Software and Systems Modeling (2021) 20:611–639
https://doi.org/10.1007/s10270-021-00876-z

THEME SECTION PAPER

Multi-paradigm modelling for cyber–physical systems: a descriptive framework

Moussa Amrani[1] · Dominique Blouin[2] · Robert Heinrich[3] · Arend Rensink[4] · Hans Vangheluwe[5] · Andreas Wortmann[6,7]

Received: 6 April 2020 / Revised: 28 September 2020 / Accepted: 8 December 2020 / Published online: 9 June 2021

Abstract
The complexity of cyber–physical systems (CPSs) is commonly addressed through complex *workflows*, involving models in a plethora of different *formalisms*, each with their own methods, techniques, and tools. Some *workflow patterns*, combined with particular *types of formalisms* and *operations* on models in these formalisms, are used successfully in engineering practice. To identify and reuse them, we refer to these combinations of workflow and formalism patterns as modelling *paradigms*. This paper proposes a *unifying (Descriptive) Framework* to describe these paradigms, as well as their combinations. This work is set in the context of Multi-Paradigm Modelling (MPM), which is based on the principle to model every part and aspect of a system explicitly, at the most appropriate level(s) of abstraction, using the most appropriate modelling formalism(s) and workflows. The purpose of the Descriptive Framework presented in this paper is to serve as a basis to reason about these formalisms, workflows, and their combinations. One crucial part of the framework is the ability to capture the *structural* essence of a paradigm through the concept of a *paradigmatic structure*. This is illustrated informally by means of two example paradigms commonly used in CPS: Discrete Event Dynamic Systems and Synchronous Data Flow. The presented framework also identifies the need to establish whether a paradigm *candidate* follows, or qualifies as, a (given) paradigm. To illustrate the ability of the framework to support *combining* paradigms, the paper shows examples of both workflow and formalism combinations. The presented framework is intended as a basis for characterisation and classification of paradigms, as a starting point for a rigorous formalisation of the framework (allowing formal analyses), and as a foundation for MPM tool development.

Keywords Multi-paradigm modelling · Foundations of model-based systems engineering · Cyber–physical systems

1 Introduction

Cyber–Physical Systems (CPSs) are engineered systems that emerge from the networking of multi-physical processes (mechanical, electrical, biochemical, etc.) and computational

Communicated by Eugene Syriani and Manuel Wimmer.

✉ Arend Rensink
arend.rensink@utwente.nl

Moussa Amrani
Moussa.Amrani@unamur.be

Dominique Blouin
Dominique.Blouin@telecom-paris.fr

Robert Heinrich
Robert.Heinrich@kit.edu

Hans Vangheluwe
Hans.Vangheluwe@uantwerpen.be

Andreas Wortmann
Andreas.Wortmann@isw.uni-stuttgart.de

1 Namur Digital Institute (NaDi), University of Namur, Namur, Belgium

2 Télécom Paris, Institut Polytechnique de Paris, Palaiseau, France

3 Karlsruhe Institute of Technology, Karlsruhe, Germany

4 University of Twente, Enschede, The Netherlands

5 University of Antwerp – Flanders Make, Antwerp, Belgium

6 RWTH Aachen, Aachen, Germany

7 University of Stuttgart, Stuttgart, Germany

processes (control, signal processing, logical inference, planning, etc.) that typically interact with a highly uncertain environment, including human actors, in a socio-economic context. These systems enable many of our daily activities and have become innovation drivers in important domains, such as automotive, avionics, civil engineering, Industry 4.0, and robotics.

Engineering CPSs requires the contribution of experts from different domains to solve the challenges related to their own discipline, but also to collaborate to make all parts work together. Because CPSs are generally costly to fully build and maintain, early modelling and simulation is a de facto technique crucial in their development. This enables reconciling the multifaceted aspects of a CPS, studying safety-critical and emerging properties, and planning for deployment even before the physical parts of the system are available (e.g. via Hardware-in-the-Loop (HIL) simulation).

The full complexity of CPS engineering is not covered by single modelling paradigms. For instance, the Equation-Based paradigm only covers the physical parts of the system; the Object-oriented paradigm only covers the code parts of the system; and the Agile paradigm only covers workflow aspects of system development. Consequently, the heterogeneity and complexity of CPSs and their design activities require the combination of multiple paradigms to describe the entire system while including all relevant aspects.

In this context, what is a *paradigm* then? The science philosopher Kuhn defines it as "an open-ended contribution that frames the thinking of an object study with concepts, results and procedures that structures future achievements" [42]. Though seemingly far from the concerns in the discipline of computer science, this definition does highlight the emergence of a *structure* that captures the object of discourse and the existence of *procedure*s that guides achievements.

In computer science, paradigms are probably best known as a means for classifying General-purpose Programming Languages (GPLs). For example, Eiffel is Object-Oriented and supports Contract-Based Design, Prolog is considered Declarative, while Lisp is Functional. A paradigm characterises both the syntax *and* semantics of the language including principles that govern it: Object Orientation imposes viewing the world in terms of communicating objects typed by classes, whereas the declarative paradigm relies on term substitution and rewriting. The idea of combining several paradigms into a single GPL led to more expressive, powerful programming languages such as Java [28] (which is Imperative, Object-Oriented, Concurrent, Real-Time, and Functional) and Maude [13] (which is Declarative, Object-Oriented, Concurrent, and Real-Time), among many others.

Multi-Paradigm Modelling (MPM) has only recently been recognized as a powerful paradigm on its own that can help to design, as well as communicate and reason about, CPSs. The term MPM finds its origin in the Modelling and Simulation community in 1996, when the EU ESPRIT Basic Research Working Group 8467 "Simulation in Europe" (SiE) formulated a collection of research directions and policy guidelines [69] identifying the need for "a multi-paradigm methodology to express model knowledge using a blend of different abstract representations rather than inventing some new super-paradigm". The main result was a vision where all parts and aspects of a complex system are modelled explicitly, using the most appropriate modelling formalisms to deal with engineering heterogeneity. The important aspect of workflow was not yet present. At first, only problems were identified, but later on, the same group focused on combining multiple formalisms [70] through architectural composition (as opposed to view composition). One main merit of the SiE work was the inclusion of **a-causal** modelling to model physical phenomena, an effort that led to the design of the Modelica language.

Physical systems are often modelled using continuous abstractions, e.g. Differential Algebraic Equations (DAEs) to express constituent equations relating physical variables of interest. Software systems are often modelled using discrete abstractions, e.g. State Automata to express the discrete changes made to data stored in memory by executing program instructions. A consequence of the fact that CPSs combine cyber (software) and physical components is that they are naturally modelled using *hybrid* modelling languages that combine continuous and discrete abstractions [79]. The meaningful and usable integration of discrete and continuous domains is at the heart of dealing with CPS. More generally, dealing with heterogeneity, both in the levels of abstraction and in the formalisms used, is one of the major challenges in modelling CPSs.

The main contribution of this paper is a Descriptive Framework for MPM applied to CPSs. The framework is based on a special kind of metamodel where *placeholders* can be used, capturing various *structural* and *process* patterns. Such metamodels support expressing property expressions that we call *paradigmatic* properties: they are used to capture the essence of a paradigm and can be bound to existing elements of candidate formalism/workflow metamodels (as well as their semantics) to determine if the candidate formalism(s)/workflow(s) effectively follow the paradigm.

Although not completely formal, our framework allows experts to better grasp the essence of how their CPSs are designed, while providing a common ground for a rigorous engineering of CPSs based on their MPM components. Ultimately, in a next step not covered by this paper, this framework aims to support tool builders, language developers, analysis engineers and other experts to reason about CPSs and figure out which formalisms, abstractions, workflows and supporting methods, techniques, and tools are the *most appropriate* to carry out their task(s), thus minimising accidental complexity due to non-optimal **tool** selection. Note that this paper does not intend to present a classification of formalisms or workflows that could be used to engineer CPSs. However, our Descriptive Framework could be used to better classify these elements by providing more precise descriptions for them.

This paper is a continuation of an effort started during the COST Action IC1404[1] "Multi-Paradigm Modelling for Cyber-Physical Systems" (MPM4CPS), which surveyed languages and tools used for engineering CPSs [12] and captured the relationships between them in an ontology. Moreover, it significantly extends, and complements, a preliminary version of our Descriptive Framework [3] by (i) capturing the various components of a paradigm explicitly and (ii) demonstrating a simple paradigm *combination* resulting in a valid paradigm, which could suggest that our framework is closed under the usual combination operators required for the *multi-paradigms* necessary for modelling CPSs.

We organised the paper as follows. Section 2 presents an informal notion of paradigm to serve as a tutorial introduction to our Descriptive Framework, which itself is described in Sect. 3. Section 4 exemplifies the framework with two well-known paradigms used for CPS development. Section 5 defines a paradigm combinator, namely *embedding*, and shows how to systematically build a paradigm candidate from candidates of the combined paradigms. Section 6 highlights and discusses related work from other communities, and Sect. 7 reflects on our results. Section 8 proposes future lines of research and concluding remarks.

2 What is a paradigm?

Broadly speaking, a *paradigm* acts as a *pattern* for describing a whole class of artefacts sharing similar characteristics or designates a framework that encapsulates theories inside a scientific domain. We aim to capture the meaning of the paradigm concept precisely enough to make it ultimately amenable to computer-based analysis and reasoning.

This section provides an intuitive and lightweight introduction to what a paradigm is. We start by a small detour in linguistics and epistemology to revisit the classical definitions in these fields, before focusing again on their meaning in computer science. Using two well-known paradigm examples from computer science, namely the Object Orientation and *Agile development* paradigms, we clarify the core components of our Descriptive Framework. The structure of this framework is then described by means of a metamodel and illustrated through typical usage scenarios.

[1] http://mpm4cps.eu.

2.1 General definitions

From a *linguistic* viewpoint, a *paradigm* has three definitions from the English dictionary:

- A *framework* containing basic assumptions, ways of thinking, and methodology that are commonly accepted by members of a scientific community [57];
- A philosophical and theoretical framework of a scientific school [of thought] or discipline within which are formulated theories, laws, and generalisations, as well as the experiments performed in support of [53].
- A model of something, or a very clear and typical *example* of something [11].

Although very general in nature, there are several aspects of these linguistic definitions that are worth pointing out. First, in each of the above definitions, a paradigm defines, in some sense, a *structure* that is shared by several elements the paradigm is intended to capture. Second, a paradigm also provides a way of *deciding* whether an element under analysis possesses those "basic assumptions" for fitting the structure. Third, a paradigm organises the elements it characterises in such a way that it becomes possible to *reason* about them (with the help of "theories", "laws" and suitable "generalisations"). Finally, a paradigm results from an *agreement* between "members of a [scientific] community": the precise definition may change over time and may be slightly different from different "schools of thought", though sharing "basic assumptions".

In the field of philosophy of science, the most popular and commonly agreed-upon definition of the concept of paradigm was formulated by Kuhn [42], who distinguishes the following:

- The subject matter, i.e. what is to be observed and scrutinised;
- The kind of questions that are supposed to be asked and probed for answers in relation to the subject;
- How these questions are to be structured;
- What predictions are made by the primary theory within the discipline;
- How the results of scientific investigations should be interpreted;
- How an experiment is to be conducted and which equipment is available to conduct these experiments.

The aspects highlighted by this philosophical definition are similar to the linguistic ones pointed out above, although differently framed. Kuhn gives some details about how the reasoning takes place: he emphasises that a paradigm is *questioned* in a *structured* way, and that some of these questions may be general enough to form the basis of predictions about the subject matter.

Let us summarise what we learnt about the nature and functions of a paradigm:

1. A paradigm captures the *essence* of a collection of elements that have a substantial impact in a scientific discipline.
2. As a consequence, a paradigm is ontologically *distinct* from those elements.
3. The essence captured by a paradigm is expressed through "questions" or, in the case of computer science, *properties of interest* that are supported by various *structures*.
4. Those properties enable *reasoning* and *drawing suitable generalisations*, and *predictions*. They also offer a way of *deciding* whether an element of interest (that we later call a "*candidate*") *qualifies as*, *follows*, or *embodies* this paradigm, typically by human assessment.

We claim that in computer science, the "questions" for a paradigm, or *paradigmatic properties* as we will call them, always rely on structures that are supported by *processes*, or *workflows*, for capturing the dynamic nature of computations, processes that ultimately manipulate *formalisms*.

In the next section, we purposely study two paradigm examples (in a simplified version) that are widely recognised as having significantly shifted the scientific field of computer science, namely Object Orientation and Agile Programming. Note that these are programming paradigms, which constitute a specific subclass of modelling paradigms, with the advantage of being readily understood by readers from the Software Engineering community. Both are chosen on purpose: the former pertains to formalisms, whereas the latter pertains to processes.

For the purpose of the presentation, we had to choose a particular way of describing those concepts using *supportive formalisms* (which correspond to meta-metamodels, or technical spaces, see Wimmer and Kramler [77]). Note, however, that our Descriptive Framework does not depend on any particular choice of supportive formalism(s): only the expression of the (paradigmatic) properties and their underlying structures depend on them for reasoning and deciding whether a (candidate) element follows a given paradigm. We further discuss this point at the end of each example.

2.2 Two simple examples: Object Orientation and Agile Programming

An important feature for paradigms, which is crucial to clarify the discourse, is the ability to explicitly *name* both the properties a paradigm relies on, and as well as variations of a paradigm. We present in this section two (versions of) well-known paradigms in computer science and discuss some of their characteristic properties. For each paradigm p, we adopt a similar presentation:

1. We provide background information on paradigm p to point out why it significantly impacted programming;
2. We focus on *one* singular property π of p that is common enough to make it easy to grasp, and simple enough to be easily demonstrated without introducing too much notation;
3. We present two *candidate* elements $\mathcal{C}1$ and $\mathcal{C}2$, one for which π is satisfied, and the other for which it is not;
4. We list the required *supporting formalisms* necessary for building our Descriptive Framework and illustrate them on the basis of our candidates.

2.2.1 Object Orientation: a formalism-oriented paradigm

Object Orientation (OO) emerged in the 1960s in response to a need to structure the way programs were specified. Instead of seeing a computation as just imperative processing of sequential instructions, OO defines and structures computation through organised, communicating objects that are typed by means of classes, which define their structure as well as their computation and communication capabilities. OO concepts are applicable in software engineering sub-domains such as analysis, design, and software development. Whether a GPL is classified as OO depends on how tightly integrated the OO concepts are into the programming language: from "pure" OO GPLs where every programming construct is an object (e.g. in Eiffel or Scala), over GPLs that still contain some procedural elements (e.g. Java or C), to GPLs that integrate some specific concepts (e.g. Ada or MATLAB).

There exist many variations of the definition of the OO paradigm for GPLs (cf. among others, [1,75]). As a possible classification, Wegner [75] distinguishes the notions of object-based and object-oriented GPLs that may support (or fail to support) data abstraction, strong typing, and delegation. For illustrative purposes, let us only consider a very basic feature, namely *inheritance*, as a language mechanism to share and factor out properties, thus promoting reuse. When a (sub-)class C inherits from a (super-)class C', then semantically, all objects that are instances of C automatically inherit the state and behaviour of C'. Of course, many other more complex properties define the OO paradigm, and potentially several variations of the same property (e.g. allowing *multiple* inheritance) may be considered. As described previously, a paradigm is often an agreement or a common understanding in a scientific school of thought, but nothing prevents the co-existence of several variations of definitions that are similar. Discriminating between them may be achieved through distinct *names* relating to different (variations of) the set of properties that characterise a given paradigm.

One may be interested in checking that a given *candidate* GPL actually qualifies as OO. Let us consider Java [28] and Pascal [15] for the purpose of the discussion. For doing so, one needs to check whether the properties defining (the particular flavour of) OO are indeed satisfied by such a candidate GPL. Note that a given candidate GPL is itself a language specified with candidate formalisms: one for capturing its concrete syntax the programmer manipulates and one for providing executability through an operational and/or a translational semantics. We will qualify those as *candidate* formalisms, to distinguish them from the *paradigmatic* formalisms used for capturing the specifics of a given paradigm.

Completely formalising those properties still requires the use of appropriate *supporting* formalisms for capturing them and a way to relate the descriptions to the formalisms defining the candidates, to check the properties' satisfaction.

To summarise, we considered the paradigm p as being Object Orientation, for which one of the characteristic properties π is *inheritance*, with two potential candidate elements $\mathcal{C}1$ as Java, and $\mathcal{C}2$ as Pascal. To be able to actually check whether $\mathcal{C}1$ and $\mathcal{C}2$ qualify as Object Oriented, we need at least four kinds of formalisms:

1. A *structural* (paradigmatic) formalism for describing structures, to name, organise and relate the concepts required by the paradigm. In the case of *inheritance*, this (paradigmatic) formalism would capture the notions of class, fields and objects and their relationships, as described, e.g. by Wegner [75]. Depending on the properties of interest characterising a given paradigm, this (paradigmatic) formalism may be used to capture patterns at both the *syntactic* and *semantic* level of a candidate, since paradigmatic properties often concern both (as it is the case for the inheritance property described earlier anyway).
   Figure 1 (top) illustrates one way to capture the structure necessary for expressing the inheritance property using a Placeholder Class Diagram inspired by the UML MOF syntax (where placeholders are represented as double rectangle "classes").
2. In the context of GPLs, candidates are usually already existing programming languages, defined in a given (candidate) formalism. Java and Pascal certainly have a BNF grammar definition historically, and Java may have a UML Class Diagram-based (e.g. as a metamodel in the Eclipse platform) or a Graph Grammar-based definition (e.g. Corradini et al. [16], among others).
   Figure 1 (bottom) represents the (simplified) metamodels of two GPLs, Java and Pascal, as *candidates* for the OO paradigm, using a MOF Class Diagram.
3. A *mapping* formalism for relating the structural (paradigmatic) formalism with the candidate formalisms. This mapping is essential because the patterns captured by the paradigm p need to be related to specific (sub-)structures in the candidates. Precisely defining this mapping formalism is out of the scope of this paper; we explain only informally how this mapping would occur (or fail to) for our candidates Java and Pascal.
   We need to check whether the topological structure from Fig. 1 may be matched against both GPLs' metamodels and if so, whether the property is satisfied (modulo the matching) on the corresponding structures.
   A Pascal Program is composed of Blocks, which are either constant, variable, or type definitions, or alternatively function and procedure declarations. None of these concepts would fully match against the C placeholder, because no association can be appropriately matched against the super reflexive association, nor with an appropriate match with VF and its own associations. Without further analysis, one can confidently conclude that Pascal does not qualify as OO.
   In the Java metamodel, however, the NormalClassDeclaration is a good candidate for a match with the C placeholder, since it also contains ClassMemberDeclarations where FieldDeclarations may potentially match the TF placeholder, with the super relationship being expressed with extends (as the textual representation of super in the left of Fig. 1). Notice that Java is actually richer: interfaces may also match with C, but would fail for the rest (since Java's interfaces do not declare fields); and Java allows field overloading.
4. Finally, a *property* (paradigmatic) formalism for specifying properties over the structural (paradigmatic) formalism, as well as an appropriate checking procedure allowing to validate, via the mapping, that a candidate GPL satisfies the expressed (paradigmatic) properties. Following our choice of Placeholder Class Diagram as a structural paradigmatic formalism, a natural choice for expressing our inheritance property would leverage the OCL language that could accommodate with placeholders. Again, without going into too much formal specification, we rely on the usual OCL syntax to try and express inheritance, in two steps.
   First, the set of accessible fields for an object is recursively computed by climbing up the super relationship in the object's typing class.

```
context O inv valuedFieldsMatchAccessibleFields :
    let valFieldNames : Set(String) =
        o.valFields.name
    in o.type.accessibleFields()
        ->collect(tf | tf.name)-> forAll(tfName |
            valFieldNames.exists(tfName))
```

Since Pascal presented no match for the structural patterns of the *inheritance* property, there is no need to try and check the property itself. For the Java case,

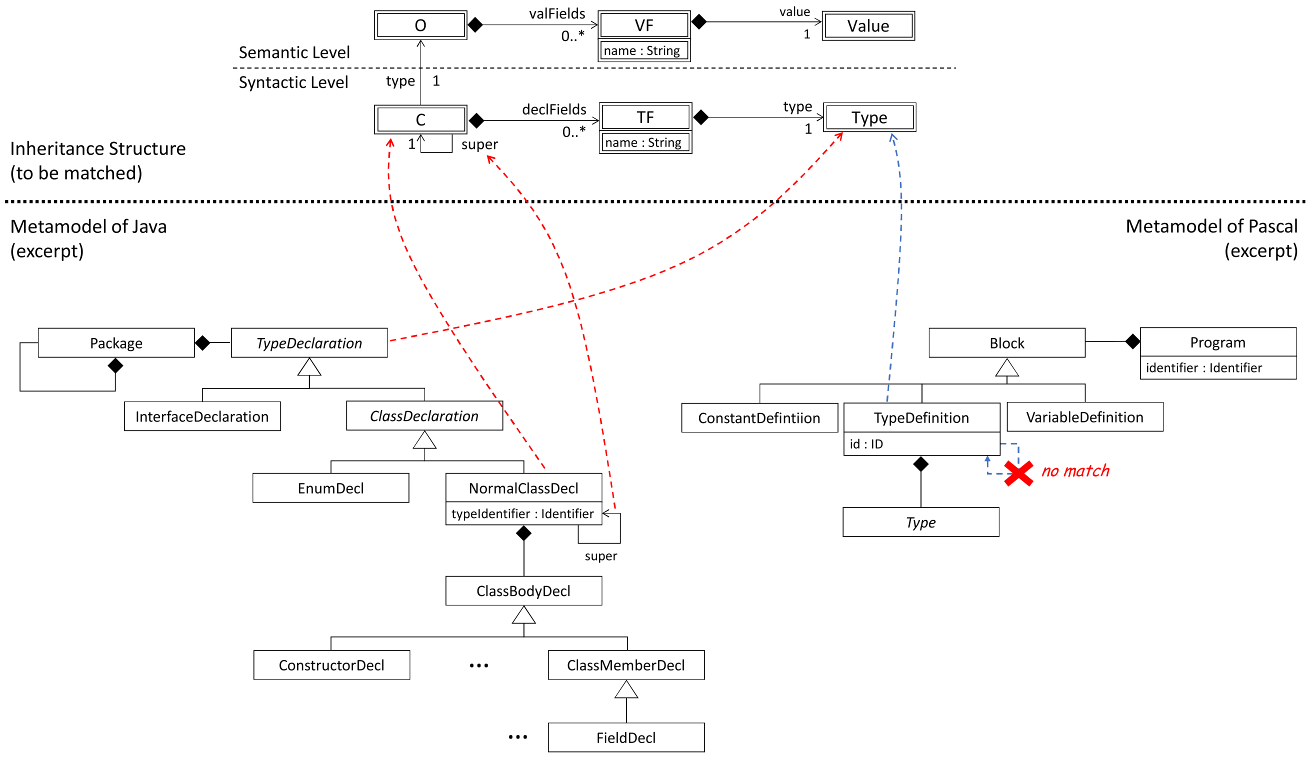


Fig. 1 On top, an example of a Placeholder Class Diagram for capturing concepts and relationships needed for expressing paradigmatic properties, using a UML MOF-based syntax: "placeholder" classes are depicted with double rectangles (instead of the regular rectangles for UML Class Diagram), to indicate that classes are meant to be matched into a candidate metamodel. The example illustrates (part of) the *inheritance* property of the OO paradigm. On the left, an excerpt of a metamodel for the Java GPL, and one for the Pascal GPL on the right, showing how (syntactic) may be appropriately matched or not

the nature of *inheritance* requires to have a look at the semantic level to check for a similar mechanism. Stärk et al. [61] proposed a formal semantics for Java based on Abstract State Machines, which are directly executable, and compared their specification with the Java Compiler. Their specification defines (algebraic) functions for class (namely $classFieldValues$) and instance field ($instFieldValue$) declarations, and models the dynamic state of objects through their reference; both collect the so-called *accessible* fields for an object and are updated with the semantic rules translating the effect of field access and assignment. The Inheritance property π is enforced in their semantic specification by simply ensuring that the (algebraic) total functions share appropriate domains (thus forcing accessible fields to possess a value, be it the value used at initialisation).

A formal proof is obviously out of this paper's scope, but this simple example already demonstrates how it may be difficult to relate and check properties expressed in different supporting formalisms (an OCL-like expression for the paradigmatic property and an algebraic expression for the Java candidate).

2.2.2 Agile development: a workflow-oriented paradigm

Agile development (AD) emerged in the early 2000s as an alternative to the so-called heavyweight software development processes (such as the traditional V-model), because many software development projects required less regulation, a shorter response time to requirement changes from customers during the course of a project, and the processes were perceived as overly constraining for developers, hampering creativity. The general principles of AD were summarised in the Agile Manifesto [50], a general guide that places people and software deliverables at the centre of the software development process, rather than more rigid and procedural processes that may lose the final objective of delivering high-quality software out of sight.

Here again, multiple variations for the definition of the AD paradigm as a software development process exist (cf. Merkow [52], Przybyłek and Morales–Trujillo [59], among others). A key feature of AD that distinguishes it from classical software development approaches is its iterative nature. Organising shorter "full cycle" phases (from requirements to delivered software), in each of which a smaller set of requirements are addressed, actually helps both parties: the stakeholders gain confidence in the developed software, which enables them to express their needs more precisely, while the developers deliver solid, well-tested pieces of the final product, responding quickly to new insights and updated needs. Selecting a feasible set of functionalities is crucial for the success of the so-called sprint phases: it is because the tasks are voluntarily reduced to covering meaningful, small increments in functionality, that it becomes possible to achieve a "full cycle" in a limited time.

For illustrative purposes, let us consider a generic Design activity that performs what is considered as a "full cycle" or Sprint. For each Sprint, a limited set of requirements needs to be selected from the complete set of requirement, thus capturing the stakeholders' priorities. The selected set must be small enough such that the sprint can be performed in a reasonable short time. Some variants of AD even require fixed-length sprints. At the end of the sprint, an assessment of the maturity of the requirements' fulfilment is performed, leading to a new evaluation of the priorities, thus entering a new sprint.

To formalise the key features of AD, one needs the means to again manipulate concepts at both the *syntactic* and *semantic* levels. Syntactically, we need to describe the notion of "activity" that takes as input (a subset of) the requirements, expressed in an appropriate formalism; and the control flow associated with the loop enclosing a sprint. Semantically, we need to ensure that any sprint execution is performed within some time limit.

In summary, in order to precisely formalise our paradigm P of choice, in this case, Agile Development, for which one characteristic property π is the fact that a sprint is performed in a reasonably short time, we consider two potential candidate elements $\mathcal{C}_1$, being the (simplified) SystemDesignPhase of the V-model, and $\mathcal{C}_2$, being a (simplified form of) SystemDesignExploration. For checking whether $\mathcal{C}_1$ and $\mathcal{C}_2$ qualify as agile development, we would need at least four kinds of formalisms:

1. A *structural* (paradigmatic) formalism for describing a *workflow* that enables distinguishing between control and artefact flows. Depending on the properties of interest characterising a given paradigm, this (paradigmatic) formalism may be used to capture patterns at both the syntactic and semantic level (i.e. over the execution traces of the paradigmatic workflow), since paradigmatic properties often concern both (as it is, for example, for the requirement that Agile loops span over short periods.

   Figure 2 (middle) depicts the (paradigmatic, structural) workflow associated with the AD key features using a UML Activity Diagram-like (our choice for the structural paradigmatic formalism listed above): the short ShortDesignActivity, contained in the Sprint activity, is a placeholder activity (note double-rounded rectangle used as a symbol, in contrast to the regular rounded rectangle in in UML Activity Diagrams)
2. In the context of Workflow specifications (cf. discussions in Sect. 6.4), candidates are usually already expressed in a given formalism. We sketch in Fig. 2 (top and bottom) (simplified versions of) parts of the V-Model development lifecycle and DesignSpaceExploration. We also use UML Activity Diagrams as a formalism to simplify the description.

   The upper part of Fig. 2 depicts a (simplified) SystemDesignPhase of the V-Model, with only requirements analysis and design activities shown (it is assumed that the design artefacts produced are executable and have been tested).
3. A *mapping* (paradigmatic) formalism for relating the structural (paradigmatic) formalism elements with a candidate formalism used for specifying the abstract syntax of potential candidate workflows: BPMN, UML Activity Diagrams, etc. Precisely defining this mapping formalism is out of this paper's scope; we only informally visualise it through the red dashed lines in Fig. 2.

   Although initially, a match of the SystemDesignPhase AD candidate seems possible (the dashed mapping arrows), it soon becomes obvious that the mapping cannot be completed as no control loop can be found in the SystemDesignPhase AD candidate. This comes as no surprise, as the essence of the V-model and its phases is its linear arrangement of activities. One may thus conclude that the V-Model's SystemDesignPhase does not qualify as Agile, since not even the syntactic components match.

   Consider now a multi-objective SystemDesignExploration process where many variants of a CPS may be explored, thus eliminating poor designs and keeping the ones that satisfy a set of global constraints to be further analysed against non-functional criteria such as performance, cost, power consumption, etc [48]. As the Agile *pattern* leaves the ShortDesignActivity unspecified, it will match any workflow candidate which contains, in the place of the ShortDesignActivity, a workflow that matches this activity's interface, and whose execution time qualifies as "short". As shown in Fig. 2, *substituting* a DesignSpaceExploration (DSE) workflow while respecting the appropriate "interface" for ShortDesignActivity guarantees acceptance as following AD.
4. A *property* (paradigmatic) formalism for specifying *properties* over the structural (paradigmatic) workflow, as well as an appropriate checking procedure to validate, via the mapping, that a candidate workflow satisfies the (paradigmatic) properties.

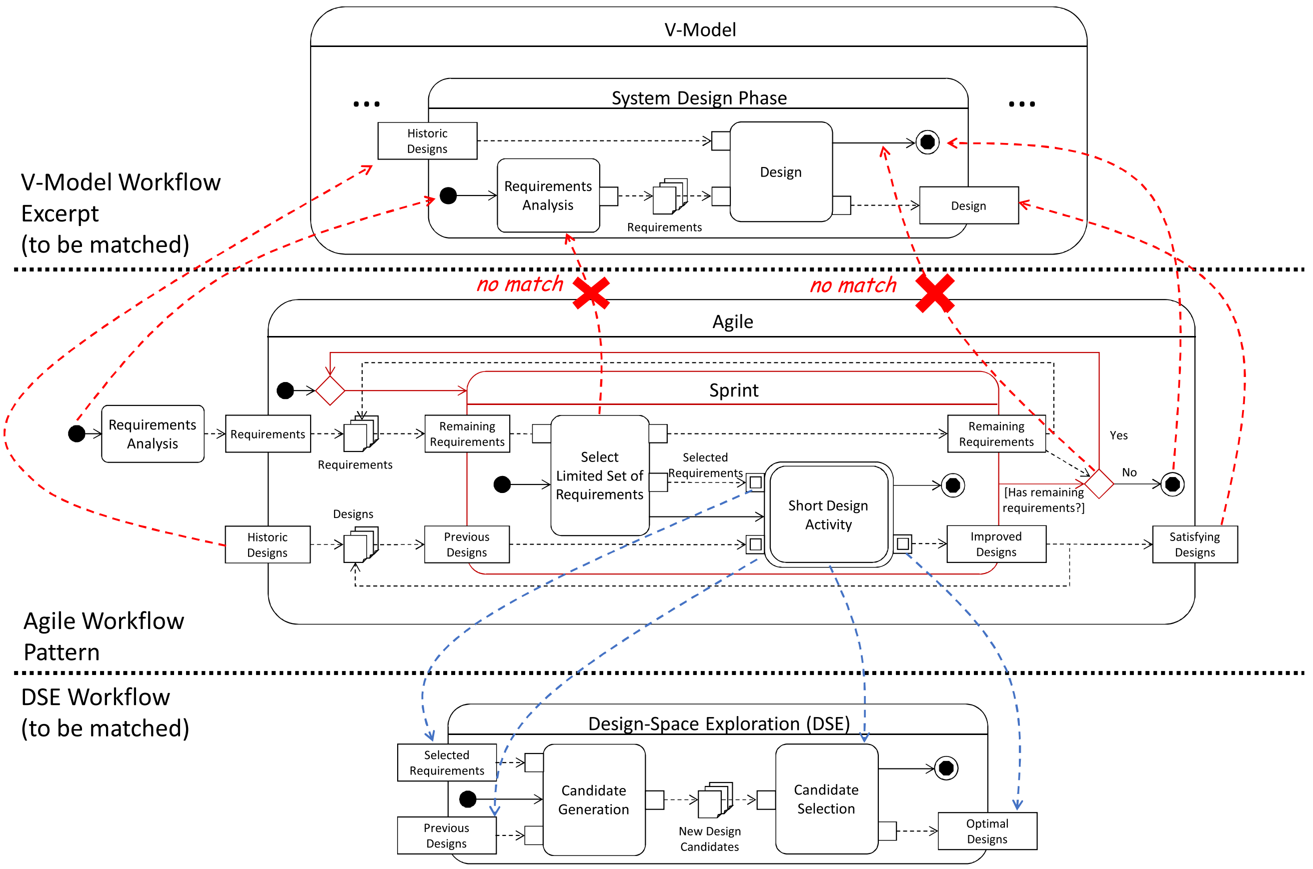


Fig. 2 A proposal for capturing the Agile development (AD) life cycle pattern, as a WorkflowPH in the middle. On the top, a representation of the SystemDesignPhase of the V-Model, which fails to fully match the AD pattern. The V-Model workflow essentially lacks a loop (so-called Sprint) that addresses a small, self-contained subset of requirements (it actually aims at the full set). On the bottom, a successful match between the AD workflow pattern and SystemDesignExploration if the latter is embedded in ShortDesignActivity

We would have to define a paradigmatic property for π in a formalism that would allow constraining the (potentially parameterisable) duration of the placeholder activity ShortDesignActivity in the Agile workflow pattern. Note that this property refers to the *trace semantics* of the structural paradigmatic formalism.

Design Space Exploration may in itself be characterised as a paradigm on its own, since it describes a characteristic way of producing valid, optimal designs that satisfy a selection of requirements. The use of patterns within the structural (paradigmatic) formalism (based here on UML Activity Diagrams) allows to easily describe *Agile* Design Space Exploration compositionally by separating the workflow for the Design Space Exploration paradigm from the one specifying its Agile nature. This leads to the notion of the proper combination of multiple paradigms: we further investigate one possible combination operator in Sect. 5.

3 A Descriptive Framework for capturing modelling paradigms

The complexity of (designing) CPSs is commonly addressed through complex *workflows*, involving models in a plethora of different *formalisms*, each with their own methods, techniques and tools, and combining particular *types of formalisms* and *operations* on models in these formalisms. MPM proposes to model everything explicitly, at the most appropriate level of abstraction, using the most appropriate modelling formalisms.

In the previous section, we offered a tutorial presentation and example of each constitutive element of a paradigm, as well as the intuition behind how to check whether a candidate qualifies as a paradigm's element: object orientation illustrated the *formalism* aspect with Java and Pascal as candidates. Agile development focused on the *workflow* aspect, with classical V-model and design space exploration life cycles. In this section, we go one step further and capture precisely, through a metamodel, the structuring elements of

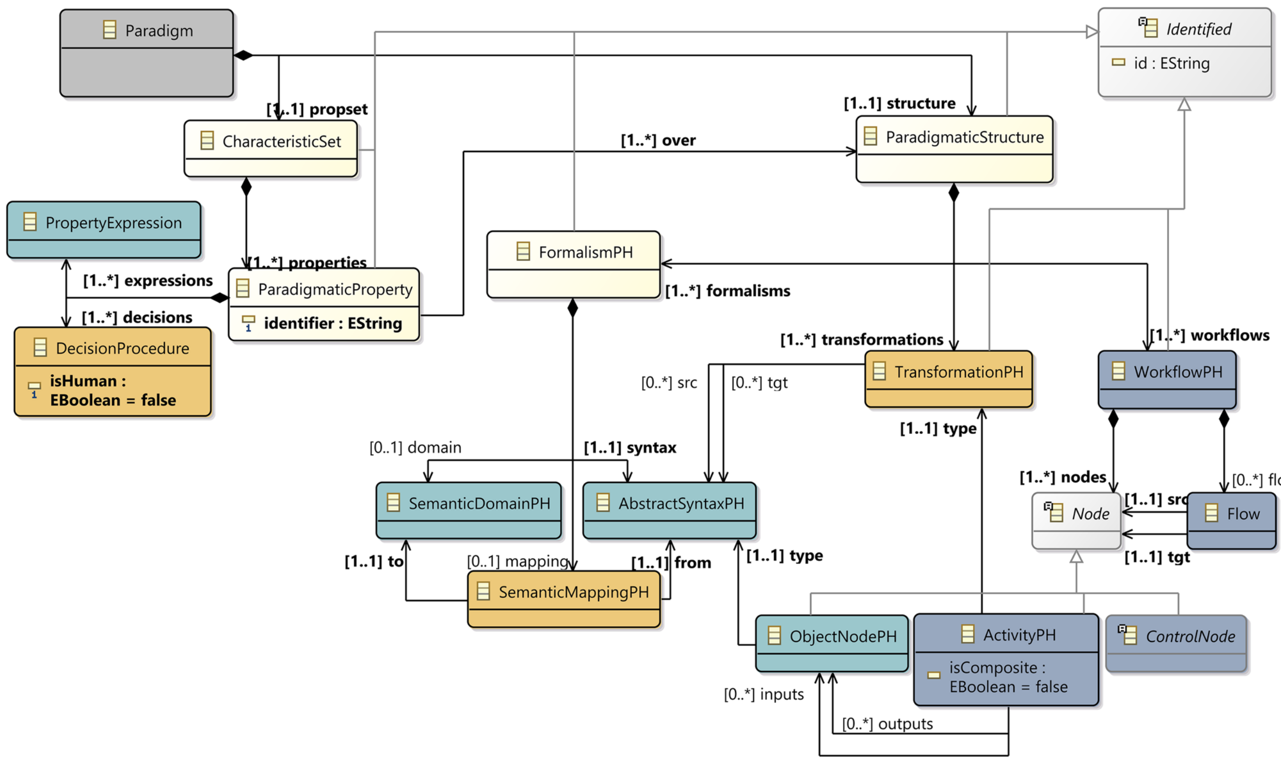


Fig. 3 A metamodel describing the concepts and structure of paradigms. A Paradigm is defined by a set of ParadigmaticPropertys that characterise components consisting of placeholders: FormalismPHs, TransformationPHs and WorkflowPHs

paradigms, namely *properties* over formalisms and workflows. This metamodel, as pictured in Fig. 3, as well as the general principle behind effectively checking whether a given candidate qualifies as, or follows a paradigm, as pictured in Fig. 4, constitutes together our Descriptive Framework for MPM.

From Fig. 3, a paradigm (name) denotes a set of ParadigmaticProperties that capture the essence of the intended paradigm. Many variations or combinations of those properties, grouped in CharacteristicSets, lead to conceptually different paradigms in our framework: for example, Object Orientation with single or multiple inheritance should be named differently. The necessary components of these properties are formally captured by a ParadigmaticStructure, which consists of three interrelated parts: a WorkflowPH capturing the dynamics of how appropriate elements are produced, consumed, and exchanged in an organised fashion within the paradigm, where both activities and objects are typed against TransformationPHs and FormalismPHs, respectively.

We describe in detail each component of the Descriptive Framework, before explaining how to use it concretely to check whether a candidate follows a given paradigm.

3.1 Paradigmatic properties

A ParadigmaticProperty is a property that captures one aspect of the paradigm's essence that is shared by all artefacts that follow it. In other words, such a property is defined "universally" at the level of the paradigm and holds for all artefacts following this paradigm. To check whether it holds or not, a ParadigmaticProperty defines explicitly a DecisionProcedure, which may be automated, or performed by a human (or any combination of both): it may be a mathematical proof, or it may be so difficult to prove that only an agreement among those interested in the Paradigm may be feasible and provide the decision. When all paradigmatic properties are checked to be valid, the artefact then becomes an artefact that qualifies as, or follows, the corresponding paradigm.

3.2 Paradigmatic structure

For a ParadigmaticProperty to be expressed (formally or not), a paradigm needs to define a minimal structure that captures the vocabulary, the concepts and their relationships that the property is about. A ParadigmaticProperty is applied over a ParadigmaticStructure, which is composed of one (or several) workflow(s) with placeholders (WorkflowPH); one (or several) formalism(s) with placeholders (FormalismPH), or one (or several) transformation(s) with placeholders (TransformationPH).

A WorkflowPH links activities with placeholders (ActivityPH) and their object nodes with placeholders (ObjectNodePH) in various ways (sequential or concurrent), described as flows driven by ControlNodes (at this abstraction level, there is no need to distinguish between so-called object and control flows).

A TransformationPH types an ActivityPH by defining a *signature*, i.e. which source(s) and target(s) placeholder formalisms (FormalismPH) the placeholder transformation operates on. We require TransformationPHs to be at least *terminating* (since they are combined in so-called transformation chains [25], they shall always produce outputs), or to *fail* when inputs are not conforming to their source FormalismPH.

A FormalismPH shall at least define, through an AbstractSyntaxPH, the expected structure supporting a ParadigmaticProperty; it may eventually specify a (partial) semantic specification through a SemanticMappingPH that maps elements from the AbstractSyntaxPH to an appropriate SemanticDomainPH. All three of them contain placeholders (as illustrated in Fig. 1 for the inheritance property in OO), allowing arbitrary precision for a ParadigmProperty.

As an example, Fig. 1 describes (part of) the supporting FormalismPH and PropertyExpression for expressing the *inheritance* ParadigmaticProperty, as part of the characteristic set for the Object Orientation paradigm

Note that in this example, we expressed the structure supporting the Inheritance property, and the property itself, in specific formalisms: for the structural part, we selected a MOF-like formalism; for the property, we naturally turned to OCL as it is a standard, and expressive enough for capturing our property of interest. To obtain an explicit specification, many languages of our descriptive framework need to be expressed as valid models of an appropriate formalism. In Fig. 3, we denote by light blue *structural* formalisms (e.g. BNF/Graph Grammars, metamodels, Entity/Relations, or any other suitable ones), and in orange *behavioural* formalisms (GPLs, transformation languages, graph transformations, and so on). Note that both need to be extended to capture *patterns* over candidates (as we suggested and demonstrated using placeholders for the pattern mechanism).

Although the activities comprising a WorkflowPH may be combined freely using ControlNodes and Flows, we require the following conditions to hold, for a WorkflowPH to be well-defined:

– Each ActivityPH is *typed* by a TransformationPH appearing in the same ParadigmaticStructure;
– Each ObjectNode used as input or output of an ActivityPH is *typed* by a FormalismPH, so that
– the type(s) of the ObjectNodes used as input and output of the ActivityPH match the signature of the TransformationPH that types the ActivityPH.

Similar to the structural and behavioural formalisms required for other various components of a ParadigmaticStructure, the elements comprising a WorkflowPH and coloured in dark blue may be part of a (richer) formalism dedicated to the description of workflows (such as UML Activity Diagrams, Business Process Models, etc.); the only constraint is that the Node and Flow concepts are in that formalism. In this paper, we choose Activity Diagrams for this purpose (cf. Sect. 6.4 for a discussion).

As already noticed, our Descriptive Framework admits as valid paradigms definitions that are restricted:

– to only FormalismPH: we assume in this case that there always exists a *default* associated WorkflowPH that allows creating appropriate instances of the formalism it is matched to; or
– to only WorkflowPH: we assume in this case that there exists a *generic, default* FormalismPH that is used by one of the ActivityPH defined inside the ProcessPHs.

This is precisely the case for the examples given in Sect. 2.2, making them valid paradigm definitions in our framework (assuming all ParadigmaticPropertys are effectively specified).

3.3 Checking whether a candidate follows a paradigm

A typical usage for our Descriptive Framework is checking whether a Candidate artefact indeed follows a given paradigm. A candidate is structurally similar to a paradigm's ParadigmaticStructure, with the fundamental difference that components are not merely placeholders anymore. A Candidate may exhibit arbitrarily complex components: the Formalisms may have complicated, intricate syntax and semantics; and the Workflows and associated Transformations (chains) may describe large real-life (industrial, or conceptual) processes related to CPS engineering.

Conceptually, checking that a ParadigmaticProperty holds on a Candidate requires the definition of a Mapping that binds (all) placeholders appearing in the property to the constituents of the Candidate. A mapping may be arbitrarily complex: the languages (metamodels) defining the Candidate may differ radically from the ones used for specifying the ParadigmaticProperty; the semantics of a Candidate may be expressed candidate may be expressed in a different "style" (it is certainly operational for the Candidate in order for it to be executable, while the FormalismPH may use an axiomatic definition to provide constraints over the

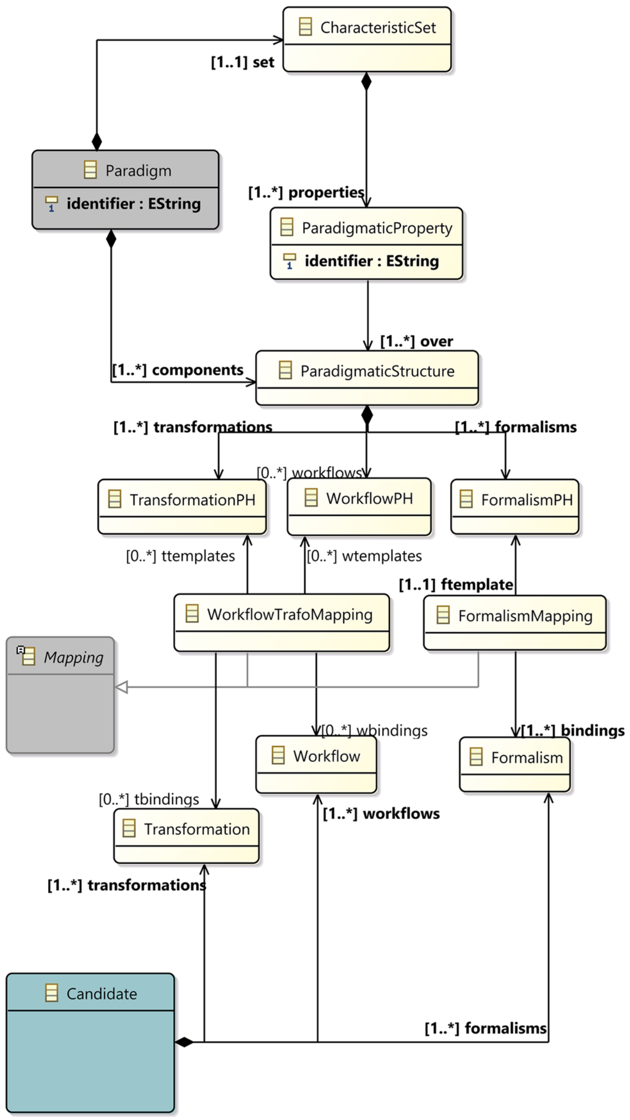


Fig. 4 Checking whether a Candidate follows a given paradigm through a mapping that binds all placeholders in the ParadigmaticStructure's components with concrete elements constituting the Candidate, then applying all DecisionProcedure

semantic domain); and a ParadigmaticProperty may operate at various levels at the same time (syntactic and semantic, just like the inheritance property for OO), making the Mapping sensitive to implementation details. Formally speaking, checking the validity of a ParadigmaticProperty consists of invoking the DecisionProcedure *over* the components the properties apply to (either Transformations, Workflows or Formalisms). but via the Mappings. Note that we define a WorkflowTrafoMapping referencing both the TransformationPH and the WorkflowPH, because candidates may abstract away or refine some parts in the other (i.e. a complex Workflow Placeholder may be realised through a transformation delegated to an external tool, which is then perceived from the Candidate viewpoint as a black box without further control on the internals, thus preventing matching to explicit placeholders).

Formally proving all of the paradigmatic properties required for a candidate to follow a given paradigm may prove extremely tedious, assuming Mappings are actually available. This explains why we expect that the DecisionProcedures associated with a ParadigmaticProperty may well be conducted by humans to overcome this difficult task. Furthermore, as described in the previous section, the formalism choices for expressing the required elements of the Descriptive Framework introduce another burden for performing the proof: as an illustration, if a Candidate for the Object Orientation paradigm captures the Formalism using a different formalism language than the ones we used in Fig. 1, then checking that the Inheritance ParadigmaticProperty holds requires not only a Mapping but additionally an *equivalence* proof between formalisms.

3.4 Final remarks

From our point of view, CPS engineering has largely undervalued the importance of *workflows* in the engineering process. Although manipulating various artefacts (which corresponds to the ActivityPH in our Descriptive Framework, as part of the overall WorkflowPH) is *de facto* a core concern, we believe that explicitly representing *how*, *when*, and to *which purpose* those artefacts interact with each other towards the greater goal of reaching an end product is a crucial part for ensuring deeper understanding of the methodologies and construction processes, but also promotes reuse and adaptation to new constraints. Making workflow pattern descriptions an integral part of our Descriptive Framework is a first step towards recognising this fact and also enables support for the underlying activities with adequate tooling at the level of paradigms (just the way it is for other engineering disciplines, as emphasised, e.g. by Pahl et al. [58] for mechanical engineering).

In our framework, nothing prevents a candidate from being involved in several mappings, allowing it to qualify as various paradigms. As an example, Java, our witness candidate for the *object-oriented* paradigm in Sect. 2.2, may well qualify as an object-oriented, but also as a concurrent GPL, assuming one can provide a proper property characterisation of what concurrency for imperative programming languages may look like. As a consequence, the Mapping component of our Descriptive Framework needs to be separated from the potential candidates; if not for conceptual reasons (as above) then for legacy reasons, because often paradigm elements are built without thinking much of the paradigm they belong, but rather on which kind of issues the element is intended to solve.

When describing informally the kind of formalisms required for capturing the nature of a paradigm, we referred to "*supportive formalisms*" to designate the so-called *meta*-formalisms, i.e. the formalisms in which the listed formalisms (paradigmatic structural, mapping and property, but also the candidate's formalism(s) themselves) are expressed in. In our conceptual metamodel of Fig. 3, we further classified them into two categories: *structural* metaformalisms in blue, which describe structures, and *behavioural* metaformalisms in yellow, which describe computations. We also showed in our tutorial examples from Sect. 2.2 that it is often the case that already existing formalisms may be extended to provide adequate pattern languages for capturing the various components of our descriptive framework (namely ParadigmaticProperty, FormalismPH, TransformationPH and WorkflowPH): we used an extension of UML Class Diagrams and Activity Diagrams to convey the idea of "patterns" that need to be filled by elements of a potential candidate (note that the exact specification and semantics of such extended *placeholder* formalisms remains future work).

We believe that many formalisms may be suitable to be promoted as *pattern/placeholder* formalisms for capturing paradigms' properties when considering suitable research on model typing [17,62,63]. The nature of the relationship between the paradigm's "patterns" and the candidate's matched elements are different from the classical class/instance relationship, since a whole submodel may be matched into a single placeholder. Such "extended" pattern/placeholder languages may be partially obtained through semi-automated processes (e.g. RAMification (Kühne et al [43])), but a precise (semantic) design, specification, and matching process of such languages is left as future work.

The diversity of supporting formalisms gives rise to two crucial, and related, issues:

1. Having different choices for supportive formalisms for the paradigm and a potential candidate requires either that extra effort is put to *translate* one of them (typically, the candidate, which may be defined in various forms) into an appropriate formalism, or to perform mathematical equivalence (or rather, simulation) proofs in order to appropriately match elements. For simplification purposes, we stick to supportive formalisms (UML Class Diagrams and Class Diagrams with placeholders; and Activity Diagrams and Activity Diagrams with placeholders) that correspond to the ones used for potential candidate, to avoid another level of complication; but in practice, this may happen often.
2. Similarly, having different choices impacts the decision procedure, since the paradigmatic properties, as well as the matchings, rely on the paradigm's supportive formalisms. The decision procedure may be seen as a procedure *modulo* the formalisms: here again, equivalence proofs taking into account both the supporting formalisms and the properties are necessary to prove we are manipulating the "same" paradigm.

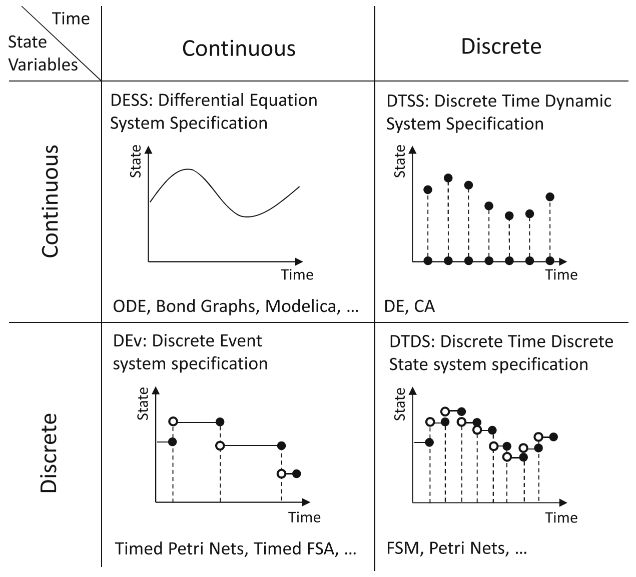

Time
State Variables
Continuous
Discrete
Continuous
DESS: Differential Equation System Specification
State
Time
ODE, Bond Graphs, Modelica, ...
DTSS: Discrete Time Dynamic System Specification
State
Time
DE, CA
Discrete
DEv: Discrete Event system specification
State
Time
Timed Petri Nets, Timed FSA, ...
DTDS: Discrete Time Discrete State system specification
State
Time
FSM, Petri Nets, ...

Fig. 5 Classification of modelling abstractions for dynamic systems according to the nature of the *time* and *state* variables [72,79]

4 Two paradigms for CPS: discrete event dynamic systems and synchronous data flow

This section presents two compact examples of paradigms relevant to the engineering of CPSs that have been selected and abstracted to illustrate the concepts of MPM that we strive to convey.

Among the many classifications for CPS modelling abstractions and associated formalisms (cf. Sect. 1 for a quick survey), the simplest and most widespread ones are based on the nature of the representations of the characteristic quantities of a CPS: the *time base* over which the CPS evolves and the *state variables*. Both quantities may be *continuous*, i.e. their domains range over dense domains (such as reals), or *discrete*, i.e. they range over discrete, enumerable domains (such as integers).

Taking a helicopter view, the behaviour of a CPS may be seen as a trajectory that depicts the evolution of state variables over time, which are falling into one of the following categories (cf. Fig. 5 and [79]):

Continuous Variables/Continuous Time leads to complex Differential Equations System Specifications (DESS) where the *constituent* relationships between quantities are captured in the form of differential algebraic equations. Such specifications often require numerical solvers to obtain approximate solutions on digital computers. Typical realisations of this paradigm are Ordinary Differential Equations, Bond Graphs, Equation-Based Object-Oriented Languages such as Modelica, and Analog Electrical Circuit Diagrams.

Continuous Variables/Discrete Time leads to Discrete Time System Specifications (DTSS). These are for example used in sampled system models, representing data periodically obtained from a physical system through sensors. Typical realisations of this paradigm are Difference Equations (DE), and Cellular Automata (CA).

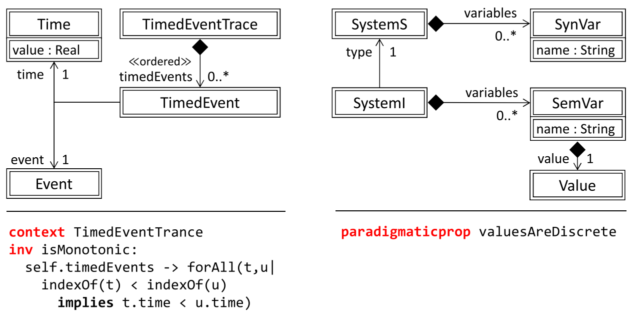
Discrete Variables/Continuous Time leads to Discrete Event dynamic system specifications (DEv). Discrete Event specifications start from the insight that discrete state changes only occur at times of pertinent "events". In between those events, the state does not change and the state trajectory is hence piecewise constant. In a finite time interval, only a finite number of events may occur. Typical realisations of this paradigm are Timed Finite State Machines, Event Graphs and the Discrete Event System Specification (Ziegler's DEVS Formalism [79] which, though Discrete Event, does permit a continuous state space).

Discrete Variables/Discrete Time The other end of the spectrum leads to Discrete Event System Specificatio (DTDS) where discrete state changes only occur at equidistant times. Typical realisations of this paradigm are State Machines.

This section presents the *Discrete Event dynamic systems specification* (abbreviated as *DEv*) and Synchronous Data Flow (abbreviated as SDF) paradigms. We describe both in details within our Descriptive Framework. This choice is guided by three criteria. First, we have selected systems that have opposite natures for the characteristic variables. Second, they are simple enough to convey the necessary concepts for illustrating our Descriptive Framework, while serving as a basis for generalisation to more elaborate CPS models. Third, the combination of those paradigms covers a large spectrum of CPS models used in practice, making them illustrative of the various combinations that exist.

Each paradigm is described systematically using the following approach:

1. We first capture the general requirements from a well-known source that informally describes the paradigm;
2. We translate these requirements within our Descriptive Framework (cf. Fig. 3), using appropriate formalisms;
3. We then present a potential Candidate, specifying its various components (Formalisms, Transformations and Workflows) to a certain extent.
4. We finally apply the checking scenario of Sect. 3.3: we show how Mappings may be (informally) defined, validating that the Candidate indeed follows the paradigm mentioned above.



```
context TimedEventTrance
inv isMonotonic:
  self.timedEvents -> forAll(t,u|
    indexOf(t) < indexOf(u)
      implies t.time < u.time)
```

```
paradigmaticprop valuesAreDiscrete
```

Fig. 6 FormalismPHs and ParadigmaticProperties for the specification of the *DEv* paradigm

4.1 Discrete Event dynamic systems (DEv) paradigm

The *discrete event dynamic system* paradigm uses discrete state variables with continuous time. We illustrate it with the *Timed Finite State Automata* [22].

4.1.1 Paradigm description

From the previous categorisation, we summarise the relevant properties of the *DEv* paradigm and express them in our Descriptive Framework, as depicted in Fig. 6

– The time is continuous: the FormalismPH Time mandates the use of real values for elements matched with Time.
– The system's *dynamics* is captured through timed events: the FormalismPH TimedEventTrace expresses the fact that some elements may be considered as Events that occur at specific time occurrences; the ParadigmaticProperty `isMonotonic` (as expressed in pseudo-OCL) ensures that Events occur at monotonically increasing timestamps.
– The system's (dynamic) state is composed of variables that range over discrete domains: two FormalismPHs describe system specifications (SystemS) and instances (SystemI). A SystemSpecification describes dynamic systems at a high abstraction level, assuming only the declaration of variables (SynVar), while a SystemInstance imposes that variables (SemVar) have values, together with a ParadigmaticProperty that enforces values are actually discrete.

To simplify the description of the *DEv* paradigm, we only consider one fundamental TransformationPH, named Execute, with a trivial WorkflowPH that allows executing the system assuming a given trace.

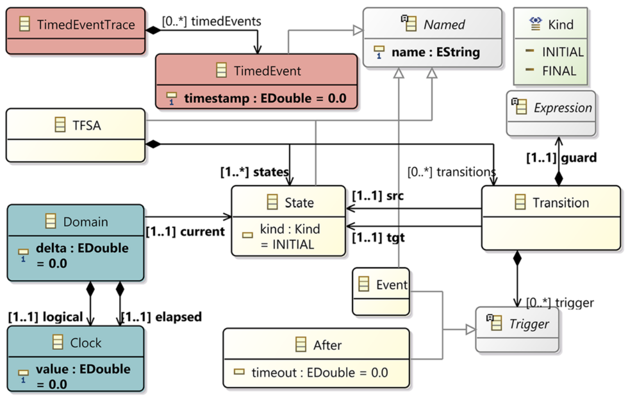


Fig. 7 Metamodels for specifying a TFSA (from Class TFSA); its semantic domain (Class Domain) for accepting a (finite) TimedEvent-Trace

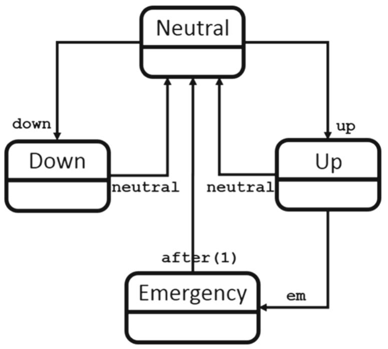


Fig. 8 A simple TFSA conforming to the TFSA domain metamodel of Fig. 7

4.1.2 Candidate: timed finite state automata (TFSA)

When augmented with time constraints, Timed Finite State Automata (TFSA) are powerful formal models, suitable for describing engineered and natural systems in various application domains, which range from sequential circuits, communication protocols, reactive and biological systems. We describe here a simplified *conceptual* formalism for TFSA that may represent concrete implementations in various tools.

Figure 7 describes the TFSA formalism. A TFSA is a Finite State Automaton with an INITIAL and some FINAL states interconnected by Transitions. A TimedEventTrace is a finite list of TimedEvent, consisting of a pair of timestamped event names. A Transition may fire when its Trigger occurs, assuming its guard evaluates to true (the Expression language is left unspecified, as it is not necessary for understanding). When there is an Event Trigger, it should match the current TimedEvent; otherwise, when the Trigger is an After, the Transition fires only when the associated timeout has elapsed, when no other TimedEvent occurs before. The TFSA formalism defines a semantic Domain (also called *configuration*) for specifying an accepting behaviour, provided a specific finite TimedEventTrace: a TFSA accepts a trace iff consuming the TimedEvents composing the trace, in order, results in a FINAL State. The Domain references the current State within the TFSA and manipulates two Clocks: a logical one that records the global time elapse; and a clock used for tracking the elapsed time locally to a State.

Figure 8 shows a simple TFSA that models the behaviour of a (simplified) car Power Window [56] equipped with a three-position command button: when pressed up or down, it indicates the window should move in the appropriate direction; when released, the button produces the neutral event. For safety reasons, when a force is detected resisting the window moving up, the system produces an emergency event, bringing the system into the Emergency mode: after one millisecond, the window stops moving, allowing whatever is obstructing the upward movement to be removed safely.

Listing 1 specifies a procedure execute capturing the behavioural semantics of a TFSA. It operates on a(n instance of a) Domain, assuming a(n instance of a) TFSA and a given (instance of a finite) TimedEventTrace, and proceeds as follows:

Initialise During this phase (Lines 5–7), the various time and state variables are set, pointing the currentState pointing the current State of the computation to the (unique) INITIAL State in the TFSA.

Check Stopping Conditions A loop captures the computation, which runs until no new TimedEvent (Line 9) is present within the given TimedTraceEvent tet, after the currentState is compared to the list of FINAL State of the TFSA.

Perform Step A computation step (Lines 10 – 25) depends on the list of outgoing Transitions of the currentState:

– If an Event Transition labelled with the same name as the current TimeEvent te exists, the Transition is fired (*must*-semantics), changing the currentState to the Transition's tgt;

– Otherwise, if an After Transition is present, it is fired assuming it already reached its timeout (i.e. timeout $\leq$ elapsed). After that, a discrete time step is taken, incrementing both clocks (elapsed and logical) by the predefined delta).

Terminate It remains to check (Line 27) whether the currentState at the end of the computation is a FINAL State.

While explaining the behavioural semantics, we explicitly distinguished separate activities whose dynamics are captured in the Activity Diagram of Fig. 9.

```
procedure execute(d    : Domain,
                  tfsa : TFSA,
                  tet  : TimedEventTrace)
do
    d.logical.value = d.elapsed.value = 0
    d.current = tfsa.getInitialState()
    currentState = d.current

    foreach(tevent : tet.timedEvents) do
        outs = tfsa.outgoingTransitions(currentState)
        transition = outs.filter[Event]
            .find[name = tevent.name]
        if(transition != null) then
            currentState = transition.tgt
            d.elapsed.value = 0
        else
            transition = outs.filter[After]
            if(transition != null &&
                transition.timeout <= d.elapsed) then
                currentState = transition.tgt
                d.elapsed.value = 0
            endif
        endif
        d.logical.value += d.delta
        d.elapsed.value += d.delta
    endfor
    return tfsa.getFinalStates().contains(currentState)
endprocedure
```

Listing 1 Algorithmic for the Execute transformation, specifying the behavioural semantics for TFSA.

4.1.3 Mapping

We briefly discuss how to (partially) build the Mapping between the ParadigmaticStructure defining our *DEv* paradigm and the components of our TFSA Candidate, as an instance of the metamodel defined in Fig. 3.

First, the TimedEventTrace metamodel in Fig. 7 maps directly to the TimedEventTrace FormalismPH of Fig. 6: names were kept identical on purpose, since TimedTraceEvents are a rather simple collection structure.

Second, the SystemSpecification may correspond to the TFSA concept, assuming the rest binds appropriately. As state variables for TFSA, which are required by a ParadigmaticProperty to be discrete, we may bind the State concept. As it occurs for TFSA, the class State appears both as a component for the class TFSA, which is matched to SystemS, and as an element in the semantic Domain, which should therefore be bound to SystemI. Since the number of States is always finite (the usual meaning of the "*" in the states reference), it defines a *discrete* domain, thereby validating the ParadigmaticProperty.

Third, the execute procedure presented in Listing 1 maps in a straightforward way to the trivial WorfkflowPH containing the Execute TransformationPH mentioned at the end of Sect. 4.2.2.

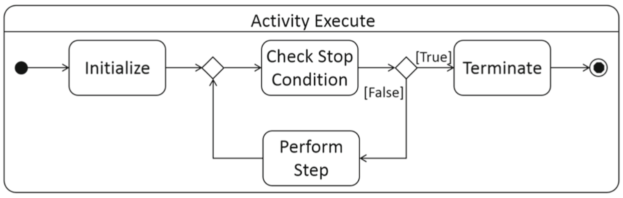


Fig. 9 **Activity Diagram** capturing the dynamics of the activities composing the behavioural semantics common to a TFSA (Listing 1) and a CBD (Listing 2)

4.2 Synchronous Data Flow (SDF) paradigm

The **Synchronous Data Flow** paradigm uses continuous time and state variables, and is illustrated with **Causal Block Diagram** , a formalism representative for many tools such as SIMULINK and SCADE.

4.2.1 Presentation

The Data Flow paradigm [74] describes computations as a special directed graph, with the following features:

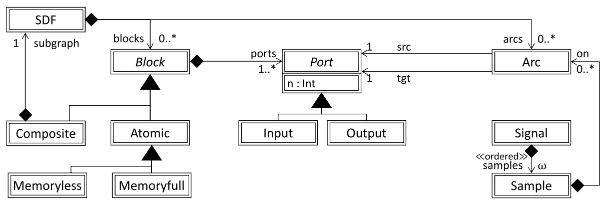
Signals represent infinite streams of data, where each data piece is called a *sample*.
Nodes also called *blocks*, represent computation units that *execute* (or *fire*) whenever enough input data become available. Blocks without input can fire at any time. Nodes may be atomic, i.e. performing basic computations (such as adders or multipliers), or composite, thereby encapsulating themselves a subgraph.
Arcs connect nodes, thus describing how data streams flow throughout the computation blocks.

Executing a Data Flow graph consists of accumulating enough samples within the system, produced by blocks without inputs, and performing the computations within the blocks, thus consuming a number of samples on each input and producing samples on all outputs in a concurrent way. Samples may be reused within the system (for example, in case of cycles) to be used as old samples Messerschmitt [54], but they will not be considered as new once consumed.

The synchronous data flow paradigm [47] is a specialisation of the data flow paradigm where all blocks appearing in a data flow graph are required to be *synchronous*, i.e. each block explicitly defines how many samples are consumed and produced.

4.2.2 Paradigm description

The previous description leads to the following proposal in our Descriptive Framework, as illustrated in Fig. 10:



```
context Arc inv noShortcut:
  self.src.oclIsTypeOf(Input) implies
  not self.tgt.oclIsTypeOf(Output)
```

Fig. 10 FormalismPHs and ParadigmaticProperties for the specification of the *SDF* paradigm (the plain arrow denotes inheritance over placeholder classes)

– Signals are composed of an infinite, ordered stream of Samples (note the ω multiplicity denoting a collection with an infinite, dynamic number of elements, as suggested by Combemale et al. [14]).
– An SDF has the structure of a *directed* graph with Arcs and Blocks as nodes.
– Blocks possess Ports that explicitly define how many Samples are used (consumed by Inputs, or produced by Outputs).
– Arcs connect Ports, and flow Signals that travel on them instantaneously. Note that a Port may be plugged to several Arcs; only shortcuts are prevented by the noShortcut ParadigmaticProperty, which forbids Arcs to connect as src and tgt Ports of the same Type.
– A memoryfull Block should always define an extra Port corresponding to initial conditions.

To simplify the description of the SDF paradigm, we only consider one fundamental TransformationPH, named Execute, with a trivial WorkflowPH that allows executing the system assuming valid inputs.

4.2.3 Causal block diagrams CBDs

Viewing a CPS as a set of interacting components that may be further decomposed is a natural and intuitive way for breaking its internal complexity. Because they offer an intuitive graphical description in terms of interconnected nodes, Causal Block Diagrams (CBDs) represent a natural formalism for capturing the dynamics of CPSs in a so-called feedback control loop: the evolution of a physical plant is monitored through sensors (thereby introducing a time discretisation), which provide a data stream constantly monitored and analysed by a software that influences back the software plant through actuators. CBDs come in different flavours, depending on the type of blocks that are available for describing a system [20,27]:

– *Algebraic* CBDs only expose mathematical computation blocks (over integers and **boolean** data flows). There is no time progression. They may describe steady-state CPSs occurring once the system has reached a steady state (e.g. an engine after its transition phase);
– *Discrete Time* CBDs extend algebraic CBDs with blocks that introduce *delay*, forcing all algebraic blocks to update their output streams whenever the delay is evaluated. They naturally describe discrete time dynamic systems.
– *Continuous Time* CBDs also extend CBDs, but in a different way: instead of introducing a time step notion with a delay, it extends algebraic CBDs with continuous time, using the mathematical integration and derivative operators. Although theoretically more powerful and more complex than the previous CBD class, they are still suitable for dynamic systems but require numerical discretisation.

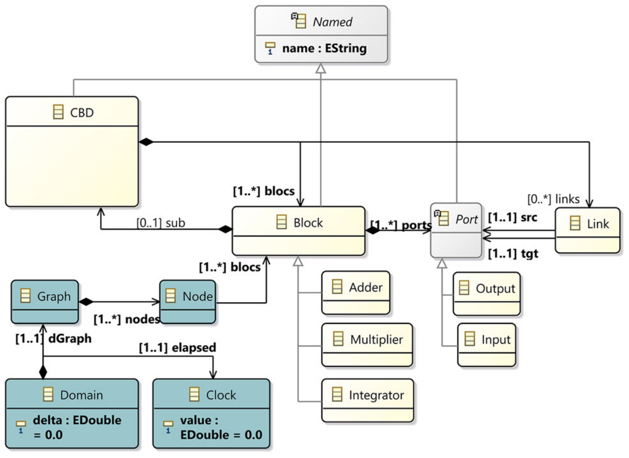


Fig. 11 Metamodels for specifying a CBD (from Class CBD) and its semantic domain (Class Domain) for executing it

CBDs have strong mathematical foundations and largely leverage recent advances in numerical solvers, making their use widespread within several tools (e.g. MathWorks' MATLAB/SimuLink; ANSYS/Esterel SCADE, to only name the most renowned ones). Without loss of generality, and to simplify the presentation, we will also consider the SDF paradigm a *conceptual* formalism for Continuous Time CBD that may be part of concrete tool implementations.

Figure 11 describes the CBD formalism. A CBD is composed of Blocks that possess a number of InputPorts and OutputPorts. Those Ports are Linked appropriately (i.e. a Link connects an output to an input). To simplify the presentation, we only consider three kinds of Blocks: an Adder and a Multiplier (which are both Memoryless) and an Integrator (which is Memoryfull). The semantic Domain for executing a CBD consists of a time step delta, and a dependency Graph (edges are not explicitly represented here) whose Nodes aggregate those Blocks that are cyclically interdependent.

```
 1 procedure execute(d   : Domain, cbd : CBD)
 2 do
 3   d.dGraph =
 4       cbd.computeDependencyGraphWithStrongComponents()
 5   d.logical.time = 0
 6   while(not end_condition) do
 7     foreach (scomponent : d.dGraph.nodes) do
 8       if(scomponent.size() = 1) then
 9         scomponent.nodes.pop().compute()
10       else — Strong Component: compute the whole cycle!
11         scomponent.nodes.compute()
12       endif
13     endfor
14   endwhile
15   d.logical.time += d.delta
16 endprocedure
```

Listing 2 Algorithmic for the Execute transformation, specifying the behavioural semantics for CBD.

Listing 2 describes a procedure execute for capturing the behavioural semantics of a CBD. It operates on a(n instance of a) Domain and a(n instance of a) CBD, and proceeds as follows [27]:

Initialise During this phase (Lines 3–5), the various time and state variables are set: the logical clock is initialised, and the dependency graph with strong components is computed.

Check Stopping Conditions The stopping condition is provided by the user (captured by the end_condition predicate in Line 6) since a CBD computes values at each time step.

Perform Step This step consists of iterating over each Block, in the order of the dependency graph. The (overloaded) Compute procedure depends on the nature of the Block:

- Memoryless A memoryless Block (such as our Adder or Multiplier) simply applies a stepwise basic operation (here, an arithmetic one) on the Samples available on the Input Port, and delivers the result on the Output Port.
- Memoryfull Blocks are split in two categories: a *delay* performs a discrete operation based on previous values of Inputs, thus requiring memory to store such values, while an *accumulator* (like our Integrator) performs an approximation of a continuous behaviour by accumulating the Input (cf. Gomes et al [27] for a detailed explanation; cf. Burden and Faires [9] for details on how numerical approximations may be used for these Blocks).

After having completed the computation of all Block, the logical clock progresses by a delta step value.

The execute procedure may be described as an activity diagram in a similar way as TFSA were, as depicted in Fig. 9. Note that for CBD, the terminate activity is, in fact, empty.

4.2.4 Mapping

Some of the Mappings between the ParadigmaticStructure defining our SDF paradigm and the components of our CBD Candidate are almost straightforward: the CBD metamodel is similar to the FormalismPH for SDF, aside from renaming (e.g. Link trivially binds to Arc), and tagging the proposed Block appropriately (Adder and Multiplier are MemoryLess, while Integrator is MemoryFull). Each Block consumes and produces exactly one Sample on each of its Input and Output Port (assuming the value on the extra Input of MemoryFull Blocks for initial conditions does not change). Note that the timestep in a CBD is implicit, as no syntactic element manipulates it directly. Rather, the timestep corresponds to an evaluation of the full CBD (as shown by the execute procedure, where the time progresses after each full iteration).

Note that the execute procedure described in Listing 2 trivially matches the Execute TransformationPH required in Sect. 4.2.2.

5 Multi-paradigm modelling: combining paradigms

Since CPSs combine physical phenomena with logical decision making, mostly implemented in software, modelling their complex behaviour requires the use of a combination of continuous time models to capture the physical aspects, with discrete time and discrete event models to represent logical computations. Depending on the level of abstraction used, the networking part of CPS may be modelled using either type of models. Furthermore, for many complex CPSs, in order to address the diverse concerns stakeholders may have, complexity is tackled through orthogonal, yet complementary viewpoints. Not only the individual views need to be modelled explicitly, but above all, their often complex interactions and integration.

This section starts by presenting some general mechanisms in engineering that govern the design of a complex CPS. It then proceeds to precisely define one example MPM combinator, namely *embedding*, before applying it to our two CPS-oriented candidates, namely TFSA for the Discrete Event Dynamic System paradigm, and CBD for the synchronous **Synchronous** Data Flow paradigm.

We are aware that embedding is just one of the many combinators applicable to formalisms and workflows, such as extension, unification or self-extension [23], merging [19], and aggregation [36]. However, embedding is popular in practice, and simple enough for us to explain our paradigm combinator concepts concisely. Future work will investigate other paradigm combinators.

5.1 General mechanisms for tackling complexity

Benveniste et al. [5] argue that three basic mechanisms, namely *model abstraction/refinement*, *architectural decomposition* and *view decomposition/merge*, are sufficient to describe any complex CPS engineering effort. In our descriptive framework, these mechanisms may be captured by a combination of TransformationPHs and/or WorkflowPHs, depending on the available machinery, the granularity at which a design needs to be tackled at any point of the CPS engineering life cycle, and the details different engineers need to know about the complete CPS. At this point, it is still not clear whether these mechanisms may themselves be considered as paradigms on their own, or as relationships that paradigms may leverage to capture complex engineering processes (in a similar way to operations over the algebraic structure of paradigm). We simply describe them succinctly, leaving their integration as an extension of our Descriptive Framework.

5.1.1 Model abstraction/refinement

Model abstraction (and its dual, refinement) is used when focusing on a particular set of *properties* of interest. A relationship A between a detailed model m_d and a more abstract model m_a is an *abstraction* with respect to a *set of properties* Π, iff for all properties $\pi \in \Pi$, the satisfaction of π by the more abstract m_a implies the satisfaction of π by the more detailed m_d. This allows one to *substitute* m_d by m_a whenever questions about the properties in Π need to be answered. Substitution is useful as the analysis of properties on the more detailed model is usually more costly than on the abstracted model. Note that the abstraction relationship may hold between models in the same or in different formalisms, as long as for both, the semantics allows for the evaluation of the same properties. When modelling physical systems, continuous domains are frequently used. In that case, a more relaxed notion of substitutability based on *approximation* may be appropriate.

5.1.2 Architectural decomposition/component composition

Architectural decomposition (and its dual, component composition) is used when the problem can be broken into parts, each with an appropriate *interface*. Such an encapsulation reduces a problem to (i) a number of sub-problems, each requiring the satisfaction of its own properties, and each leading to the design of a component and (ii) the design of an appropriate architecture connecting the components in such a way that the composition satisfies the original required properties. Such a breakdown often comes naturally at some levels of abstraction, using appropriate formalisms (which support hierarchy). This may occur when the problem/solution domain exhibits locality or continuity properties. Note that the component models may again be described in different formalisms, as long their interfaces match and the multi-formalism composition has a precise semantics.

5.1.3 View decomposition/merge

View decomposition (and its dual, view merge) is used in the collaboration between multiple stakeholders, each with different concerns. Each viewpoint allows the evaluation of a stakeholder-specific set of properties. When concrete views are merged, the conjunction of all the views' properties must hold. In the software realm, IEEE Standard 1471 defines the relationships between viewpoints and their realisations views. Note that the views may be described in different formalisms.

5.2 Embedding: a simple, powerful MPM combinator

As an orthogonal view to the general mechanisms presented above, there exists the possibility to combine paradigms to form new paradigms through *combinators*, i.e. operators that allow the combination of two artefacts that follow two paradigms (distinct or not). Combinators may even have higher arities, allowing combination of a finite collection of artefacts.

Given the way our Descriptive Framework captures the notion of paradigm, a natural (yet not completely general) way to describe combinators is to proceed in a component-wise fashion:

F-Combinator Combining Formalisms, keeping their default Workflows separate, while ensuring ParadigmaticPropertys that ensure soundness of the operation; or
W-Combinator Combining Workflows, assuming their default Formalisms are distinct, while ensuring soundness.

In this section, we propose to capture a simple binary F-Combinator named *embedding* that we note $\oplus$:

$$\begin{aligned} \oplus : \mathsf{Formalism} \times \mathsf{Formalism} &\rightarrow \mathsf{Formalism} \\ (\mathsf{Host}, \mathsf{Guest}) &\mapsto \mathsf{New} \end{aligned}$$

An embedding takes two *source* formalisms (together with their default workflows), the Host and the Guest, each following its own paradigm, and produces a New formalism with two separate, default workflows that may be improved to help co-design the new formalism instances. Note that $\oplus$ is a non-commutative combinator: switching Host, i.e. the formalism that embeds, or is extended with, the Guest, generally results in two radically different results, as we will illustrate in Sects. 5.4 and 5.3.

For the new formalism to be valid, an embedding should:

- Define a new, valid *abstract syntax* based on the abstract syntaxes of the Host and Guest source formalisms;
- Define a new *semantics* that is *conservative*, i.e. if the embedded (syntactic) elements are removed from the new formalism instances, the execution semantics shall *coincide*, as a projection, with each one of the source formalism instance execution semantics.

At a high level, one can see the execution (operational semantics) of an embedding as a three-step process:

1. The host starts the execution, following its semantics;
2. At some specific steps during the execution, corresponding to the embedding, the host delegates the execution to the guest;
3. The guest then proceeds with its own execution semantics;
4. At some predefined steps during the guest's execution, or when something global occurs for the host, the delegation stops and returns to the host.

The specific point where the delegation occurs is defined syntactically, while the mechanisms for delegating from the higher, *macro*-level of the host, to the lower, *micro*-level of the guest and back, is defined in a semantic adaptation (embedding).

For illustrative purpose, we will describe the following embedding, which results in the well-known *hierarchical* TFSA (HTFSA):

$$\text{HTFSA} \triangleq \text{TFSA} \oplus \text{TFSA}$$

5.2.1 Abstract syntax

The pattern described in Fig. 12 (bottom) captures how the resulting paradigm's abstract syntax is constructed: a Director class from the host is extended with a Delegate class from the guest. The Delegate then contains a Delegation where the micro-steps occur. As a guideline for helping identify potential matches, a Director is often a super class extended with particular cases that behave slightly differently from each other.

For building a HTFSA by embedding, we need to match the previous pattern (cf. Fig. 12, top, unnecessary details omitted). We identify as a natural candidate the State class as a Director, which leads to internal computations inside Composite states, performed by an full TFSA as a Delegation.

5.2.2 Execution semantics

The Activity Diagram of Fig. 13 describes a possible recursive operationalisation of the execution semantics in

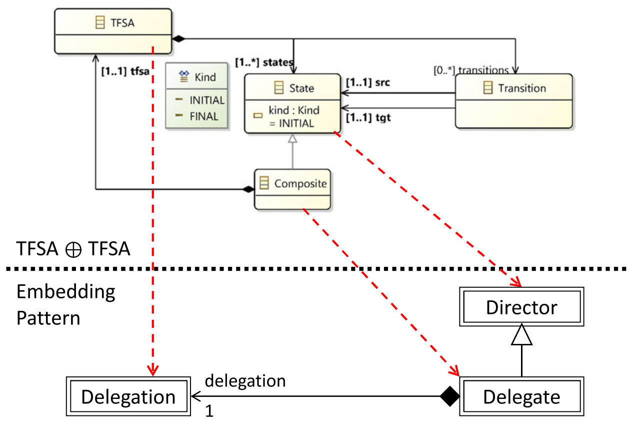


Fig. 12 The Embedding Pattern (bottom) defines how the Abstract Syntax may be built from Host and Guest abstract syntax elements: in the Host, a Director performs macro-steps, and sometimes Delegates computations to the Guest, resulting in micro-steps performed by the Guest's Delegation. In the case of HTFSA, the State class is matched as the Director, using a a full TFSA as a Delegation.

an embedding, by implementing the following algorithm expressed as Activity Diagrams:

1. Starting from the Host, an Initialise phase sets time and system state variables for preparing the computation steps;
2. A CheckStopConditions checks whether this (hierarchical) level's halting conditions are fulfilled. If they are, this level's computation halts: control is transferred back the outer level, eventually performing a Terminate activity for final settings; or the whole computation terminates.
3. If CheckStopConditions are not fulfilled, a PerformStep occurs, making progress for this level's computation;
4. Then, a CheckForDelegation checks whether the current element embeds an internal instance: if this is the case, control is transferred to the inner structure (Delegate::Execute); otherwise, the control loops back to CheckStopConditions for another (macro) step.

The check and eventual call to the Delegate's Execute Transformation (depicted in green) transfers control to the lower level, performing the *micro*-steps embedded inside the current level's *macro*-step (depicted in red). Note that this pattern may occur finitely many times, allowing the embedding of an arbitrary number of levels.

Applying this pattern to the particular case of the HTFSA embedding performs a transfer to the sub-TFSA, while keeping the same Execute specification. Note that this pattern produces a behaviour for HTFSA that is opposite to the one promoted by UML: in case of competition between transitions at different hierarchical levels with identical Events, the *outer*most transition takes priority, following Harel's

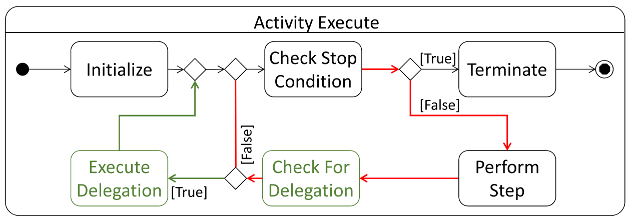


Fig. 13 Pattern expressing the behavioural semantics of an embedding. After a preliminary phase that Initialises necessary variables, a *macro*-step (in red) is performed by the Host. In case of a Delegation at this step (CheckForDelegation), control is transferred to the *micro*-level, after a preliminary phase (InitialiseInternal, corresponding to the Initialise phase, but at the micro-level). When CheckStopConditions fails, it transfers control back to the micro-level, or stops the whole computation if there is none

Statecharts semantics [32] (as opposed to the conventional *inner*most choice in UML).

5.3 Embedding CBD into TFSA

Many CPSs evolve through so-called *running modes* [51], i.e. their behaviour changes significantly depending on high-level, clearly identifiable *modes*. For example, regulatory systems in biology identify potential deviations from a normal course of action (such as cell mitosis, DNA replication, metabolic regulation and so on), and take measures to recover, thus exhibiting two clear modes; robot arms in a factory exhibit different behaviour depending on the way they move in space in order to avoid hurting the humans working around them, or to hit an obstacle, thus making clear distinctions when operating in either secure or risky environments; autonomous electric vehicles introduce several driving modes for handling snow, allowing user-controlled drifting for circuit driving, or avoiding obstacles dangerous for the occupants, thus exhibiting clear distinctions on how to manage power, drive trains and so on depending on potential dangers or road conditions.

Consider as a small example of such a CPS, a bouncing ball that may be kicked from time to time [73]: a ball starts free-falling from a predefined height; it will eventually collide with the ground, then bounces up again with reduced energy; sometimes it is kicked, adding a predefined velocity. To model such a system, we immediately notice three modes: a FreeFall mode describes the ball's free fall, following Newton's laws; the (artificial, infinitesimally short) Collision mode describes the moment the ball hits the ground and bounces up, going again in free fall; and the Kick mode represent a kick, adding to the ball's upward velocity. At a high level of abstraction, this small system switches from one of those modes to one of the others, depending on clearly defined events, where each mode describes the system's dynamics with continuous, physical (Newtonian) laws. There are two paradigms at play in this scenario:

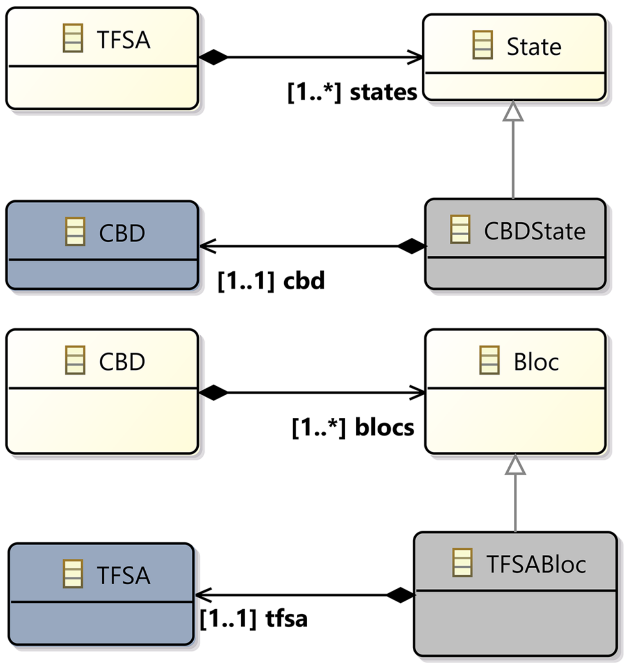


Fig. 14 Concretising the Embedding Pattern of Fig. 12 (bottom) for TFSA ⊕ CBD (top) and CBD ⊕ TFSA (bottom) (Delegate in grey and Delegation in red)

– at an outer level, describing modes where switching occurs when particular events are identified corresponds to the Discrete Event Dynamic System paradigm;
– at an inner level, in each mode, representing the dynamics of a Newtonian physical system may be approximated in a Synchronous Data Flow paradigm.

The presence of an outer and an inner level suggests to embed an SDF formalism into a DEv formalism, following the procedure described previously:

Abstract Syntax Modes may be captured by States; therefore, matching the State class as a Delegate seems appropriate. The Delegation is composed of a full CBD for capturing the various physical laws governing the free falls (up and down) and the kick-up (cf. Fig. 14).

Execution Semantics Two instances of the Execute Activity Diagram of Fig. 9 may be composed following the pattern described in Sect. 5.2.2: the macro-step would follow the activities described for the behavioural semantics of TFSA (Listing 1), while the micro-step would embed the activities for the behavioural semantics of CBD (Listing 2). Note that to effectively allow a simulation of the whole system, a new time step delta should be computed as the maximum common divisor of the TFSA semantic Domain's delta and the CBD semantic Domain's delta.

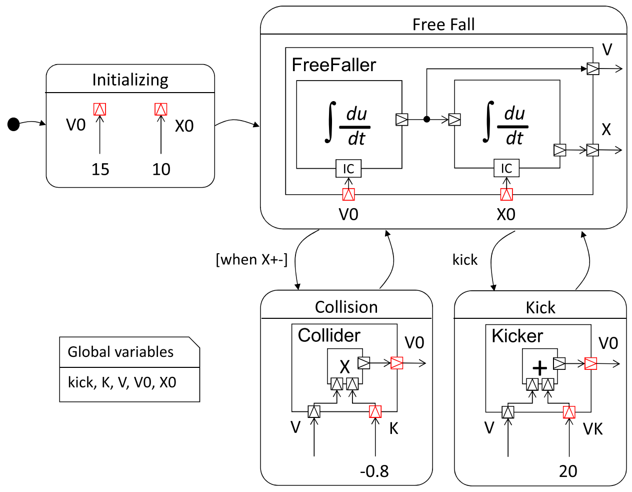


Fig. 15 Model of the Kickable Bouncing Ball: after Initialising the necessary state variables, the ball is in FreeFall, following Netwton's Free Fall Law, then Colliding, thus bouncing. Sometimes, the ball gets Kicked, allowing it to momentarily gain upward speed. The Global Variables are shared by the CBD inside the States

The ⊕ combinator only provides guidelines for embedding: some syntactic and semantic adjustments need to be provided to obtain a full fledged formalism. In this case, two elements need to be taken care of to enable communication between both formalisms:

– Since CBDs continuously compute outputs from input when activated inside a State, they need persistent GlobalVariables to enable communication between instances in various State.
– Possibly new Triggers may need to be defined to capture the so-called *level/zero-crossing* phenomenon, i.e. producing inside the environment an Event when some continuous variables exceed a predefined threshold [80].

As a result, Fig. 15 depicts a possible instance of the embedding TFSA ⊕ CBD that captures the behaviour of the kickable bouncing ball. The GlobalVariables are declared outside the TFSA, and two specific Events (namely when +- and when -+) detect the moments when the ball reaches the lowest (on the ground) and highest positions during falls.

5.4 Embedding TFSA into CBD

Many CPSs are, from an abstract viewpoint, so-called feedback control systems ([4]), i.e. they are composed of two (or more) subsystems that are connected so that each influences the other(s), with the particularity that at least one of these subsystems (often realised as a software component) senses the operations of the other subsystems through various sensors, compares the sensed behaviour to a desired behaviour, and computes corrective actions that are applied through actuators. Such interconnected, strongly coupled CPSs are notoriously difficult to analyse, making modelling and simulation a crucial enabler in the large-scale development of such systems.

Consider again the small CPS example of a car's Power Window, introduced earlier in Sect. 4.1.2, but now taking physical effects into consideration. A driver has at his disposal two buttons Up and Down, which manually command a motor that moves a driver-side window. For safety reasons, the Window is also equipped with a sensor that detects a resistive Force against the upward movement, helping detect whether an object obstructs the Window's course. One possible way to check the safety of the system is to simulate it and to check that a reasonable Force always leads to halting the window's upward motion. A possible (simplified) model would compute the position of the Window, given the multiple inputs (provided by the user's manual commands and the sensor) and some predefined parameters (corresponding to the Friction the window's frame imposes on the Window during its movement, and the motor's linear force Motor). We can distinguish two different paradigms that are involved in a Control/Command pair:

– at an outer level, the Window's movement simply follows Newton's Second Law, since the overall mass of the system (frame + motor + window) stays constant.
– at an inner level, deciding which direction the Window should move in may be modelled in a discrete way by analysing the window's state over time and detecting the emergency cases due to excessive resistive force.

From an abstract viewpoint, the Window's movement may be simulated using the SDF paradigm (using a discretisation of its equations of motion), while the control part may be described through the *DEv* paradigm, suggesting to embed a *Dev* formalism into an SDF one, following the procedure presented above:

Abstract Syntax In a CBD, the Block class plays the role of the Director, attaching as a Delegation a full TFSA (cf. Fig. 14).
Execution Semantics A construction similar to the previous case may be used, this time using CBD as the macro-step and TFSA as a micro-step.

As a result, Fig. 16 depicts a possible instance of the embedding CBD ⊕ TFSA, for modelling (part of) the behaviour of the Power Window CPS.

6 Related work

Our work proposes an explicit definition of the notion of modelling paradigm, which is a generalisation of the notion of programming paradigm to the more general domain of

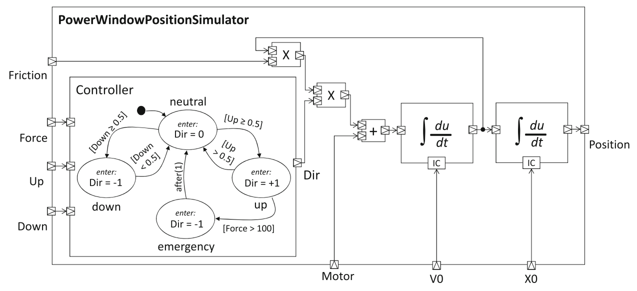


Fig. 16 Partial model of a power window in a car. The first CBD Block continuously computes in which direction the Window (motor) moves, transmitting the information to the second Block, which effectively realises the movement

modelling. Over the years, a plethora of programming languages have been developed to deal with the heterogeneity of software systems. The notion of programming paradigm [31,78] was proposed more than 30 years ago to categorise the different approaches or styles used by the many different programming languages. This lead to the rich research field of multi-paradigm programming. We did, however, not find any work making the notion of programming paradigm as explicit and precise as we propose in this work. Rather, the notion of programming paradigm is expressed in natural language and varies slightly from one author to another. Van Roy [68] proposed a more precise definition where a programming paradigm is defined as "[...] a set of programming concepts, organised into a simple core language called the paradigm's kernel language". Even this definition is neither precise, nor does it propose, or even identify the need for a procedure to decide whether a programming language is based on a given paradigm.

We find the same issue with MPM, which as mentioned earlier, originated from the Modelling and Simulation Community in 1996. While the initial work of the COST Action IC1404 MPM4CPS[2] proposed an ontology for the domain of MPM for CPS, we do not find a precise definition of the notion of a modelling paradigm in the ontology. There is, however, a body of work well suited to support multi-paradigm modelling, such as the composition of modelling languages, the composition of analyses, the composition of tools and the composition of workflows. Furthermore, while research has produced a variety of interesting textbooks on modelling for CPS, such as Alur [2], Lee and Seshia [47], Taha et al [65], these usually employ selected modelling techniques and do not cover the multi-paradigm aspects of CPSs.

The related work regarding each of these topics is discussed in the following subsections.

[2] http://mpm4cps.eu.

6.1 Composition of Modelling Languages

The composition of modelling languages is closely related to multi-paradigm modelling as composed languages often comprise different formalisms (e.g. UML class diagrams for the Object-Oriented description of structures, State Machines that describe state-based behaviour, and OCL to describe constraints of the overall system). To this end, software language engineering [36] produced a variety of formalisms, such as

– the integrated syntax definitions of MontiCore [35], Neverlang [66], and Xtext [6];
– methods to define well-formedness rules, including OCL [33] or the Name-Binding Language NaBL [71] of Spoofax;
– model transformation techniques and frameworks, such as ATL [40], T-Core [64], or the Epsilon Transformation Language ETL [41].

For such language definition formalisms, the composition techniques range from embedding and merging of abstract syntax definitions [19,35], over matching grammar nonterminals by name [66] and importing syntactic elements from other DSLs [6], to the integration of interpreters or code generators [10,45]. All of these have in common that their compositionality is limited to the level of their formalisms, i.e. yet there is no software language reuse technology that considers composing the intended usage workflows (e.g. compiling, transforming, validating) that come with them.

Language workbenches [24] span technological spaces by providing and combining multiple formalisms, such as grammars and template languages for code generation [66] or reference architectures for metamodels [34] and interpreters for model execution [76] to support engineering multiple aspects of modelling languages. Such workbenches come with powerful tools and documented workflows describing how to engineer languages with the given formalisms. The workflows are usually given in natural language, which severely hampers reasoning about the compatibility with other workflows. Also, often the mappings between their supporting formalisms (e.g. metamodelling techniques and code generators) are not modelled explicitly but encoded in the tooling. A formal basis for MPM can enable the making explicit of workflows and their relations to formalisms, thus facilitating language composition not only across formalisms, but also across workbenches.

6.2 Composition of analyses

Another field of research closely related to MPM is the composition of analyses, since composed analyses often comprise different formalisms (e.g. discrete event systems for reasoning about event-based communication, queuing networks for resource utilisation analysis and logical programming for constraint checking).

One established way to realise analysis composition is simulator coupling and co-simulation. Distributed Interactive Simulation (DIS) [38] is a decentralised approach to simulator coupling, where the state of the analysis is shared between all participants of the simulation. The successor of DIS is High-Level Architecture (HLA) [37] standard. Coupling information is stored by a central manager which enables the combination of analyses. An overview of the state of the art of co-simulation is presented by Gomes et al. [26].

All of these approaches have in common that their compositionality is limited to the level of their formalisms and to information exchange between partial analyses implemented in tools. There is, however, no modularisation and composition concept for analyses on a semantic level. A formal basis for MPM can serve as a foundation for modularisation and composition of analyses on a semantic level.

6.3 Composition of software tools

Combining different formalisms usually implies combining the tools that operate on models in these formalisms. Tools are commonly integrated to form so-called toolchains. However, the field lacks methodical or theoretical foundations for systematically combining such tools across various domains.

Co-Simulation [26], as already mentioned in the previous section, is one specific area where tools composition is supported by foundational work. Co-simulation allows one to combine existing simulation tools into a integrated simulator, with some guarantees of correctness. The Functional Mockup Interface (FMI)[3] standard provides Functional Mock-Up Units (FMU) that can be combined/orchestrated using standardised interfaces.

The Open Services for Lifecycle Collaboration (OSLC) Initiative[4] proposes a set of specifications that enable the integration of any kind of software development tools. It builds on the W3C RDF (Resource Description Framework) to describe resources shared by tools, linked data to relate these resources and a REST (REpresentation State Tansfer) interface to expos the tool APIs as Web services for their integration, as techniques to enable the preview, creation and selection of links between resources. OSLC strongly relies on Web technologies which may limit performance. It also lacks technology agnostic foundations.

The SPIRIT framework [39] for model and data integration and toolchain development tries to provide a more all-encompassing foundation. One advantage of this approach is that it considers the evaluation of how well the developed toolchains perform and adopts a service-oriented approach. For evaluation, metrics are defined for the capabilities of individual tools within a toolchain, and the interoperability of the whole toolchain. SPIRIT integrates several open standards such as the GOPPRR (Graph Object Property Point Role Relationship) metametamodel, the Web Ontology Language (OWL), the FMI and Business Process Modelling Notation (BPMN) for workflow modelling. However, it is not clear whether the approach requires that existing formalisms are re-implemented based on this metametamodel.

6.4 Composition of workflows

While the MPM community recognised that the explicit specification of MBSE workflows (as, for example, described in this paper) is needed, the workflow management community has long understood the usefulness of explicitly modelling service composition and choreography (e.g. [18,67]) using appropriate formalisms.

Service Composition and Business Process Composition [21,44] are two well-known approaches. Service Composition is usually split into two broad categories: *static* composition, which includes orchestration, i.e. one service orchestrating the others, and choreography, i.e. each service describes its interactions, for which different formalisms and languages have been developed over the years such as WS-BPEL, WS-CDL and OWL-S; and *dynamic* composition, which uses semantic annotations as proposed by Lautenbacher and Bauer [44].

Many Business Process Composition algorithms have been proposed based on graphical notations: for example, Brockmans et al. [7] proposed to model business processes through Petri Nets, which are annotated with domain ontologies using similarity computation and aggregation. However, none of these approaches from the workflow management community considers the composition of multiple formalisms.

7 Discussion

Our Descriptive Framework for MPM is the first approach to enable the systematic integration, use, and evaluation of the variety of paradigms necessary to successfully engineer CPSs. Built on the generic concepts of formalisms, workflows, model operations, and their integration, the framework is agnostic to the kind of systems it is applied to. While we consider this beneficial for the applicability of our theory, specific instances of the use of MPM, with predefined formalisms and workflows for specific challenges, can be useful in more limited contexts.

Our Framework shows that specific metaformalisms for describing formalisms, transformations, and workflows are

[3] https://fmi-standard.org/.

[4] https://open-services.net/.

necessary to achieve enough precision to be able to explicitly check whether a paradigm candidate follows a paradigm. This choice of metaformalisms imposes a particular view on MPM. Other choices such as functional or logical views might yield different results. However, as metamodels have been successfully employed to describe (parts of) the world in software engineering, we consider this choice well-suited to describe the foundations of MPM. Nonetheless, freedom in this choice entails that when paradigm candidates are described using formalisms different from those used in this paper to capture the paradigmatic structure, mechanisms are required to (dis)prove their equivalence and compatibility. This will complicate establishing relations between these paradigms and demands further research. The existing research on the topic of semantic equivalence is seen as a possible starting point.

This freedom of choice also extends to the property languages of choice (such as the metamodel patterns used in this paper). When the formalisms to specify paradigms are fixed and integrated, suitable and more specific property languages can be derived automatically using modelling language engineering techniques (such as ProMoBox [55], among other approaches). When this choice is not fixed, the paradigms used for property description and property checking need to be integrated properly as well. When no automated methods exist to check the equivalence of properties expressed in different formalisms, a manual proof might be required.

The *Physical* part in CPS introduces the need for computationally *a*causal models in order to capture the constitutive physical laws of such systems, which may well be expressed through an *acausal paradigm*. Note how Equation-based Object-Oriented Languages such as Modelica supporting the acausal paradigm still only capture mathematical relationships. Still, they may not capture all constraints imposed by the laws of physics. That requires even more physics-oriented paradigms, those based on Power Flows [29] as used in Bond Graphs [8]. On the other hand, combining inherently causal "*cyber*" components and aspects of CPSs, also implies the combination of causal paradigms (i.e. refinements such as the Data Flow or the State Automata paradigms that we illustrated in this paper) with acausal paradigms. This combination does fit in our proposal for a Descriptive Framework and is indeed the heart and soul of CPSs. However, for pedagogical reasons and for the purpose of illustration when presenting this Framework, we have restricted the illustrative formalisms and workflows to the simplest possible, yet keeping the key ingredients to illustrate how combinations of such paradigms may look like. Further application of our Descriptive Framework to more elaborated acausal paradigms and their possible combinations is left as future work.

Our framework is also completely agnostic with respect to the specific way the integration of two (or more) paradigms is done, as this integration highly depends on the constituent paradigms, as well as on the purpose of the integration. We illustrated paradigm integration with an example (namely TFSA $\oplus$ CBD for mode automata) where one paradigm describes the system's structure, while the other captures the state-based behaviour of the system's elements. Moving towards, for example, assembly lines would require the integration of geometry (supported by Computer-Aided Design) with kinematics and rule-based assembly knowledge, opening the way to radically different types of integration. Our extensional perspective for joining the paradigms' specifications is the foundation for integration in our MPM theory. This does require further restrictions to be identified, e.g. information about the formalisms, workflows, and the intentions of integration (as it has been identified for Model Transformations [49]).

The combinations of paradigms discussed in the examples focus on the formalisms and detail corresponding compositions. As sketched in Fig. 2, the combination of workflows demands for composition operations such as activity embedding (or similar combinations) that, again, depend on the formalisms of choice. Note also that in practice, many operations that should be enforced at the formalism level may be delegated to the workflow part as an external operation. For example, checking the validity of a CPS design may require the knowledge of other views or components of the overall system, thereby relying on legacy procedures to ensure consistency, which will naturally take the form of an activity. In future work, we will detail this for various workflow formalisms and their usage in concrete situations.

Our vision of applying MPM includes structuring the engineering of CPSs by making the paradigms of the different stakeholders explicit and machine-processable, and, ultimately, applying our theory to foster automation in Systems Engineering. This may include building, deriving, and validating tool ensembles for engineering specific CPSs as well as making the cooperation between the different stakeholders through the paradigms explicit. Moreover, we expect this vision to enable predicting various qualities of the engineering process as well as of the CPS product.

To properly integrate different paradigms and prevent operating on incompatible paradigms, we must be able to identify formalisms and workflows belonging to a certain well-defined paradigm. To this end, *decision procedures* need to be established to determine whether formalisms and workflows belong to a certain paradigm, that can subsequently be integrated. Of course, these procedures are also highly dependent on the paradigms in question and may not be automatically decidable in all cases.

Paradigms may be related to one another, e.g. through extension, refinement, or substitutability. These relationships may form the basis for reuse of paradigm-based analyses, proofs, tools, etc. We illustrated a rather simplistic example for Object Orientation (with single vs. multiple inheritance).

This is subject to ongoing work and might relate to the notion of model types [17,30] for the structure of formalisms and to the notion of semantic refinement [60] for the behaviour specified in workflows.

To capitalise on the foundations of MPM, the construction, analysis, and integration of paradigms should, ultimately, be supported through automation. Therefore, software tools are needed that capture workflows related to paradigms, such as analysing whether a set of paradigms can be integrated to achieve certain results, to store, query, and retrieve paradigms from repositories, and more. The functional requirements for such tools and repositories demand further research and form the core of MPM engineering as a discipline.

The examples presented throughout the paper are carefully selected to clarify the concepts of MPM that we strive to convey. Consequently, these are compact and cannot cover the complete landscape of paradigms and their combinations necessary to engineer a sophisticated CPS. In the future, we will investigate selecting and combining specific paradigms to design, engineer, and deploy CPSs to present the application of our framework in the large.

MPM advocates using the most appropriate formalisms. This may lead to different components and views expressed in different formalisms with different semantics. The burden is on the modeller to prove equivalence—if that is needed or indeed even possible. However, once formalisms and workflows have been explicitly modelled, implementing a decision procedure becomes possible. This will most likely require a community to stick to a particular "style" of modelling. If multiple styles are needed, proofs of equivalence may not be possible. Then again, a community may agree on equivalence until the converse is proven.

8 Conclusion

This paper proposed a structural Descriptive Framework for Multi-Paradigm Modelling. A paradigm P is defined as a set of characteristics, so-called *paradigmatic properties*, that requires a *paradigmatic structure* to be expressed explicitly. This paradigmatic structure captures "universal" concepts expressed through placeholders and shared by all artefacts qualifying as, or following P. The placeholders are intended to be mapped to the concrete constructs defining potential candidates. This enables decision procedures associated with the paradigmatic properties to be performed to validate whether a paradigm candidate follows P. To tackle the heterogeneity and complexity of CPSs, it is often necessary to combine multiple paradigms to adequately capture all facets of a CPS. This calls for Multi-Paradigm Modelling. To that end, we have explored a first paradigm combinator, namely *embedding*, and have shown how to systematically build a valid paradigm candidate for the resulting multi-paradigm combination.

The Descriptive Framework presented in this paper is a first step towards more formal foundations for MPM for CPSs, which future research can build upon. For instance, during the COST Action IC1404, we actively collected and classified several industrial paradigmatic scenarios involving various workflows and formalisms. From this work, an interesting library of CPS paradigms currently used in industry may follow. Such a library will allow researchers and practitioners to reflect and build upon. It will also provide a further validation of the structures described in our Descriptive Framework. Capturing other paradigm combination operators observed in practice would also contribute to the exploration of the various ways MPM is already used in industry for modelling and simulating complex CPSs. Based on the understanding gained in this work, of different paradigms and their interaction for modelling and analysing complex CPSs, we will explore a feature-based decomposition and composition of modelling languages and analyses. Ultimately, a better understanding of the different paradigms that are in place to model CPSs and of their integration can yield better modelling, analysis, design, and optimisation tools. This will contribute to more efficient engineering practices of future CPSs.

Acknowledgements Moussa Amrani was funded by the Walloon Region Cluster of Excellence SkyWin SW_D-DAMS Project. Dominique Blouin was partially supported by the ISC Chair (*Ingénierie des Systèmes Complexes*) on cyber–physical systems and distributed control systems. Robert Heinrich was partially funded by the German Federal Ministry of Education and Research under grant 01IS18067D (RESPOND), and the KASTEL institutional funding. Hans Vangheluwe's Research Group was partially supported by Flanders Make, the strategic research centre for the Flemish manufacturing industry. Andreas Wortmann was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany Excellence Strategy—EXC 2023 Internet of Production.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Springer, New York (1998)

2. Alur, R.: Principles of Cyber-Physical Systems. MIT Press, Cambridge (2015)
3. Amrani, M., Blouin, D., Heinrich, R., Rensink, A., Vangheluwe, H., Wortmann, A.: Towards a formal specification of multi-paradigm modelling. In: International Conference on Model Driven Engineering Languages and Systems Companion, pp. 419–424 (2019)
4. Åström, K.J., Murray, R.M.: Feedback Systems: An Introduction for Scientists and Engineers. Princeton University Press, Princeton (2012)
5. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T.A., Larsen, K.G.: Contracts for system design. Foundations and trends in electronic design automation. **12**(2–3) (2018). https://www.amazon.co.uk/Contracts-System-Foundations-Electronic-Automation/dp/1680834029
6. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing Ltd, Birmingham (2016)
7. Brockmans, S., Ehrig, M., Koschmider, A., Oberweis, A., Studer, R.: Semantic alignment of business processes. In: Proceedings of the Eighth International Conference on Enterprise Information Systems: Databases and Information Systems Integration, pp. 191–196 (2006)
8. Broenink, J.F.: Bond graphs: A unifying framework for modelling of physical systems. In: Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems, pp. 15–44. Springer (2020)
9. Burden, R.L., Faires, J.D.: Numerical Analysis. Cengage Learning (2015)
10. Butting, A., Eikermann, R., Kautz, O., Rumpe, B., Wortmann, A.: Modeling language variability with reusable language components. In: International Conference on Systems and Software Product Line (SPLC'18). ACM (2018)
11. Cambridge University Press: Cambridge Dictionary (2020). https://dictionary.cambridge.org/dictionary/english/paradigm. Accessed 25 Mar 2020
12. Carreira, P., Amaral, V., Vangheluwe, H.: Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems. Springer, New York (2020)
13. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude. How to Specify, Program and Verify Systems in Rewriting Logic. Springer, A High-Performance Logical Framework (2007)
14. Combemale, B., Thirioux, X., Baudry, B.: Formally defining and iterating infinite models. In: International Conference on Model Driven Engineering Languages and Systems, pp. 119–133 (2012)
15. Cooper, D.: Standard Pascal User Reference Manual. W. W. Norton, New York (1983)
16. Corradini, A., Dotti, F.L., Foss, L., Ribeiro, L.: Translating java code to graph transformation systems. In: International Conference on Graph Transformation, pp. 383–398 (2004)
17. De Lara, J., Guerra, E.: A posteriori typing for model-driven engineering: concepts, analysis, and applications. ACM Trans. Softw. Eng. Methodol. **25**(4), 1–31 (2017)
18. Decker, G., Weske, M.: Interaction-centric modeling of process choreographies. Inf. Syst. **36**(2), 292–312 (2011). https://doi.org/10.1016/j.is.2010.06.005
19. Degueule, T., Combemale, B., Blouin, A., Barais, O., Jézéquel, J.M.: Melange: A meta-language for modular and reusable development of DSLs. Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pp. 25–36 (2015b)
20. Denckla, B., Mosterman, P.J.: Formalizing causal block diagrams for modeling a class of hybrid dynamic systems. In: Proceedings of the 44th IEEE Conference on Decision and Control, pp. 4193–4198, (2005). https://doi.org/10.1109/CDC.2005.1582820
21. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management, 2nd edn. Springer, New York (2018)
22. El-Fakih, K., Gromov, M., Shabaldina, N.V., Yevtushenko, N.: Distinguishing experiments for timed nondeterministic finite state machines. Acta Cybern. **21**(2), 205–222 (2013)
23. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, pp. 1–8 (2012)
24. Erdweg, S., Van Der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., et al.: Evaluating and comparing language workbenches: existing results and benchmarks for the future. Comput. Lang. Syst. Struct. **44**, 24–47 (2015)
25. Fritzsche, M., Gilani, W., Lämmel, R., Jouault, F.: Model transformation chains in model-driven performance engineering: experiences and future research needs. In: Modellierung, pp. 213–220 (2011)
26. Gomes, C., et al.: Co-simulation: a survey. ACM Comput. Surv. **51**(3), 49 (2018)
27. Gomes, C., Denil, J., Vangheluwe, H.: Causal-block diagrams: a family of languages for causal modelling of cyber-physical systems. In: Carreira, P., Amaral, V., Vangheluwe, H. (eds.) Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems, pp. 97–125. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-43946-0_4
28. Gosling, J., Joy, B., Steele, G.L., Bracha, G., Buckley, A.: The Java Language Specification. Java SE, 8th edn. Addison-Wesley Professional, Redwood City (2014)
29. Grainger, J.J., Stevenson, W.D., Chang, G.W.: Power Systems Analysis. Mc Graw Hill, New York (2015)
30. Guy, C., Combemale, B., Derrien, S., Steel, J.R., Jézéquel, J.M.: On model subtyping. In: European Conference on Modelling Foundations and Applications, pp. 400–415. Springer (2012)
31. Hailpern, B.: Guest editor's introduction multiparadigm languages and environments. IEEE Softw. **3**(01), 6–9 (1986). https://doi.org/10.1109/MS.1986.232426
32. Harel, D., Naamad, A.: The STATEMATE Semantics of Statecharts. ACM Trans. Softw. Eng. Methodol. **5**, 293–333 (1996)
33. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Thiele, M., Wende, C., Wilke, C.: Integrating OCL and textual modelling languages. In: International Conference on Model Driven Engineering Languages and Systems, pp 349–363. Springer (2010)
34. Heinrich, R., Strittmatter, M., Reussner, R.H.: A layered reference architecture for metamodels to tailor quality modeling and analysis. IEEE Trans. Softw. Eng. (2019). https://doi.org/10.1109/TSE.2019.2903797
35. Hölldobler, K., Rumpe, B.: MontiCore 5 Language Workbench Edition 2017. Aachener Informatik-Berichte, Software Engineering, Band, vol. 32. Shaker Verlag (2017)
36. Hölldobler, K., Rumpe, B., Wortmann, A.: software language engineering in the large: towards composing and deriving languages. Comput. Lang. Syst. Struct. **54**, 386–405 (2018)
37. IEEE Ieee standard for modeling and simulation (m amp;s) high level architecture (hla)—framework and rules. IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000) pp 1–38 (2010). https://doi.org/10.1109/IEEESTD.2010.5553440
38. IEEE: IEEE Standard for distributed interactive simulation—communication services and profiles. IEEE Std **12782–1995**, 1–20 (1996). https://doi.org/10.1109/IEEESTD.1996.80824
39. Jinzhi, L.: A framework for cyber-physical system tool-chain development: A service-oriented and model-based systems engineering approach. PhD thesis, KTH Royal Institute of Technology (2019)
40. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: a QVT-like transformation language. In: Companion to the 21st

ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pp. 719–720. ACM (2006)
41. Kolovos, D.S., Paige, R.F., Polack, F.A.: The epsilon transformation language. In: International Conference on Theory and Practice of Model Transformations, pp. 46–60. Springer (2008)
42. Kuhn, T.: The Structure of Scientific Revolutions. Chicago Press, Chicago (2012)
43. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Explicit transformation modeling. In: MODELS Workshops, pp. 240–255 (2009)
44. Lautenbacher, F., Bauer, B.: A survey on workflow annotation and composition approaches. In: Proceedings of the Workshop on Semantic Business Process and Product Lifecycle Management (2007)
45. Leduc, M., Degueule, T., Combemale, B.: Modular Language Composition for the Masses. In: Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, pp 47–59. ACM (2018)
46. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **36**(1), 24–35 (1987)
47. Lee, E.A., Seshia, S.A.: Introduction to Embedded Systems: A Cyber-Physical Systems Approach. MIT Press, Cambridge (2016)
48. Liu, S.H.: Design Space Exploration for Distributed Real-Time: A Software Product Line Engineering Approach for Design and Analysis of Component-Based Distributed Real-Time and Embedded Systems. VDM Verlag (2010)
49. Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G., Syriani, E., Wimmer, M.: Model transformation intents and their properties. SoSyM **15**(3), 647–684 (2014)
50. Manifesto, A.: Manifesto for agile software development. (2001) See https://agilemanifesto.org/. Accessed 2 Aug 2020
51. Maraninchi, F., Rémond, Y.: Mode Automata. A New Domain-Specific Construct For The Development of Safe Critical Systems. Science of Computer Programming (2003)
52. Merkow, M.: Secure, Resilient, and Agile Software Development. Auerbach Publications, New York (2019)
53. Merriam-Webster : Merriam-Webster Online Dictionary. (2020). https://www.merriam-webster.com/dictionary/paradigm. Accessed: 25 Mar 2020
54. Messerschmitt, D.G.: A tool for structured functional simulation. IEEE J. Sel. Areas Commun. **2**(1), 137–147 (1984)
55. Meyers, B., Vangheluwe, H., Denil, J., Salay, R.: A framework for temporal verification support in domain-specific modelling. IEEE Trans. Softw. Eng. (TSE) **93**(12), 1045–1061 (2017)
56. Mosterman, P.J., Sztipanovits, J., Engell, S.: Computer-automated multiparadigm modeling in control systems technology. IEEE Trans. Control Syst. Technol. **12**(2), 223–234 (2012)
57. Oxford University Press (2020) Oxford English Dictionary. https://www.oed.com/view/Entry/137329. Accessed: 25 Mar 2020
58. Pahl, G., Beitz, W., Feldhusen, J., Grote, K.H.: Engineering Design. A Systematic Approach. Springer, New York (2007)
59. Przybyłek, A., Morales-Trujillo, M.E. (eds.): Advances in Agile and User-Centred Software Engineering. Lecture Notes in Business Information Processing, vol. 376. Springer, New York (2019)
60. Rumpe, B., Wortmann, A.: Abstraction and refinement in hierarchically decomposable and underspecified CPS-architectures. In: Lohstroh, M., Derler, P., Sirjani, M. (eds.) Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday, LNCS 10760, pp. 383–406. Springer, New York (2018)
61. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition. Verification Validation. Springer, New York (2014)
62. Steel, J., Jézéquel, J.M.: On Model Typing. Software And Systems Modeling (2007)
63. Sun, W., Combemale, B., Derrien, S., France, R.B.: Using Model Types to Support Contract-Aware Model Substitutability. In: European Conference on Modelling Foundations and Applications, pp. 118–133 (2013)
64. Syriani, E., Vangheluwe, H., LaShomb, B.: T-core: a framework for custom-built model transformation engines. Softw. Syst. Model. **14**(3), 1215–1243 (2015)
65. Taha, W., Taha, A.E.M., Thunberg, J.: Cyber-Physical Systems: A Model-Based Approach. Springer, New York (2020)
66. Vacchi, E., Cazzola, W.: Neverlang: a framework for feature-oriented language development. Comput. Lang. Syst. Struct. **43**, 1–40 (2015)
67. van der Aalst, W.M.P., Mooij, A.J., Stahl, C., Wolf, K.: Service Interaction: Patterns, Formalization, and Analysis, pp. 42–88. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-01918-0_2
68. Van Roy, P.: Concepts, Techniques, and Models of Computer Programming, Chap Programming Paradigms for Dummies: What Every Programmer Should Know, pp. 9–47. MIT Press, Cambridge (2012)
69. Vangheluwe, H., Vansteenkiste, G., Kerckhoffs, E.: Simulation for the Future: Progress of the ESPRIT Basic Research working group 8467. In: European Simulation Symposium (ESS), SCS (1996)
70. Vangheluwe, H., Vansteenkiste, G.: A multi-paradigm modeling and simulation methodology: Formalisms and languages. In: European Simulation Symposium (ESS), SCS, pp. 168–172 (1996)
71. Wachsmuth, G.H., Konat, G.D.P., Visser, E.: language design with the Spoofax language Workbench. IEEE Softw. **31**(5), 35–43 (2014)
72. Wainer, G.A.: Discrete-Event Modeling and Simulation: A Practitioner's Approach. CRC Press, Boca Raton (2009)
73. Wainer, G.A., Mosterman, P.J. (eds.): Discrete-Event Modeling and Simulation: Theory and Applications. CRC Press, Boca Raton (2010)
74. Watson, I., Gürd, J.G.: A practical data flow computer. IEEE Comput. **15**, 51–57 (1982)
75. Wegner, P.: Dimensions of object-based language design. SIGPLAN Not. **22**(12), 168–182 (1987)
76. Wende, C.: Language Family Engineering—with Features and Role-Based Composition. PhD thesis, Technische Universitaet Dresden (2012)
77. Wimmer, M., Kramler, G.: Bridging Grammarware and Modelware. In: Workshop in Software Model Engineering, pp. 159–168 (2005)
78. Zave, P.: A compositional approach to multiparadigm programming. IEEE Softw. **6**(05), 15–18 (1989). https://doi.org/10.1109/52.795096
79. Zeigler, B.P., Muzy, A., Kofman, E.: Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations. Academic Press, London (2018)
80. Zhang, F., Yeddanapudi, M., Mosterman, P.J.: Zero-crossing location and detection algorithms for hybrid system simulation. Int. Fed. Autom. Control **41**(2), 7967–7972 (2008)

Moussa Amrani is currently a postdoc researcher at the University of Namur (Belgium). His research topics focus on formal verification of model transformations and visual domain-specific modelling. He is working on a Belgian Cluster of Excellence SkyWin Project called D-DAMS, centred on DO-178C certification of flying software, where he pushes model-based development approaches for facilitating the certification activities. He received a Ph.D. in computer science in 2013 from University of Luxembourg, and a M.Sc. in computer science from University Grenoble Alpes in 1998. Prior to his Ph.D., he worked for a decade in R&D departments of various research laboratories and companies.

Dominique Blouin is a permanent research engineer at the Telecom Paris engineering school of the Institut Polytechnique de Paris in France. His research interests are model management, (bidirectional) model transformation and synchronization, domain-specific languages development, require ments engineering, and architecture description languages. He was the vice-chair of working group 1 of the Multi-Paradigm Modelling for Cyber-Physical Systems (MPM4CPS) COST action on foundations for MPM4CPS. He is an active member of the SAE AADL standardisation committee where he initiated the RDAL language, which lead to the ALISA (Architecture-led Incremental System Assurance) framework for AADL. He is also an active contributor to the RAMSES analysis and code generation tool for AADL.

Robert Heinrich is head of the Quality-driven System Evolution research group at Karlsruhe Institute of Technology (Germany). He holds a doctoral degree from Heidelberg University and a degree in Computer Science from University of Applied Sciences Kaiserslautern. His research interests include decomposition and composition of model-based analysis for several quality properties, such as performance, confidentiality, and maintainability, as well as for several domains, such as information systems, business processes, and automated production systems. One core asset of his work is the Palladio software architecture simulator. He is involved in the organisation committees of several international conferences, established and organised various workshops, is reviewer for international premium journals, like IEEE Transactions on Software Engineering and IEEE Software, and is reviewer for international academic funding agencies. Robert is principal investigator or chief coordinator in several grants from the German Research Foundation and governmental funding agencies. He has (co-)authored more than 80 peer-reviewed publications and spent research visits in Chongqing (China) and Tel Aviv (Israel).

Arend Rensink is full professor at the Formal Methods and Tools group, on the topic of Software Modelling, Transformation and Verification. As means for this, he focuses on process algebra and graph transformation, addressing especially their usability in practice. He has developed and maintains the graph transformation tool GROOVE. Besides his research, Arend is Programme Director of the Computer Science Bachelor and Master programmes at the University of Twente, which have grown fivefold in the last decade and are rated among the top programmes in the Netherlands.

Hans Vangheluwe is a professor in the Computer Science Department of the University of Antwerp. His Modelling, Simulation and Design Lab (MSDL) is a core laboratory in the Design and Optimization cluster of Flanders Make, the strategic research centre for the Flemish manufacturing industry. In his research on multi-paradigm modelling, he studies the foundations and applications of modelling language engineering. This covers the entire spectrum, from acausal modelling languages such as Modelica for lumped parameter modelling of physical systems, to discrete-event simulation languages such as DEVS and GPSS to model software and production systems. He investigates modular combinations of these formalisms, of views and of abstractions. He develops scalable (meta-)modelling and (co-)simulation tools to help engineers design, build, optimise and maintain Cyber-Physical Systems. He was the co-founder and coordinator of the EU ESPRIT Basic Research Working Group 8467 “Simulation in Europe”, a founding member of the Modelica language Design Team, and the chair of the EU COST Action IC1404 Multi-Paradigm Modelling for Cyber-Physical Systems (MPM4CPS).

Andreas Wortmann is a professor at the Institute for Machine Tools and Manufacturing Units (ISW) of the University of Stuttgart. Before, he was a senior researcher at the Chair for Software Engineering at RWTH Aachen University and head of the model-driven systems engineering research group. His main research interests are model-driven software and systems engineering, software language engineering, and digital twins. He served as PC member and chair for several international workshops and conferences related to model-driven systems engineering and language engineering. He also is a member of ACM and ACM SIGSOFT and co-author of over 80 peer-reviewed international publications. Moreover, he serves on the board of the European Association for Programming Languages and Systems (EAPLS), on the editorial board of the Journal of Automotive Software Engineering (JASE), and co-organises the working group on Model-Based Systems Engineering of the German Society for Systems Engineering (GfSE). For more information, consult http://www.wortmann.ac.