Concrete Syntax: A Multi-paradigm Modelling Approach

Yentl Van Tendeloo
University of Antwerp
Antwerp, Belgium
Yentl.VanTendeloo@uantwerpen.be

Simon Van Mierlo
University of Antwerp
Antwerp, Belgium
Simon.VanMierlo@uantwerpen.be

Bart Meyers
University of Antwerp
Antwerp, Belgium
Flanders Make vzw
Belgium
Bart.Meyers@uantwerpen.be

Hans Vangheluwe
University of Antwerp
Antwerp, Belgium
Hans.Vangheluwe@uantwerpen.be
Flanders Make vzw
Belgium
McGill University
Montréal, Canada
hv@cs.mcgill.ca

Abstract

Domain-Specific Modelling Languages (DSLs) allow domain experts to create models using abstractions they are most familiar with. A DSL's syntax is specified in two parts: the abstract syntax defines the language's concepts and their allowed combinations, and the concrete syntax defines how those concepts are presented to the user (typically using a graphical or textual notation). However important concrete syntax is for the usability of the language, current modelling tools offer limited possibilities for defining the mapping between abstract and concrete syntax. Often, the language designer is restricted to defining a single icon representation of each concept, which is then rendered to the user in a (fixed) graphical interface. This paper presents a framework that explicitly models the bi-directional mapping between the abstract and concrete syntax, thereby making these restrictions easy to overcome. It is more flexible and allows, amongst others, for a model to be represented in multiple front-ends, using multiple representation formats, and multiple mappings. Our approach is evaluated with an implementation in our prototype tool, the Modelverse, and by applying it on an example language.

SLE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5525-4/17/10…$15.00
https://doi.org/10.1145/3136014.3136017

CCS Concepts • **Software and its engineering** → **Domain specific languages**; Integrated and visual development environments; • **Computing methodologies** → *Model development and analysis*;

Keywords Concrete Syntax, Abstract Syntax, Visual, Plotting, Simulation, Model Transformation

ACM Reference Format:
Yentl Van Tendeloo, Simon Van Mierlo, Bart Meyers, and Hans Vangheluwe. 2017. Concrete Syntax: A Multi-paradigm Modelling Approach. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3136014.3136017

1 Introduction

Domain-Specific Modelling Languages (DSLs) are defined by their abstract and concrete syntax [13, 24]. The abstract syntax defines the concepts of the language, which can be instantiated and used as the building blocks of models. For example, the abstract syntax of UML Class Diagrams defines concepts such as *Class*, *Association*, and *Attributes*. The concrete syntax defines the visualization, or rendering, of these abstract syntax concepts. For example, the concrete syntax of UML Class Diagrams defines the mapping of a Class instance to a rectangle with the name of the class on top and a list of all attributes below it. Significant restrictions exist in current tools for the definition of concrete syntax, thereby restricting the language engineer, who is responsible for creating languages that are intuitive to use for domain experts.

Code-based solutions (i.e., tool plugins) are now often used to implement advanced concrete syntax functionality. While feasible, the creation of plugins is not always for the faint-hearted [18], as it relies on tool details (e.g., API) and advanced functionality becomes non-intuitive to express. Additionally, creating the concrete syntax is part of the job

of the language engineer, who is not necessarily an expert in tool plugin creation. To address these problems, we present a different angle of attack, where we apply the Multi-Paradigm Modelling (MPM) [27] approach to concrete syntax. MPM facilitates the analysis, transformation, simulation, optimisation, documentation, evolution, integration, platform independence, and code synthesis of artefacts. Explicit modelling of complex systems includes the explicit modelling of modelling languages; in MPM, they often become part of the software development cycle (cfr. domain-specific languages). This means that abstract syntax, semantics, and concrete syntax need to be modelled explicitly. We identify several limitations in the concrete syntax of state-of-the-art approaches, which are now addressed with (coded, language-specific) plugins. All these limitations become easy to solve using our MPM-based approach, without the need for plugins.

We identify five common limitations: (1) A **single front-end** (or visualization tool) is provided, which is largely aware of the concepts of (meta-)modelling. Existing visualization libraries therefore require a lot of additional code, as these (meta-)modelling concepts need to be introduced. (2) A **single representation** is used for all languages, such as one consisting of rectangles and lines, often arranged in a graph-like manner. While these can be used as primitives for many types of visualization, some models are ideally expressed using a plot, or completely different modes of perceptualization. Note the use of the term *perceptualization*, as we do not wish to limit ourselves to visual representations of models, but want to include, for example, text and sound as well. (3) A **single mapping** to the representation is used, such as to UML Object Diagrams, which can be used for all (graph-based) models, but is seldom the most appropriate. Even when a domain-specific concrete syntax is defined, it is often restricted to only one such mapping. (4) **No extra concrete syntax operations** are available, such as domain-specific lay-outing [7], which aids users in understanding the model. As these algorithms are domain-specific, they must be part of the specification of the domain-specific language. (5) A **one-to-many** mapping between abstract syntax and the visualized model is used, as an icon definition is used. While this often suffices, many-to-many mappings offer additional possibilities to the language engineer.

The remainder of this paper is organised as follows. Section 2 presents an example domain-specific language, motivating the need for a flexible concrete syntax. Section 3 presents our framework, describing the different phases. Section 4 presents the flexibility that we achieve with our approach, linking back to the motivating example. Section 5 evaluates our approach using a prototype implementation. Section 6 discusses identified shortcomings and further extensions of our approach. Section 7 presents related work. Section 8 concludes the paper.

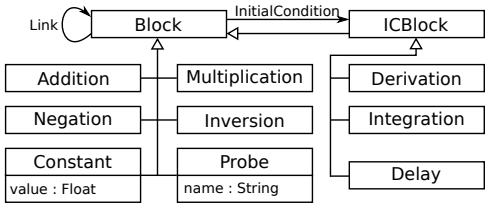

Figure 1. Causal Block Diagrams metamodel.

2 Motivating Example

To show the need for more flexibility in concrete syntax definitions, we use the Causal Block Diagrams (CBD) language [2] as a running example. While this language can be created and used in current tools, its concrete syntax can not easily be implemented as we would like. For an optimal interaction between the modeller and the model, several extensions to concrete syntax are proposed next. The previously mentioned restrictions are now elaborated on in the context of this motivating example: the CBD language. Note that our contribution lies in the explicitly modelled framework for concrete syntax, and not in the extensions offered for this specific domain specific language.

2.1 Causal Block Diagrams

The CBD language is a simple yet realistic language, used to model complex mathematical equations. The language consists of a set of blocks with inputs and outputs. Connections between these blocks carry a signal, which the blocks manipulate. The types of blocks include simple blocks, such as addition blocks, but also more advanced blocks, such as integration blocks. The CBD metamodel, shown in Figure 1, lists all possible blocks and their configuration.

An example instance of the language, with an often used concrete syntax, is shown in Figure 2. The example models a mass suspended by a vertical spring. We consider two forces: the gravitational force and the restoring force of the spring. The set of equations is shown in Equation 1.

$$\begin{cases} F_{spring} & = k \cdot y \\ F_{gravity} & = m \cdot g \\ F_{net} & = F_{gravity} - F_{spring} \\ a & = \frac{F_{net}}{m} \\ \frac{dv}{dt} & = a \\ \frac{dy}{dt} & = v \end{cases} \tag{1}$$

The semantic domain of CBDs is a trace language whose instances contain the values in the probe blocks, paired with the simulation time at which the value was recorded. An appropriate perceptualization of a real value changing over time is a plot, as shown in Figure 3. From this figure, the evolution of the value throughout time becomes immediately obvious. In our case, there are two plots: for the velocity of the mass (v) and the current position (y).

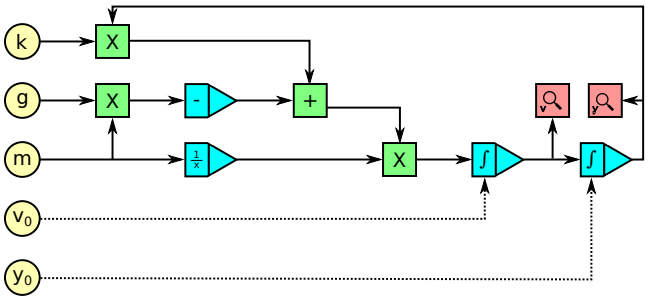

Figure 2. Example Causal Block Diagrams model of a mass suspended on a vertical spring.

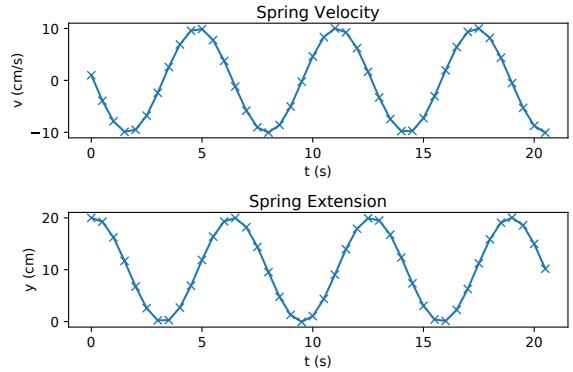

Figure 3. Plotted trace of the CBD model in Figure 2, with $k = 1$, $m = 1kg$, $y_0 = 20cm$, $v_0 = 1\frac{cm}{s}$, and $g = 10\frac{cm}{s^2}$.

2.2 Existing Limitations

As previously introduced, many limitations currently exist in the perceptualization and rendering of models. For each of these limitations, we present a potential requirement of the CBD concrete syntax, and how current tools fail to adequately address them. In our related work section, we further discuss specific tools and the techniques they use.

2.2.1 Multiple GUIs

When modelling CBDs, different users might have different preferences in how they interact with their model. Some users prefer an online browser-based application, requiring no installation nor local code execution, while other users prefer an offline application which executes locally. Nonetheless, the model should be visualized in (almost) the same way, similar to how it is shown in Figure 2. As these users want to collaborate, they should share the same back-end, while their front-ends are different.

Although many (meta-)modelling tools explicitly make the distinction between a back-end and front-end, or expose a modelling API, the distinction between front-end and back-end is often not as expected. Most of the time, the front-ends still need to be aware of most meta-modelling concepts, as they receive the abstract syntax model, the metamodel, and a concrete syntax definition. Changes to the model are then performed in the front-end, and only the abstract syntax changes are propagated to the back-end. Multiple front-ends therefore duplicate this modelling code, while it should only be concerned with the binding to the platform (e.g., TkInter).

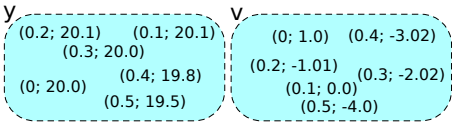

Figure 4. Graphical representation of the trace in Figure 3.

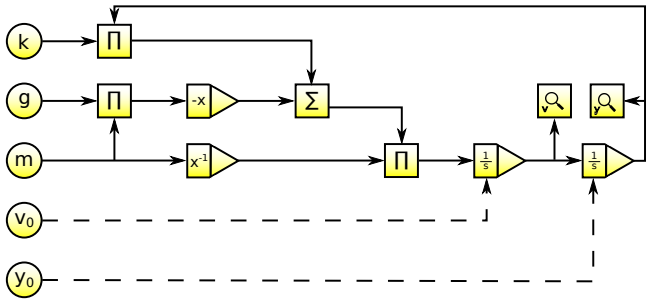

Figure 5. Same model as in Figure 2, but with other icons.

2.2.2 Multiple Perceptualization Formats

Visualizing CBDs is completely different from visualizing their semantics. The semantics of a CBD, expressed as a trace of its signals, is ideally shown as a plot, instead of a graph-like structure. A possible rendering of a trace with the same perceptualization format as the CBD model is shown in Figure 4. Clearly, the trace is better visualized as a plot, previously shown in Figure 3: the plot immediately shows the oscillating behaviour, which cannot easily be derived from the set of tuples. Similarly, some modellers prefer text over a graphical representation [16], though all representations have their limitations [10].

While different front-ends exist today, most are restricted to a graph-like or text-only representation of the models. Other perceptualizations might reason about different concepts, such as datapoints (for plots) or music notes (for sonification), instead of graphical primitives.

2.2.3 Multiple Mappings

The ideal visualization of a CBD model depends on the domain expert looking at it, even when the visualization is relatively similar (e.g., both block-based). Some elements might have a different icon attached to them, depending on the background of the user. For example, users with a Simulink® background are familiar with the symbols $1/s$ for an integrator, and Σ for an addition block. Other users might prefer the symbols $\int$ and +, respectively. A visualization with an alternative set of icons is shown in Figure 5.

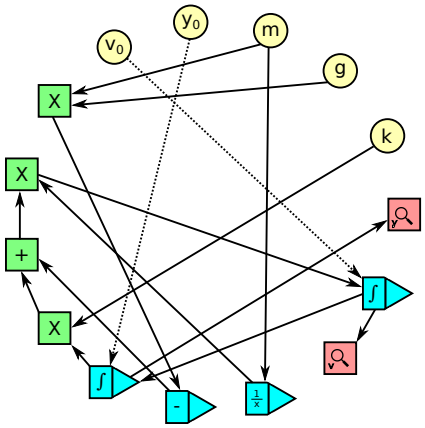

Figure 6. Circle lay-out version of the model in Figure 2.

Even though many tools nowadays support the definition of custom icons for a language, there is often only one possible visualization attached to it. As such, when a different visualization is required, the complete model, including abstract syntax, must be copied. While some tools allow for workarounds, such as defining both icons and only showing one, depending on a configuration option, this is not an elegant solution.

2.2.4 Lay-Outing

The ideal lay-out of CBD elements is closely related to its dataflow. If the flow goes left-to-right, with the exception of feedback loops (e.g., Figure 2), the semantics is easier to interpret than if the position seems random, as in a circle lay-out [17] (e.g., Figure 6).. The flow of the data, and therefore the ideal lay-out, is specific to CBDs, as it depends on the topological sort of the dependency tree [2]. This is specific to CBDs and should therefore not be hard-coded in either the back-end or front-end: it should be defined and maintained by the language engineer.

While current tools often implement generic lay-out algorithms, such as circle and spring lay-out, they have no support for lay-out algorithms provided by the language itself (i.e., domain-specific lay-out algorithms). Lay-outing can be generalized as a "post-processing operation" on the rendered model, where the visualized model is reordered. There is thus a need to define algorithms on the rendered model, ideally included in the concrete syntax model.

2.2.5 Many-to-many Mapping and Parsing

While we have previously allowed for multiple mappings, thereby allowing for a single element to be visualized in multiple ways, the modeller might have additional preferences. For example, CBDs are sometimes visualized with a conjoined addition/subtraction block (e.g., in Ptolemy/Kepler [1]): a single block has an addition and subtraction port, where all signals are summed, but the signals on the subtraction port are negated first. This is syntactic sugar for a single addition block, with negation blocks for each input on the subtraction port, as shown in Figure 7. Whichever representation is used depends on the domain expert, though we want the abstract syntax to be identical, independent of the used representation.

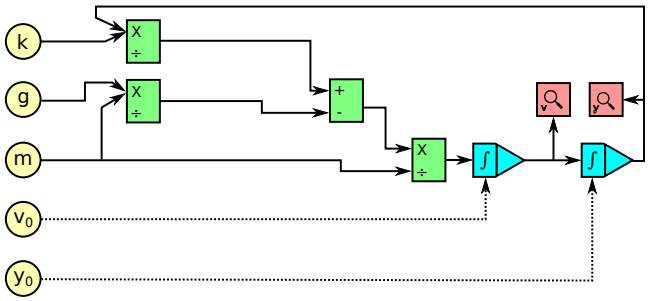

Figure 7. Alternative representation of the model in Figure 2, now using a more complex mapping.

While this problem seems highly related to the multiple mappings problem, it is fundamentally different: the conjoined addition/subtraction block is a single concrete syntax element with multiple abstract syntax elements underlying it. Indeed, each connection to the subtraction port has a (hidden) negator block in the abstract syntax. While it is possible to change the abstract syntax, this would create problems for the other operations, where the negation block is explicitly present. The problem is therefore the restriction of many tools to a one-to-many mapping: a single abstract syntax element is rendered by several concrete syntax elements, independently of other abstract syntax elements. A possible workaround is the introduction of an intermediate language, which expands or collapses the addition/subtraction block, though such an intermediate language causes additional consistency problems.

3 Explicitly Modelling Concrete Syntax

We now present our multi-paradigm modelling approach to concrete syntax, where we explicitly model all aspects.

Our approach makes a clear distinction between the responsibilities of the back-end and front-end. The back-end is responsible for all (meta-)modelling related concepts, including how models are perceptualized and comprehended. The front-end is responsible only for how this perceptible model is rendered using a specific platform, such as TkInter. Instead of transferring the abstract syntax of the model (using domain-specific concepts, such as *Constant*), the back-end transforms this model to the MM_{Render} language (which uses perceptualization concepts, such as *Ellipse*).

Our approach is centered around four activities, as shown in Figure 8: *Perceptualization*, *Rendering*, *Recognition*, and *Comprehension*. Our approach is independent of how these

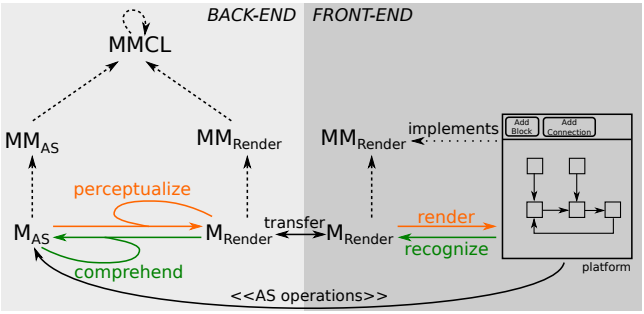

Figure 8. Overview of the approach.

activities are implemented (e.g., in code, using model transformations, or manually).

In the remainder of this section, we elaborate on each step of our approach, where we link to a minimal example in the context of CBDs, shown in Figure 9. We start at the top left in the figure, with M_{AS}. M_{AS} is first (1) perceptualized, resulting in an M_{Render}. For example, instances of the *Constant* class are translated to a *Group* containing an *Ellipse* and *Text* instance. This M_{Render} is (2) transferred to the front-end in some way, which is independent of our approach. In the example, a JSON serialization of the source model is shown. The front-end has a copy of M_{Render}, which is (3) rendered for that specific platform. For example, all instances of *Ellipse* are iterated over, and a `create_oval` TkInter function is invoked. The TkInter front-end listens to user events (e.g., mouse clicks), thereby (4) altering the rendered model. For example, the text entry "1" is altered to "2". Such changes are (5) recognized (e.g., via callbacks), resulting to changes on M_{Render}. For example, the *Text* instance has its attribute *text* updated to "2". Changes are (2) transferred to the back-end again, this can be incremental or overwrite the complete model. Finally, the new M_{Render} is (6) comprehended, thereby changing M_{AS}. For example, the changed text results in an update to the value of the constant block. Each of these steps is further elaborated on next.

3.1 Perceptualization

The first step to our approach is perceptualization, where a model in a domain-specific language MM_{AS} is mapped to a perceptualization language MM_{Render}. This defines how the model is presented to the user. For each language that we want to visualize, it is important to define a perceptualization activity, which is the concrete syntax definition.

This activity needs to map to an MM_{Render}, which defines the mode of presentation to the user. MM_{Render} defines the platform primitives that can be used, such as *Ellipse*, *Rectangle*, and *Line*. In our example we focus on graphical languages, as this is easiest to present on paper, and therefore our MM_{Render} is defined as in Figure 10. Note that this MM_{Render} is not yet linked to any specific platform, such as TkInter or Scalable Vector Graphics (SVG). The used concepts are generic to many graphical visualization libraries.

Our approach is not restricted to any specific MM_{Render}, although we demonstrate our approach using a metamodel for graphical visualization. It is straightforward to come up with different MM_{Render} specifications, such as one for plots (e.g., for signal traces), text (e.g., for action language), or even sound (e.g., for music sheets [19]). We envision a small library of different kinds of MM_{Render} to capture all necessary perceptualizations. Of course, a front-end should also be defined which can render models in that language.

Traceability can be constructed between M_{AS} and M_{Render}, to be used for incremental perceptualization, where we only perceptualize elements in M_{AS} that have no associated elements in M_{Render} yet. This is the reason for the loop in Figure 8, where perceptualization takes in both M_{AS} and the current M_{Render}. It remains up to the language engineer whether or not to use incremental perceptualization.

In our example, we transform the single CBD instance of *Constant* to instances of *Group*, *Ellipse*, and *Text*, conforming to the MM_{Render} metamodel in Figure 10. This defines how constant blocks are to be presented to the user: as a group of an ellipse and some text. We defined this activity using model transformations. An example model transformation rule is shown in Figure 11, which creates a *Group*, *Ellipse*, and *Text* instance for each *Constant* element that not yet has an associated group. The values of their attributes are hidden due to space restrictions, but are mostly trivial (e.g., the colour of the ellipse and value of the text). In a model transformation rule, the Left-Hand Side (LHS) pattern is matched in the model, and is replaced by the Right-Hand Side (RHS), unless the Negative Application Condition (NAC, shown in the dashed rectangle) also matches. The (purple) numerical annotations link elements in the LHS to elements in the RHS.

3.2 Model Transfer

As there is an explicit difference between the back-end and the front-end, there needs to be a way to transfer the models. We want this to be as general as possible, as both the back-end and front-end could be physically distributed and implemented in different programming languages. In our example, the model is serialized using JSON, and transferred over network sockets. Nonetheless, our approach is independent of the implementation details of model transfer, and we therefore do not elaborate on this aspect. It is only important that an exact copy of M_{Render} is present on both the back-end and front-end; this can be achieved in many different ways.

Note that, thanks to our approach, only models in the MM_{Render} language must be transferred, potentially allowing for additional optimizations in the serialization.

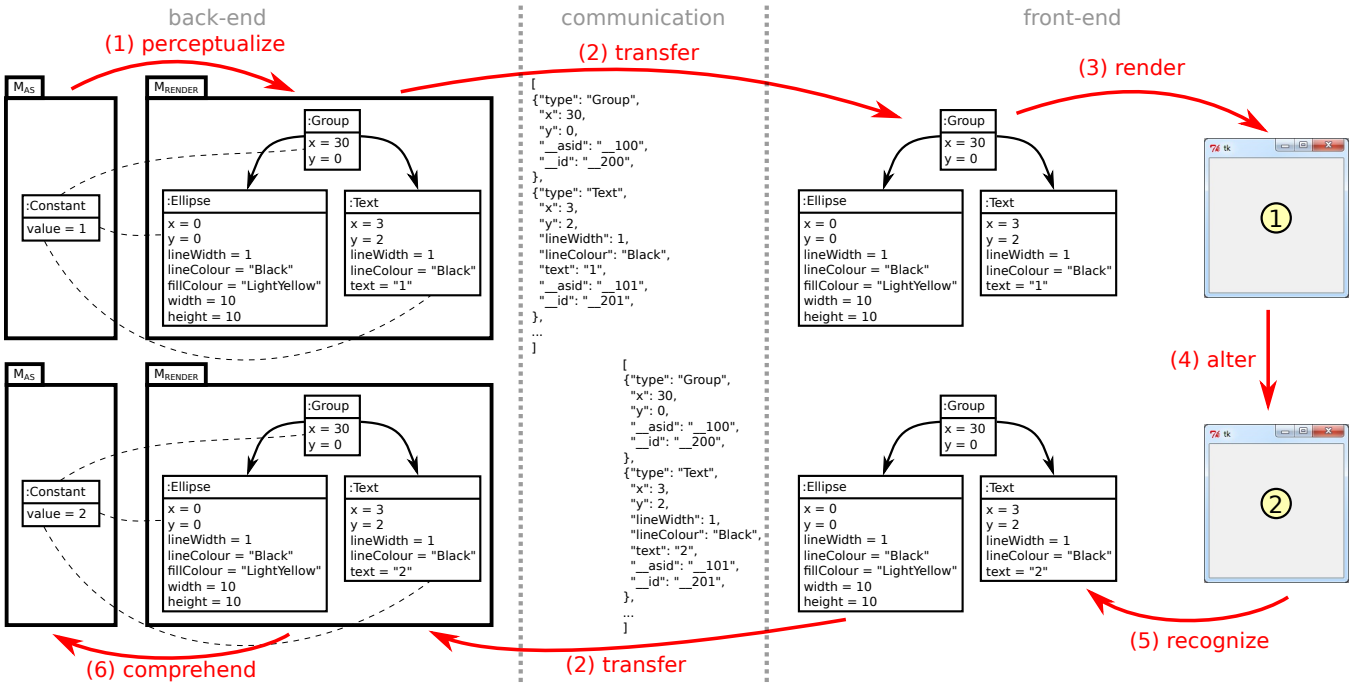

Figure 9. Overview of the approach with an example for CBDs.

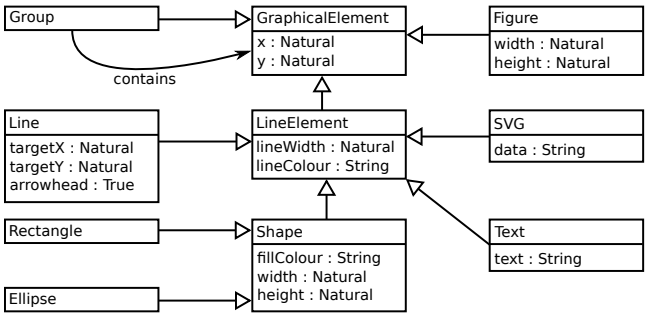

Figure 10. MM_{render} for graphical visualization.

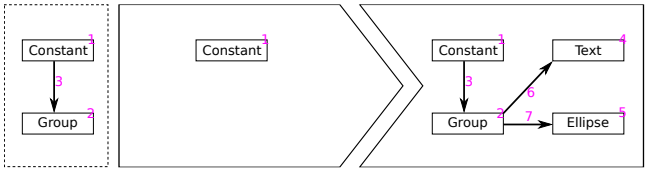

Figure 11. Example rule for CBD perceptualization.

3.3 Rendering

When the M_{Render} arrives at the front-end, it needs to be presented to the user. This is done by mapping the concepts of M_{Render} to the platform operations responsible for the presentation. As such, the front-end's interface is described in a platform-independent way using MM_{Render}. It is thus important that the front-end and back-end agree on the same MM_{Render}. Rendering can be seen as a transformation from concepts in MM_{Render} to concepts in the platform.

While our approach explicitly represents both M_{Render} and MM_{Render} in the front-end, this does not necessarily have to be the case. For example, the front-end could just iterate over the JSON serialization it gets in, directly invoking platform functions. And even while the models are not explicitly present in the front-end, the front-end still makes implicit use of these models and the back-end ensures well-formedness.

In our example, the front-end maps concepts such as *Ellipse* to the `create_oval` TkInter function, also translating the attributes to arguments for that function. The complexity of the mapping on how close the concepts of MM_{Render} match those of the platform. For example, if a platform does not support rectangles, elements of the *Rectangle* class have to be mapped internally to four seperate lines (or whatever operation the platform provides).

3.4 Altering

Some front-ends allow altering the rendered model in some way. Straightforward examples are moving around elements, changing their size, and so on. Such changes occur in the platform, and are based on platform events (e.g., button press, mouse move, mouse click), which need to be mapped to model operations. As the detection of such events is highly platform-dependent, and can be considered an implementation detail, we do not elaborate on this. For our approach, it is only important that the rendered model can be altered, as we are independent of how these changes actually occur.

Even though simple operations are common, altering the model can happen in any way, for example through sketch interpretation [3, 15], where sketches are recognized as changes in the platform (e.g., a drawing of a circle is mapped to the TkInter circle concept).

3.5 Recognition

When changes are made to the rendered model, these changes have to be propagated to the M_{Render}, as this is the common exchange format between back-end and front-end. While this mapping is often trivial, it depends on the match between MM_{Render} and the platform concepts. For example, for a trivial mapping, moving a rectangle in the platform merely maps to moving that same rectangle element in M_{Render}. For a complex mapping, however, the rectangle might be a set of lines in the platform, where moving one of these lines affects the three other lines as well.

Recognition does not attach semantics to the change. Indeed, changing the value of the text merely alters the text value, and the associated constant block still has the value 1. As such, recognition is limited to syntactical changes.

In our example, the mapping is trivial: updating the text value in the platform merely requires us to update the *text* attribute of the *Text* instance in M_{Render}.

3.6 Comprehension

Comprehension maps changes on M_{Render} back to changes on M_{AS}. As such, it attaches semantics to the change that was made. Note that this operation often makes use of the tracability information that was previously created during perceptualization, as it needs to map between both formalisms. Therefore, comprehension can make use of the original M_{AS}, being the reason for the loop in the overview figure.

Often, a front-end only allow syntactical changes that have no influence on semantics. For example, moving an element of a topological formalism changes the x and y attributes in M_{Render}, though it has no effect on the semantics of the model. In many cases, therefore, comprehension is skipped completely. Nonetheless, it is an essential activity in the context of free-hand editors, where all changes are made purely in concrete syntax.

The distinction between recognition and comprehension is important. For example, recognition recognizes when a rectangle is dragged to a different location (changing its x and y attributes), and comprehension comprehends that this implies containment (creating a *Containment* link). In contrast to perceptualization, comprehension might fail if the user creates a structure that cannot be comprehended (i.e., a *parsing error*). While we are sure that the modified M_{Render} conforms to MM_{Render}, it does not necessarily represent a comprehensible model (e.g., a circle has no meaning in CBDs without a text value in it).

In our example, comprehension maps the text value of the *Text* element back to the value of the *Constant* block. Note that this is one of the only changes on concrete syntax that would have any semantical effect. For example, altering the x and y attributes of any of these elements would have no semantical effect, as CBDs are a topological formalism. When the *Text* element is deleted altogether, comprehension fails.

4 Degrees of Flexibility

With our approach explained, we present how this approach addresses the various restrictions of existing tools. For each restriction, we explain how our approach is flexible enough to support it, applied to our motivating example.

As we did not code our approach, many of these degrees of flexibility are just the creation of a new model, in which meta-modelling tools are specialized. The presented degrees of flexibility can therefore be explained at a high level of abstraction, without going into implementation details. This would not be the case for a plugin-based approach, for example, as we would have to rely on tool-specific API calls.

4.1 Multiple GUIs

The first restriction was related to having multiple front-ends, possibly implemented in different implementation languages, though all with similar semantics. We addressed this problem by presenting the MM_{Render} model as the "interface" for model rendering: all front-ends must accept the same set of models. As long as the back-end and front-end agree on a certain MM_{Render}, specified in the back-end, all front-ends that implement it are supported. In contrast to other tools, where the front-end is offered some kind of fixed modelling API on abstract syntax, our front-end only receives a serialized model, in a known format, which it must render as-is: all processing has already been done. The back-end is completely independent of the front-end and, subsequently, the platform used for rendering. This is shown in Figure 12, where the same M_{Render} and MM_{Render} is used for two different front-ends, rendering the same representation of the model.

For CBDs, we implemented a front-end in Python/TkInter and JavaScript/SVG. Both are similar in use and visualization, though their underlying mapping to the platform drastically differs. There is still much freedom left in the front-end, specifically for elements not defined in MM_{Render}, such as the supported operations (e.g., zooming, scaling) and interaction with the user (e.g., mouse-driven, keyboard-driven).

4.2 Multiple Perceptualization Formats

As our approach explicitly models MM_{Render}, it is possible to have several variants of it, each defining a different format. Each front-end merely needs to ensure that its MM_{Render} is comprehended by the back-end, and can from then on visualize models in that language. A different MM_{Render} often requires a different front-end, though this is not required.

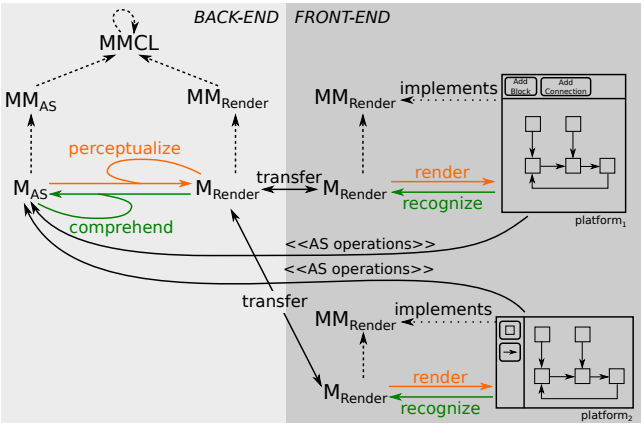

Figure 12. Approach with multiple GUI front-ends.

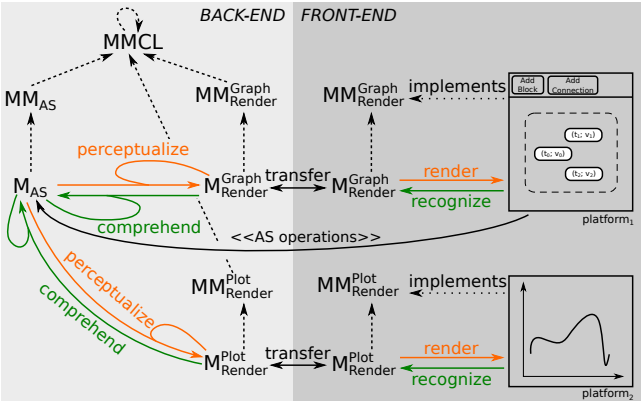

Figure 13. Approach with multiple MM_{Render} models.

For example, a TkInter front-end can visualize a text-only MM_{Render} as a TkInter text widget.

Figure 13 shows this in the context of our CBD example, where we have two rendering formats: one for graphical models (MM^{Graph}_{Render}), and one for plots (MM^{Plot}_{Render}). Each MM_{Render} has its own front-end. Both front-ends are connected to the same back-end and share the same models and API to these models. Through this API, a graphical front-end receives a model conforming to MM^{Graph}_{Render}, and the plotting front-end receives a model conforming to MM^{Plot}_{Render}.

4.3 Multiple Perceptualizations

Since the mapping from MM_{AS} to MM_{Render} is explicitly modelled, it is possible to change it, or have multiple. Any mapping is fine, as long as it generates a valid instance of MM_{Render}, and can therefore be rendered. These mappings can target different versions of MM_{Render}, as was already shown in the previous point, but can also go to the same MM_{Render}.

Figure 14 shows this in the context of our CBD example, where we have two mappings to the same MM_{Render}. One defines the integration block icon as a rectangle with $1/s$ in it (MM^{V1}_{Render}), whereas the other defines it using a triangle and

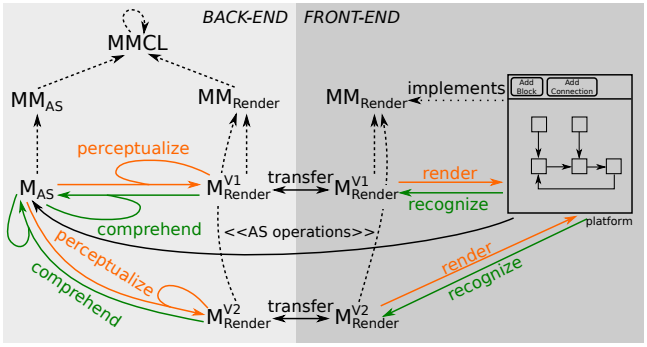

Figure 14. Approach with multiple mappers to the same MM_{Render}. The same tool is used for both models, though different instances.

the $\int$ symbol in it (MM^{V2}_{Render}). Both mappings are equally correct and can be used interchangeably: all changes on one representation are automatically mimicked on the other representations, as they share the same M_{AS}.

4.4 Lay-outing

Lay-outing is an additional operation executed after perceptualization, as we need to operate on the current visualization. Therefore, it is often shifted to the front-end completely. In our approach, the perceptualized model is available in the back-end, where the lay-outing can happen using, for example, model transformations. This not only makes it possible to share the same lay-out algorithms between front-ends, but also allows domain-specific lay-outing algorithms. For practical reasons, the lay-out algorithm, and any other pre- or post-processing operations, are implemented as part of the perceptualization phase.

For our CBD example, we can implement a new domain-specific lay-out algorithm as part of the perceptualization. When new elements are added, users can add them wherever they want, but they will automatically be placed at the ideal location in the CBD model. With lay-outing happening at the back-end, all users sharing the same perceptualized model will also see the lay-out propagated.

4.5 Many-to-Many Perceptualization

As our mapping for the perceptualization and comprehension is any kind of operation, we can use any executable language to define it in. In contrast to icon definitions, we can map multiple abstract syntax elements to multiple concrete syntax elements, as the mapping itself is generic. This can be used during perceptualization to create complex rules that cannot be expressed with the usual icon definitions: multiple abstract syntax elements are condensed into a single icon. Thanks to the use of traceability links in our approach, from M_{Render} to M_{AS}, it is also possible to incrementally update the concrete syntax, by linking previously rendered elements.

For our CBD example, we are able to utilize model transformations to map elements from the source language (M_{AS}) to elements in the target language(M_{Render}). In general, model transformation language are not limited to a one-to-many mapping, in contrast to most icon definition languages.

5 Evaluation

We now evaluate our framework based on an implementation in our prototype tool: the Modelverse[1] [25, 26].

5.1 Research Questions

We distill our motivating example into five research questions:

1. **R1**: Can new front-ends be implemented fast?
2. **R2**: Can models be perceptualized in different ways?
3. **R3**: Can multiple perceptualizations be defined?
4. **R4**: Can domain-specific lay-outing be defined?
5. **R5**: Can many-to-many perceptualizations be defined?

5.2 Results

R1: Lightweight Front-ends We have implemented two separate front-ends, for two different platforms: TkInter and Matplotlib, both using Python. The Matplotlib front-end only visualizes the model and does not offer any manipulation operations. The TkInter front-end includes basic concrete syntax operations, such as moving around elements, and basic abstract syntax operations, such as modifying attributes. Each front-end was implemented by a different developer, familiar with the platform. Each individual front-end took less than one day to implement up to the point where they could visualize the models, exactly as received from the back-end. Each front-end has a small code base: approximately 250 lines of Python code for the front-end with Matplotlib, and 350 lines for the front-end with TkInter. For both front-ends, no (meta-)modelling information had to be coded, except for the implementation of MM_{Render}. This can be considered fast for front-end development, which usually takes a significant amount of time. In our case, perceptualization was only defined once in the back-end, instead of once for each front-end.

R2: Different Perceptualizations Using the two previously implemented front-ends, we have also shown the feasibility of different perceptualizations. The first front-end provides a plot-based perceptualization of a trace model. In this perceptualization, the model is visualized as a graph, and all operations, such as zooming, are provided natively by the Matplotlib platform. The second front-end provides a graphical perceptualization of the original CBD and resulting trace model. In this perceptualization, we rely on the TkInter visualization primitives. The trace model can therefore be perceptualized in two significantly different ways.

[1] https://msdl.uantwerpen.be/git/yentl/modelverse

R3: Multiple Similar Perceptualizations Using the previously implemented graphical front-end, with TkInter, we have implemented two different perceptualizations as model transformations. This front-end therefore has a drop-down menu for the model to show, and a drop-down menu for the available perceptualizations. Both are automatically populated by querying the back-end. The same model can therefore be visualized with two slightly different transformations.

R4: Domain-Specific Layouting We have implemented a simple lay-out algorithm in the perceptualization transformation. Combined with the two different perceptualization transformations, we were able not only to alter the icons of the different concrete syntax elements, but also to change their relative position. As such, when switching from one perceptualization to the other, the model not only changes its icons, but the position of these icons changes as well.

R5: Many-to-many Perceptualization As our approach is based on generic activities, it stands to reason that we can support many-to-many perceptualization. A simple many-to-many perceptualization was implemented, as presented before in the motivating example. After the usual icon mapping, mapping an addition block to a rectangle with the addition symbol in it, additional model transformation rules are added to search for a negation block that is connected to the addition block. When such a pattern is found, the concrete syntax representation of the negation block is removed, and the connection is redrawn to the negated input port of the addition block. As such, a one-to-many mapping between M_{AS} and M_{Render} is shown to be possible.

5.3 Threats to Validity

For **construct validity**, our primary threat is the measures used for R1. We used two measures: the time needed to develop the tool, and the number of lines of code. Development time highly depends on the skill of the developer and the familiarity with the used libraries. To minimize the time needed to get familiar with the libraries, developers were familiar with the library they had to use up to the level that they had no technical problems. The number of lines of code is not too reliable to determine the difficulty of writing the front-end. The codebase of the two front-ends mostly consists of linear code and does not include non-trivial algorithms. For example, out of the 250 lines of code for the plotting front-end, 50 lines are dedicated to the translation of terminology (e.g., "solid" line types in MM_{Render} to "-" in Matplotlib).

For **external validity**, our primary threat is the application to only a single language (CBDs), with a single back-end (the Modelverse), and only a single implementation language (Python). Nonetheless, we believe that each of these is representative, and our approach does not depend on any of these in particular.

For **reliability**, we note that we depend on the familiarity of the researchers with the used tools. As we have used our own prototype tools, we knew all details relevant to the application of our approach. Lack of documentation about these tools might hinder other researchers from implementing the same functionality in this tool. Another threat to reliability is the small amount of experiments that were conducted.

6 Discussion

We briefly present three directions in which our work is currently still limited, but can be further extended: performance, GUI interaction and concrete syntax definitions.

6.1 Performance

Performance has not been discussed up to now, as it is not one of the concerns that we want to tackle. Nonetheless, concrete syntax can only be deemed usable if it is also sufficiently efficient to use: model perceptualization and comprehension require a relatively low latency, as otherwise the interface does not seem responsive, leading to user frustration. Model transformations are the crucial factor in our approach: benchmarking our approach would actually be benchmarking the underlying model transformation engine. Many model transformation optimizations have been discussed in the literature, such as incrementality [22], distributed queries [21], or scope discovery [11]. Our approach itself is independent of the underlying model transformation algorithm.

6.2 GUI Interaction

Up to now, the behaviour of the front-end was considered as a black box. While we did term its operations as *rendering* and *recognition*, nothing is said about how this happens. Many differences are possible here as well, which can ideally be domain-specific. For example, what operations does a modeller have to do to delete an element? Must an element be left clicked and then the *delete* key pressed, or is there a button to do this? Depending on the domain, any of these modes of interaction might be more natural to the user.

The behaviour of the GUI, and in particular its interaction with the user, should ideally also be explicitly modelled, similar to the concrete syntax. This timed, reactive, and possibly concurrent behaviour is best described by a specialized formalism, such as SCCD [23].

6.3 Concrete Syntax Definition

While our proposed framework offers a lot of flexibility to language engineers, defining a concrete syntax mapping is not as easy as an icon definition. To increase usability, we propose an additional language, the MM_{CS}, which is a language for concrete syntax definitions. A concrete syntax definition is a DSL for the definition of concrete syntax. An example is an icon definition language. Instances in this language, termed M_{CS}, can be used to generate the perceptualization and comprehension model transformation. So while we use the full-blown infrastructure, it becomes possible to use a similar workflow as before, if so desired. This is shown in Figure 15, where we show that both model transformations are generated from M_{CS}. MM_{CS} is also tightly related to both MM_{AS} and MM_{Render}, as it uses concepts from both. Again, we are not restricted to a single MM_{CS}, as it is possible to define and use several, all of which define DSLs for the domain of concrete syntax definitions.

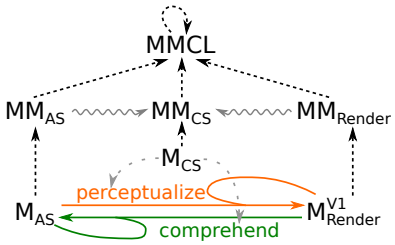

Figure 15. Concrete Syntax definition to automatically generate the perceptualization and comprehension operations.

7 Related Work

Most (visual) modelling environments support customizing the concrete syntax of modelling languages. We consider a number of representative examples and explore to which extent they support the features listed in the previous sections. Without exception, these tools hardcode MM_{Render}, meaning that even when they offer some degrees of flexibility, it is constrained to a specific type of perceptualization.

AToMPM [20] is a graphical meta-modelling environment, implemented in Javascript/SVG. It allows language engineers to develop their languages' abstract syntax using a class-diagram language. For the concrete syntax, an icon definition language is provided. The language engineer has to create an icon for each class, and a link for each association. A class' icon and an association's link define the graphical appearance of the instances of that class or association; it can consist of several graphical primitives such as rectangles, circles, and lines. The graphical primitives have a number of attributes, such as *colour*, *size*, *font* (for text), etc. The value of these concrete syntax attributes can depend on the value of abstract syntax attributes: this can be defined in a *mapper*. Conversely, changes on the concrete syntax (e.g., dragging an icon) can be *parsed*, which results in changes to the value of the abstract syntax attributes. AToMPM is restricted to one-to-many perceptualization. Multiple concrete syntaxes can be defined for the same abstract syntax definition; the front-end allows to switch between different renderings of the same abstract syntax model. Due to AToMPM's client-server architecture, an alternative front-end could be developed using a different platform. Layout algorithms are not supported.

AToM^3 [6], the predecessor of AToMPM, is implemented in Python/Tkinter. Model storage and visualization are tightly coupled. Similar to AToMPM, visualization is defined using an icon editor, though only one concrete syntax definition is supported for each language, as they are tightly interwoven. No comprehension from concrete to abstract syntax is supported and perceptualization is limited to displaying the value of an attribute in a text field. The language engineer can, however, code actions that are triggered by events, such as editing an object, moving it, selecting it, etc. These scripts can access both the abstract syntax and concrete syntax (Python) objects, though they are not governed by well-formedness rules: invalid configurations can be reached. Some layout algorithms are provided, such as circle layout and spring layout, though all of them are generic; domain-specific layout algorithms are not supported. For AToM^3, a multi-view component was previously introduced [5], though this was mostly focused on the abstract syntax and associated semantics.

MetaEdit+ [12] is a commercial domain-specific metamodelling environment. To define the abstract syntax of a language, a metamodel is created in the feature-rich GOPPRR (Graph-Object-Property-Port-Role-Relationship) language. A symbol editor allows to customize the concrete syntax of the language; again, each class is given a graphical representation. Mapping is limited to text elements, whose value can be defined based on the abstract syntax of the model, and visibility of graphical elements, based on a condition on the abstract syntax of the model. Custom layout algorithms nor comprehension are not supported. While MetaEdit+ is a commercial, proprietary tool, it does implement a SOAP API with which external tools can query and modify the models stored in the tool. No access is given to the graphical info of the models. Therefore, it is impossible to implement a minimal user interface with MetaEdit+ as a back-end, unless perceptualization is implemented from scratch.

A number of frameworks exist that allow language engineers to create graphical user interfaces in Eclipse EMF (https://www.eclipse.org/emf). GMF (https://www.eclipse.org/modeling/gmp) allows the generation of a modelling tool from a concrete syntax definition, a perceptualization and a tool definition, which are all explicitly modelled. Users can generate an editor as an Eclipse plug-in or as a Rich Client Platform (RCP) application. Reusing existing libraries, however, is not as straightforward. Sirius builds on GMF and aims to ease the development of modelling tools, while primarily focusing on multi-view modelling [14]. Multiple concrete syntaxes for the same abstract syntax are supported, for example by providing multiple viewpoints depending on the level of abstraction. Papyrus [9] is a tool for modelling UML or SysML diagrams. Focusing on such standards, the tool allows users to specify tailored concrete syntax for their UML profile. All these EMF approaches are based on the generation of a modelling tool.

In the domain of textual languages, abstract syntax and concrete syntax are usually defined together by means of a grammar. In this context, comprehension is equivalent to parsing. Any (general-purpose) text editor can be used as a front-end for free-hand editing. A parser is used to determine the text's conformance to the language. Nowadays, smart text editors are used to parse the text dynamically during editing, thereby supporting syntax highlighting, error detection, auto-completion, etc. Xtext is a framework that supports implementing textual DSLs and such smart editors [8]. A DSL is defined by an Xtext grammar, from which it is possible to parse an EMF-based abstract syntax tree by using a generated ANTLR parser. A textual environment can be generated, which includes syntax highlighting, error visualization, content-assist, folding, jump-to-declaration and reverse-reference lookup across multiple files. Xtext supports multiple front-end frameworks, such as Eclipse, IntelliJ, and web browser support, but the user is not expected to define support for his own framework. Xtext is defined for textual languages exclusively, unlike our approach.

Textual concrete syntax definition for DSLs is also supported in MetaDepth, based on ANTLR [4]. In MetaDepth, concrete syntax and abstract syntax definition are separated, unlike typical approaches for textual syntax. There is no dedicated support for a user interface; instead, an external general-purpose text editor must be used.

Similar to our approach is Monto [18], which addresses the problem of extending existing integrated development environments. But whereas their approach sticks to the same approach as before, trying to make plugins easier to define, our approach takes a radically different approach by modelling all aspects explicitly. In our approach, plugins disappear, and effectively become just new models in the tool, which are used to augment the behaviour of the tool. Projectional editing [28] is an alternative approach, where the abstract syntax, instead of the concrete syntax, is manipulated.

The overview of our approach bears similarity to the megamodel on parsing and unparsing [29], where 12 classes of artefacts were identified, along with a set of transformations between them. This overview is mostly oriented towards textual languages. In contrast, our approach covers different types of perceptualization: textual or graphical perceptualization is handled similarly in our approach. Related to this, our approach is capable of handling other perceptualization strategies as well, such as sonification, as long as there is an MM_{Render} and supporting front-end.

8 Conclusion

We have shown several restrictions in current approaches to concrete syntax, and in particular graphical concrete syntax. In this paper, we identified five restrictions with regard to their (graphical) concrete syntax, which we address by presenting a Multi-Paradigm Modelling (MPM) approach.

With our approach, the to-be-rendered model is represented in abstract syntax as well, making it possible to do the perceptualization in the back-end. The perceptualized model is transferred to a compatible front-end, which merely renders the result. Changes to the rendered model are recognized and comprehended to update the abstract syntax model.

We have shown the various degrees of flexibility offered by this approach, and described our implementation in the Modelverse. Future work is possible in many directions. First, a front-end interaction model could be defined, which describes the (domain-specific) behaviour of the front-end. Second, a concrete syntax definition language could be defined, which allows the automated generation of the required operations. Third, the link with textual languages, where the concepts of parsing and unparsing are already well-known, could be further explored.

Acknowledgments

This work was partly funded by PhD fellowships from the Research Foundation - Flanders (FWO) and Agency for Innovation by Science and Technology in Flanders (IWT). This research was partially supported by Flanders Make vzw, the strategic research centre for the manufacturing industry.

References

[1] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludäscher, and Steve Mock. 2004. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management*. 423–424.

[2] François E. Cellier. 1991. *Continuous System Modeling* (first ed.). Springer-Verlag.

[3] Randall Davis. 2007. Magic Paper: Sketch-Understanding Research. *Computer* (2007), 34–41.

[4] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2015. Model-driven engineering with domain-specific meta-modelling languages. *Software and System Modeling* 14, 1 (2015), 429–459.

[5] Juan de Lara, Esther Guerra, and Hans Vangheluwe. 2005. A multi-view component modelling language for systems design: Checking consistency and timing constraints. In *Visual Modeling for Software Intensive Systems*. 27–34.

[6] Juan De Lara and Hans Vangheluwe. 2002. AToM3: A Tool for Multi-formalism and Meta-modelling. In *International Conference on Fundamental Approaches to Software Engineering*. 174–188.

[7] Denis Dubé. 2006. *Graph Layout for Domain-Specific Modeling*. Master's thesis. McGill University.

[8] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: implement your language faster than the quick and dirty way. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (SPLASH/OOPSLA)*. 307–309.

[9] Sébastien Gérard. 2015. Once upon a Time, There Was Papyrus.... In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*. IS–7.

[10] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. 2007. Text-Based Modeling. In *Proceedings of the 4th International Workshop on Software Language Engineering*.

[11] Māris Jukšs, Clark Verbrugge, Maged Elaasar, and Hans Vangheluwe. 2016. Scope in model transformations. *Software & Systems Modeling* (2016), 1–26.

[12] Steven Kelly and Juha-Pekka Tolvanen. 2008. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons.

[13] Anneke Kleppe. 2007. A language description is more than a metamodel. In *Fourth International Workshop on Software Language Engineering*.

[14] Frédéric Madiot and Marc Paganelli. 2015. Eclipse Sirius Demonstration. In *Proceedings of the MoDELS 2015 Demo and Poster Session*. 9–11.

[15] Matt Notowidigdo and Robert C. Miller. 2004. Off-line Sketch Interpretation. In *AAAI Fall Symposium on Making Pen-Based Interaction Intelligent and Natural*. 120–126.

[16] Marian Petre. 1995. Why looking isn't always seeing: Readership skills and graphical programming. *Commun. ACM* 38, 6 (1995), 33–44.

[17] Janet M. Six and Ioannis G. Tollis. 1999. Circular Drawings of Biconnected Graphs. In *Algorithm Engineering and Experimentation*. 57–73.

[18] Anthony Sloane, Matthew Roberts, Scott Buckley, and Shaun Muscat. 2014. Monto: A Disintegrated Development Environment. In *Proceedings of the International Conference on Software Language Engineering*. 211–220.

[19] Vasco Sousa and Eugene Syriani. 2015. An Expeditious Approach to Modeling IDE Interaction Design. In *Joint Proceedings of the 3rd International Workshop on the Globalization Of Modeling Languages and the 9th International Workshop on Multi-Paradigm Modeling*. 52–61.

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

[21] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. 2014. IncQuery-D: A distributed incremental model query framework in the cloud. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. 653 – 669.

[22] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. 2015. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming* 98, 1 (2015), 80–99.

[23] Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, Joeri Exelmans, and Hans Vangheluwe. 2016. SCCD: SCXML Extended with Class Diagrams. In *Proceedings of the Workshop on Engineering Interactive Systems with SCXML*. 2:1–2:6.

[24] Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, and Hans Vangheluwe. 2017. Domain-Specific Modelling for Human-Computer Interaction. In *The Handbook of Formal Methods in Human-Computer Interaction*. 435–463.

[25] Yentl Van Tendeloo. 2015. Foundations of a Multi-Paradigm Modelling Tool. In *Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015*. 52 – 57.

[26] Yentl Van Tendeloo and Hans Vangheluwe. 2017. The Modelverse: a tool for multi-paradigm modelling and simulation. In *Proceedings of the Winter Simulation Conference*. (accepted).

[27] Hans Vangheluwe, Juan de Lara, and Pieter J. Mosterman. 2002. An Introduction to Multi-Paradigm Modelling and Simulation. In *Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems)*. 9 – 20.

[28] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Proceedings of the International Conference on Software Language Engineering*. 41–61.

[29] Vadim Zaytsev and Anya Helene Bagge. 2014. Parsing in a broad sense. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. 50 – 67.