Towards a Formal Specification of Multi-Paradigm Modelling

Moussa Amrani
Faculty of Science
University of Namur / NaDI
Namur, Belgium
Moussa.Amrani@unamur.be

Dominique Blouin
LTCI, Telecom Paris
Institut Polytechnique de Paris
Paris, France
dominique.blouin@telecom-paris.fr

Robert Heinrich
Inst. Program Struct. and Data Org.
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
robert.heinrich@kit.edu

Arend Rensink
Formal Methods and Tools
University of Twente
Twente, Netherlands
Arend.Rensink@utwente.nl

Hans Vangheluwe
Modelling, Simulation and Design Lab
University of Antwerp – Flanders Make
Antwerp, Belgium
Hans.Vangheluwe@uantwerpen.be

Andreas Wortmann
Software Engineering
RWTH Aachen University
Aachen, Germany
Wortmann@se-rwth.de

***Abstract*—The notion of a programming paradigm is used to classify programming languages and their accompanying workflows based on their salient features. Similarly, the notion of a *modelling paradigm* can be used to characterise the plethora of modelling approaches used to engineer complex Cyber-Physical Systems (CPS). Modelling paradigms encompass formalisms, abstractions, workflows and supporting tool(chain)s. A precise definition of this modelling paradigm notion is lacking however. Such a definition will increase insight, will allow for formal reasoning about the consistency of modelling frameworks and may serve as the basis for the construction of new modelling, simulation, verification, synthesis, ... environments to support design of CPS. We present a formal framework aimed at capturing the notion of modelling paradigm, as a first step towards a comprehensive formalisation of multi-paradigm modelling. Our formalisation is illustrated by CookieCAD, a simple Computer-Aided Design paradigm used in the development of cookie stencils.**

I. Introduction

Modern General-Purpose Programming Languages (GPLs) can be classified according the paradigm(s) they support. For example, Eiffel is object-oriented and supports the contract-based-design paradigm, Prolog is declarative, and Lisp is functional. The paradigm characterises the underlying syntactic and semantic structures and principles that govern these General Purpose Languages (GPLs): object orientation is imperative in nature and imposes viewing the world in terms of classes and communicating objects whereas the declarative style relies on term substitution and rewriting. As a consequence, a statement in Eiffel has very little in common with a Prolog sentence due to the very different view supported by both languages. A programming paradigm directly translates into different concepts encoded in the GPL's syntax definition (known as a metamodel in the Model-Driven Engineering world). Very naturally, the idea of combining several paradigms at the level of GPLs led to more expressive, powerful programming languages such as Java (which is imperative, object-oriented, concurrent, and real-time and, recently, functional) and Maude (which is declarative, object-oriented and also concurrent and real-time). What is a *paradigm* then? The science philosopher Kuhn [1] defines it as an open-ended contribution that frames the thinking of an object study with concepts, results and procedures that structures future achievements. 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 notion of *procedure* that guides achievements.

Cyber-Physical Systems (CPS) emerge from the networking of multi-physical (mechanical, electrical, biochemical, ...) and computational (control, signal processing, logical inference, planning, ...) processes, often interacting with a highly uncertain environment, including human actors, in a socio-economic context. CPS are notoriously complex because they cross discipline borders, leading to inter-domain interactions, in applications that are often safety-critical.

Multi-Paradigm Modelling (MPM) has been recognised lately as a powerful paradigm in its own right that may be helpful in designing, as well as communicating and reasoning about, CPS. Originating from the Modelling and Simulation Community, the term MPM finds its origin in 1996, when the EU ESPRIT Basic Research Working Group 8467 formulated a series of simulation policy guidelines [2] 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", and later on proposing a methodology focusing on combining multiple formalisms [3]. The recent COST Action IC1404 MPM4CPS (http://mpm4cps.eu) surveyed some existing languages and tools commonly used for CPS, and related the languages and techniques for modelling CPS in an ontology. This work triggered the need to propose a theoretical, formal specification of MPM. This formal framework aims to facilitate the communication between experts to help them better grasp the essence of how their CPS are built, but also to facilitate a rigorous comparison of systems based on their core MPM com-

ponents. Ultimately, this framework aims to support (meta-)tool builders who assist practitioners to reason about CPS and figure out which formalisms, abstractions, workflows and supporting methods, techniques and tools are *most appropriate* to carry out their task(s).

Section II motivates our work. Section III sketches a formal framework proposal addressing the notion of paradigms. Section IV introduces the small CookieCAD paradigm. Section V discusses related work and concludes.

II. Motivation

To develop a CPS, project managers and engineers need to select the most appropriate development languages, software lifecycles and "interfaces" to specify the different views, components and their interactions of the system with as little "accidental complexity" [4] as possible. For example, when it is known that system/software requirements are likely to change frequently during the project's course, selecting an Agile development process may help cope with evolution and change. If the system's behaviour requires that operations are triggered when data become available, similar to reactive systems, Data Flow languages may help specify the most critical parts of the software behaviour precisely, making it amenable for timing analysis.

MPM requires to *model everything explicitly*, using *the most appropriate formalism(s)*, *at the most appropriate abstraction level(s)* [5]. This suggests that a *paradigm* is a placeholder for the properties in each of the dimensions described above: the *formalisms*, the *abstraction levels*, and the *processes* used in the modelling activities. Ultimately, MPM aims to select, organise and manage the three dimensions above. We aim to clarify the formal foundations of MPM to help design supporting tools. In this paper, we ignore the abstraction dimension for now and focus on the two others, though we do motivate here why the three dimensions are necessary and how they are related.

Formalisms/Languages. The first dimension relates to what is known as *Modelling Language Engineering*, i.e., the explicit modelling of the key components of (modelling) languages: concrete and abstract syntaxes, as well as semantics, together with the usual activities: analysis, simulation, execution, debugging, etc. that should be supported by tools. Such tools may be largely synthesized from high-level specifications of these languages (such as metamodels for abstract syntax) and their usage. *All* modelling artefacts, including language specifications as well as their instances, are organised in a repository which we call the *Modelverse*. The state of this repository evolves over time as the artefacts in it are modified.

Processes. Workflows or life-cycles are processes relating the various MPM activities. In particular, the above three primitive activities are combined. This is often supported by a toolchain whereby different tools support different activities. Processes may be *descriptive*, charting the sequence of activities carried out as well as the artefacts involved, *proscriptive* by declaratively specifying constraints on the allowed activities and their combinations, and *prescriptive* allowing enactment.

TABLE I: Properties of two paradigms: Object Orientation (OO[7]) and Computer-Aided Design (CAD[8])

ι_1: Object Orientation (OO)	
OO_1	Possess the concepts of Object and Class
OO_2	Objects possess a state and a set of capabilities / operations
OO_3	Possess an inheritance mechanism
OO_4	Inheritance allows to reuse operations
ι_2: Computer-Aided Design (CAD)	
CAD_1	Comprises concepts of (2D/3D) points and lines
CAD_2	Shapes are defined by lines
CAD_3	Supports transformation of shapes into (2D/3D) products

Abstraction/Refinement, Architectural and View decomposition. During the course of system development, three basic approaches are commonly combined to tackle complexity. One particular combination of these approaches leads to Contract-Based System Design [6].

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 if 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.

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 (1) a number of sub-problems, each requiring the satisfaction of its own properties, and each leading to the design of a component and (2) 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 is for example thanks to locality or continuity in the problem/solution domain. Note that the component models may be described in different formalisms, as long their interfaces match and the multi-formalism composition has a precise semantics.

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.

III. Formalisation

Our formal framework defines a *paradigm* as a *label* or *name* ι denoting a set of properties $\pi \in \Pi$. We define a *paradigmatic structure* $\mathsf{PS} \in \mathbb{PS}$ as a candidate structure that *qualifies*, or *embodies*, or *follows* the paradigm ι if PS satisfies π. PS describes a set of workflows that capture various activities seen to be desirable for realising ι. Table I illustrates some relevant properties for two paradigms.

Since every aspects of the paradigmatic structure needs to be explicitly modelled, we first introduce the two established notions to manipulate languages explicitly, namely metamodels and models on the one hand, and transformation specification and execution on the other.

A *metamodel* $\mathsf{MM} \in \mathcal{M}$ specifies the *abstract syntax* of a language L. MM is expressed in a specific language called meta-metamodel. A *model* $\mathsf{M} \in \mathbb{M}$ is a particular instance of L, also specified using a language that may be visual / diagrammatic, textual or hybrid (and normally different from the meta-metamodel). When M is a valid instance of MM, M is said to *conform to* MM and noted $\mathsf{M} \triangleright \mathsf{MM}$.

A *transformation specification* $\mathsf{T} \in \mathbb{T}$ is a triple $\mathsf{T} = ((\mathsf{MM}^{\mathsf{i}}_{\mathsf{s}})_{\mathsf{i}\in[1..n]}, (\mathsf{MM}^{\mathsf{j}}_{\mathsf{t}})_{\mathsf{j}\in[1..m]}, \mathsf{spec})$, where $(\mathsf{MM}^{\mathsf{i}}_{\mathsf{s}})_{\mathsf{i}\in[1..n]}$ and $(\mathsf{MM}^{\mathsf{j}}_{\mathsf{t}})_{\mathsf{j}\in[1..m]}$ are indexed sets of source and target languages, respectively, and constitute the transformation's *signature*; while spec is a well-formed transformation definition written in a transformation language. A *transformation execution* $\mathsf{TE}_{\mathsf{T}} \in \mathbb{TE}$ is a general computation performed on (a) language instance(s) that conform(s) to the source language(s) of the transformation $\mathsf{T} \in \mathbb{T}$.

As an example, the javac compiler is a transformation $\mathsf{T}_{compil} = (\mathsf{MM}_{\mathsf{Java}}, \mathsf{MM}_{\mathsf{BC}}, \mathsf{javac})$ that outputs byte code (BC) from any (valid, conforming) Java program $\mathsf{J} \triangleright \mathsf{MM}_{\mathsf{Java}}$. Note that javac is itself specified as a Java program (i.e. $\mathsf{javac} \triangleright \mathsf{MM}_{\mathsf{Java}}$).

A. Templates

Templates are plain metamodels and transformations, specified with the usual meta-metamodel(s) and transformation languages. A key difference however is that they are not complete, in the sense that they require a regular metamodel (or transformations) to be matched. Templates are used as placeholders for enforcing transformation signatures, expressing high-level properties for metamodels and transformations (specifications), or capturing the necessary steps in a transformation. Suppose we want to formally specify property OO_3 for inheritance in Table I. Figure 1a presents a minimal template defining the basic placeholders: to even allow discussing about super- and sub-classes, one need at least distinguishing between Class and Objects, and defining a partial order (super) between classes. The inheritance property then states that any object of a subclass has access to the state defined by its class, but also to the state defined by its superclass. Note that using a MOF-like meta-metamodel for expressing the template provides navigational notation for free: the property would start with something like $\mathsf{c1}, \mathsf{c2} : \mathsf{Class}$ such that $\mathsf{c2} \in \mathsf{c1.super}$.

Using templates requires a powerful, customisable matching process. For metamodels, contributions on model typing [9], a posteriori typing [10], and metamodel morphisms [11], [12] may be appropriate; for transformations, the rich literature on reuse [13] provides some guidelines on how to realise that. Hence, matching Fig. 1a's template into the Java metamodel [14] would match ClassDeclaration to C and the extends clause to super. We simply capture this important relationship in the following: for a template metamodel $\mathsf{TMM} \in \mathcal{M}$ and a regular metamodel $\mathsf{MM} \in \mathcal{M}$, we note $\mathsf{TMM} \rightsquigarrow \mathsf{MM}$ when TMM is appropriately matched by MM.

B. Workflow

Transformations typically support the realisation of activities that are essential to (the engineering of) languages, typically for parsing and/or pretty printing, defining their semantics (translationally or operationnaly), analysing relevant properties, debugging, and of course executing, simulating, animating, etc. [15]. These activities, and others essential for a paradigm, are typically captured through a workflow.

In our framework, workflows describe precisely the set of activities captured by a candidate paradigmatic structure. A workflow is composed of two elements: a *Formalism Transformation Graph* (FTG) describes explicitly the links between formalisms / languages, stating which possible transformations may be used; and a *Process Model* describes how language instances are combined together towards achieving a particular activity. Combining both elements results in an FTG+PM, as described in [16], [17], [18].

Instead of directly manipulating (meta-)models and/or transformations, our framework relies on *names* that allow to retrieve the corresponding artefacts from a repository (eventually centralised as a Modelverse). This linking is captured by the following definition.

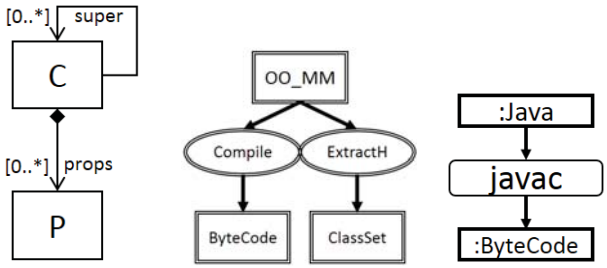
Definition 1 (Naming)**.** *The* naming functions *associate names to their actual item:*

$$\begin{array}{rcl} model & : & \mathsf{MName} \twoheadrightarrow \mathbb{M} \\ mmodel & : & \mathsf{MMName} \twoheadrightarrow \mathcal{M} \\ tmm & : & \mathsf{TMMName} \twoheadrightarrow \mathcal{M} \\ trans & : & \mathsf{TName} \twoheadrightarrow \mathbb{T} \\ tt & : & \mathsf{TTName} \twoheadrightarrow \mathbb{T} \end{array}$$

All functions are partial, returning an undefined item (noted $\perp$) when the repository stores no item with the asked name. Template names for metamodels (TMMName) and transformations (TTName) are in a separate namespace than those for regular ones. The repository is ultimately responsible to retrieve appropriate items: for example, querying the repo with the template of Fig. 1a would eventually allow the selection of the Java metamodel if inheritance is required.

To accomodate with templates, we introduce *extended* FTGs, defined as a (name-restricted) mapping of (template) transformations (names) into a signature.

Definition 2. *(Extended) Formalism Transformation Graph (x*FTG*) An* Extended Formalism Transformation Graph $\mathsf{FTG} \in$



(a) Template for expressing inheritance. (b) FTG for compilation / hierarchy extraction. (c) Compiling PM.

Fig. 1: Examples for (a) Templates, (b) FTG, and (c) PM.

$\mathbb{FTG}(TR, MM)$ *is a function* $\mathsf{FTG}\colon TR \to \langle MM \rangle \times \langle MM \rangle \times \mathbb{B}$.

We parameterise $\mathbb{FTG}$ with two sets of names $TR \subseteq (\mathsf{TName} \cup \mathsf{TTName})$ for regular or template transformations, and $MM \subseteq (\mathsf{MMName} \cup \mathsf{TMMName})$ for regular or template metamodels. Let $tr \mapsto (\langle mm_1, \ldots, mm_n \rangle, \langle mm'_1, \ldots, mm'_m \rangle, b)$ be such a mapping in $\mathbb{FTG}(TR, MM)$: when b is set to true, tr refers to an *automatic* transformation (and human guided otherwise); and mm_is (resp. mm'_js) names are the *source* (resp. *target*) of tr. Figure 1b pictures a simple FTG using two template, automatic transformations (names) (represented with double-rounded, white items) called $\mathsf{Compile}, \mathsf{ExtractH} \in \mathsf{TTransName}$ that respectively compile and extract the class hierarchy of a template metamodel $\mathsf{OO\_MM} \in \mathsf{TMMName}$.

As its name indicates, a PM describes a process, i.e., a set of activities that are combined together towards achieving a particular goal. Instead of reinventing a Domain-Specific Language for this well-studied domain, we simply specialise one standard and well-known language that covers our needs, namely UML's Activity Diagrams.

Definition 3 (Process Model (PM))**.** *A* process model $\mathsf{P} \in \mathbb{PM}$ *is an instance of a* UML *Activity Diagram where*

- ActionNode*s are labelled by transformation instance names typed by their conforming transformation specifications, and may be* hierarchical *and may contain input or output* Pin*s; and*
- ObjectNode*s are labelled by language instance names typed by their conforming languages; and*
- ControlNode*s include* Decision/Merge, Fork/Join *and* Init/Final*s nodes.*

Figure 1c pictures a compilation process between one ObjectNode representing a Java program : Java producing a ByteCode instance : ByteCode through the ActionNode javac. The following definition puts things together: a workflow is composed of an FTG together with a *well-formed* PM.

Definition 4 (Workflow)**.** *A* workflow $\mathsf{W} \in \mathbb{W}(TR, MM)$ *is a pair* $\mathsf{W} = (\mathsf{FTG}, \mathsf{P})$ *where* $\mathsf{FTG} \in \mathbb{FTG}(TR, MM)$ *and* $\mathsf{P} \in \mathbb{PM}$*, such that* P *is well-formed (noted* $\mathsf{P} \blacktriangleright \mathsf{FTG}$*) wrt.* FTG*, i.e., for each* ActionNode *inside* P *with name* tr*,*

- *there exists a transformation name* $\mathsf{t} \in \mathcal{D}\mathsf{om}(\mathsf{FTG})$ *that refers to a transformation specification* T *(i.e.* $trans(\mathsf{t}) = \mathsf{T}$ *or* $tt(\mathsf{t}) = \mathsf{T}$*) such that* tr *conforms to* T *(i.e.* $tr_{\mathsf{T}} \in \mathbb{TE}$*)*
- *for each source or target* mm_i *of* t *(i.e.* $\mathsf{FTG} = (\langle \ldots, mm_i, \ldots \rangle, \langle \ldots \rangle, b)$ *or* $\mathsf{FTG} = (\langle \ldots \rangle, \langle \ldots, mm_i, \ldots \rangle, b)$*) at rank* i *refering to a (template) metamodel* $\mathsf{MM_i}$ *(i.e.* $mmodel(mm_i) = \mathsf{MM_i}$ *or* $tmmodel(mm_i) = \mathsf{MM_i}$*), there exists a* Pin *at rank* i *connected (as input or output) to an input* ObjectNode *carrying a model name* mname *that refers to a model conforming to a metamodel* MM′ *(i.e.* $model(\mathsf{mname}) \rhd \mathsf{MM}'$*, such that either* MM *is the source/target* i*-metamodel of* T *(i.e.* $\mathsf{MM} = \mathsf{MM}'$*), or it matches it (i.e.,* $\mathsf{MM} \rightsquigarrow \mathsf{MM}$*).*

When clear from context, or unnecessarily detailed, we simply note the set of workflows $\mathbb{W}$, without any indication of languages or transformations (sets).

For instance, the PM in Fig. 1c is well-formed wrt. the FTG in Fig. 1b: as established earlier, : Java refers to a Java model, whose metamodel matches OO_MM and javac is an appropriate match for Compile; furthermore, no transformation (execution) are exhibited for matching ExtractH.

Our definition for XFTGs relies solely on *names*: we heavily rely on the repository (Modelverse) capabilities in order to first retrieve the appropriate items (metamodels, models and transformations) as well as checking that matching between them is appropriately achieved. This was abstracted away in Def. 1, but requires a proper formalisation in the future.

Similar to the original [16], [17], [18], our definition does not impose a particular topology on the PM as long as transformations are well-formed. However, the original definition simply rely on a name-matching typing, similar to what is depicted in Fig. 1c, leverage UML-like Object/Class instanciation. In contrast, we introduced a relaxed matching mechanism based on metamodel/transformation matching relying on the $\rightsquigarrow$ relationship.

C. Paradigm

Properties characterising a paradigm (as briefly described in Table I) may span all the previous element. For example, expressing the inheritance formally would require the (template) metamodel of Fig. 1a, but also access to its semantic domain to check that the state of a superclass becomes available to all objects of any subclass. Similarly, properties over transformations may restrict or constraint their applicability (e.g., ExtractH would have to define preconditions for object-oriented metamodels allowing multiple inheritance, and have a post-condition on the datastructure of its output). Properties may also characterise the topology, the usage and the matching process in an FTG or its corresponding PM (by, for example, ensuring frequent loops for the Agile paradigm).

We first collect all previous elements in a single mathematical structure called *paradigmatic structure* on which properties may hold.

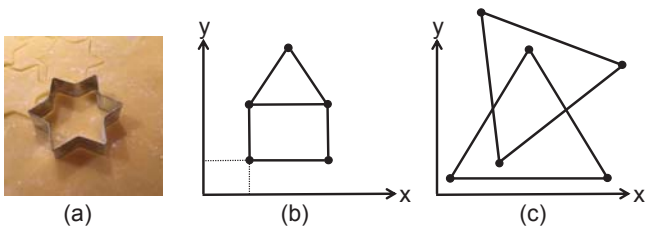


Fig. 2: CookieCAD: (a) Real-life cookie cutter. (b) House-like shape with line-sharing. (c) Invalid shape (due to overlaps).

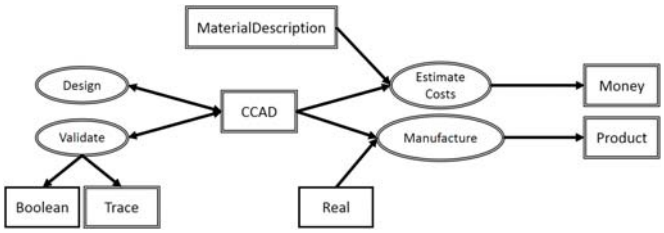


Fig. 3: $\mathsf{FTG}_{\mathsf{CCAD}}$: an xFTG for CCAD activities.

Definition 5 (Paradigmatic Structure)**.** *A* paradigmatic structure $\mathsf{PS} \in \mathbb{PS}$ *is a pair* $\mathsf{PS} = (MM, W)$ *where* $MM \in \wp(\mathcal{M})$ *is a set of metamodels and* $W \in \wp(\mathbb{W})$ *is a set of workflows.*

Following the mantra of modelling "at the most appropriate level of abstraction", it becomes impossible at the abstraction level of this presentation to formally (i.e., intensionally) define the nature of such a large class of properties. We therefore provide an extensional definition: we note $\mathcal{P}(S)$ the set of all possible properties expressible over a structure S.

Definition 6 (Paradigmatic Properties)**.** *A* paradigmatic property *is a tuple* $\pi = (\pi_{\mathsf{MM}}, \pi_{\mathsf{W}}, \pi_{\mathsf{PS}}) \in \Pi$ *where*

- $\pi_{\mathsf{MM}} \in \mathcal{P}(\mathcal{M})$ *is a set of properties over metamodels (and their semantics);*
- $\pi_{\mathsf{W}} \in \mathcal{P}(\mathbb{W})$ *is a set of properties over workflows (and their matching procedure);*
- $\pi_{\mathsf{PS}} \in \mathcal{P}(\mathbb{PS})$ *is a set of properties spanning over all components of paradigmatic structures).*

Paradigmatic properties may be expressed through *pattern languages*, e.g., for ensuring the presence of certain concepts, or through *dedicated logics*, e.g., for ensuring semantic properties. Finally, we associate a *name* to a set of properties for referring to a specific paradigm.

Definition 7. *Paradigm Let* ParadigmName *be a set of (paradigm) names, that we associate to properties through a function* $\iota \in [\mathsf{IntentN} \to \Pi]$.

For $\mathsf{paradigm} \in \mathsf{ParadigmName}$ *such that* $\iota(\mathsf{paradigm}) = (\pi_{\mathsf{MM}}(\mathsf{p}), \pi_{\mathsf{W}}(\mathsf{p}), \pi_{\mathsf{PS}}(\mathsf{p}))$*, we say that* $\mathsf{PS} = (MM, W) \in \mathbb{PS}$ embodies *(alternatively,* follows*,* qualifies as*)* paradigm *iff*

- *the properties* $\pi_{\mathsf{MM}}(\mathsf{p})$ *hold on* MM*;*
- *the properties* $\pi_{\mathsf{W}}(\mathsf{p})$ *hold on* W*; and*
- *the properties* $\pi_{\mathsf{PS}}(\mathsf{p}))$ *hold on* PS.

Notice that it may be interesting to introduce namespaces for paradigm names, since it is likely that similar denominations would be used for slightly different sets of properties: e.g., one may define object orientation in the context of single or multiple inheritance, the latter requiring to take care of conflicting properties (e.g., via Eiffel's renaming mechanism).

IV. Case Study: CookieCAD

Computer-Aided Design (CAD) [8] promotes the use of computer systems to assist the creation, modification, analysis and optimisation of a design. A design may describe any physical, 3D object, e.g., engines or planes for analysing heat and fluid transfers, bridges and wind mills to estimate the dynamic response of the mechanical structure to various environmental variations. CAD is often paired with Computer-Aided Manufacturing (CAM) to help plan, manage and control the operations around the process of bringing 3D designs to life. Historically, techniques and tools heavily rely on geometry, solid and surface mathematical formalisms. This particular set of processes operated over a selected bunch of formalisms and languages makes CAD a paradigm on its own.

We propose to illustrate our formal framework with a simplified variant of CAD named CookieCAD (CCAD) that helps design simple cookie cutters, as illustrated in Fig. 2(a). A cookie cutter adopts a simple shape (triangle, rectangle, star, etc.) represented by 2D geometric lines (noted L) which are specified based on the definition of two points in no particular order (noted P) placed in a cartesian plane (using x-/y- coordinates). In order to be manufacturable, points shall not be placed too close to each other (the exact distance tolerance depends on the machinery used), and shall represent closed polygons that may share some lines (as illustrated in Fig. 2(b) for a house-like cutter, with the triangular roof placed over the rectangular base). Consequently, lines shall not cross each others as illustrated in Fig. 2(c), which results in an *invalid* cookie cutter. At manufacturing time, such a design shall be associated with a width to build a 3D physical object: the precision of the machine may finally discard some of the designs presented on Fig. 2(a) if it is not able to cut or fold metal pieces that size.

Figure 3 depicts a possible, simplified xFTG for CCAD named $\mathsf{FTG}_{\mathsf{CCAD}}$. It includes four transformations templates represented as double-rounded ellipses named Design, Validate, EstimateCosts and Manufacture. As an example, a repository would have to retrieve a transformation corresponding to the following signature: $\mathsf{Validate} \mapsto (\langle \mathsf{CCAD} \rangle, \langle \mathsf{CCAD}, \mathsf{Boolean}, \mathsf{Trace} \rangle, \top)$, meaning that Validate is automatic and takes as source a CCAD model and produces back the source CCAD (notice the double arrow from/to CCAD), and a boolean value, indicating whether the source model is valid or not, in which case a Trace model is produced. The target CCAD model may be discarded by some tools, but nothing prevents a repository to select transformations that may additionnally decorate the target with information from the trace model to help CCAD designers grasp their errors' origins more easily, or keep traces and models cleanly separated, since both behaviours match Validate's signature. All

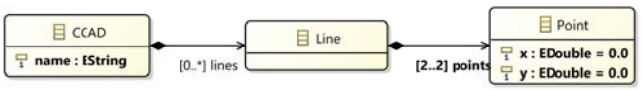


Fig. 4: Template metamodel CCAD in $\mathsf{FTG}_{\mathsf{CCAD}}$ (cf. Fig. 3).

transformation and metamodel names in Fig. 3 are templates (meaning they belong to TMMName or TTName) except Boolean and Real, which is used as a source parameterising the width of the cookie cutter design in order to produce the physical 3D cutter.

These template transformations make a central use of the CCAD language that would handle the visual representation, design, validation, debugging, etc. of a cookie cutters designs, based on the template metamodel of Fig. 4: a CCAD model is defined by a set of lines delimited by exactly two points, in no particular order, having real coordinates on a 2D plane. The semantics of such a metamodel captures all points in an Euclidian plane between any pair of points defining a line.

Fig. 5a depicts a candidate, well-formed PMs named $\mathsf{PM}_{\mathsf{Full}}$. It arranges three transformations from $\mathsf{FTG}_{\mathsf{CCAD}}$ in a sequential fashion: starting from scratch, a designer starts iteratively create a design, then manufacture it once it has been validated. This workflow does not prohibit manufacturing of expensive designs, since EstimateCosts is left out. Enforcing such a scenario requires specifying properties specifically targeting $\mathsf{PM}_{\mathsf{Full}}$'s topology, asking at least EstimateCosts's presence in the PM in order to break $\mathsf{PM}_{\mathsf{Full}}$ well-formedness. Refering to Tab. I, CAD_1 and CAD_2 are easily recognised as property patterns over a metamodel manipulating CAD designs; while CAD_3 specifies a property characterising the existence of a transformation that produces a final, real-life 3D product. Fig. 4 shows possible patterns for capturing them in a MOF-like syntax. Assuming one has explicit languages for expressing the required properties, this would built a set of paradigmatic properties $\pi_{\mathsf{CAD}} \in \Pi$ that constitute part of our CAD paradigm (i.e. $\mathsf{CAD} \mapsto \pi_{\mathsf{CAD}}$ in the sense of Def. 7).

We have built a paradigmatic structure $\mathsf{PS}_{\mathsf{CCAD}} = (MM, \{\mathsf{W}_{\mathsf{CCAD}}\}) \in \mathbb{PS}$ with only one workflow $\mathsf{W}_{\mathsf{CCAD}} = (\mathsf{FTG}_{\mathsf{CCAD}}, \mathsf{PM}_{\mathsf{Full}}) \in \mathbb{W}(TR, MM)$ relying on transformation TR and metamodel names MM as described in Fig. 3. *Now, does* $\mathsf{PS}_{\mathsf{CCAD}}$ *qualify as a* CAD *(simplified) paradigm?*

Properties CAD_1 and CAD_2 are easily checked on the CCAD metamodel by matching the classes from the template properties to their corresponding class (namely, L and P to Line and Point, then references and b to x and y, and finally S to CCAD itself). Property CAD_2 may be matched to the transformation template Manufacture, with a template Product yet to be retrieved from the repository.

V. Conclusion

To deal with the complexities in designing CPS, MPM is seen as a potential candidate for helping project managers and engineers.

This paper motivated the need of MPM for CPS and identified three important dimensions: formalisms / languages; abstractions / viewpoints; and processes. It proposes, as a first step, a partial formalisation of two of these to capture the notion of paradigm: after defining the core characterising properties of a paradigm, we offer a decision procedure to check that a structure comprising two of the three previous dimensions satisfy these properties. This was illustrated with a (simplified) CAD paradigm. Many directions still need to be explored: adding multi-view properties as well as going from a single to multiple paradigms as well as specifying how they relate to each other (through abstractions/multiviews) and characterising such relationships; then validating the approach on realistic/industrial CPS examples.

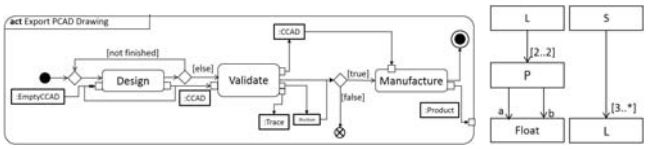


Fig. 5: (Left) Candidate PM for manufacturing CCAD designs. (Right) Pattern properties for CAD_1 and CAD_2.

References

[1] T. Kuhn, *The Structure of Scientific Revolutions*. Chicago Press, 2012.

[2] H. Vangheluwe, G. Vansteenkiste, and E. Kerckhoffs, "Simulation for the Future: Progress of the ESPRIT Basic Research working group 8467," in *European Simulation Symposium (ESS)*. SCS, 1996.

[3] H. Vangheluwe and G. Vansteenkiste, "A multi-paradigm modeling and simulation methodology: Formalisms and languages," in *European Simulation Symposium (ESS)*. SCS, 1996, pp. 168–172.

[4] F. P. Brooks, Jr., "No silver bullet – essence and accidents of software engineering," *Computer*, vol. 20, no. 4, pp. 10–19, Apr. 1987.

[5] Y. Van Tendeloo, "A Foundation for Multi-Paradigm Modelling," Ph.D. dissertation, University of Antwerp, 2017.

[6] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. A. Henzinger, and K. G. Larsen, "Contracts for system design," *Foundations and Trends in Electronic Design Automation*, vol. 12, no. 2-3, 2018.

[7] P. Wegner, "Dimensions of Object-Based Language Design," *SIGPLAN Notices*, vol. 22, no. 12, pp. 168–182, Dec. 1987.

[8] M. P. Groover and E. W. J. Zimmers, *CAD/CAM: Computer-Aided Design and Manufacturing*, P. Hall, Ed. Prentice Hall, 2008.

[9] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, "Safe Model Polymorphism for Flexible Modeling," *Computer Languages, Systems and Structures*, vol. 49, no. C, pp. 176–195, 2017.

[10] J. De Lara and E. Guerra, "A posteriori typing for model-driven engineering: Concepts, analysis, and applications," *ACM Transactions on Software Engineering Methodology*, vol. 25, no. 4, pp. 1–31, 2017.

[11] R. Salay, J. Mylopoulos, and S. Esterbrook, "Using Macromodels to Manage Collections of Related Models," in *CAiSE*, 2009, pp. 141–155.

[12] F. Durán, S. Zschaler, and J. Troya, "On the Reusable Specification of Non-functional Properties in DSLs," in *SLE*, 2012, pp. 332–351.

[13] A. Kusel, J. Schonbock, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Swinger, "Reuse in Model-To-Model Transformation Languages: Are We There Yet?" *SoSyM*, vol. 14, no. 2, pp. 537–572, 2015.

[14] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith, *The Java Language Specification*, se 12 ed. Oracle USA, 2019.

[15] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. Selim, E. Syriani, and M. Wimmer, "Model Transformation Intents and Their Properties," SOSYM, vol. 15, no. 3, pp. 647–684, 2014.

[16] S. Mustafiz, J. Denil, L. Lúcio, and H. Vangheluwe, "The FTG+PM Framework For Multi-Paradigm Modelling: An Automotive Case Study," in *MPM Workshop*, 2012, pp. 13–18.

[17] L. Lúcio, S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukss, "FTG+PM: An Integrated Framework For Investigating Model Transformation Chains," in *International SDL Forum*, 2013, pp. 182–202.

[18] L. Lúcio, S. Mustafiz, J. Denil, B. Meyers, and H. Vangheluwe, "The Formalism Transformation Graph As A Guide To Model-Driven Engineering," McGill University, Tech. Rep. SOCS-TR2012, 2012.