The ARTEMIS Vision,” Computer, vol. 53, no. 11, pp. 65–69, 2020.
[5] A. Sangiovanni-Vincentelli and G. Martin, “Platform-based design and software design methodology for embedded systems,” IEEE Design Test of Computers, vol. 18, no. 6, pp. 23–33, 2001.
[6] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin, “Actor-Oriented Design of Embedded Hardware and Software Systems,” Journal of Circuits, Systems & Computers, vol. 12, no. 3, pp. 231–260, 2003.
[7] V. Pinkvich and A. Platunov, “Method for Testing and Debugging Flow Formal Specification in Full-Stack Embedded Systems Designs,” in 2020 9th Mediterranean Conference on Embedded Computing (MECO), 2020.
[8] G. Zervakis, H. Amrouch, and J. Henkel, “Design Automation of Approximate Circuits With Runtime Reconfigurable Accuracy,” IEEE Access, vol. 8, pp. 53,522–53,538, 2020.
[9] J. Emmert, C. Stroud, B. Skaggs, and M. Abramovici, “Dynamic fault tolerance in FPGAs via partial reconfiguration,” in Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00871), 2000, pp. 165–174.
[10] A. Kian, H. Hosseintalaei, and A. Jahanian, “Protecting the FPGA IPs against Higher-order Side Channel Attacks using Dynamic Partial Reconfiguration,” in 20th International Symposium on Computer Architecture and Digital Systems (CADS). IEEE, Aug. 2020.
[11] S. A. Seshia, S. Hu, W. Li, and Q. Zhu, “Design automation of cyber-physical systems: Challenges, advances, and opportunities,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 36, no. 9, pp. 1421–1434, 2017.
[12] L. Belhamel, A. Buscarino, A. Cuccuccion, L. Fortuna, and G. Rascona, “Model-based design streamlines for stm32 motor control embedded software,” in 2020 7th International Conference on Control, Decision and Information Technologies (CoDiT), vol. 1, 2020, pp. 223–228.
[13] T. Dörn, T. Sandmann, and I. Becker, “A formal model for the automatic configuration of access protection units in mpSoC-based embedded systems,” in 2020 23rd Euromicro Conference on Digital System Design (DSD), 2020, pp. 596–603.
DENIS S. LOUBACH was born in São José dos Campos, São Paulo, Brazil in 1982. He received the PhD degree in Electronic Engineering and Computer Science from the Aeronautics Institute of Technology – ITA in 2012. This author became a Member (M) of IEEE in 2006. From 2014 to 2018, he was an Assistant Professor with the University of Campinas – UNICAMP working in the area of real-time embedded systems and digital systems. Since July of 2018, he has been an Assistant Professor with the Department of Computer Systems, Computer Science Division, ITA. His research interests include reconfigurable computing, real-time embedded systems, and models of computation theory. Dr. Loubach had also worked for a couple of years in the aeronautics industry as an avionics systems and embedded software engineer.

RICARDO BONNA was born in the city of São Paulo, São Paulo, Brazil in 1990. He received the PhD degree in Mechatronics from the University of Campinas (UNICAMP) in 2021. He has a Master of Science (MSc) degree in control systems engineering. Since 2019, he has been working as an embedded software engineer at Embraer. His research interests include both embedded and control systems design. He is also an enthusiast of functional programming for system modeling and design.

GEORGE UNGUREANU was born in Tulcea, Romania in 1988. He is a PhD candidate in embedded systems design at KTH Royal Institute of Technology, Sweden. He received a Master of Science (MSc) degree in system-on-chip design from KTH Royal Institute of Technology, Sweden, in 2013. Since 2021, he has been working as a software research engineer in the telecommunications industry. His research interests are cyber-physical systems, modeling and design of embedded systems, massively-parallel programming paradigms, and he is a functional programming enthusiast.

INGO SANDER Ingo Sander received the MSc degree in Electrical Engineering from the Technical University of Braunschweig, Germany, in 1990 and the PhD degree and Docent degree from KTH Royal Institute of Technology, Sweden, in 2003 and 2009, respectively. Between 1991 and 1993 he worked as system design engineer at Ericsson, Sweden. In 1993 he joined KTH, where since 2018 he has a position of Professor in Electronic System Design. His main research interests are located in the area of design methodologies for embedded systems, including system modeling, design space exploration and system synthesis.

INGEMAR SÖDERQUIST was born in Borghamn, Sweden, in 1960. He received his M.S. degree in Electronic and Electrical Engineering from the University of Linköping, Sweden, in 1985, and the Ph.D. degree in Electronic Devices from the University of Linköping, Sweden, in 2002. From 1985 to 2000, he was with Saab Dynamics AB, where he worked with analysis and development of high performance signal processing, radar systems, and microwave products. From 2000 to 2003 he was with Saab Avionics AB where he worked as manager of the Digital RF Systems group at the Electronic Warfare Division. In 2003 he moved to Saab AB working as project manager in the Product Area Recording and Monitoring Systems. Currently he is working with broad-based view for electronics in the Avionics Computer Systems group at the Business Area Aeronautics. In 2008 he was appointed the position Technical Fellow within the strategic area Electronics Design in Avionics. He is also at the Department of Electrical Engineering at Linköping University. His research interest is high performance circuits and systems in CMOS, failure modes and mechanisms in electronic systems, and reliable computer architectures.

***