Towards The Systematic Design of Model Animation: Key Ingredients and General Guidelines

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

Abdelkader Ouared
Abdelkader.Ouared@unamur.be
Faculty of Computer Science / NaDI,
University of Namur
Namur, Belgium

Pierre-Yves Schobbens
Pierre-Yves.Schobbens@unamur.be
Faculty of Computer Science / NaDI,
University of Namur
Namur, Belgium

ABSTRACT

Model Animation (MA) is a practical technique for providing modellers and Model Transformation designers an insurance that their models behave as expected. This is specially relevant in expertise domains where models have a natural visual representation, i.e. a dedicated concrete syntax. MA can be seen as the visual representation of Model Transformation simulation. It supports Model Transformation designers understand, trace, monitor, and ultimately debug their specification using visual clues.

In contrast to other techniques surrounding Model-Driven Engineering, MA has received less attention, in contrast to, e.g. testing or debugging. This paper is a first step towards the systematic engineering of model animators. It identifies three key challenges: (i) how to effectively, explicitly and precisely define the concrete syntax to enable MA; (ii) how to build an MA language to express animations units in a compositional way, so that animations become flexible in their definition, and reusable across several Dsls; and finally (iii) how to explicitly relate MA units with their transformation counterparts, to avoid reimplementing the transformation scheduling.

We analyse these challenges to extract some requirements for future animators, and give a partial conceptual proposal that fulfil them, paving the way towards the creation of a family of animation tools that would work alongside transformation engines. We then show, on simple examples, how these propositions apply and to which extent they promote flexibility and reuse.

CCS CONCEPTS

• **Software and its engineering → Visual languages**; • **General and reference → Surveys and overviews**.

KEYWORDS

Model Transformation, Animation, Visual Representation

ACM Reference Format:

Moussa Amrani, Abdelkader Ouared, and Pierre-Yves Schobbens. 2022. Towards The Systematic Design of Model Animation: Key Ingredients and General Guidelines. In *Proceedings of ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3550356.3561607

MODELS '22 Companion, October 23–28, 2022, Montreal, QC, Canada

ACM ISBN 978-1-4503-9467-3/22/10…$15.00
https://doi.org/10.1145/3550356.3561607

1 INTRODUCTION

Investing the time and efforts to develop the tooling associated with general-purpose programming or modelling langages, such as analysers, debuggers, and testing frameworks, is a no-brainer, because of the large audience it will benefit. The same question for Domain-Specific (Modelling) Languages (Dsmls) has a more mitigated answer. For example, in recent years, many efforts have been invested in defining guidelines for the systematic design and development of Dsl debuggers [5, 8, 31], but also various analysis tools [22], resulting in tool support that automate most part of the development. This helps reduce the time and efforts Dsl tool builders have to invest to offer, alongside of their Dsl, supporting tools that are nowadays considered essential for the activity of Dsl modelling.

A complementary, lightweight approach for ensuring correctness and quality of Dsls, in particular concerning their behaviour, is the adoption of Model Animation (MA). MA can be seen as the visual representation of the simulation of Model Transformations (MTs) [21]: it gives a visual counterpart to the changes and updates operated on a model during its execution. Just like debugging, MA assumes that an MT captures the behavioural semantics of a Dsl, namely an inplace, endogenous *simulation*, and that the MT engine is capable of interrupting its execution in key places so that the modeller can inspect visually what is happening. However, since MA heavily relies on model visualisation, another crucial element is necessary: the definition of a *visual* concrete syntax for the model, and an associated MA engine that has the ability to alter graphical features of the concrete syntax fast enough to confer the illusion of movement, hence the animation. Relying on visual information, MA provides an insight and feedback about what is going on during a simulation that would help modellers to understand, analyse, and predict what their models do by relying on visual information, especially those who have little background in programming. It should also support MT designers for understanding, tracing, monitoring, and ultimately debugging their specifications based on visual clues.

Many Dsl tools already offer the ability to perform MA alongside of MT. eProvide, [28] (now discontinued) was one of the first tools that included a "visual debugger", which required to alter the Dsl metamodel to accommodate with the debugging logic, and relied on simple graphical components. More recently, GeMoC [6] and AtoMPM [30] are two well-established Dsl frameworks that both integrate debugging and MA. The animators are tightly coupled with the underlying MT Language (MTL) for performing animation

(although GeMoC offers several ones, namely Kermeta and Xtend), since the execution engine is instrumented to stop at specific locations in the MT to perform animation pieces. An interesting feature of AtoMPM is the ability to take advantage of the concrete syntax to express the MT rules themselves, reducing the gap between MT specification and MA. Other tools based on formalisms that have native graphical representations (such as Finite State Machines and Petri Nets) offer animators with little extra efforts since the MT is directly expressed using the visual MT, whose execution is simply rendered visually. However, these tools force MT designers to specify their transformations in a formalism that does not directly manipulate their metamodel's features.

Although these tools all represent interesting steps and real progress in the matter of MA, they sometimes lack *flexibility* in the MA specification, and *reuse* of MA units across DSLs. In this paper, we take a first step towards the systematic engineering of MA, by identifying key components for designing MA tools, such that they ensure our required properties of flexibility and reuse. We formulate three challenges that touch upon the core components of animators: the concrete syntax and the associated MA engine; the MA specification; and the relationship between MA and MT. We detail and elaborate these challenges by showing on simple examples what we intend by flexibility and reuse, and how a compositional DSL dedicated to animation can help with these requirements.

The paper is organised as follows. We start in §2 by clarifying the notion of MA in contrast to closely related notions, and motivate our challenges with concrete situations. We review in §3 some popular DSLs, and devise a family of animation for each of them, showing that animation should be left as a design choice, instead of being forced by an execution engine. We then revisit the challenges in §4, to identify key ingredients for each of them that would support the ability to build animators. To conceptually meet the challenges, we formulate a (partial) proposal in §5 and show on our example DSLs how some of the challenges find an interesting answer. Finally, we discuss Related Work in §6 before wrapping up in §7 with concluding remarks.

2 BACKGROUND & MOTIVATION

Figure 1 (left) depicts the various components of (a simplified, unary) MT. An input model M_{in}, conforming to a source metamodel MM_{src}, is transformed into an output model M_{out}, hopefully conforming to a target metamodel MM_{tgt}, by the execution of an MT specification.

MTs are not created equal: whereas they may seem syntactically, and sometimes semantically, similar, they may serve different purposes (known as MT intents [21] in the literature) that need to be adequately distinguished. For the purpose of this paper, three different MT intents are interesting to precisely define, because they often raise confusion in the literature. For an input model M,

Visualisation is an intent characterising an outplace, heterogeneous MT aiming at visually *representing*, or *rendering*, M through the definition of a *visual* concrete syntax mainly composed of graphical symbols.

Simulation is an intent characterising an inplace, endogeneous MT aiming at defining M's operational semantics, i.e. how M evolves over time during execution.

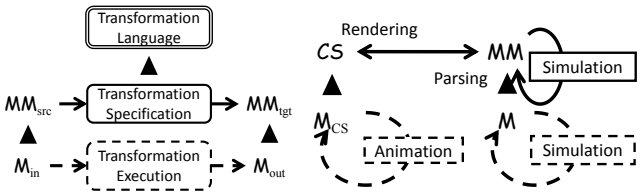


Figure 1: Model *Transformation* (MT) (left) and Model *Animation* (MA), as a special case of MT, operating on the *concrete syntax* of model M, which is involved in an endogenous, in-place *simulation* MT. Metamodels MM and MM_{CS} may be related through rendering/parsing

Animation is "*the visual representation of a simulation*" [21], i.e. it visually renders the computation steps defined by the MT simulation specification, by visually projecting the changes operated on the simulated model through its (visual) concrete syntax.

Summarising, an *animation* is tightly connected to a *simulation* through the use of a(n existing) *visualisation*. In other terms, designing an MA requires the following components, as depicted in Figure 1 (right):

- A Concrete Syntax CS that should be defined over MM, so that it becomes possible to *visualise* M as a graphical model M_{CS}.
- MM's operational semantics is specified in a *simulation* MT (specification).
- *during* each run of the simulation's execution, the computing steps are visually rendered over M_{CS}.

Just like MTs are built over transformation *units*, i.e. basic building blocks that are composed to realise larger, complex transformations, it is likely that MA requires a similar decomposition process into basic "units" to build and organise them. However, MTs are likely specified independently of any MA, primarily for capturing the DSL's behaviour, but also likely for supporting other activities (typically, code generation, and various analyses, among others). Since both an MT and an MA for a given model M exist for different purposes, it seems reasonable to build them separately, and even to entrust their specification to different persons, or teams, thus creating a new *role* in MDE: the *MA Designer*, who need adequate methodologies and tools to perform the job.

Although serving different concerns, MT and MA are by nature tightly connected, since they both encode a DSL's behaviour. A Model Transformation Unit (TU) is the smallest, coherent unit of computation visible from the outside. An interesting approach would consist in directly relating, or connecting, the (execution of) a Model Animation Unit (AU) to (the execution of) a TU, so that the changes performed inside the TU are visually rendered.

However, to fully unlock the potential of MA engineering, the specification of AUs should be completely decoupled from the specification of (and languages for) TUs. As they serve a common purpose, a mechanism for explicitly relating AUs to corresponding TU(s) is required for this vision to work: when the execution engine of the MTL executes a TU, it passes information to the MA engine to concurrently run the animation. This vision is directly inspired

Figure 2: Relationship between Transformation (TU) and Animation (AU) Units for promoting multi-abstraction, and reuse.

by existing methodologies for designing debuggers for MTLs [5, 31]: to avoid inspecting models that may be inconsistent during the execution of a TU, a so-called debugging *step* is defined by annotating TUs relevant for inspection. An execution *step* provides the support for animation steps.

Conceptually, TUs and AUs should be in an N:N relationship, as depicted in Figure 2

- A TU may be connected to several AUs, in order to provide different abstraction levels, and multiple views on the same execution (part). Typically, a beginner may need detailed visual information to understand a specific part of a model's execution, while a more advanced user may be happy with minimal animation. We illustrate this by providing several animations for typical DSLs in §3.
- An AU may be connected to many, but different, TUs, likely integrated in simulations for different models. Associated with appropriate composition operators, this approach promotes the definition of AUs libraries that capture popular and widely used animation *patterns*, thus enabling modularity and reuse across DSLs. We show in §4.3 how similar AUs may be reused for rendering different MTs.

To summarise our vision for enabling a systematic design of MA, we assume that a DSL Engineer has already defined the core components, i.e. a metamodel capturing the abstract syntax, one concrete syntax allowing to create, edit and manage conforming models, and a simulation that captures the DSL's behaviour, specified in an MTL that defines identifiable TUs. The missing steps, and corresponding challenges, for an MA Engineer, would be the following:

C1. How to explicitly model a Concrete Syntax, with sufficient details to support animation? Since MA basically consists of continuously updating the concrete syntax of a DSL along its execution, an explicit representation of the concrete syntax that exposes the graphical features to the MA designer is required.

C2. How to explicitly model an MA *compositionally*, i.e. by creating an MA using basic components that are progressively composed in a fully-fledged animation?

C3. How to abstract away from the peculiar details of a given DSL, thus promoting *reuse*, and to explicitly map TUs with their related AU(s), to support *flexibility*?

3 DSL EXAMPLES

We now present three simple DSMLs, namely PacMan, Finite State Machines (FSM), and Petri Nets (PN), following the same outline:

- The DSL's structure is specified using a metamodel expressed in MOF; we also informally specify a possible *concrete syntax*, then provide a simple witness model.
- The DSL's semantics is sketched, without details of implementations that highly depend on (i) the MTL used, and (ii) the modelling style of the MT designer. We however show how possible implementations in different MTLs, for PacMan, may lead to a similar MT structure (and TUs supporting animation), in a way that is agnostic of the underlying MTL.
- A family of animations are then described in detail, and numbered for future reference.

Note that we assume that the DSL structure follows the Executable DSML Pattern [7], which consists of two separate metamodels for defining executable DSMLs:

- The so-called *Domain Definition* metamodel captures the *static* structure of the DSL, which typically serves for defining valid models using various textual and/or visual syntaxes;
- The *State Definition* metamodel adds new information on top of the Domain Definition metamodel to enable state-based execution, thus defining those elements that are modified during execution.

Both metamodels typically need to be merged, using different techniques (e.g., using MOF's "package merge" approach [24], or other composition techniques [1]). For clarity and presentation simplification purposes, we depicts both metamodels in a merged fashion, although we visually distinguish each part: the Domain Definition metamodel elements are represented using the regular font with plain arrows for MOF references; while the State Definition elements use curvy fonts and dotted MOF references.

3.1 PacMan: A DSL for the PacMan game

The PacMan game is a popular arcade game that gained interest in the MDE community because it captures a well-known, simple reactive DSL with an easily understandable concrete syntax, and presents interesting real-time features for execution.

3.1.1 Specification. On the top left compartment is represented a metamodel MM_{PM}, as created by a DSL engineer. A Game consists of a sequence of Levels, each displaying a Maze where Personas evolve. A Level terminates when PacMan is eaten by a Ghost, or when it has eaten all Cookies. Several players may compete for the highest Score.

On the top right compartment, a basic model M_{2x2} with a unique Level of size 2x2, as possibly created by a modeller, is depicted using three different concrete syntaxes: textual for ${}^{TXT}M_{2x2}$ (inspired by eMotions [26]); based on UML Object Diagram [27] for ${}^{OD}M_{2x2}$; and freely inspired by the real game for ${}^{Viz}M_{2x2}$, the two latters being graphical.

3.1.2 Execution. PacMan is a *reactive* DSL, i.e. it reacts to external stimuli given by a player deciding which way to go. The bottom compartment of Figure 3 describes a simple rule *moveR* (moving Pac-Man on a Cell at its right, if available) as part of the simulation MT specification TS_{PM}, implementing a fragment of MM_{PM}'s executable semantics. The first MT, TS_{Xtend}, uses metaprogramming (based on Xtend with GeMoC [19]). The next and last ones are based on Graph Transformations: TS_{GBT} relies on ${}^{OD}M_{2x2}$ to express the rewriting (as would be expressed e.g. in Henshin [3]); while TS_{VIZ} relies on ${}^{Viz}M_{2x2}$ (as would be expressed e.g. in AtoMPM [29]). Finally, TS_{FSM} presents an MT fragment expressed

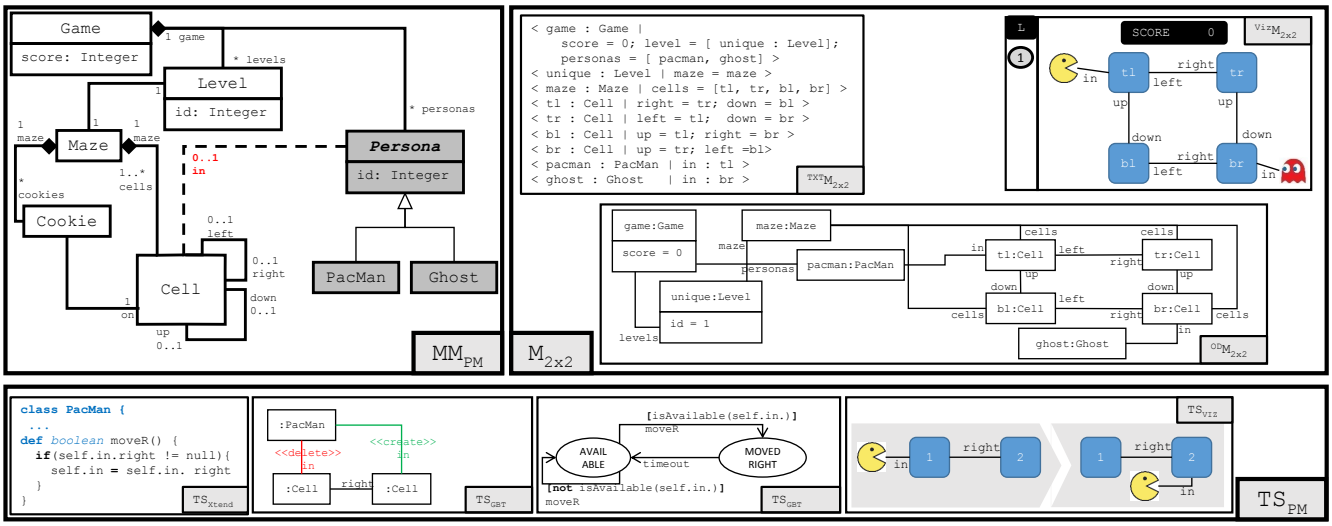


Figure 3: Specification of a DSL for the PacMan game.The DSL Engineer creates the metamodel MM_{PM} (following the Executable DSML Pattern [7]); an MT designer specifies a transformation TS_{PM} (composed of several transformation units such as moveR depicted here).

with a UML's Finite State Machine [27]. Note that all MT specifications (fragments) but TS_{Xtend} present a graphical representation, showing that MT specifications may well be graphically visualised as well.

3.1.3 Animations. The PacMan game typically uses three kinds of animations for different situations:

A Persona **moves.** Typically, a specific animation could be attached to each movement, as they are traditionally encoded in different rules/operations (cf. e.g. *moveR* as specified in Figure 3, but also *moveL*, *moveU* and *moveD* for moving left, up and down). Depending on whether the MTL allows using abstract classes (which would be Persona here) for MT specification, these rules/operations may need to be duplicated.

PacMan **eats a** Cookie**.** This occurs when PacMan is on a Cell that contains a Cookie, making it disappear and triggering a Score update.

PacMan **is eaten by a** Ghost**.** This occurs when PacMan is on the same Cell as a Ghost, which may occur by one Persona making a move to an already occupied Cell.

To summarise, we would have to define four different animations to fully animate the PacMan DSL:

PM.1 A Persona disappears from one Cell and reappears on another (adjacent) Cell.

PM.2 Assuming PacMan is on a Cell containing a Cookie, the Cookie disappears.

PM.3 The Score is updated by a predefined increment.

PM.4 Assuming PacMan and a Ghost are on the same Cell, PacMan disappears.

Note that those animations are not completely unrelated. First, **PM.2** and **PM.3** need to be conducted sequentially quickly enough to not notice a time gap. Second, **PM.2** and **PM.4** appear to be very similar in nature: they both assume that two objects are located on the same Cell before making one of them disappear.

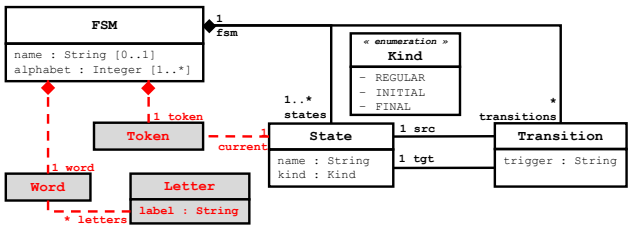


Figure 4: A metamodel for Finite State Machines.

3.2 FSM: A DSL for Finite-State Machines

Finite State Machines (FSM) represent a common DSL for capturing state-based behaviour of various biological, but also computational domains (e.g. Chomsky's Regular Grammars, among others). We consider FSMs that are simplified in various ways: in particular, we target a *word-acceptance* semantics, where transitions do not contain guards and triggers are reduced to their simplest expression, namely simple strings.

3.2.1 Specification. Figure 4 specifies the metamodel of the Finite State Machine DSL. An FSM is composed of States identified by a name, and Transitions that contain a simple trigger. Figure 5 depicts a simple model consisting of three States and three Transitions, to recognise the regular expression $(a \cdot b)^\star\, b$.

We assume the classical visual representation for FSM: a State is represented by a circle labelled with its name; and a Transition is represented by an arrow pointing to its tgt and labelled by its trigger. We represent the current State by surimposing a red

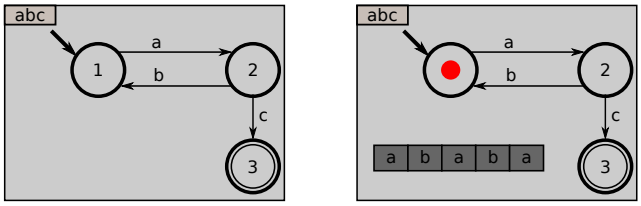
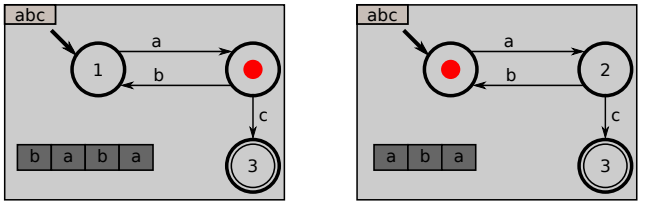
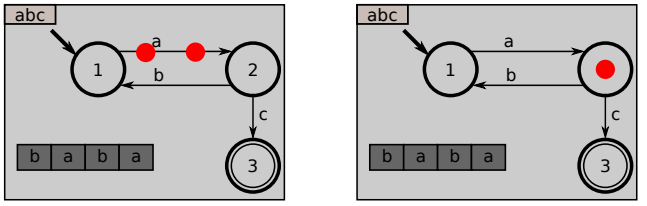


Figure 5: The abc **FSM: a simple model conforming to the specification of Figure 4 (left), depicted with the usual concrete syntax, and the starting point of animation of the acceptance of the word** a · b · a · b · a **(right)**



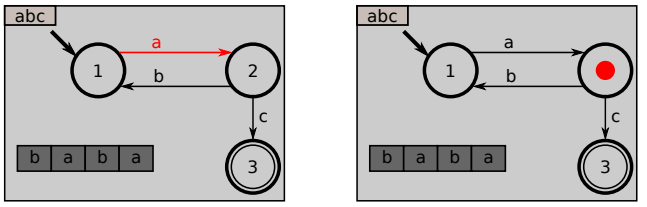
(a) ***FSM.2.1:*** **Firing** Transition a **and reaching** State **2, while consuming the first** Letter a.

(b) ***FSM.21:*** **Firing** Transition b **and reaching** State **1 again, while consuming** b.



(c) ***FSM.2.2:*** **First, sliding the** Token **along the fired** Transition **(with two** Token **snapshots to reproduce the visual effect).**

(d) ***FSM.2.2:*** **Then, moving the** Token **to the target** State



(e) ***FSM.2.3:*** **First highlighting in red the fired** Transition **(together with its** trigger**).**

(f) ***FSM.2.3:*** **Then, moving the** Token **to the target** State

Figure 6: Typical animation steps illustrated: ***FSM.2.1*** **with two acceptance steps (top);** ***FSM.2.2*** **for sliding along (middle);** ***FSM.2.3*** **with transition blink (bottom).**

rounded form (that we call *token*) over the corresponding State (cf. Figure 5).

3.2.2 Execution. The word-accepting semantics is encoded in a transformation called accept that reads a Word and traverses the FSM. A Word is accepted iff the State reached when the Word becomes empty is FINAL .

3.2.3 Animations. There are typically two categories of related animations, one for showing the progression on the word, the other showing the FSM's current state.

FSM.1 A Letter is consumed from the Word being accepted. This may be performed in two ways:
- **FSM.1.1** The Letter (and its box) is removed from the Word;
- **FSM.1.2** The Letter is ~~struck out~~, but is present and still readable.

FSM.2 A Transition is fired, thus deactivating the current State and activating the State the Transition points to. This may be realised in different ways:
- **FSM.2.1** The token disappears from the current State and simply reappears inside the updated current State;
- **FSM.2.2** The token disappears from the current State, and slides along the entire arrow of the fired Transition, then appears in the updated current;
- **FSM.2.3** The fired Transition blinks in red for 2 seconds, then the token disappears from the current State, before reappearing in the updated current.

Here again, **FSM.1** and **FSM.2** are timely related. An MA designer may choose different solutions: a sequential, concurrent (i.e. indeterministically performing them in an interleaved way), or a precisely timed approach may suit different needs. Note that the three versions of **FSM.2** display different MA *abstraction details* for the same TU, corresponding to different levels of expertise.

3.3 PN: A DSL for (Simple) Petri Nets

The formalism of Petri Nets (PNs) is popular for modelling various physical as well as computer systems that include concurrency, as it allows formal verification of interesting properties (typically, reachability, safety and liveness) [13]. We consider here simple Place/Transition PNs with weighted arcs with their traditional firing semantics.

3.3.1 Specification. Figure 7 specifies the metamodel of a PN DSL. A PN is a bipartite graph whose nodes are Places and Transitions, and whose edges are called *weighted* Arcs. The marking represents tokens hosted by a Place; it will change during execution along Transition firing.

We assume the traditional visual concrete syntax of PNs: a Place is represented as a circle, a Transition as a black thin box, and an Arc as a directed arrow carrying a label representing its weight (omitted when equals to 1).

3.3.2 Execution. Starting with an initial marking, i.e. setting the values of marking for each Place appearing in a model, executing a PN consists in sequentially *firing* enabled Transitions, until no Transitions are enabled any more. A Transition t is *enabled* iff each input Place (i.e. a Place connected to t by an Arc directed to t) is marked with at least the weight of the Arc. Firing t consumes tokens from all input Places and creates tokens on output Places, according to the corresponding output Arcs' weight. Note that

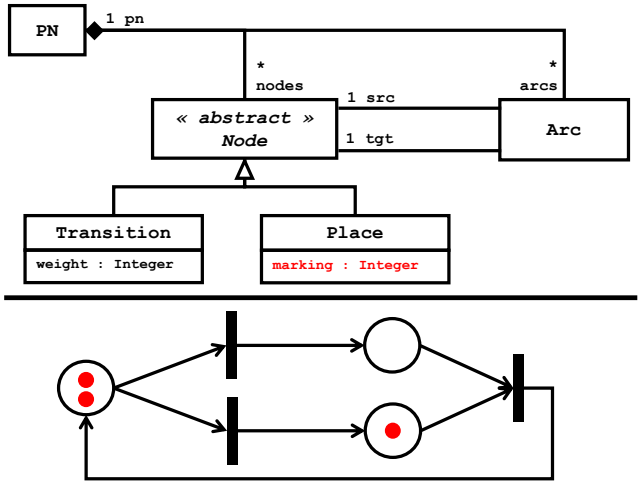


Figure 7: A metamodel (top) and model (bottom) for Petri Nets.

an enabled Transition may not fire, denoting concurrency and non-determinism.

3.3.3 Animations. The animations are similar to the ones for the FSM Dsl, except that Places may host multiple markings, and that the Transitions do not hold them. We highlight some variants for animating PNs.

PN.1 Highlight all Transitions that are enabled;
PN.2 Highlight *the* Transition, among all enabled ones, that has been selected for firing;
PN.3 Firing a Transition, which may be realised using different animations:
- **PN.3.1** Removing, at once, all markings from input Places and creating new markings on the output Places.
- **PN.3.2** Removing/Creating the markings, but sequentially on all input Places.
- **PN.3.2** Removing/Creating the markings as before, but "sliding" the tokens along the Arcs to visually represent the marking transfer.

Again, all those three animations are related in complex ways: it may be possible to perform them step by step, or automatically, or even interactively leave the choice to the user to select which Transition has to be fired.

4 CHALLENGES

We now revisit the Challenges formulated at the end of §2, to identify key ingredients and to provide general guidelines for building model animators that would allow flexibility and reuse.

4.1 Concrete Syntax (CS)

The first challenge, Concrete Syntax (CS), is related to a Dsml's metamodel MM through two transformations:

$$\mathrm{MM} \underset{\text{parsing}}{\overset{\text{rendering}}{\rightleftharpoons}} \mathrm{CS}$$

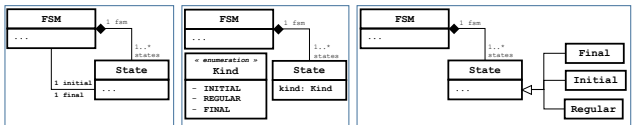


Figure 8: Modelling a State in FSM: with explicit references `initial` and `final` (left); with a distinguishing property `kind` (middle); and with inheritance (right).

In syntax-directed editing, *parsing* is not necessary because models are created by relying on the syntax, thus preventing ill-models. On the other hand, *rendering* is crucial for representing a model when appropriate, but it also represents the anchor for animation.

Having explicit MTs between MM and CS assumes indeed that CS is explicitly metamodelled: following the Mde methodology, such a Dsl should cover the definition of visual representations (geometric shapes as well as various data structure representations such as tables, graphs, etc.), but also the ability to define the rendering relationship, i.e. patterns associating MM's metamodel elements with CS's elements. We identified at least four challenges:

CS.1. Providing *simple* visualisation for data structures Providing configurable representations for common data structures such as (multi-column) tables, graphs, sets and lists present the benefit of rapidly presenting information from a model. This ability should rely on queries to identify, and eventually modify, the elements of the model used for "populating" such structures.

CS.2. Providing *complex* rendering patterns. The partial metamodels in Figure 8 specify the *initial* State in four different ways: by referencing to the appropriate State among those available, by characterising each State with a property kind, or by using type inheritance. Many tools would only support the latter, because they essentially only allow to associate various icons to classes. This approach is too simplistic and forces metamodel designers to tweak their metamodels towards MA, introducing yet another metamodel specialisation (as it is already the case for model editing and model analysis, among others). Complex rendering patterns should allow to freely associate visual counterparts to various combinations and values of metamodel elements.

CS.3. Integrating "insideness" *natively* We call *insideness* the visual equivalent of *containment* in metamodelling, i.e. the ability to uniquely put elements *inside* inside a container so that elements inside disappear along with the container's removal. This feature presents two advantages. First, it would allow to natively handle common situations like including text inside a shape (e.g. displaying a State's name within the circle for an FSM) and then treat them in an universal manner. Second, if the feature is customisable, it would enforce a natural semantics graphically with an expected behaviour.

CS.4. Providing "snapping" capabilities Snapping helps a precise arrangement of graphical elements by "gluing" them to a specific target, e.g. a canva's boundary, a grid, or points on objects.

CS.6. Supporting animations natively Having at our disposal a Dsl for defining CSs brings the ability to *visually* render

metamodels, but does not guarantee the ability to perform animation. For that, the DSL should natively support the fast update of visual features that compose the CS (e.g., the position, size, color, type of line for an FSM's Transition).

4.2 Model Animation

To tackle Challenge C2, it is important to design the MA framework with the idea of reuse at its core. One possible approach is to separate two things: the basic blocks defining animations *effects* that may be combined into comprehensive Model Animation Units (AUs), from the way they are organised and scheduled to build complex animations. The approach we propose is similar to the de-/reconstruction of MTLs described by Syriani and Vangheluwe [29]: the idea is to capture the most basic animation effects, and offer powerful mechanisms to combine them, resulting in the ability to effectively specify any kind of animation. We quickly discuss basic effects before describing two approaches for scheduling.

4.2.1 AU Effects. Effects may be roughly classified into four categories, depending on the features they act on:

Creation effects bring an element into a diagram in various ways, e.g. by simply making it appear at the right position, or by "zooming" it in, i.e. starting from a small size until reaching its regular size, while preserving the element's proportions.

Deletion effects remove an elements from a diagram, e.g. by making it disappear, or by "zooming" it out.

Action effects update one, or several of the visual features of an element, e.g. the color, size, etc. of a text, a shape or a connector.

Path Motion effects move an element along a predefined path, e.g. a straight or curved line, or the line represented by a connector.

We expect that most of the MAs would make use of a small set of basic effects. These effects could be organised into libraries and potentially exchanged among MA specialists, but their reuse across different platforms may be hindered by how effects are highly dependent on the way the graphical components of the CS are defined.

As illustrative examples, PM.3, FSM.1.2 and PN.1 and PN.2 all falls into the *Action* effects category, because only the graphical features (or values) of the score label, the Letter and the color of the Arc(s) are modified. Figure 5 (middle) graphically illustrates a possible way to realise a *Path Motion* effect. Finally, *Creation* and *Deletion* effects are combined to realise a complex animation in FSM.2.1, PM.1 and PN.3.2.

4.2.2 AU Scheduling. Although theoretically equivalent, two approaches are possible for realising this requirement.

Embedding the scheduling *inside* AUs As suggested in Fig. 9 (top), each AU is extended with pre- and post-conditions. A precondition indicates how an AU starts relatively to the previous one (at the same time, after, or delayed by a given time); while a postcondition would define whether the AU is repeated (and how many times).

Define *combinators outside* of AUs Another approach, suggested in Figure 9 (bottom) is to provide explicit *combination* operators (aka. *combinators*) that would take as operands AUs. The same combinators, i.e. sequencers with a possible delay, repetitors, and parallelisation, may be defined.

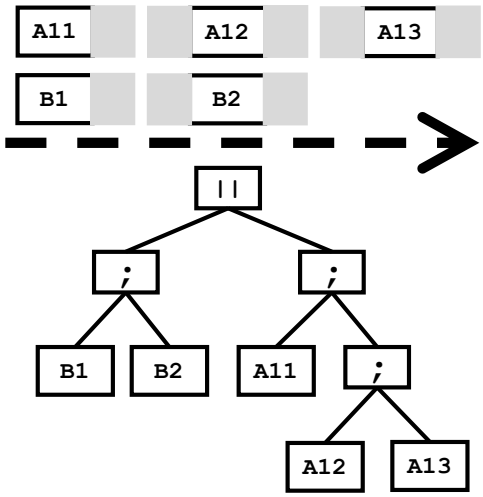


Figure 9: On top, scheduling is realised with *embedding*: each AU has pre- (except the first) and post-conditions, represented as grey blocks attached before and after the AU. On bottom, the same animations but with AUs organised with two combinators: || for parallel, and ; for sequential combinations (the temporal details are omitted).

The second approach may seem more familiar to MA designers with a programming background, because the combinators approach proposes constructions similar to what is available in GPLs. However, the first approach may be more compact visually, and familiar for MA designers accustomed to slideshow tools (the most popular and mainstream ones being Microsoft Powerpoint, Apple Keynote and Google Slides, among a plethora of other dedicated tools, many of which only existing online).

Note that both approaches may concurrently exist, allowing MA designers to choose the most appropriate approach for specifying their MA. It is left as future work to study under which conditions they may coexist (if possible at all), and to identify when and how each presents more benefits than the other.

4.3 Parameterisation of Animation

What we mean by *parameterisation* is the ability, for an AU, to separate the internal animation logic so that it becomes possible to apply it consistently to elements from different DSLs, thus promoting reuse. Taking our DSLs of §3, we see that PM.1, FSM.2.1 and PN.3.1 all involve the same animation logic (*disappear* from the current location, followed by *appear* on another location), but applied to different elements: in PM, it concerns the Persona; in FSM, it concerns the Token; and in PN, it iteratively applies to the markings.

Figure 10 shows how parameters could be integrated into animations. Two basic animation units APP (making an element appear) and DAPP (making an element disappear) with their respective parameters. They are then composed into a complex unit BLINK,

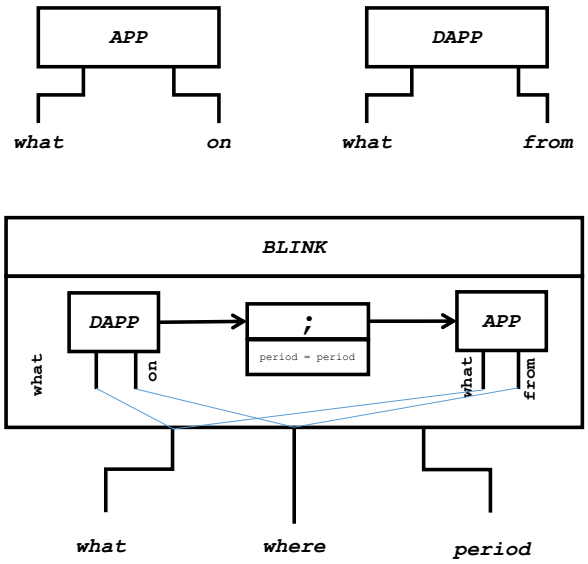


Figure 10: Two basic AUs: APP **make the element (connected to)** what ***appear*** on **a location; and** DAPP **make one *disappear*. Then,** BLINK **combines both to make an object *blink* at a defined** period**.**

which exposes the parameters of the previous units and adds a new one for controlling the blinking delay.

The exact mechanism required for implementing this feature into MA is beyond this paper's scope. However, there is essentially two ways to realise it (we illustrate this point later in §5.2.2):

Using model's elements The first one consists of explicitly referencing the model's elements (either objects or links). Retrieving the corresponding graphical elements can be achieved through the mapping to the CS. This approach has the advantage of directly reusing the models, preventing MA designers from any extra work. However, the task may be complicated by the presence of complex rendering patterns that need to be computed before determining which graphical object the AU logic applies to.

Using graphical elements The second, and opposite, approach consists of directly referencing the graphical elements impacted by an AU. In this case, the AU does not depend on the complexity of rendering patterns, but has the drawback of preventing to easily retrieve which elements in the model have been impacted by an animation.

4.4 Linking AUs and TUs

We already assumed that AUs are "mapped" to specific TUs of the MT defining a DSL's execution semantics. Depending on the underlying MTL, this mapping may be given in two ways: by annotating the TUs with the information (typically, an AU name that allows to retrieve an associated MA specification); conversely, by annotating the AUs with the TUs they refer to; or using an explicit map defined outside both the MT and MA specifications. The first approach has already been successfully used for mapping TUs with debugging steps in [5].

Relating AUs with TUs provides a simple advantage: since the MT already captures the internal logic and scheduling between TUs, no need to reproduce them for the AUs. However, this imposes that the execution engine is instrumented for enabling simple communication with the MA engine. As a first starting step, adopting the generic interface for debugging proposed by Bousse et al. [5] provides what is necessary: it indicates the MA engine that a simulation has started and stopped, and that a TU has started (together with appropriate parameters) and stopped.

5 PROPOSAL

We finally wrap up the previous considerations by making a partial proposal for the missing elements, so that it becomes possible to demonstrate that our vision works. We demonstrate that (i) it becomes possible to define MAs independently of the MT specifying a DSL's execution semantics; that (ii) several MAs may be linked to the same MT for animating a model according to the viewer's preferences; and finally that (iii) MAs may be reused accross DSLs.

5.1 VCS: A Visual Concrete Syntax DSL

Figure 11 shows the metamodel of VCS, a simplified DSL designed for the specification of Visual Concrete Syntaxes. It consists of a Canva of variable size where graphical elments (GElement) may be geometrically placed. A GElement follows the Composite Pattern [10]: Shapes may be *composed*, i.e. the containee is displayed *inside* the container with specific alignment features.

A Shape is either a Table, a TextBox, or a Geometric element, which is either a Form or a Connector. Each Shape is inscribed into a bounding box with specific dimensions that helps precisely positioning it on the Canva, and also defines specific features: for example, a Rectangle may define the thickness of its external box, and an Arrow may define different types and sizes for their heads. A Shape may also define a Style that can be named and reused across other Shapes, to specify various features such as the background or line colour, the line transparency, etc.

This metamodel captures insideness with Composer, and may easily be extended with additional Shapes that only need to be encapsulated in a box (thus, giving value to the attributes length and height in Shape).

5.2 Revisiting the FSM

5.2.1 Concrete Syntax. The first step is to define a mapping from the FSM metamodel (cf. Figure 4) to (a model of) VCS. Since defining the mapping rigorously requires the definition of a specific DSL, we rely on the reader's intuition and only explain it informally.

Following the traditional approach in MDE, defining a mapping results in an (semi-)automatically generated so-called *palette* presenting to a modeller the visual components that can be used for creating, and editing models. The palette in itself is not difficult to create from the mapping, but to obtain a fully-fledged syntax-oriented editor, additional constraints for well-formedness need to be added, which is out of our scope for now. Furthermore, editing flexibility for some visual components is required to ease model creation: typically, any kind of arrow, eventually supported by routing

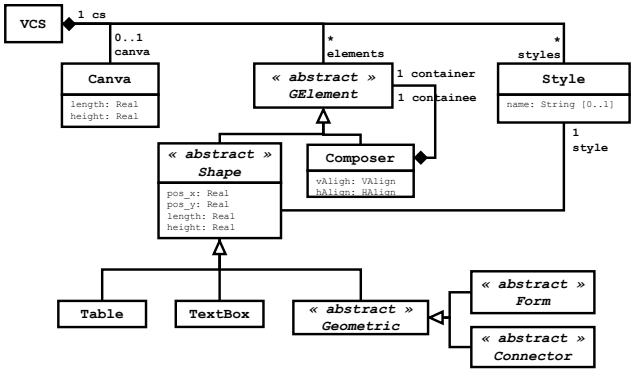


Figure 11: VCS: A simplified Dsl for specifying *V*isual *C*oncrete *S*yntaxes

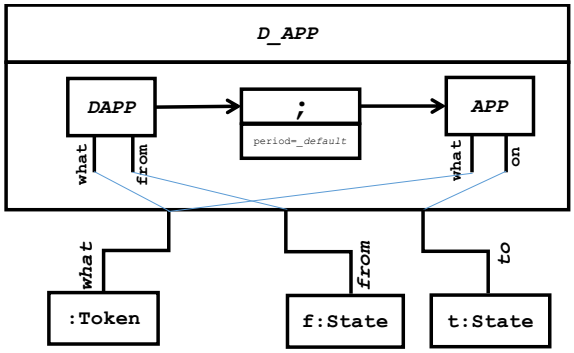


Figure 12: D_APP: Combining disappearance (DAPP) and appearance (APP) in sequence (with combinator ;), and routing the parameters accordingly.

algorithms, may be at disposal of the modeller instead of forcing one kind of arrow for any diagram.

The FSM class is mapped to the overall diagram supported by the Canva, accompanied with a TextBox that will display FSM::name. A (REGULAR) State is represented by an Ellipse with equal length and height (thus creating a circle) drawn in plain black. A FINAL one is mapped to a Composite: two Ellipses vertically and horizontally centered, with length and height only differing by the same delta. An INITIAL State will be represented similar to a REGULAR one, except it will be drawn in dotted red to distinguish it. This allows us to not bother with aligning an arrow into an Ellipse for now. Finally, a Transition may be mapped to any instance of an Connector drawn in black, plain line with the same thickness as the State's Ellipses, assuming they present an ArrowHead with a specific size and type. Finally, Token is mapped to a plain red Ellipse with dimensions the fourth of the ones used for State, and are systematically placed at the horizontal and vertical center of the State's Ellipse it currently points to.

5.2.2 Selected Animations. The second step is to define, and attach, animations (units) to Transformation Units. We will consider two versions of the transformation accept(word : Word), which iteratively calls a TU fire that performs the firing, until the word is empty (in which case, it remains to check whether the current State is FINAL), or no Transition is fireable (i.e. fire does not return anything, in which case the word is rejected).

(1) The fire TU simply loops over all possible outgoing Transition to find one that is fireable, i.e. whose trigger matches the current Letter.
(2) The fire TU performs the same algorithm, except that it calls another TU getEnabledTransition that returns one (of the possible) matching Transition(s).

We now define AUs for those two transformations. The animation D_APP, depicted in Figure 12, implements FSM.2.1: it makes the Token's Ellipse disappear from the current State (referenced as f), then reappear into the target one (named t). Our previous AU BLINK presented in Figure 10 implements FSM.2.3: it makes the Transition's Connector blink for 3 seconds. With these definitions, we can proceed to the mapping between TUs and AUs:

Transformation (1)	**Transformation (2)**		
fire ↦ D_APP	fire	↦	D_APP
	getEnabledTransition	↦	BLINK

Regarding parameterisation (cf. §4.3), we have chosen here the first approach using *models' elements*. To properly identify which model elements participate in an AU, the TU annotations may explicitly list them for the MA designer to choose from.

5.2.3 Reusing AUs. D_APP could be reused for PM.1, with :PacMan, :Cell and :Cell as parameters, and for PN.3.1 with markings, :Place and :Place. Similarly, BLINK could be reused for PN.2 with :Transition and a predefined period. Note that all these AUs should be effectively linked to the respective TUs of the Dsls to fully work.

6 RELATED WORK

MA is integrated in various tools that differ in the way the underlying MT is specified. GeMoC [6] relies on Obeo Sirius for performing the MA of Dsls whose behavioural semantics is expressed in the Kermeta3 or Xtend languages. AtoMPM [30] supports the specification of MTs based on Graph Rewriting, and provides support for specifying a concrete syntax for a Dsl that allows to both express MT rules using the concrete syntax (by relying on the RAMification process [18]), and to debug and animate a model. Earlier, Sadilek and Wachsmuth [28] proposed eProvide, an Eclipse-based tool that, given the specification of the "runtime state" and a visual concrete syntax, automatically generates a visual interpreter and a visual debugger for executing a Dsl. ProB [20] is a model-checker and visualiser for B specifications. It has been used in Meeduse [16], an visualiser and analyser for Dsls whose semantics is specified with B, and allows step-by-step execution and animation.

Many frameworks that allow modellers to express Dsl semantics using formalisms that natively have a visual representation provide animators for free. Such examples include Finite-State Machines [2, 9, 11] and Petri Nets [23, 25, 32]. Although the animation facilities are greatly simplified, this approach does not support domain-specific visual concrete syntax: the animation is realised directly on the underlying formalism, preventing flexibility and adaptivity.

MA can also be performed *offline*, i.e. aside from the MT execution itself. In this case, information about one, or several, execution(s) of a DSL need to be collected beforehand, and "replayed" by the MA engine in order to visually represent the execution(s). Generally, this information is collected in the form of execution traces [15] that represent the dynamic state of the DSL along the execution. Although this approach does not strictly fit our definition of MA, it is interesting because it highlights the importance of properly identifying what a dynamic state is, and the possibility to reuse the MA engine for different purposes. Several contributions follow this approach [12, 14, 17], with different MT languages. Since our approach promotes decoupling transformations from animations, our vision allows to disconnect the original animations used when collecting information, from the animations used afterwards, when the animation is replayed offline.

Debugging, and in particular omniscient debugging [5, 8, 31], is a topic closely related to MA, as it shares some of the underlying mechanisms, such as the definition of a TU considered as a candidate *step* for debugging, and the protocols used to communicate between the transformation engine and the debugger. Bousse et al. [5] introduce two interesting notions that we have reused in our vision. First, *annotation steps* indicate which parts of a MT (our TUs) may safely be interrupted for inspection during debugging. Second, an *interruption pattern* (based on the more well-known Observer pattern [10]) specifies a minimal interface for external services, typically the debugger, to interrupt the execution engine at the beginning and end of the execution, and at key moments when annotation steps are reached. Van Mierlo et al. [31] provide a reusable architecture and an explicitly modelled workflow to support debuggers construction. An interesting feature is the ability to rely on a visual representation of the models being simulating, thus providing a form of animation.

Modelling & Simulation [4] is a closely related research area addressing large Cyber-Physical Systems, where the behaviour is usually expressed through formalisms that include real-time, and sometimes continuous space variables, based on physical phenomena that require complex behaviour captured through, among others, differential equations. Although the behaviour of such systems is clearly out of scope of MDE, the tools supporting these specifications often propose *visual* simulation (e.g. ANSYS Simplorer, 20-sim, MapleSoft MapleSim, etc.) They do not strictly compare to our proposal, but many ideas and concerns from the literature of this research area may help design better animators.

7 CONCLUSION

Model Animation (MA) is a popular approach for providing visual clues for model transformation designers, as well as modellers, that should contribute to help them better understand, and ultimately correct, their models and the associated transformations. This is because MA is closely related to the execution of DSLs: MA visually represent meaningful steps of the model transformations.

As a first step towards a systematic approach for MA engineering, we have identified in this paper three important challenges. First, designing concrete syntaxes for DSLs is not trivial, and this should support complex patterns in order to explicitly express the so-called *mapping*, i.e. the rendering transformation between the DSL metamodel capturing its abstract syntax, into the metamodel describing graphical components. This latter metamodel is a DSL on its own right, and should natively support animation, i.e. the timely and effective change in the graphical components's features (e.g. size, color, thickness, and general topology of elements such as tables, rectangles, etc.)

Second, we plea for a compositional approach for defining complex MA, because relying on basic constructions that are combined to form complex animations promotes flexibility and reuse. The flexibility of a dedicated MA DSL should open the ability to parameterise MA definitions, so that the internal logic of the MA becomes independent from the graphical elements it is applied to. Just like methods may be used in different context captured by the method's parameters, an MA should have parameters to explicitly designate which elements are taken into account, and how. The reuse of an MA DSL should allow to apply the same MA, which uses the same logic underneath, across multiple DSLs that have the same animation logic. We have shown on simple examples that both properties, flexibility and reuse, are effectively found in very simple DSLs, and should appear in bigger ones as well. In turn, these properties should promote the exchange and reuse of well-defined libraries of animations, thus facilitating the specification of future MAs.

Third, in order to not reinvent the logic behind the Model Transformation capturing a DSL's behavioural semantics, an MA should be linked to appropriate Transformations Units (TUs) that would drive the MAs. This way, it becomes possible to extensively reuse the Model Transformation specification as well as its scheduling logic, and only trigger appropriate MA pieces (that we called Animation Units) that would visually reflect what a TU is achieving.

We are aware that MA is a novel field, thus lacks maturity both in terms of methodologies and tooling. However, by identifying these challenges, by providing some leads on how it could be possible to at least partially meet them, while still applying the MDE approach (and in particular, defining carefully designed DSLs for each of the above tasks), we hope to pave the way towards a new and systematic approach for MA. This, in turn, should create a new role for MDE: the *MA designer*, alongside the usual role of modeller, DSL builder and MT specialist. As the field is still in its infancy, we are planning to approach the literature more systematically, in order to uncover what the current practice is, how it may be possible to formalise the DSLs dedicated to the tasks we identified, and identify the various features of animators as a tool in the collection of tools for MDE workbenches.

REFERENCES

[1] Anas Abouzahra, Ayoub Sabraoui, and Karim Afdel. Model Composition In Model-Driven Engineering: A Systematic Literature Review. *Journal of Information and Software Technology*, 125, 2020.
[2] Nils Bandener, Christian Soltenborn, and Gregor Engels. Extending DMM Behavior Specifications for Visual Execution and Debugging. In *International Conference on Software Language Engineering*, pages 357–376, 2010.
[3] Robert Bill, Sebastian Gabmeyer, Petra Kaufmann, and Martina Seidl. Model Checking of CTL-Extended OCL Specifications. In *Conference on Software Language Engineering*, pages 221–240, Västerås, Sweden, 2014. Springer.
[4] Louis G. Birta and Gilbert. Arbez. *Modelling and Simulation: Exploring Dynamic System Behaviour.* Springer, 2019.
[5] Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. Omniscient debugging for executable dsls. *Journal of Systems and Software*, 137:261–288, 2018.
[6] Benoit Combemale, Cédric Brun, Joël Champeau, Xavier Crégut, Julien Deantoni, and Jérome Le Noir. A Tool-Supported Approach for Concurrent Execution of

Heterogeneous Models. In *European Congress on Embedded Real Time Software and Systems*, Toulouse, France, 2016. SEE & 3AF.

[7] Benoît Combemale, Xavier Crégut, and Marc Pantel. A Design Pattern to Build Executable DSMLs and Associated V&V Tools. In *Asia-Pacific Software Engineering Conference*, pages 282–287, 2012.

[8] Jonathan Corley, Brian P. Eddy, Eugene Syriani, and Jeff Gray. Efficient And Scalable Omniscient Debugging For Model Transformations. *Software Quality Journal*, 25:7–48, 2016.

[9] Nondini Das, Suchita Ganesan, Leo Jweda, Mojtaba Bagherzadeh, Nicolas Hili, and Jürgen Dingel. Supporting the model-driven development of real-time embedded systems with run-time monitoring and animation via highly customizable code generation. *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, 2016.

[10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] Heather Goldsby, Betty H. C. Cheng, Sascha Konrad, and Stephane Kamdoum. A Visualization Framework for the Modeling and Formal Analysis of High Assurance Systems. In *International Conference on Model Driven Engineering Languages and Systems*, pages 707–721, 2006.

[12] Philip Guin and Eugene Syriani. Model-based Animation of Micro-Traffic Simulation. In *Symposium on Theory of Modeling and Simulation. DEVS Integrative M&S Symposium*, pages 1–6, San Diego, CA, USA, 2013. Society for Computer Simulation.

[13] Xudong He and Tadao Murata. *The Electrical Engineering Handbook*, chapter High-Level Petri Nets: Extensions, Analysis, and Applications, pages 459–475. Academic Press, 2005.

[14] Ábel Hegedüs, István Ráth, and Dániel Varró. Replaying Execution Trace Models for Dynamic Modeling Languages. *Periodica Polytechnica - Electrical Engineering and Computer Science*, 56(3):71–82, 2012.

[15] Fazilat Hojaji, Tanja Mayerhofer, Bahman Zamani, Abdelwahab Hamou-Lhadj, and Erwan Bousse. Model Execution Tracing: A Systematic Mapping Study. *Jounral of Software and Systems Modeling*, 18:3461–3485, 2019.

[16] Akram Idani. Meeduse: A Tool To Build And Run Proved DSLs. In *International Conference on Integrated Formal Methods*, pages 349–367, Lugano, Switzerland, 2020. Springer, Springer.

[17] Akram Idani. Formal Model-Driven Executable DSLs. Application to Petri Nets. *Innovations in Systems and Software Engineering*, 18(5):1–24, 2021.

[18] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Systematic Transformation Development. In *International Workshop onMulti-Paradigm Modeling*, 2009.

[19] Dorian Leroy, Erwan Bousse, Manuel Wimmer, Benoit Combemale, and Wieland Schwinger. Create and Play your Pac-Man Game with the GEMOC Studio. In *Workshop on Executable Modeling*, pages 1–5, Austin, TX, USA, 2017. IEEE.

[20] Michael Leuschel and Michael Butler. Prob: an automated analysis toolset for the b method. *International Journal on Software Tools for Technology Transfer*, 10 (2):185–203, 2008.

[21] Levi Lúcio, Moussa Amrani, Jürgen Dingel, Leen Lambers, Rick Salay, Gehan Selim, Eugene Syriani, and Manuel Wimmer. Model Transformation Intents and Their Properties. *Journal of Software And Systems (SoSyM)*, 15:647–684, 2014.

[22] Bart Meyers, Romuald Deshayes, Levi Lùcio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. ProMoBox: A Framework for Generating Domain-Specific Property Languages. In *Conference on Software Language Engineering*, pages 1–20, Västerås, Sweden, 2014. Springer.

[23] David Mosteller, Michael Haustermann, Daniel Moldt, and Dennis Schmitz. Integrated Simulation of Domain-Specific Modeling Languages With Petri Net-Based Transformational Semantics. In *Transactions on Petri Nets and Other Models of Concurrency XIV*, pages 101–125. Springer, Germany, 2019.

[24] Object Management Group. Meta Object Facility (MOF) Core Specification. Technical Report v2.5.1, Object Management Group, 2016. URL https://www.omg.org/spec/MOF.

[25] Philippe Palanque, Jean-François Ladry, David Navarre, and Eric Barboni. High-Fidelity Prototyping of Interactive Systems Can Be Formal Too. In *International Conference on Human-Computer Interaction*, pages 667–676, 2009.

[26] José E. Rivera, Francisco Durán, and Antonio Vallecillo. Formal Specification and Analysis of Domain Specific Models Using Maude. *Simulation*, 85(11–12): 778–792, 2009.

[27] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, New York, NY, USA, 2004.

[28] Daniel A. Sadilek and Guido Wachsmuth. Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages. In *European Conference on Model Driven Architecture Foundations and Applications*, pages 63–78, 2008.

[29] Eugene Syriani and Hans Vangheluwe. A Modular Timed Graph Transformation Language for Simulation-Based Design. *Software and Systems Modeling*, 12(2): 387–414, 2013.

[30] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. AToMPM: A Web-based Modeling Environment. In *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition*, pages 21–25, Miami, USA, 2013. CEUR.

[31] Simon Van Mierlo, Hans Vangheluwe, Simon Breslav, Rhys Goldstein, and Azam Khan. Extending Explicitly Modelled Simulation Debugging Environments with Dynamic Structure. *ACM Transactions on Modeling and Computer Simulation*, 30 (1):1–25, 2020.

[32] Manuel Wimmer, Angelika Kusel, Johannes Schönböck, T Reiter, Werner Retschitzegger, and Werner Schwinger. Lets's Play the Token Game. Model Transformations Powered By Transformation Nets. In *International Workshop on Petri Nets and Software Engineering*, pages 35–50, Paris, France, 2009. Hamburg University.