Operationalizing the Integration of User Interaction Specifications in the Synthesis of Modeling Editors

Vasco Sousa
Université of Montréal
Canada
dasilvav@iro.umontreal.ca

Eugene Syriani
Université of Montréal
Canada
syriani@iro.umontreal.ca

Khady Fall
Université of Montréal
Canada
khady.fall@umontreal.ca

Abstract

A long shortcoming in the automatic generation of domain-specific modeling (DSM) editors has been the lack of user experience, in particular, user interaction adapted to its user. The current practice relies solely on the abstract and concrete syntax of the language, restricting the user interaction with the editor to a set of generic interactions built-in the tool. To increase the user experience with DSM editors, we propose to specify the different viewpoints of interactions (e.g., I/O devices, component of the interface, behavior of the editor) each modeled at the right level of abstraction for its user expert. The goal of this paper is to demonstrate the feasibility of the approach, by anchoring the operational semantics of all these viewpoints in a Statecharts model that controls the DSM editor. We report on the complex transformation that takes as input different viewpoints are expressed in at distinct levels of abstraction, to produce a custom interaction with the DSM editor based on a RETE algorithm. Our implementation shows that we can produce correct and responsive results by emulating existing DSM editors.

CCS Concepts • **Software and its engineering → Domain specific languages**; *Visual languages*; Software configuration management and version control systems.

Keywords IDE generation, User experience, Domain-specific modeling

ACM Reference Format:
Vasco Sousa, Eugene Syriani, and Khady Fall. 2019. Operationalizing the Integration of User Interaction Specifications in the Synthesis of Modeling Editors. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE '19), October 20–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3357766.3359539

1 Introduction

It is common practice in model-driven engineering (MDE) to automatically generate modeling editors for domain-specific modeling (DSM) languages (DSL). All streamlined frameworks, such as those based on the Eclipse Modeling Framework (EMF) [36] (e.g., Sirius, Xtext), MetaEdit+ [23], and AToMPM [40], rely solely on the abstract and concrete syntax of the language to produce the editors. As such, the user interaction with the editor is restricted to a set of built-in generic interactions. These interactions are common to all DSM editors generated from the framework and often follow a drag-and-drop or click-popup interaction style. Along with these interactions, a fixed layout is also enforced: toolbars are always placed in the same position and populated with the elements of the DSL.

In the past few years, we have seen keynotes, vision papers and panel discussions to promote the integration of User Experience (UX) design in DSM editors [1, 8, 35]. There have been some attempts to address interaction at the design level, like IFML [13]. However, such attempts are too specific to the target platform. For instance, IFML requires the definition of the interactions at the level of UML objects and the information on the specific system calls of the interaction, instead of staying at abstractions closer to the domain and UX practices. Our long-term goal is to tailor the interaction with the editor to the domain , to improve the UX.

Sousa et al. [35] coined the idea of a set of languages to model the different viewpoints of interactions. These allow the specifications of multiple modes of interaction with different input/output (I/O) devices (e.g., mouse, touch), custom editor layouts, and to configure the behavior of the editor, to deploy it on various platforms and adapted to the domain. However, Sousa et al. have not provided a way to produce functional modeling editors from the different models specifying the interactions. The main challenge is that the viewpoints are specified in various languages, expressed at different levels of abstraction. Although the languages proposed are at an adequate level of abstraction for UX experts [34], an operational semantics that yields a sound and usable modeling editor is still missing. Therefore, in this paper, we propose to anchor the semantics of all the different aspects of the user interaction into an executable artifact that opens the interaction possibilities of a DSM editor to the needs of its target users. This anchoring entails transforming all the viewpoint models of the DSM editor into a single executable model.

To achieve this, we decided to operationalize the semantics of the user interaction using Statecharts [19], given their reactive and modal behavior. To tackle the complexity of the transformation and to avoid performance issues, our solution is based on a novel RETE algorithm to construct the statechart incrementally. In the following section, we present a running example to help illustrate the different aspects of our approach in a more practical manner. In Section 3, we briefly describe the required notions of the formalisms that specify the different viewpoints of the user interaction of a DSM editor, and define a process to synthesize such editors. In Section 4, we present our main contribution: how to operationally produce an executable statechart that controls the DSM editor from the different viewpoints of interactions. In Section 5, we outline how we implemented the generation process. To evaluate the generation process, we evaluate the performance of the process and demonstrate that we can produced DSM editors with similar user interactions as existing ones in Section 6. In Section 7, we discuss challenges based on our experience with designing these editors. Finally, we overview related work in Section 8 and conclude in Section 9.

2 Running Example

As a running example, let us define a simple DSM editor to model a configuration of Conway's Game of Life (GoL) [14]. We assume DSM editors with a graphical user interface (GUI) and a visual canvas to manipulate models in their graphical concrete syntax. GoL consists of a cell grid. Each cell can be either dead or alive (blank and black respectively in our figures). The cells are subject to a series of rules that change their life status. In this example, we construct a DSM editor to model configurations of the cell grid. We consider that the operational semantic rules are specified and simulated using a model transformation.

If we were to synthesize a graphical GoL editor with common MDE frameworks, we would first create a metamodel of GoL. For instance, this metamodel would consist of a single class `Cell` with a boolean attribute `alive`, two integer attributes `xPos` and `yPos` for the coordinates of the cell, and an association `neighbors` to itself allowing up to eight neighbors. A possible graphical concrete syntax would represent a cell as a square that changes color based on the value of `alive` and a geometrical constraint to snap cells according to their `neighbors` relation. From the metamodel and the concrete syntax definitions, we can generate a graphical GoL editor using common MDE frameworks. To create a GoL model, the user needs to manually instantiate each cell from the toolbar and set the `alive` property of the cells alive.

Due to the simplicity of GoL, its editor should be reduced to its minimum, such as traditional GoL engines[1], thus removing accidental complexity. Ideally, the editor should only provide a single language element `Cell` with the attribute `life` set to `TRUE`, forgoing any language element selection toolbar or attribute view. Additionally, this editor should provide direct access to running the simulation.

3 Viewpoints of the user interaction

Our approach allows the designer to customize the interface of the editor, its behavior, and the I/O devices involved in the interaction. Therefore, we augment the artifacts required to generate a DSM editor with additional models, each representing different viewpoints of the user interaction of the editor. On top of the abstract and concrete syntaxes of the DSM language, we add an *interface model* to define the elements of the GUI and their layout, an *interaction model* to define the behavior of the editor, and an *event mapping model* to map I/O events to a specific platform and implementation. We now briefly introduce each model in what follows.

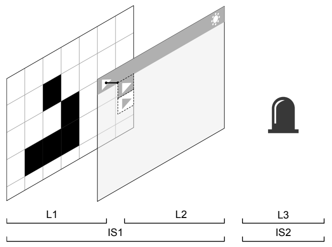

Figure 1. Desktop interface layers example

3.1 Interface model

The Interface modeling language is aimed at UX designers. It specifies what elements (visual and non-visual) the user can interact with, their representation, and their distribution in the interaction space. As such, this model takes its cues from approaches like sketching [5]. On top of providing designers the freedom to design the appearance of the editor, this model also affects user interaction. For example in Figure 1, the *play* button used to start the simulation of the GoL model is placed at the top left of the screen for a desktop application. Similarly, the user interaction influences other properties of the interface model, such as the size of elements, their color, and other forms of feedback.

Figure 2 presents an excerpt of the metamodel of the interface modeling language. It is based on OMG's Diagram Definition standard [15]. The elements of the interface model are very different from DSL elements. **Graphical elements** are used to represent toolbars, buttons, icons, widgets, and any other shape in the graphical editor. In a desktop version of our example (Figure 1), we define one toolbar with two buttons. Given that there is only one element type in this DSL, there is no need for a palette and button to instantiate a cell. Instead, we leave cell instantiation as a default interaction via, for example, interacting with the canvas.

[1] http://pmav.eu/stuff/javascript-game-of-life-v3.1.1/

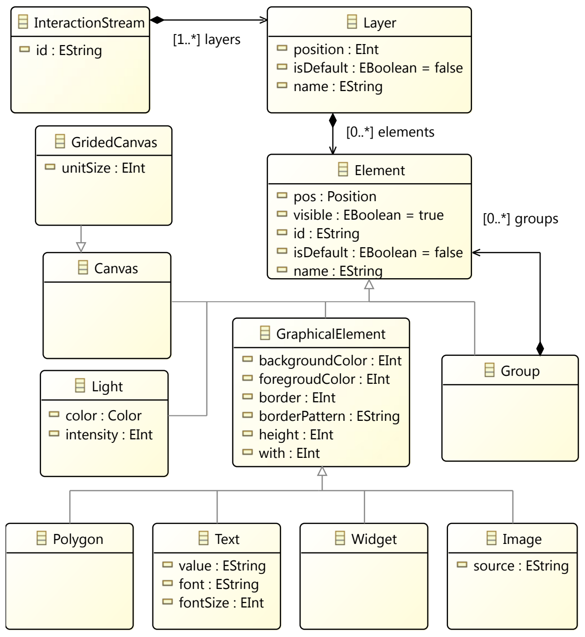

Figure 2. Excerpt of the Interface metamodel

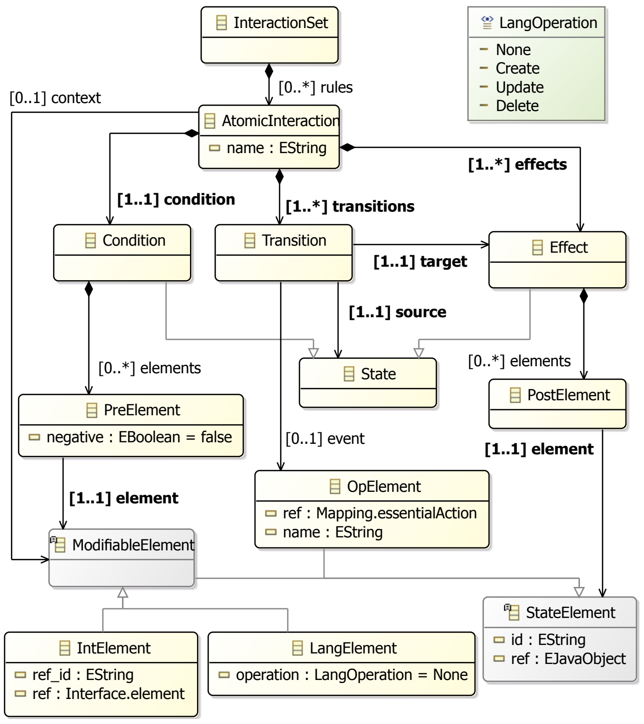

Figure 3. Interaction metamodel

The **canvas**, in which the user performs the modeling actions, is typically a grid where DSL elements are instantiated in their concrete syntax. The size of the cells of the grid depends on the editor and the language for which it is designed. In our example, the canvas is set up so that a grid cell matches the size of a GoL cell, as to provide the look and feel of traditional GoL editors. The canvas contains all instantiated DSM elements represented in their concrete syntax. Therefore, all interactions with them (e.g., selection, resize) are performed through the canvas. This allows us to treat the canvas and the DSM as a black-box that connects the editor to any model-aware back-end.

We add the concept of **layer** to segment the interface by related elements. In visual representations, this also implies that the elements in one layer are rendered overlaying the elements of lower layers. In Figure 1, there is one layer for the canvas (L1) and one for the toolbar (L2). In this interface, the toolbar will always be rendered on top of the canvas and its elements. **Interaction Streams** segment forms of interaction that are in different dimensions, e.g., screen elements, keyboard inputs, sound. In GoL, we add a **light** element in a separate interaction stream, which could be available on the mobile device or externally connected to the computer. For simplicity, we change the visual color of the icon to reflect the value of its RGB color attribute.

3.2 Interaction model

The Interaction modeling language allows UX designers to define the semantics of elements of the interface. Once the interface model and abstract/concrete syntax of the DSL are defined, the user can tailor the user interaction of the DSM editor to the target users. As defined by its metamodel in Figure 3, an interaction model consists of a set of **interaction rules**, like the five needed for the GoL editor in Figure 4.

As commonly represented in UX using Trigger-Action rules [42], interaction rules represent declarative context-dependent effects on the editor: when an event occurs and the interface or the DSM satisfies a specific condition, the rule produces an effect on either of them. Semantically, an interaction model takes its root from the model transformation paradigm and Event-Condition-Action (ECA) rules [11]. A rule has a precondition (**condition**) that describes the requirements on the state of the editor for the interaction to occur. The condition can comprise elements from the interface model (`IntElement`) or the DSL (`LangElement`). One of the elements in the condition is the **context** of the rule, represented in Figure 4 by a mouse arrow. It is used to determine the exact point of interaction. The postcondition of a rule is a set of **effects** that describe what actions to take on the interface or what operations specific to the DSL it should apply. Like in ECA, an interaction rule must be explicitly triggered by an event, typically as a result of the user using an input device. For example, in Figure 4, rule A specifies the creation of a cell: if the focus is on the canvas and we receive the `select` event, the rule instantiates an alive cell element on the canvas at the position in focus on the grid. Here, the `LangElement` refers to the `Cell` class in the GoL metamodel through its `ref` attribute. Thus, we can set the value of all of its attributes by reusing that class, such as `alive=true`. When

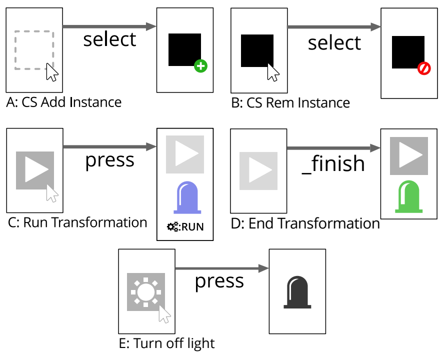

Figure 4. Interaction rules for GoL

a mapping from the metamodel to a concrete syntax is provided by the modeling back-end, we can reuse the concrete syntax definition from the DSL inside interaction rules.

Rule B specifies the removal of a `cell`: when the context is an alive `cell` element and we receive the `select` event, the rule deletes that cell. An effect can create instances of the abstract syntax of the DSL, delete them, or update their attribute values. An effect can also modify properties of elements of the interface model (such as their visibility, position, size). In the metamodel in Figure 3, `IntfElement` reuses classes from the interface metamodel, as we do for `LangElement`. The `ref_id` attribute ensures that the interface element corresponds to an existin one in the interface model. An effect may also call an external operation, defined in the event mapping model, such as `RUN` in rule C to execute the model transformation that simulates the GoL model.

The running system can also internally signal events that trigger an interaction rule. For example, in rule D, the `_finish` event corresponds to an event that is raised when the execution of the `RUN` operation terminates. The rule then enables the play button and turns the notification light to green. Note how no context is needed when the event is internal.

The interaction model expresses interactions at a level of abstraction that is closer to the intent than the implementation of the interaction within the DSM editor. For instance, rule C states that a button is pressed independently from whether it is a mouse click or a touch on the screen.

3.3 Event mapping model

Interaction rules rely on domain-specific events, called **essential actions** to keep the interaction rules independent from the implementation and match the intent of the user. There are three types of essential actions. **Input events** represent an interaction of the user on the editor (e.g., pressing a button) or complex interactions using frameworks such as [33]. **Operation events** represent an operation to be performed on an interface element, often corresponding to a function call, e.g., running the simulation. **Internal events** represent the reaction of the system to an operation event, e.g., the termination of a function. An underscore precedes the name of internal events like `_finish`.

The event mapping model defines all the available essential actions and provides a matrix that translates each one to a **system action**, a platform-specific function offered by the implementation of the system. The mapping is left-total: all essential actions require a mapping. However, that mapping does not need to be unique, as the same essential action can be mapped to different system actions. This enables the interaction rules to be applicable regardless of the I/O device or to support multiple I/O to accomplish the same interaction. For example, the event `press` can be mapped to both a touch and a left mouse click. Conversely, we can also map two different essential actions to the same system action. This is especially useful when the I/O device offers fewer interactions than the interaction model permits.

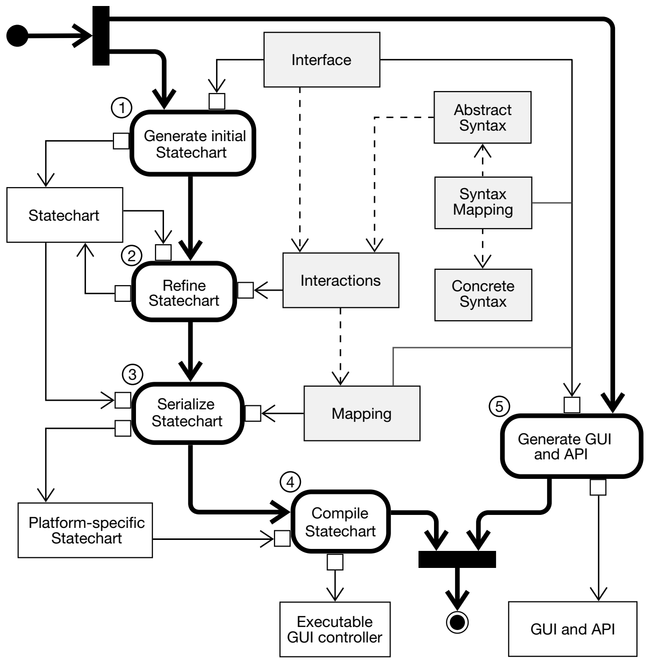

Figure 5. Process to synthesize DSM editors

3.4 Synthesis of DSM editor

Figure 5 illustrates the process to produce custom DSM editors. The generation process follows typical MDE automation techniques. In activity ①, a **code generator** takes as input the mapping between the abstract and concrete syntaxes of the DSL, the interface model, and the event mapping model to produce the GUI of the editor. For example, this can be the HTML and Javascript code of the web-based front-end editor or the front-end code of a smartphone application. It also generates the function calls to the modeling back-end to retrieve and manipulate the abstract and concrete syntax DSM elements. These functions are encapsulated as an API provided by the canvas.

The challenge of the operationalization process resides in the generation of the controller code of the GUI that defines the custom user interaction in the DSM editor. We model this behavior with a Statecharts model, which serves as semantic anchoring of the interaction model. Statecharts is a common formalism to control the behavior of GUI [20]. Thanks to its event-driven and modal paradigm, it simplifies the development process for coding, debugging, and testing the behavior of GUI. However, transforming the interface, interaction, and event mapping models to a statechart is a complex transformation that requires multiple steps.

In activity ②, we generate an initial statechart based on the structure of the interface model. This is a straightforward **model-to-model transformation** that creates default states placed in parallel regions in the statechart for each layer of the interface model. This is followed by activity ③, where we perform successive incremental refinements of the statechart to encode the semantics of every interaction rule. This is a complex **in-place transformation**. We must evaluate the applicability of each interaction rule as we are adding new states and transitions to the statechart by symbolically applying the rules. Then, with activity ④, we serialize the statechart to be compatible with the implementation framework. A **model-to-text transformation** replaces all essential action in the statechart with system actions, given the target platform, and serializes it to a standard state machine format (e.g., SCXML). Finally, in activity ⑤ we compile the state machine into code that controls the behavior of the generated GUI of the editor using off-the-shelf state machine compilers.

4 Generating the user interaction controller

4.1 Mapping onto Statecharts

We produce a statechart to control when the effects of interaction rules are applied, according to the triggering event and the satisfaction of their condition. Thus, the statechart must encode all possible rule applications for every state of the DSM editor, i.e., all reachable configurations of the GUI elements specified by the interface model. We assume that the interface model is partitioned into distinct layers and that a single user interacts with the editor. Therefore, at any given time, the state of the GUI involves at most one configuration of the elements from each layer. Hence, the structure of the hierarchical statechart shall consist of one high-level AND-state containing one parallel region (a.k.a. orthogonal component [19]) per layer as in Figure 6. Since we assume all graphical DSM editors have a canvas, there is always one layer dedicated to the canvas.

We define a **statechart** by $SC = \langle L, S, T, layer \rangle$ to contain a set of layers L, a set of basic states S, and the function $layer : S \rightarrow L$ identifying the layer of each state. Transitions in $T = \langle S, S, Ev, \wp(A), g \rangle$ connect states within the same

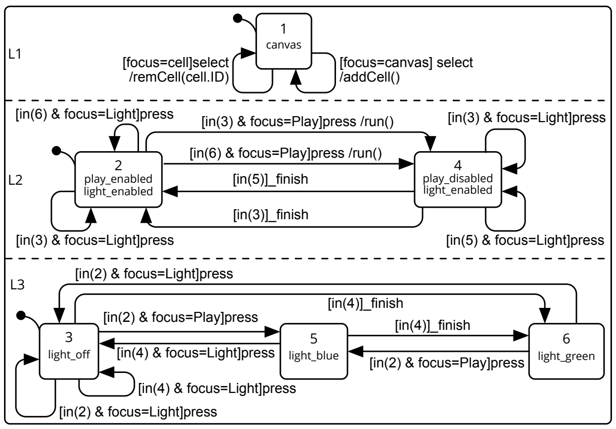
L1
1
canvas
[focus=cell]select
/remCell(cell.ID)
[focus=canvas] select
/addCell()
L2
[in(6) & focus=Light]press
[in(3) & focus=Play]press /run()
[in(3) & focus=Light]press
2
play_enabled
light_enabled
[in(6) & focus=Play]press /run()
[in(5)]_finish
4
play_disabled
light_enabled
[in(3)]_finish
[in(3) & focus=Light]press
[in(5) & focus=Light]press
L3
[in(2) & focus=Light]press
[in(4)]_finish
[in(2) & focus=Play]press
[in(4)]_finish
3
light_off
[in(4) & focus=Light]press
5
light_blue
[in(2) & focus=Play]press
6
light_green
[in(4) & focus=Light]press
[in(2) & focus=Light]press

Figure 6. The resulting statechart after refinements according to the five rules

layer, triggered by an event in Ev. They perform a set of actions A and have a guard g as triggering condition.

States within a layer represent the state of all interface elements within that layer at a given point of interaction. For example in GoL, layer L3 contains only the light. After several refinement iterations, there will be three states in the orthogonal component L3: one for every color and intensity of the light present in the interaction rules (see the bottom layer of Figure 6). Any statechart produced will have a layer corresponding to the modeling canvas (L1). In the scope of this work, we consider the canvas as a black-box; thus we only need one state to represent its reachable configuration.

Transitions represent how the state of the interface elements can change according to the application of a rule. For example, because of rule E, each state in L3 has a transition to state 3 where the light is off. The **event** of the rule triggers each transition it corresponds to, `press` in this example. However, the light button is the context of rule E, meaning that we cannot take a `press` transition unless the corresponding states in L2 are active and the context (focus) is on the light. Therefore, we must add a **guard** on transitions to guarantee that we are in the required states in all layers referenced by the rule. The **actions** of a transition apply the effects of the interaction rule. So changes to interface elements are all encoded as actions in the transitions. For example, all transitions incoming to state 3 must set the light to `RGB(0,0,0)`. Actions can also manipulate DSL elements, such as the creation of a `cell` in rule A. Any operation invoked in the effect of a rule is added to the actions of the transition, such as `RUN`.

We denote a **configuration** by $config : S \longrightarrow \wp(ModifiableElement)$ to map each state to a set ME of interface or language elements. It represents the reachable snapshot of the state of the editor at a given time. For $(s, ME) \in config$, s is the active state of the orthogonal component corresponding to $layer(s)$. ME encodes the attribute values of every interface element in that layer corresponding to s, possibly with instances of the metamodel of the DSL. The **initial configuration** represents the default states of each layer, i.e., the way they are set up at the startup of the editor. For GoL, the initial configuration is thus $\{(1, ME1), (2, ME2), (3, ME3)\}$ respectively representing an idle canvas, a toolbar with the play and light buttons enabled, and the light turned off.

4.2 Challenge of incremental refinement

The initial statechart consists of one orthogonal component per layer, as defined in the interface model, each containing one state. As we have not considered any interaction rule yet, it represents an unresponsive editor. The following transformation refines the initial statechart by detecting which rules are applicable at any given state configuration. It applies each interaction rule incrementally, by adding new states and transitions that represent these interactions. Each step also gives us a new configuration representing the state of the editor resulting from the rule application. It is then, in turn, evaluated to detect which rules are now applicable. In this manner, after all five rules are processed, the statechart evolves to its final form as depicted in Figure 6.

One major challenge of this transformation is that we do not know how many states and configurations will be evaluated apriori. Therefore, we must consider all the states of the statechart in all configurations as potential candidates to apply every interaction rule. Suppose that we are now applying rule C in the GoL example. The transformation creates a new state and transition in the statechart that disables the play button, turns on the blue light, and runs the simulation. However, we now have to verify if this new state enables new rule applications, such as rule E to turn off the light.

This will lead to an explosion of the number of state configurations and require a considerable time and memory footprint for complex interaction models. However, typically each interaction rule only modifies a few elements, and thus it is not necessary to re-evaluate complete configurations every time. Ideally, we only evaluate the changes of the elements affected by a rule. Therefore, we need a transformation engine that allows us to control how much of a state configuration should be evaluated and combine that partial evaluation with previous evaluation results. Similar challenges have been identified in the model transformation community which led to the rise of *incremental transformations* [44]. Inspired by these approaches, we define an incremental refinement procedure to construct the statechart, while reusing evaluations of previous state configurations.

4.3 Incremental rule evaluation

To address the re-evaluation of state configurations we provide a RETE-based algorithm [12]. The algorithm takes a series of facts, in our case configurations, and evaluates what interaction rules to apply on these facts. The algorithm relies on a **RETE network** composed of $\langle A, B_\vee, B_\wedge, E, elem, layer, rule \rangle$. Figure 7 shows the RETE network for GoL.

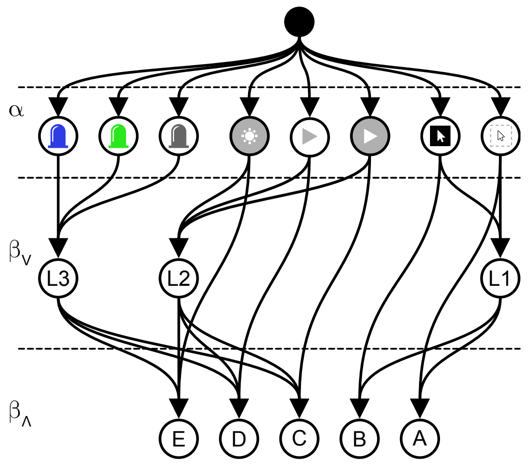

Figure 7. Structure of the RETE network for GoL

4.3.1 Construction of the network

Alpha nodes $\alpha \in A$ filter the facts to values required by interaction rules. Therefore, an α-node represents a modifiable (interface or language) element present in the condition or effects of a rule with a distinct set of attribute values. For instance, we will have three α-nodes for the light, one for each color/intensity defined in the rules. The function $elem : A \rightarrow ME$ assigns a modifiable element to each node.

Beta nodes aggregate the results from its incoming nodes. We distinguish between two sets of beta nodes. A $\beta_\vee \in B_\vee \subset B$ node groups α-nodes by layer and a $\beta_\wedge \in B_\wedge \subset B$ node represents each interaction rule. Consequently, $layer : B_\vee \rightarrow L$ assigns a layer to a $\beta_\vee$-node and $rule : B_\wedge \rightarrow IR$ assigns an interaction rule to a $\beta_\wedge$-node. Edges in E connect nodes according to their structure. An edge connects α to $\beta_\vee$ if $elem(\alpha)$ pertains to $layer(\beta_\vee)$. An edge connects $\beta_\vee$ to $\beta_\wedge$ if there is an element in $layer(\beta_\vee)$ involved in $rule(\beta_\wedge)$. To make sure that the whole condition of an interaction rule is satisfied, we also connect α to $\beta_\wedge$ if $elem(\alpha)$ is part of the condition of $rule(\beta_\wedge)$.

For example, to encode rule C in the network, we need three α-nodes: one for an enabled play button, one for a disabled play button, and one for a blue light. The play buttons belong to layer L2, so their α-nodes are connected to the same $\beta_\vee$, whereas the light belongs to L3. Furthermore, the condition of the rule is to have the button enabled. Thus we have an edge from its α-node to the $\beta_\wedge$-node corresponding to rule C. We also connect edges from the $\beta_\vee$-nodes for L2 and L3 to that $\beta_\wedge$-node.

4.3.2 Evaluation of the network

The purpose of the RETE network is to detect when an interaction rule is applicable during the refinement process. Instead of naively evaluating the applicability of each rule on every possible configuration, the network should tell us if a given configuration induces a change that triggers the application of a rule. To do so, we define a **token function** $\langle A \rightarrow \text{bool} \times \text{bool}, B \rightarrow [\text{bool}]_{in} \rangle$ for each type of node. This function assigns two booleans to each α-node. The first boolean is the **presence token**. It is set to *TRUE* only if, for a given configuration $config$, $\forall (s, ME) \in config, \exists ME$ such that $elem(\alpha) \in ME$. The second boolean is the **change token** representing if the presence token has changed. Each beta node is assigned a boolean array of the size of its incoming edges *in* to identify the provenance of each token. Initially, all tokens are set to *FALSE* throughout the network, but the change tokens are set to *TRUE* to ensure that all rules will be considered.

To determine which rules are applicable from a given configuration, we propagate tokens in three stages: first from α-nodes, then from $\beta_\vee$-nodes, and finally from $\beta_\wedge$-nodes. If modified, the presence token of an α-node is propagated via the edge to the $\beta_\wedge$-node. This contributes to deciding whether the condition of $rule(\beta_\wedge)$ is satisfied. If modified, the change token of an α-node is propagated via the edge to the $\beta_\vee$-node. This indicates that the effect of a rule leading to $config$ has changed $elem(\alpha)$ which may require to re-evaluate any rule it is involved in. When the configuration is processed, and all tokens from α-nodes have propagated to the beta nodes, we evaluate the $\beta_\vee$-nodes. Since it suffices that one element of the layer involved in an interaction rule is changed to trigger it potentially, $\beta_\vee$-nodes perform the disjunction of all the tokens they hold (newly received and from the previous iteration). We propagate the result to the $\beta_\wedge$-nodes they are connected to. Finally, a $\beta_\wedge$-node performs the conjunction of all the tokens it holds from incoming α-nodes to verify if the condition of its rule is satisfied. However, a change may have happened in a layer that is not involved in the condition of the rule. Therefore, to intersect all layers involved in the rule, $\beta_\wedge$-nodes perform the conjunction of all the tokens they hold, including those from incoming $\beta_\vee$-nodes.

This strategy is particularly efficient when we have a large set of facts that incur minimal changes with each execution of an interaction rule. To illustrate the evaluation of the network in Figure 7, suppose it receives a configuration encoding that the `play` button is disabled (e.g., after rule C was applied). Note that there are two α-nodes assigned to this button for when it is enabled and disabled. The configuration entails the reevaluation of these α-nodes, flipping their presence token in relation to the previous configuration. Consequently, the change token of `play_disabled` is set to *TRUE* and propagated to the $\beta_\wedge$-node of rule D. Similarly we propagate the *FALSE* value token of `play_enabled`. The presence token of the disabled node is propagated to the $\beta_\vee$-node corresponding to layer L2. Through disjunction, it propagates *TRUE* to the $\beta_\wedge$-nodes. Finally, we perform the conjunction of the $\beta_\wedge$-nodes of rules C, D, and E. The result is that, from the given configuration, rule D is applicable but not rule C.

4.4 Incremental statechart refinement

Algorithm 1 defines the procedure to refine the initial statechart incrementally as new configurations arise from the application of the rules. It first creates a RETE network from the set of interaction rules and initializes its token function, as described in Section 4.3.1. Then, on line 7, it evaluates the network as described in Section 4.3.2 by propagating the tokens according to the current configuration. This gives us the set of applicable rules from this configuration and the new token function. It then applies each rule to refine the statechart (explained in Section 4.5), which produces a potentially new configuration. It is crucial to couple new configurations with the token assignments that led to it so that, when it will be processed, the new configuration will be evaluated on that specific token assignment. The procedure continues processing configurations until all reachable configurations have been processed.

Algorithm 1: RefineStatechart(SC, config, interactions)

```
Input: the initial statechart, the initial configuration, and the
       interaction set
1  network ← GenerateNetwork(interactions)
2  token ← network.initToken()
3  evalQueue ← {⟨config, token⟩} ; configHist ← ∅
4  while evalQueue ≠ ∅ do
5      ⟨config, token⟩ ← evalQueue.dequeue()
6      (AppRules, new_token) ← network.eval(config, token)
7      for rule ∈ AppRules // applicable rules do
8          new_config ← ApplyRule(SC, rule, config, configHist)
9          if new_config ∉ configHist then
10             evalQueue.enqueue(⟨new_config, new_token⟩)
11             configHist ← configHist ∪ { new_config }
```

The procedure keeps track of the configurations already processed; therefore each configuration is processed once. Hence, at any point in time, `evalQueue` will only contain new configurations. Since the RETE network evaluations drive the procedure, it limits the explosion of exploring all possible state combinations of the interface. In the worst case, a change in a modifiable element triggers the reevaluation of all α-nodes. Therefore, a strict upper bound of the state-space of the `RefineStatechart` algorithm is $|A|^2$. In practice, this will be lower since many interface elements have mutually exclusive states, like buttons and lights.

4.5 Construction of the statechart

Algorithm 2 refines the statechart by applying the applicable rules on the current configuration provided by the evaluation of the RETE network. These refinements adhere to the refinement method and rules defined in [37].

Recall that a state *s* in the statechart represents a particular state of all interface elements of the layer corresponding to

Algorithm 2: ApplyRule(SC, rule, config, configHist)

```
Input: a statechart, the interaction rule to be applied, the
       configuration to apply the rule on, and the
       configuration history
Output: a new configuration and updates the statechart to the
        rule
1  $L$ ← rule.getAllLayers() // from condition and effect
2  ActiveStates ← $\{s \mid (s, ME) \in$ config $\wedge$ layer($s$) $\cap L \neq \varnothing\}$
3  config' ← clone(config)
4  actions ← rule.effects.getOperationElems()
5  event ← rule.getEventName()
6  for $s \in$ ActiveStates do
7      guard ← ⟨ ActiveStates \{$s$}, rule.getContextName()⟩
8      for $e \in$ rule.effects.getModifElems(layer($s$)) do
9          actions ← actions ∪{$e$}
10         config'[$s$].replace($e$)
11     ME' ← config'[$s$]
12     target ← configHist.findState(ME',layer($s$))
13     if target = NULL then
14         target ← SC.newState(layer($s$))
15     SC.newTransition($s$, target, event, guard, actions)
16     config[target] ← config'[$s$]
17     actions ← ∅
18 return config
```

the orthogonal component containing s. On lines 1–2, we identify the set of active states from the configuration that is affected by the interaction rule. Thus, this set contains at most one state per orthogonal component. If there are language elements in the interaction rule, the canvas state is also part of the set so those elements can be treated in the same way as interface elements. Then, we compute the effects of the rule on the elements of these active states (lines 8–10). Afterward, we create a new transition starting from each active state to a state corresponding to the effects of the rule on the same layer (line 15), be it a new state, or an existing state.

The transition is triggered by the event that corresponds to the event of the rule (line 6). The set of actions of a transition is defined by the attribute values of interface elements (line 9), the CRUD operations on DSL elements (line 9), and the external operations to invoke (line 4). The function `getModifElems` returns language elements (e.g., deleted cell in rule B) or interface elements (e.g., unlit light in rule E) of a specific layer in the effect of the rule. In rule C, the function `getOperationElems` returns the light with a blue color and the `RUN` operation with its parameters.

The condition and effects of a rule may involve elements from different layers, such as rule C. Therefore, as explained in Section 4.1, we must synchronize the transitions across layers to satisfy the condition of the rule. On line 7, we see that a guard has two components: a state and a context condition. The state condition synchronizes transitions across layers. Suppose we are in the configuration $\{(1, ME1), (4, ME4), (3, ME3)\}$ and receive a `press` event on the play button in Figure 6. If we did not have the state condition $in(2)$ on the guard of transition from 3 to 5, then the light could turn blue even when the play button is disabled. The context condition distinguishes between transitions sharing the same source state triggered by the same event. For example, if we are processing state 3, we will eventually apply rules C and E. Since their transition relies on the same event and state condition, we must distinguish their trigger with a guard on the context of the rule. The function `getContextName` returns the value of `ref_id` if the context is an interface element, the type name when it is a DSL element, or `null` otherwise.

Along the way, we construct the new configuration resulting from the application of a rule, by modifying the previous configuration with the elements in the effect (line 10). If another rule already created the new configuration, then the statechart already has a state corresponding to it (line 11). Then, this state should be the target of the new transition. The function `findState` searches through the configuration history for a state corresponding to the set of modifiable elements in the new configuration within the layer of s. The refinement of the statechart ends when Algorithm 1 is completed, resulting in the statechart in Figure 6.

4.6 Adaptation to a specific platform

The serialization process involves two parts. We translate the final statechart into an executable format, such as SCXML, typically using a model-to-text transformation engine. Additionally, we must adapt the events and actions of every transition to be compatible with the target platform. Two APIs must be readily available for the platform: one to manipulate DSM elements and one to operate interface elements. The event mapping model defines the required translations. The modeling framework, such as EMF, provides the first API. Actions on transitions in the canvas layer are translated into the CRUD operations of the modeling framework API. The second API must be readily available for the I/O devices with which the user will interact. Actions on transitions are simply setters of interface elements. For example, if the light element in GoL is mapped to an RGB LED in Arduino, then the value of its color attribute is mapped to the `setColor(r,g,b)` function call with the right parameters. Also, the operation event `RUN` is replaced by calling the function corresponding to the execution of the simulation.

5 Implementation

To demonstrate the feasibility of our approach, we implemented a prototype as an Eclipse project available online[2]. We defined the input modeling languages in EMF using Ecore

[2]https://github.com/geodes-sms/DSMEditorGenerator

metamodels and deployed as plugins. We defined a custom simple hierarchical Statecharts language where we enrich states with references to interface elements they encapsulate and their values to optimize the configuration lookups at runtime. We implemented all transformations using the Epsilon tool suite [24] because it integrates different types of transformations seamlessly and allow us to run them in headless mode outside of Eclipse.

We use the declarative model-to-model transformation language ETL [25] for the transformation *Generate Initial Statechart* (see Figure 5). It is composed of a series of declarative patterns that translate each layer of the interface model into an orthogonal component with a single initial state. We then populate that state with the references to all the elements within that layer. Finally, we aggregate all orthogonal components inside a root state. We developed the complex transformation *Refine Statechart* in EOL [24]. It implements the algorithms presented in Section 4: constructing and evaluating the RETE network, and applying interaction rules. Here, we developed a temporary data structure to represent tokens and configuration history. For the *Serialize Statechart* transformation, we used the template-based code generator EGL [30]. We transform the statechart into the SCCD [43] format, which is compatible with Harel's semantics [19]. Apart from specifying the serialization format in the templates, we convert essential actions to system actions according to the event mapping model (see Section 4.1) and discard references from states to interface elements.

The tool generates web-based graphical DSM editors. Therefore, we generate the GUI in HTML/CSS/Javascript. We add listeners specified in the event mapping model to raise the appropriate event in the statechart. For instance, we add a listener for a mouse left click event where, if the context is the play button, it raises `pressPlay`. We provide a Javascript API for the statechart to control the interaction with the GUI.

6 Evaluation

There are many aspects to evaluate whether our approach to generate and execute DSM editors with custom interactions is appropriate. Since the main contribution of this paper is the generation process, we focus on the following two objectives: *(1) Is our approach expressive enough to mimic user interactions in existing DSM editors? (2) Does the generation of the editor scale with the increasing complexity of user interactions?* For (1), we qualitatively assess the user interaction with the generated editors compared to existing ones. For (2), we measure the time performance and memory usage of the transformation process with varying complexities of interaction models.

6.1 Reproducing existing editors

To evaluate that the declarative specification of interactions results in a correct interaction environment, we implemented three editors that replicate the main interaction aspects of existing editors. All three editors are available online[2]. Table 1 outlines the number of interaction rules, interface model elements, and layers required to generate each editor. It also depicts the size of the generated network and statechart.

Number of	**GoL**	**Pacman**	**Music**
Rules	5	42	66
Interface elements	6	13	32
Layers	3	3	4
α-nodes	8	38	12
States	6	17	23
Transitions	18	135	515

Table 1. Characteristics of each case study

6.1.1 GoL editor

The web-based GoL application[1] allows one to define the initial alive cells by clicking on a grid and offers a button to run the simulation, as described in Section 2. There are other widgets available on the web application, such as buttons to create predefined cell patterns and statistics on the running simulation. We generated the GoL editor following the description throughout this paper, which concentrates on its essential interaction. The goal was to have a minimal editor capable of illustrating our approach with which we can verify its correctness manually. To illustrate the ability to interact with multiple interaction streams, we added the light functionality to the editor.

6.1.2 Pacman editor

The goal of this case is to replicate a DSM editor generated inside a modeling language workbench. The Pacman editor present in AToMPM [38] (which has similar behavior to a Sirius editor in Eclipse) allows the modeler to define game configurations where Pacman navigates through grid nodes searching for food to eat, while ghosts try to kill him. The particularity of this editor is its emphasis on language elements. It requires a toolbar with the different DSL elements to instantiate and the ability to drag-and-drop elements on the canvas. As reported in Table 1, the statechart needed 17 states and 135 transitions, 21 of which are on the canvas state to manipulate language elements. The three layers correspond to the canvas, toolbar, and popup window to view the properties of language elements.

The main challenge, in this case, is to enable dragging DSM elements. Each element type (e.g., Pacman, ghost) required three interaction rules. One rule marks the element to be dragged, triggered by a `mouseDown` event. A second rule allows us to move the element on the canvas triggered by a `mouseMove` event. A third rule fixes the element at a specific position upon receiving a `mouseUp` event. Furthermore, we required 30 interaction rules dedicated to selecting the DSL types on the toolbar to make sure exactly one type is selected when creating an element on the canvas. This is due to interaction rules not being capable of sharing information.

6.1.3 Music sheet editor

Software tools targeted at non-programmers involve various kinds of interactions [31]. For example, Flat[3] allows musicians to compose music sheets by playing on a MIDI keyboard. To replicate some of its unique methods of interaction, we generated a music sheet editor where the user is presented with a series staffs aligned with the gridded canvas. Via a MIDI or regular keyboard, the user can play a sound which places a note next to the previous one on the canvas. The key pressed places the note on the corresponding position (e.g., Do, Re, Mi). The note length (e.g., whole, half, quarter, eighth) can be selected in the toolbar, or the user can switch on a timed mode. In timed mode, the duration of a key press determines the note length. The editor requires 26 interaction rules for the toolbar interactions, 28 for note input with the toolbar, and 32 for timed note input. The interface model consists of four layers: canvas, toolbar, cursor, and staff/ledger lines. The states in the staff layer have no transitions since the user cannot directly interact with the staffs.

The main challenge, in this case, is the extensive reliance on an input method where the user does not select points on the canvas. Therefore, we need to add a visual cursor and move it after placing each note. Another challenge is to select the right language element determined by time durations. To do so, we separate a keystroke input into a rule to mark the beginning and another to mark the end of a note. We must also keep track of the passage of time between the trigger of these rules.

6.2 Evaluation of the performance of the generation

From a performance point of view, the most important phase in the generation process of the DSM editor is the construction of the statechart. As described in Section 3.4, it requires three steps: generating the initial statechart, refining it, and serializing it. We report the time and memory consumption of the construction of the statechart.

6.2.1 Experiment setup

In Section 4, we showed that the number of α-nodes influences the construction and evaluation of the RETE network, as well as the refinement of the statechart. We hypothesize that it is highly correlated with the number of interaction rules. To evaluate this hypothesis we performed a Monte Carlo simulation on a synthesized suite of interaction models containing from 1 up to 300 rules. The synthesis of these rules followed a uniform distribution for the following parameters. The condition and effect of each rule contained at most 10 modifiable elements, which corresponds to the order

[3]https://flat.io/

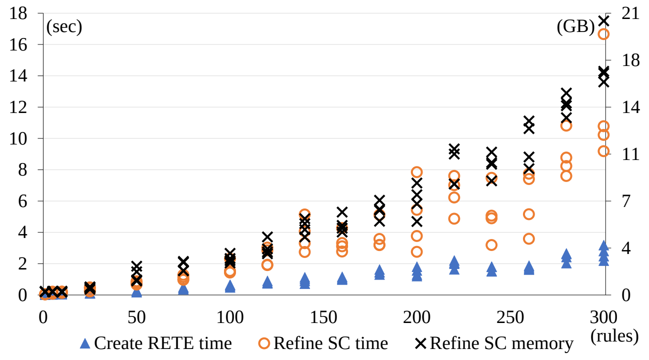

Figure 8. Time and memory usage of the `RefineStatechart` procedure in seconds and gigabytes per number of rules

of magnitude we observed when mimicking the interaction of existing editors. We gave an equal probability for each element to be an interface or language element. We also gave a 60% chance for an element in the condition of a rule to match an element in the effect of another rule (like rules A and B in GoL) to provoke sequences of interactions. To ensure only the interaction model influences the results, we fixed the interface model to consist of 10 layers, each containing one interface element. We also fixed the metamodel of the DSL to contain 10 language elements. All the synthesized models and characteristics are available online[2]. We ran the experiments on a Windows 10 machine with an i7-4770 processor at 3.8GHz, 32GB of RAM on Eclipse Oxygen with JDK 1.8 and the heap size set to 24GB. We measured execution times with Epsilon Profiler and collected memory usage with VisualVM.

6.2.2 Methodology

We ran the transformations on each interaction model 30 times and collected the average time and memory consumption. To mitigate the risk of biasing the results because of the random choices in the construction of the rules, we replicated the synthesis of the models five times with different seeds. We conducted Levene's test to assess between-group variance differences and one-way ANOVAs with post-hoc tests to assess between-group mean differences. For both time and memory performance of the refinement phase, there are no differences in variance ($p = .421, p = .950$ resp.) or in mean ($p = .854, p = .994$ resp.) among the replicas. Therefore, for a given number of rules, the random choices to construct the rules do not influence the performance on our sample. Furthermore, a Pearson correlation showed an almost perfect association between the number of rules and α-nodes ($r = .980, p = .001$). Therefore, we can safely report the performance as a function of the number of rules. All the detailed results are available online[2].

6.2.3 Results

Since we fixed the number of layers, the generation of the initial statechart in ETL requires negligible time and memory usage. The serialization in EGL grows with the size of the resulting statechart, staying below 550ms. As expected, the only significant influence on the performance is the refinement transformation presented in Algorithm 1. Figure 8 shows the results of running this step on our data set. Overall, the execution time increases with the number of rules. We also note more variability as the number of rules increases. This is due to the randomness injected in the data set, but also to the Eclipse environment and its impact on the JVM. The figure also shows the time needed to create the RETE network. A regression model fitting the data well ($R^2 = .960$) indicates that it increases linearly with the number of rules ($b = .008$), yet very slightly as the number of rules grows. However, for interaction models with less than 50 rules, the creation of the network takes most of the resources.

The refinement is an incremental transformation and, therefore, uses a significant amount of memory. Figure 8 shows that memory usage follows the same trend as time performance. Compared to the DSM editors we modeled in Section 6.1, generating an editor with 200 rules takes less than 10 seconds and under 8GB of memory.

6.2.4 Limitations

The performance evaluation is highly dependent on the synthesized rules and our prototypical implementation of the general approach. Nevertheless, it provides a preliminary idea of the performance to expect. However, our approach aims at creating DSM editors, not complete development environments, like Eclipse. Therefore, as observed with the three qualitative cases, the complexity and number of rules we synthesized in the data set remains reasonable. Optimizing the implementation of the generator and measuring performances with a headless deployment are among the items to mitigate threats to validity. Also, we shall investigate further examples with increasing variability — for instance, interaction models where a higher number of rule conditions depend on rule effects.

7 Discussion

Having experimented with this novel approach of generating DSM editors, we report some future challenges and discuss their impact on the approach.

7.1 Coping with the number of interaction rules

The reproduction of the presented case studies has shown an elevated number of interaction rules. This stemed mostly from dealing with determinism between interactions whose conditions where not were not mutually exclusive at a conceptual level, but required to be so for implementation reasons. In the Pacman editor case, the number of interaction rules would be significantly reduced had the interaction language allowed us to share information between rules. Some model transformation languages [39, 44] allow rules to use a context provided from other rules when finding matches. Likewise, we could augment interaction rules with a construct that explicitly marks certain elements of a rule to be reused in other rules, e.g., to retain which language element in the toolbar is selected. The impact on Algorithm 2 should be minimal.

In the Music sheet editor case, some rules are dedicated to placing a note on the staff and others to move the cursor right after. Rules having sequences of effects are very useful to apply a sequence of actions in a single rule elegantly. One future objective is to reduce the amount of rules by enhancing the interaction language with specific constructs that dictate a partial order in the application of the rules.

The procedure to construct the statechart from interaction rules ensures the statechart is deterministic thanks to the use of guards. However, in general, two interaction rules could be applicable at the same time, or their condition could overlap. Extending the interaction model with a mechanism to prioritize or explicitly control the application of rules is another line of research to explore. Some support to detect overlapping or conflicting rules would be useful to the user. All these improvements will contribute to reducing the number of rules to design.

7.2 Analysis of interaction rules

Given the number of interaction rules the user has to define, we should support the user with general analysis of the rules to reduce the cognitive effort. The choice of Statecharts as semantic domain brings several advantages, in particular it offers in-depth analysis thanks to many automated verification methods [27]. These analysis can then be reported back at the level of the rules. For example, we can verify general soundness criteria, such as the presence of deadlocks [22], reachability and cycle detection [2]. Several tools offer debugging facilities for Statecharts, like Cameo [28]. It is then possible to propagate the debugging to the level of interaction rules, following the methodology in [26].

7.3 Concrete syntax manipulation

In this paper, we have limited the interaction with language elements to CRUD operations. In general, more interactions are desirable, such as resizing, rotating, or selecting multiple DSM elements on the canvas. This is achievable by adding another set of interaction rules specific for concrete syntax manipulation. This would expand the canvas layer to have more states and transitions to handle these kinds of interactions.

7.4 Incrementality and refactoring of the statchart

The generated statechart is not optimal: e.g., solitary states should be removed, and transitions can be collapsed together.

We can apply common Statecharts refactoring patterns following good design principles [20]. For example, both transitions from state 2 to state 4 in Figure 6 are triggered by the same event, have the same context condition in their guards, and perform the same action. Therefore, we can collapse them into one transition and merge their guards.

The generation process of the statechart to control the GUI of the editor doesn't necessarily have to start from the initial statechart. Algorithm 1 is incremental, meaning that we can provide it with a pre-built statechart describing the common default functionalities of DSM editors and augment it with interaction rules specific to the DSL. This would require the statechart to be structured in layers and to retain the configuration history leading to it.

7.5 Usability of DSM editors

Finally, it will be interesting to conduct a controlled user study to identify if the generated DSM editors with custom interactions are more appropriate than those generated in language workbenches and evaluate the productivity and effort required to construct interface, interaction, and event mapping models.

8 Related Work

8.1 User-defined interactions for GUI

The field of human-computer interaction is rich with a lot of work on UX that aims at improving the interfaces, actions, and feedback of the interactions between a user and the software [4]. However, MDE developers do not have an effective and efficient way of describing such interactions to integrate them in the development of DSM editor. Typically, the GUI design of these editors relies mostly on the typical approach of sketching [5], where the designer creates GUI mockups deciding on the layout of components, their relation, and the integration of data. These methods can be applied when designing the interface model in our approach.

Approaches like IFML [13] allow developers to model and generate web applications. In IFML, developers specify user interactions by describing the components, state objects, data, and control logic of the interactions. Even if it does not integrate UX concerns, IFML targets object-oriented developers with its close dependency to UML. Interactions are implemented as object-oriented functions. Our approach targets UX designers to define interactions using high-level interaction rules, involving elements of the DSL and the interface model. Like IFML, when specific operations must be executed in the effect of the rule (like `RUN` in GoL), an API must be available to implement the operation. However, the event mapping model hides this from the designer who can call a high-level essential action. Following domain-specific modeling principles, our objective is to promote end-user development, as motivated in [29].

Many tools allow UX designers to configure interaction rules [6]. The paradigm of the interaction rules we propose is inspired by ECA rules [11] and Trigger-action rules [41, 42] which allow users to specify interactions visually using domain-specific concepts. A study in the IoT domain [10] shows that ECA rules are well-suited for the expertise of end-users. In [7], the authors combined MDE techniques with WebML and ECA rules to generate web applications. Our approach integrates concepts of the DSL to generate DSM editors. The work in [32] uses a similar declarative approach to specify interactions with visual domain elements. However, these are sequences of interactions centered on data visualization and tied to the visualization platform. In the model transformation paradigm, Guerra and de Lara [16] define graph transformations that are triggered by user actions, such as scaling language elements, which, in turn, triggers GUI events.

8.2 Statecharts controlling GUI

The Statecharts formalism is considered a defacto standard for modeling reactive systems [17]. It is also very often used to synthesize source code automatically [18, 21] to a variety of platforms (e.g., web, embedded systems), and programming languages (e.g., C, Java, Python). These two considerations make it ideal for describing the reactive nature of a modeling editor and integrate that description in an automated generation process. Horrocks [20] describes design principles to control a GUI with statecharts.

In MDE, AToMPM [40] and AToM^3 [9] are two language workbenches where a statechart controls the interaction with the GUI of the DSM editor. It is based on [3] that shows how to design a statechart for drag-and-drop, selection, and, moving elements in concrete syntax on the canvas. These interactions are similar to the Pacman editor case.

9 Conclusion

This work proposes to customize the GUI and user interaction of DSM editors generated following the MDE methodology. Traditionally, the metamodel and concrete syntax definition of the DSL suffice to generate these editors. We augment this input with modeling languages geared to UX experts: an interface model, an interaction model, and an event mapping model. This paper concentrates on how to operationalize these user interaction specifications into the generation of DSM editors. We transform these models into a statechart that controls the behavior of the editor. A preliminary evaluation shows that the transformation scales with the increasing complexity of interactions and that the resulting editor is capable of mimicking interactions commonly found in DSM editors. The results of this work open many research opportunities, especially to foster collaborations between the MDE and the UX research communities.

References

[1] S. Abrahão, F. Bourdeleau, B. Cheng, S. Kokaly, R. Paige, H. Störrle, and J. Whittle. 2017. User Experience for Model-Driven Engineering: Challenges and Future Directions. In *Model Driven Engineering Languages and Systems*. IEEE, 229–236.

[2] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. 2005. Analysis of Recursive State Machines. *Transactions on Programming Language Systems* 27, 4 (2005), 786–818.

[3] J. Beard and H. Vangheluwe. 2009. Modelling the reactive behaviour of SVG-based scoped user interfaces with hierarchically-linked statecharts. In *SVG Open Conference*.

[4] J. Burgoon, J. Bonito, B. Bengtsson, C. Cederberg, M. Lundeberg, and L. Allspach. 2000. Interactivity in human–computer interaction: A study of credibility, understanding, and influence. *Computers in human behavior* 16, 6 (2000), 553–574.

[5] B. Buxton. 2007. *Sketching User Experiences: Getting the Design Right and the Right Design.* Morgan Kaufmann Publishers Inc.

[6] D. Caivano, D. Fgli, R. Lanzilotti, A. Piccinno, and F. Cassano. 2018. Supporting end users to control their smart home: design implications from a literature review and an empirical investigation. *Journal of Systems and Software* 144 (2018), 295–313.

[7] S. Ceri, F. Daniel, M. Matera, and F. Facca. 2007. Model-driven Development of Context-aware Web Applications. *ACM Transactions Internet Technology* 7, 1 (2007).

[8] L. Constantine. 2009. Interaction Design and Model-Driven Development. In *Model Driven Engineering Languages and Systems*. Springer, 377–377.

[9] J. de Lara and H. Vangheluwe. 2002. AToM^3: A Tool for Multi-formalism and Meta-Modelling. In *Fundamental Approaches to Software Engineering (LNCS)*, Vol. 2306. Springer, 174–188.

[10] G. Desolda, C. Ardito, and M. Matera. 2017. Empowering end users to customize their smart environments: model, composition paradigms, and domain-specific tools. *ACM Transactions on Computer-Human Interaction* 24, 2 (2017), 12.

[11] Klaus R. Dittrich, Stella Gatziu, and Andreas Geppert. 1995. The active database management system manifesto: A rulebase of ADBMS features. In *Rules in Database Systems*. Springer, 1–17.

[12] C. Forgy. 1982. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19, 1 (1982), 17–37.

[13] P. Fraternali and M. Brambilla. 2015. *Interaction Flow Modeling Language*. OMG. Version 1.0.

[14] M. Gardner. 1970. Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game 'life'. *Scientific American* 223, 4 (1970), 120–123.

[15] Object Management Group. 2015. *Diagram Definition, Version 1.1.*

[16] E. Guerra and J. de Lara. 2004. Event-Driven Grammars: Towards the Integration of Meta-modelling and Graph Transformation. In *International Conference on Graph Transformation (LNCS)*, Vol. 3256. Springer, 54–69.

[17] D. Harel. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8, 3 (jun 1987), 231–274.

[18] D. Harel and E. Gery. 1997. Executable Object Modeling with Statecharts. *Computer* 30, 7 (jul 1997), 31–42.

[19] D. Harel and A. Naamad. 1996. The STATEMATE semantics of statecharts. *Transactions on Software Engineering and Methodology* 5, 4 (oct 1996), 293–333.

[20] I. Horrocks. 1999. *Constructing the User Interface with Statecharts.* Addison-Wesley.

[21] Itemis. 2019. Generating state machine code. https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/codegen_generating_state_machine_code.

[22] A. Karatkevich. 2003. Deadlock Analysis in Statecharts. In *Forum on Specification and Design Languages*. 414–425.

[23] S. Kelly, K. Lyytinen, and M. Rossi. 1996. MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In *Conference on Advanced Information Systems Engineering (LNCS)*, Vol. 1080. Springer, 1–21.

[24] D. Kolovos, R. Paige, and F. Polack. 2006. The Epsilon Object Language (EOL). In *Model Driven Architecture – Foundations and Applications*. Springer, 128–142.

[25] D. Kolovos, R. Paige, and F. Polack. 2008. The Epsilon Transformation Language. In *Theory and Practice of Model Transformations*. Springer, 46–60.

[26] R. Mannadiar and H. Vangheluwe. 2011. Debugging in Domain-Specific Modelling. In *Software Language Engineering*. Springer, 276–285.

[27] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. 1998. Implementing statecharts in PROMELA/SPIN. In *Workshop on Industrial Strength Formal Specification Techniques*. IEEE, 90–101.

[28] Nomagic. 2019. Cameo Enterprise Architecture. https://www.nomagic.com/products/cameo-enterprise-architecture.

[29] F. Paternò. 2013. End user development: Survey of an emerging field for empowering people. *ISRN Software Engineering* 2013 (2013).

[30] L. Rose, R. Paige, D. Kolovos, and F. Polack. 2008. The Epsilon Generation Language. In *Model Driven Architecture – Foundations and Applications*. Springer, 1–16.

[31] J. M. Rouley, J. Orbeck, and E. Syriani. 2014. Usability and Suitability Survey of Features in Visual IDEs for Non-Programmers. In *Evaluation and Usability of Programming Languages and Tools*. ACM, 31–42.

[32] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. 2014. Declarative Interaction Design for Data Visualization. In *Symposium on User Interface Software and Technology*. ACM, 669–678.

[33] C. Scholliers, L. Hoste, B. Signer, and W. De Meuter. 2011. Midas: A Declarative Multi-touch Interaction Framework. In *International Conference on Tangible, Embedded, and Embodied Interaction*. ACM, 49–56.

[34] V. Sousa. 2017. Adapting Modeling Environments to Domain Specific Interactions. In *Symposium on Engineering Interactive Computing Systems (EICS '17)*. ACM, 145–148.

[35] V. Sousa and E. Syriani. 2015. An Expeditious Approach to Modeling IDE Interaction Design. In *Workshop on Multi-Paradigm Modeling*, Vol. 1511. CEUR-WS.org, 52–61.

[36] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. 2008. *EMF: Eclipse Modeling Framework* (2nd ed.). Addison Wesley Professional.

[37] E. Syriani, L. Lúcio, and V. Sousa. 2019. Structure and behavior preserving Statecharts refinements. *Science of Computer Programming* 170 (2019), 45–79.

[38] E. Syriani and H. Vangheluwe. 2013. A Modular Timed Graph Transformation Language for Simulation-based Design. *Software & Systems Modeling* 12, 2 (2013), 387–414.

[39] E. Syriani, H. Vangheluwe, and B. LaShomb. 2015. T-Core: A Framework for Custom-built Transformation Languages. *Journal on Software and Systems Modeling* 14, 3 (2015), 1215–1243.

[40] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. 2013. AToMPM: A Web-based Modeling Environment. In *Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition (MODELS'13)*, Vol. 1115. CEUR-WS.org, 21–25.

[41] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman. 2014. Practical Trigger-action Programming in the Smart Home. In *SIGCHI Conference on Human Factors in Computing Systems*. ACM, 803–812.

[42] B. Ur, M. Pak Yong Ho, S. Brawner, J. Lee, S. Mennicken, N. Picard, D. Schulze, and M. L. Littman. 2016. Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes. In *SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3227–3231.

[43] S. Van Mierlo, Y. Van Tendeloo, B. Meyers, J. Exelmans, and H. Vangheluwe. 2016. SCCD: SCXML Extended with Class Diagrams. In *Workshop on Engineering Interactive Systems with SCXML, part of EICS 2016*.

[44] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi. 2016. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software and System Modeling* 15, 3 (2016), 609–629.