Université de Montréal

Adapting Modeling Environments to Domain Specific Interactions

par

Vasco Sousa

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en informatique

décembre 2020

Université de Montréal

Faculté des arts et des sciences

Cette thèse intitulée

Adapting Modeling Environments to Domain Specific Interactions

présentée par

Vasco Sousa

a été évaluée par un jury composé des personnes suivantes :

Philippe Langlais, Ph.D.
(président-rapporteur)

Eugene Syriani, Ph.D.
(directeur de recherche)

Houari Sahraoui, Ph.D.
(membre du jury)

Richard Paige, Ph.D.
(examinateur externe)

Pierre Legendre, Ph.D.
(représentant du doyen de la FESP)

Résumé

Les outils logiciels sont utilisés par des experts dans une variété de domaines. Il existe de nombreux environnements de modélisation logicielle adaptés á une expertise spécifique. Cependant, il n'existe pas d'approche cohérente pour synthétiser génériquement une ligne de produits de tels environnements de modélisation qui prennent également en compte l'interaction et l'expérience utilisateur adaptées au domaine. L'objectif de ma thése est la proposition d'une solution pour modéliser explicitement les interfaces utilisateur et l'interaction des environnements de modélisation afin qu'ils puissent étre adaptés aux habitudes et aux préférences des experts du domaine. Nous étendons les techniques d'ingénierie actuelles pilotées par un modéle qui synthétisent des environnements de modélisation graphique pour prendre également en compte les modéles d'interaction. La sémantique formelle de notre cadre linguistique est basée sur des statecharts. Nous définissons un processus de développement pour générer de tels environnements de modélisation afin de maximiser la réutilisation á travers une nouveau technique de raffinement de statecharts.

Mots clés : Langages Dédiés au Domaine, Interfaces Homme-Machine, Environnements de Modélisation, Raffinement des Machines d'État.

Abstract

Software tools are being used by experts in a variety of domains. There are numerous software modeling environments tailored to a specific domain expertise. However, there is no consistent approach to generically synthesize a product line of such modeling environments that also take into account the user interaction and experience adapted to the domain. The focus of my thesis is the proposal of a solution to explicitly model user interfaces and interaction of modeling environments so that they can be tailored to the habits and preferences of domain experts. We extend current model-driven engineering techniques that synthesize graphical modeling environments to also take interaction models into account. The formal semantics of our language framework is based on statecharts. We define a development process for generating such modeling environments to maximize reuse through a novel statechart refinement technique.

Keywords: Domain Specific Modeling Languages, Human Computer Interaction, Modeling Environments, Statechart Refinement.

Contents

List of Tables

List of Figures

Liste des sigles et des abréviations

API	Application Programming Interface
CTT	Concur Task Tree
DSML	Domain Specific Modeling Language
ECA	Event Condition Action
EMF	Eclipse Modeling Framework
EGL	Epsilon Generator Language
EOL	Epsilon Object Language
ETL	Epsilon Transformation Language
GMF	Graphical Modeling Framework
GoL	Game of Life
GUI	Graphical User Interface

HCI	Human Computer Interaction
IDE	Integrated development environment
IFML	Interaction Flow Management Language
MDE	Model Driven Engineering
OCL	Object Constraint Language
OMG	Object Management Group
OO	Object Oriented
UML	Unified Modeling Language
UX	User Experience

Remerciements

In memoriam: Francelina Garcia, Américo da Silva, John Conway and Cirilla.

There are people I am thankful for, and in particular that I am thankful for, in my path to a Ph.D. Vasco Amaral and Bruno Barroca, for showing me this field of study and making me feel I had something to contribute. Levi Lucio, for being a friend whose path has crossed mine quite a few times, and will probably cross a few more. Pierre Kelsen, for the show of faith, and almost being my supervisor if I hadn't one already when the opportunity came. Marwane El Kharbili, Moussa Amrani, Denis Shirnin and Lüic Gammaitoni for convincing me I could do a Ph.D. Eugene Syriani, for the faith in me and all the hammering through these years. Khady Fall, because minions are people too. Chihab Mokaddem, Thomas Schweizer, Erick Raelijohn, Lechanceux Luhunu, Miguel Gamboa, Navpreet Kaur, Constantin Masson, for all the times they were colleagues. Marie Cattiaut, Ambra Aielli, Quentin Roy Massoteau, Martin Verciel, Laurène Rosenstein, Berangere Brais, Edwin Alibert, Maëlle Brogard, Branavan Inthiranathan, Michael Clarke for the times you made Montreal a welcoming place. Eduard Batot, for being an uplifting voice of reason. Remus Dobrican, Barbara Carvalho and João Pires, for keeping me alive. Céline Begin, because some people are indispensable.

Ana, Helder, Maria Isabel and Caetano, because I love them very much.

Chapter 1

Introduction

"Begin at the beginning", the King said gravely, "and go on till you come to the end: then stop".

Lewis Carroll, Alice in Wonderland (1866)

Intuitively, we all understand the connection between the usefulness of a tool and its use; almost to the point of platitude. How we use things is almost indissociable from why and for what. A hammer, no matter how good it is at pushing nails, has to fit the hand of the user. Well, it doesn't, a rock can also nail things — if one can wield it, albeit not as comfortably. But given the choice, the hammer will always be given a chance, and only if it fails at its job, will we turn to the rock.

The same applies to software. A user will often prefer a tool that does the job, but they are more comfortable using, than a tool that is excellent at its job, but is unwieldy to use. Of course, in the case of software, it is more complicated than if it fits the hand, as there are learned habits and cultural norms. Ctrl-v has no innate semantics, but V exists next to C in most keyboard layouts, and Ctrl-c is a good shortcut for copy. So Ctrl-v not Ctrl-p is the shortcut for paste. This illustrates the balance of cultural norms, as the choice of C is an English-centric one. Ergonomic choices, as V is right next to C making it convenient for the repetitive task of copy and paste. And learned habits, as anyone that traverses from or to Mac may attest.

There are of course exceptions, power users *who take results over ease of use, but even those are driven by the same principles if not the same rules.*

In a looser way, software modeling also follows these principles when addressing a problem, by way of using modeling languages that fit the problem and adhere to user concepts and expectations. So it is amazing that so few general-purpose modeling tools make use of these principles in their User Experience (UX) . Most of the time these tools are not user-friendly, as any student who makes their first contact with them can attest. Some may say that it is a matter of habit, you just have to learn the tools, but that falls in adapting the user to the tool and not the tool to the user.

1.1. Context

Software modeling, refers to the use of software to model a solution for a problem in a specific domain (*e.g.,* music, finance, biology). As such, there is a plethora of software tools that enable domain experts (*e.g.,* musicians, economists, biologists) to represent, manipulate, and simulate models using notations from the domain, specified at a suitable level of abstraction (*e.g.,* a music sheet, ...). In the programming domain, such tools are often called Integrated Development Environments (IDE) , like for example Eclipse [1].

Model Driven Engineering (MDE) [**44, 107**] is a SoftwareEngineering (RHS) paradigm and associated technologies that apply lessons learned from earlier efforts at developing higher-level platform and language abstractions. It has proved to be able to do so through the generation of IDEs in a variety of domains [**68, 113**]. A main aspect of MDE is characterized by creating a Domain-specific Modeling Language (DSML) that defines an abstraction of the domain, a modeling environment to produce models based on the DSML, and tools that can further manipulate these models. The creation of a DSML has traditionally focused on two aspects, the domain's concepts and structure, and how they are represented. For instance, if we consider modeling a music sheet, the notion of note and tempo are concepts of the domain and they are represented in the form of ♩ and ¢ respectively.

The use of a modeling environment provides the means to create and manipulate these domain-specific models. This helps in making sure they conform to their language, while providing feedback on the modeling process. For a music sheet model, this results in only allowing musical symbols and making sure they are correctly placed on the music sheet. Having the concepts and graphical elements fit the expectations of the users, helps to reduce their cognitive effort and promotes the adoption and use of these models [**107**].

This approach of creating languages, with the specific purpose of helping reach the best level of abstraction to address a problem, caters to the notion that language has an influence in shaping our thought process. This is a debated theory that comes from natural languages but is shown to be present in artificial languages, such as programming languages [**64**].

1. `www.eclipse.org`

DSMLs are a part of such artificial languages. Not only does the graphical elements of a language, but in general, the user interface also has an influence on the strategies of the thought process [**20**]. This is similar to the influence of the graphical elements of the language have on clarity and familiarity to reduce the cognitive effort of the user. We would argue that User Experience (UX) as the set of actions, interfaces, and feedback that characterizes the interactions of a user with the software is also part of this effort [**27**]. Having known elements (*e.g.,* the close window button) in a known location (*e.g.,* the top right of the window) with a known representation (*e.g.,* a rectangle with an $\times$) helps to reduce the cognitive effort of finding it. In the same way, having a known interaction (*e.g.,* using a mouse wheel to scroll through a document) reduces the effort of searching for the scroll bar and allows the user to stay focused on the document itself.

In this thesis, we claim that, alongside DSMLs aim to help to tackle problems at a more focused level of abstraction by reducing accidental complexity, the user interface and interaction of MDE tools used to manipulate these models are also crucial in reducing accidental complexity, and thus improving user productivity.

1.2. Problematic

One would expect that user interfaces and interaction be explicitly part of the development of modeling environments. For instance, works such as [**17, 106**] strive to expand user interactions beyond that of traditional means, in ways to tackle UX issues and grasp other requirements and ways of expressing user interactions. Still, very little effort has been done in tackling how to describe interaction and its influence on the productivity of modeling environments, as discussed during the panel session of the MoDELS 2016 conference [**9**]. The field of Human Computer Interaction (HCI) [**57**] studies how to improve interactions, however we do not have an effective and efficient way of describing such interactions in order to integrate them in the development of modeling environments.

Current frameworks (such as the Eclipse Modeling Framework (EMF) [**114**]) produce DSML environments from the specification of domain concepts and notations, and then reuse their own user interface and interactions. However, what happens when we aim at adjusting the thought process and task performance for a particular problem? The most direct approach is to provide such tools to experts on the problem domain. The problem with this approach is that the domain expert's expectations on how to use such editors might not align with the actual predefined interaction of the editor [**103**]. This may stem from either pre-established procedures on how to approach the problem or simply an expectation of using any type of interaction available. For example, the alignment of the notes in a staff is crucial in a modeling environment for music sheets. In current practices, all the modeler can do is specify what notes are and how to represent them, not how a user can interact with music sheets as musicians expect to. The interaction that we are left with remains the same as the framework that was used to produce the modeling environment. For the most part, this framework interaction that is reused in the generated DSML editors, is a drag and drop of the language elements, one at a time. This disconnect between expectations and the interaction that is provided, is what we are trying to avoid, as it goes against the purpose of using MDE tools. This is also one of the main practical reasons why domain experts continue to use IDEs dedicated to their domains (*e.g.,* MAX for music modeling [**7**]), instead of IDEs generated from generic MDE frameworks.

It should be possible to produce modeling editors in an automated way that permits the customization of languages to their use, but also to provide a better mapping between interactions and expectations. This will alleviate the cognitive effort and focus on improving the user's thought process, not only at the level of the language itself, but also at the level of how the tool is used, making the use of this type of tool more intuitive, efficient, and flexible. It is possible to create modeling environments that provide rich interactions, such as the simulation of interacting with a virtual book in virtual reality [**36**]. Still, the gap between how such interactions are defined and how graphical MDE tools are developed does not permit a proper adaptation of interactions. It becomes clear that the interaction with modeling environments needs to be adapted in new ways. Some modeling environments counter this absence of custom interaction by having these adjustments to the user interface and interaction manually hard-coded into the editor. But this solution is time-consuming, focuses on how to implement interaction rather than how to use interaction, and rapidly becomes inflexible leading to large maintenance efforts.

Furthermore, with the popularity of new computational devices and peripherals at the expense of traditional desktop environments, such as tablets with varying sizes, virtual and augmented reality goggles, and collaborative interactive tables, new human user interactions will keep on appearing. If no effort is put forward, the gap between the interaction with the modeling environments and the user expectations will only increase.

1.3. Contribution

The main goal of this thesis is to explicitly include the description of user interaction in the specification of MDE tools , so that they can be customized to the user needs, habits, and application domain. This inclusion should be integrated in the existing methodologies of creating MDE tools, which are themselves developed using MDE techniques, as a way to promote flexibility of development and facilitate their maintenance. Many users express preferences for tools that provide more functionalities than what they actually use [**38**]. This can be interpreted at the level of functionalities of the tools and at the level of interaction. An example of the latter is accessing functionalities through menus that are also available through keyboard shortcuts, a feature usually used by more experienced users. This tells us that we can, for the moment, concentrate on providing a way of describing interactions in MDE tools as a form of diversifying the user interaction, rather than search for the most efficient interaction. In this thesis, we focus on the user interface and interaction of graphical DSML editors.

In this prespective, and because we want to integrate into the established process of automaticaly generating DSML editors, we focus on the description of the interaction themselves, not in the overall tasks as those, for the most part, remain the same as the existing process. This also means that as a first hurdle, we aim at providing better and more varied interaction integration on the predicted tasks. So we perform a user study, not to compare it as a generic approach to interaction description, but instead to the flexibility of interaction integration of the existing DSML editors generation process.

The contributions of the thesis are:

(1) An explicit modeling language to describe interactions in graphical DSML editors.

(2) The automatic synthesis of the graphical DSML editors from the interaction models.

(3) A development process to maximize the reuse of existing models of interaction and integrate them with new ones.

(4) A formalized semantic anchoring of any produced interaction controller, with by construction guaranties.

1.4. Outline

In Chapter 2, we present the background knowledge on MDE and HCI, as well as related work on describing user interaction. In Chapter 3, we describe our proposal to solve the presented problem. In Chapter 4, we formalize a Statechart specific form of refinement, that we use as the operational semantic domain of our approach. In Chapter 5, we detail the operationalization and implementation of our approach. In Chapter 6, we verify the implementation and evaluate its performance. In Chapter 7, we validate our approach thanks to an empirical user study. Finally, we conclude in Chapter 8.

Chapter 2

State of the Art

We have learned nothing in twelve thousand years.

Pablo Picasso, upon exiting the Lascaux cave, France (1940)

Let us begin by looking into some background knowledge to the topics of HCI and MDE to help us better understand the presented work, alongside some comparisons with existing approaches from which lessons can be taken.

2.1. Background on Model Driven Engineering

MDE is the frame of development of our work and, as such, there must be some understanding of its inner workings, its use and the work done in this field that influences our research.

2.1.1. Domain Specific Modeling

Based on the description in [**68**], DSMLs help us specify problems and solutions within an application domain through a defined structure, terminology, behavior, requirements and notation pertinent to that domain. These domains of application are not limited to computer science and can range from the automotive industry to biology and art. The purpose of this approach is to isolate a domain problem and its possible solution from computational and implementation details and, instead, express the solution within the application domain, using models at a higher level of abstraction than code, to facilitate its use and validation by experts of that domain.

There are four main categories of activities that can be performed with or on models, Model Editing, Model Transformation, Code Generation and Model Analysis. Because it is the activity that requires the most active participation from the user, in the most immediate context we are only focused on improving the former with better user interactions.

2.1.1.1. Domain Specific Modeling Language Engineering

The process of engineering a DSML starts with the analysis of the targeted domain, and collecting the concepts that populate that domain at the targeted level of abstraction. In a more concrete example, this can be individual notes and their modes and variations, if we are modeling a music sheet, or we can have a higher level of abstraction, and use the concepts of instruments and their complete performance to model the sound mixing of a music recording. This shows us that even within the same domain the concepts that we need to encode may vary depending on the level of abstraction.

Once we establish the concepts needed, based off the domain and the level of abstraction required, we need to encode these concepts, how they relate and what variable attributes they may have. To do this we use a metamodel [**75**]. The most common approach to define a metamodel of a DSML is through the use of languages heavily based if not actual implementations of Unified Modeling Language (UML) Class Diagram [**22**]. This allows the creation of the DSML concepts based on existing Object Oriented requirements and analysis techniques. Classes represent conceptual entities of the domain, attributes their characteristics, and associations relate entities together. The static semantics of the language can be further detailed with Object Constraint Language (OCL) well-formedness invariant rules [**130**]. This is referred to in MDE as the Abstract Syntax of the language, and with it the modeling environments can restrain the models to conform to the DSML.

> *This is where Object Oriented engineering and DSML engineering start to diverge. Because the interaction with the DSML is traditionally derived from the Metamodel, its Class Diagram representation has to be adjusted to guide the interaction of the modeling process. This means that instead of a pure representation of the concepts as it would be the case in Object Oriented development, it represents instead how we want the concepts to behave. This also undermines the reliance on OCL, as it requires an existing instance to be evaluated, instead of preemptively enforcing the conformity to the DSML.*
>
> *Again to give an example, if we consider the notes of a music sheet, we can encode different types of notes as different classes that inherit from a super class note, or we can encode the different types as an attribute of the class note. The choice between the two, instead of being based purely on the analysis of the domain, it is also informed by how we want the note to behave during the editing of the model.*

We can somewhat overcome this, by creating two Metamodels, one defining a representation closer to the concepts as they ere defined in their domain, and better adjusted for analysis and other manipulations, and another Metamodel, defining how we want the language to behave. Because the domain concepts are represented in both metamodels, there is an easy one to one mapping between the two. This is actually what we do in parts of our implementation. But this approach still requires the maintenance of the two Metamodels to keep them synchronized, creating limitations in terms of scalability and long term maintenance.

This gives us how a model is structured but not how it is represented to a user. Typically, this is a textual or a diagram representation, but it can also be a combination of the two, or even other forms of information representation such as tabular. In this thesis, we focus only on graphical representations.

For that we need a Concrete Syntax for graphical representations. That is, a library of symbols and representations of the domain concepts, and a mapping of those symbols to the concepts defined in the Abstract Syntax. This mapping, is usually a conditional mapping that allows us to map not only a visual representation to each class, but also allows multiple representation depending on their attributes.

At this point, we have a definition of how a model is structured by the Abstract Syntax and how it is represented by the Concrete Syntax, we then need to specify what the model and its elements mean. We give a meaning to the models by defining a precise semantics of the DSML [**56**]. Traditionally, we achieve this in one of two ways. We can define how a configuration of the model evolves into a new configuration based on its semantics, and we call this Operational Semantics. Or we can define how a configuration of the model would be represented in a new language with a well known semantics, and we cal this Denotational or Translational Semantics. Both of these approaches, albeit in slightly different ways in their concrete application, are achieved through the use of Model transformation.

2.1.1.2. Model Transformation

Model transformation is a central part of MDE. It defines an automated and systematic manipulation of an input model to produce an output model [**80**]. This manipulation is done with a specific intent, such as defining the operational semantics of a DSML through simulation [**117**], or to translate one DSML into a language with well-defined semantics [**79**].

Transformation rules are the smallest units of a model transformation. A transformation rule has many different features according to [**34**]. The domain of a rule defines how a rule can access elements of models. In most languages dedicated to Model transformation, rules are a declarative construct that dictates what shall be transformed and not how. An advantage of using the rule-based transformation paradigm is that it allows to specify the transformation as a set of operational rewriting rules instead of using imperative programming languages. These rules, consist of pre-condition and post-condition patterns.

The pre-condition pattern determines the applicability of a rule through a pattern that must be found in the input model. This pattern is a set of class types and optionally the values their attributes and relations have that we try to find in the model. Some Model transformation languages also allow the specification of a negative patter, that is a set of elements and values that when found, it inhibits the application of the rule.

The post-condition specifies a pattern that must be found in the output model after the rule is applied, usually this pattern is achieved by creating new instances of classes or relations. Some declarative Model transformation languages also allow imperative actions to be specified over the attributes to create more complex patterns or values.

Code generation [**68**] is the process of producing executable code from the information specified in a model. This process is a derivation of model transformation and in particular model-to-text transformations. This transformation process can be achieved through the use of different tools and techniques [**102**]. One of these techniques is template based code generation. Where templates are assigned to the different element types of the language and filled out with the instance specific transformation. With these techniques it is possible to fully generate executable code without any further intervention.

2.1.2. Graphical Modeling

One way of representing DSMLs is with diagrammatic notations. As stated in [**126, 88**], some of the reasons to use this type of representation over a textual notation are, the capacity of spacial representation of concepts allowing a modeler to place conceptually close notions in specific arrangements (for example, in a service diagram all elements of a sub-service are placed in proximity on the diagram), the direct representation of interaction between elements (for example, by connecting lines we can immediately observe common ancestors in a family tree), with a rich set of graphic elements it is possible to easily differentiate them making it easier to identify (for example, the opposing kings in a representation of a chess board).

There are three main categories of diagrammatic modeling editors: syntax-directed, free-hand and projectional. Syntax directed modeling editors (*e.g.,* AToMPM [**118**]) are encoded with domain information and allow the instantiation of pre-existing concepts from the application domain, following specific modeling patterns that will make sense in the application domain. Free hand editors (*e.g.,* Flexisketch [**131**]), allow for freeform modeling, that will afterwards be interpreted. Projectional editors (*e.g.,* MPS [**128**]) of which there are no graphical editors, present a partial model that represents a defined pattern within the application domain, to be filled out with the missing information by the user. In the course of this thesis we are focusing on syntax directed modeling editors.

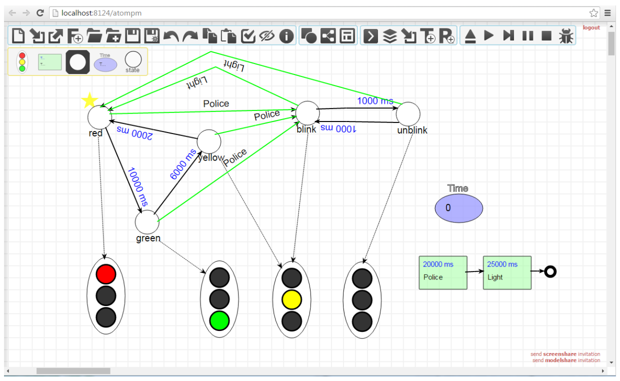
To use such syntax directed modeling editors we require a library of graphical elements that adhere to any pre-existing notation within the application domain and a mapping between the concepts expressed in the DSML and these graphical elements.

2.1.3. Current Anatomy of a Domain Specific Modeling Language Editor

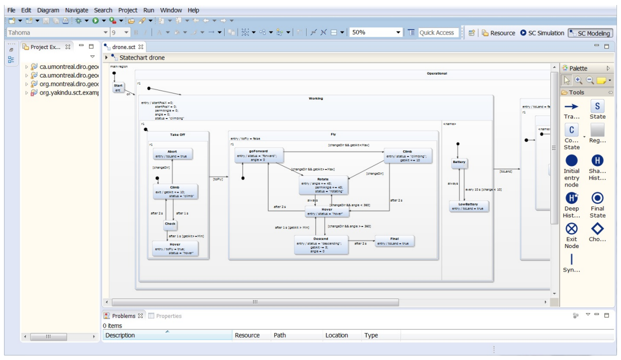
Current DSMLs development is usually done in language workbenches, that provide an automated generation of the modeling environment, as well as other features such as model exploration, debugging, and other common IDE features [**41**], as well as model specific features such as Model transformation. The most prevalent workbench platform used to produce modeling environments is Eclipse with its EMF [**114**]. In such workbenches, the language development process is often bootstrapped, relying on DSMLs and other auxiliary tools to define the metamodel and a detailed specification of a Concrete Syntax for that language; be it textual with tools such as EMFText [**1**] or graphical with the use of Eclipse's Graphical Modeling Framework (GMF) [**4**], either directly or with tools such as EuGENia [**74**], Graphiti [**2**] or Sirius [**11**]. Other notable tools used to produce modeling environments are: MetaEdit+ [**67**], one of the first commercial tools to allow the creation of such editors; AToMPM [**118**] a framework, that has introduced some innovations in the field, such as the direct use of domain specific Concrete Syntax in model transformation, and the deployment of the modeling environment on the cloud.

These frameworks make use of code generation, that relies on a metamodel and a specification of the concrete syntax. This process generates all the required artifacts to have a functional editor in the base framework. In the case of GMF, from the metamodel definition of a DSML, it will produce a set of Eclipse plugins with the Java code and configuration files that allow us to edit the model of that DSML and display it as a diagram. In the case of AToMPM, it will generate Javascript and JSON files with the operations of that particular language that can be loaded in the editor. Note that, Eclipse's GMF technically uses a *tool model*, in the editor generation process, but its use is limited to configuring the Concrete Syntax language toolbar. So in the end, although they provide mechanisms to tailor the Abstract Syntax and Concrete Syntax to a domain, all of these frameworks reuse the user interface and interactions of the language workbench used for designing a DSML, when creating an environment for manipulating models in this DSML. This effectively assumes that the interface and interactions used to create DSMLs are well fitted for any other domain.

Knowing that the process of generating these editors is done by code generation, it means that this generated code could be treated as normal code and be further customized. This is what was done in the first generation of the Business Process Modeling Notation (BPMN) editor for Eclipse, a well-polished editor that only relies minimally on the generated code. This is not a common practice, because the process of manually editing generated code relies on a deep understanding of the generated code and the platform it relies on. Any update to the DSML also puts the manual changes to the editor at risk. This may make it more cost-effective to build an editor from scratch, although still not a trivial task, and any change to the DSML comes with higher code maintenance costs attached. Some of these tools do provide mitigation of these factors with the use of extension points or annotations to protect the manual code and still allow regeneration, but the level of customization still has some limitations or still requires extensive manual coding. This is the case in Eclipse, where instead of relying on the GMF figure primitives we can declare the use of custom object encapsulated in a custom plugin, and this can keep all this custom code safe from any code regeneration. But this only provides customization at the level of the language element Concrete Syntax level, and for instance in the case of the aforementioned BPMN editor the amount of customization is such that it makes it impossible to regenerate the code derived from the metamodel without breaking the customized code.



(a) AToMPMuser interface



(b) GMF user interface

Figure 2.1. Graphical modeling editor user interface examples

In terms of usage and interaction within these editors, the landscape is similar in all of these environments, as we show with the examples in Figure 2.1. They all have a canvas space that occupies most of the screen space of the editor. In it, the user can freely place the diagram elements as long as they still satisfy the abstract syntax constraints of the language. To select what language elements to place on the canvas, a toolbar is available with all the elements available from the Concrete Syntax. In the Eclipse's GMF example, these are located on the bar to the right of the editor as shown in Figure 1(b), and in the AToMPM example they are located horizontally as the second bar at the top of the editor as shown in Figure 1(a). The usage of such toolbars to allow the instantiation of Concrete Syntax elements usually falls into one of two categories. One possibility — and the most common — is the user selects the element type to instantiate in the toolbar and then click anywhere on the canvas to instantiate an object of that type at that location. At this point some frameworks allow for multiple instances (*e.g.,* AToMPM), other reset the element selection and require it to be re-selected (*e.g.,* Eclipse, although it is possible to create multiple instances by pressing `Ctrl` during these steps). The other possibility is, the user clicks on the toolbar element and drags it to the desired position on the canvas. Additional activities such as Open, Save, Copy, Paste, etc. are made available in additional tool or menu bars, and are a reuse of the environment used to develop the editor. In both examples in Figure 2.1, these are positioned as toolbars at the top of the editor.

The drawback of this approach is that it assumes that the process, interactions and overall UX of creating a DSML is adequate for designing models of any domain. As more heterogeneous computation devices become more prevalent, such as smartphones, tablets with or without pen input, Virtual and Augmented Reality, the universality of the mouse and keyboard interaction is put into question. So if we want to adapt to the needs of specific domains, be it the preferences of its domain users, mobility, or a broader user accessibility, we need to expand how we can interact with DSML editors. One theoretical way to achieve this would be to maintain the current process but expand the universality of the interactions the present language workbenches provide. But because universality in not a realistic goal in user interaction, we aim to tackle this by making the generated DSML editors adapt to different interactions on a case by case basis. As we have seen, to do that, the specification of the Abstract Syntax and the Concrete Syntax is not enough. We need to add additional information about what interactions can be performed and their effects and how the elements that can be interacted with are displayed and organized. For this we look at how User Interaction is handled outside the presented use case, and suggest a possible approach to tackle this issue.

2.2. Background on Human Computer Interaction

HCI is a field of study that explores the design, evaluation, and implementation of interactive computation systems [**57**]. The most common representation of this interaction is a user engaging with a computer through means of a screen, mouse, and keyboard. But this is a reductive vision of the possibilities of interactive engagement and computation systems. HCI is then not limited to the classical implementations of interaction and takes knowledge from a multitude of fields such as Computer Science, Psychology (cognitive theories and human behavior), Sociology (cultural and social impact and influence in interactions), Anthropology (interaction with technology) and Industrial Design (layout, representation techniques, and ergonomics).

2.2.1. Evaluation in Human Computer Interaction

In HCI, there are very few factors that can be objectively evaluated without any user study, such as measuring that a piece of displayed text fits all displays it will be presented on. Most factors of user interaction are more nuanced and would require and absolute knowledge of human biology and psychology, to be able to encompass all the variability of users. Early approaches modeled the user, but these reduce the human factor to a series of computation actions that do not represent users correctly [**127**]. For this reason most of the HCI validation and studies comes in the form of empirical studies with targeted user groups. These can be divided into two types of studies; qualitative and quantitative studies.

Qualitative studies focus on qualifying the user interaction, such as how intuitive an interaction may be, if the interface is appreciated [**70**]. These studies are preformed with use of inquiries, questionnaires or interviews, following the guidelines in [**59**], where the test group provides direct feedback about the factors being evaluated. This type of study relies on subjective data, such as individual opinions and habits, making it harder to generalize without very large user studies. Nevertheless, it is still invaluable, specially if the focus group is specialized, such as users from a very specific domain. This is the case, because the evaluation elements will rely less on generalizing its results, and more on evaluating if the proposed solution is appropriate to the users classifiable by their domain stereotype.

Quantitative studies focus on measuring factors dependent on the user interaction, such as reaction and completion times, mechanical efforts, or eye movement [**46**]. These studies are preformed with the use of controlled experiments, to make the results reproducible and isolate factors external to the interaction itself. This type of study relies on statistical analysis of numerical data, such as times, distances, number of occurrences, and requires more participants or experimental runs, to establish statistical significance. The drawback is multiple run experiments may introduce habituation bias, and they require more participants and time to execute.

In the context presented work, both a quantitative and a qualitative study where preformed and are included in this thesis.

2.3. Methods and Languages for Specification of User Interfaces and Interaction

The field of HCI is rich with a lot of work on UX which aims at improving the interfaces, actions, and feedback of the interactions between a user and the software [**26**]. Typically, Graphical User Interface (GUI) design of these editors relies on sketching [**28**]. This is an approach where the designer creates GUI mockups deciding on the layout of components, their relation, and the integration of data. These methods can be applied or provide a conceptual basis when designing the interface model in our approach.

2.3.1. Sketching

A common approach to user interface design and specification, is sketching. In [**28**] it is pointed as the initial phase of design. With it, a designer creates a preliminary mockup of the user interface, exploring where the different elements go, how they relate, and where data is displayed. This approach uses knowledge from classical layout design, namely editorial layout, in how to use two-dimensional space. Each mockup is static and the interaction components are informally specified as the transition between different mockups.

These mockups can also be referred to as wireframes. Several tools exist to build wireframes such as wireframe.cc [**13**], Adobe Experience Design CC [**5**], UXPin [**12**], Fluid UI [**6**], Pidoco [**10**], Microsoft Visio [**8**]. They facilitate the process by providing element reuse from mockup to mockup and allowing a simple representation of the transitions between mockups to indicate points of interaction. In its essence it is a brainstorming tool to organize and explore the user interface, and, as such, has great flexibility. Still, it is usually limited to screen interfaces, providing no clear specification of more complex systems, or departure of traditional devices.

An additional shortcoming of this approach is that the mockup is not as closely related to the final product as one might think [**28**]. These mockups are also very specific, to their application and implementation. They mainly focus on well-established uses, such as webpages and mobile applications, that have very specific design parameters and a limited set of interactions. It is specific to their implementation because different devices require different mockups.

2.3.2. Flowchart

Flowcharts [**62, 89, 115**] are used when more elaborate interactions are required when designing a mockup. This can be done with the use flowchart editors to help layout the transitions between mockups in a richer form. Sill this approach does not specify what the interaction itself entails. This is also similar to the use of flat Statecharts in specifying interaction, where each state encompasses the full layout [**40**]. As such, the addition of interaction and complexity of the system creates a statespace explosion that makes the flowchart hard to maintain if no measures are taken to manage the increase in scale.

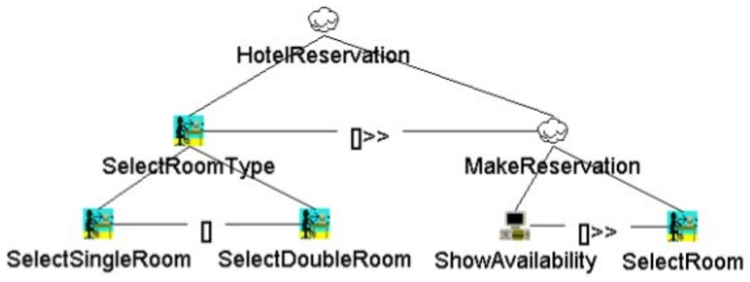


Figure 2.2. Example of a CTT [**3**]

2.3.3. Concur Task Trees

Concur Task Trees (CTT) [**98, 95, 96**], aim to help the design process of interactive applications, by providing a relational organization of interaction tasks. As the name implies, these diagrams takes the form of a tree, as we can see in Figure 2.2. The nodes of this tree represent tasks that the user can perform on the system — such as Hotel Reservation, where nodes of a sub-tree are tasks that when performed contribute to the resolution of the task at the root of that sub-tree — such as SelectRoomType and MakeReservation that contribute to accomplish Hotel Reservation.

These diagrams also have arcs between nodes of different branches of the tree, that represent an operational relation between those nodes. For example in Figure 2.2, the SelectSigleRoom and SelectDoubleRoom tasks have a choice relation (denoted by the [] operator), as the initiation of any of these tasks disables the other. The tasks SelectRoomType and MakeReservation also have a relation between them, but this time it is an order dependency relation with information sharing (denoted by the []» operator) as to enable the MakeReservation task the SelectRoomType must first be completed, and the information produced by SelectRoomType is used by MakeReservation. Similar relations can be made to indicate concurrency, independence, enabling, disabling and suspend-resume interruption of tasks.

All of this allows for the coordination of complex sequences of tasks and the reuse of common sequences of tasks. These are interesting concepts in coordinating interaction tasks, but, they provide no description of the interactions themselves on how they relate to the interface, and have a different level of granularity that what we are aiming to use. As CTT tasks a quin to use cases, that can encompass multiple steps that can be run in parralel or interrupted by other tasks. We on the other hand focus on the interaction itself (*e.g.,* a press of a button), which cannot be interrupted, and as such it is a more atomic unit of interaction description.

The authors in [**94**] provide a simulation of CTTs with interaction templates, but the interaction information used is at the implementation level and not at the conceptual level of user intent, which we are aiming at to be able to reuse the editor description in different systems.

A further development of operationalizing CTT in [**129**] provides a translation into Statechart. The main differences between this work and the one presented in this thesis, are; the strict ordering requirements needed to produce a correct Statechart, whereas we provide an approach that is independent of the order in witch the interactions are translated into Statechart. The advantage here is that with an order independent processing of the interaction rules, the development of each interaction rule can also be independent of the remaining rules. This has the consequence of facilitating an incremental development of the interactions, while still providing preservation guaranties over the pre-existing interactions. This would not be the case if at any increment in the interaction rules the order in which they will be translated into a Statechart all rules need to be checked to set the correct ordering with the new rules. Because we focus on atomic interactions instead of tasks, we assume that all interactions are parallel choices unless otherwise stated, while in their translation of CTT this is done if the choice operator is present and relies on the processing order, leading to a greater stacking of Statechart hierarchical components. They also rely on a pre-specified partial Statecharts and templates for the different operators, and thus dealing with more heterogeneous compositions, while we reduce all interactions to a uniform pattern. This makes their approach harder to statically verify and validate before the construction of the Statechart. So ultimately we are dealing with slightly different approaches to interaction specification, although it would be possible to translate their approach into our framework.

There are also approaches to extend CTTs with normalization processes and operators [**101**]. Nevertheless, the presented solution has two major drawbacks. First, the additional operators add the possibility of solving task conflicts in multiple way, leading to ambiguities on how to correctly represent these solutions. Second, it still provides no proper representation of the interactions, only their interdependency.

2.3.4. Trigger Action Rules

Trigger Action Rules [**121, 122**] are widely used in specifying control languages aimed at users without computer science backgrounds in home automation environments. These rules take the form of a condition block with a description of the values and events that trigger the rules, and a block with the actions performed. Usually these rules are presented with a visual representation, but are also limited to a predefined set of values and events defined in the target environment.

Still, the effectiveness of this approach is debated. The rigidity present in these rules is known to either lead to a limited expression, or to lead to very large amounts of rules when more complex automation patterns are attempted. This leads to unwanted behaviors and difficulty to maintain or do small adjustments to the system.

This has some similarities with our approach, and while trying to take in the accessibility benefits of this type of approach, we also define a possible approach to mitigate its drawbacks.

These also share many similarities with Event Condition Action (ECA) rules [**39**], where the main difference of with Trigger-action Rules is that the second is a term mostly used in HCI and the first is mostly used in Programming Languages and Model Transformation. A study in the Internet of Things domain [**37**] shows that ECA rules are well-suited for the expertise of end-users. In [**31**], the authors combined MDE techniques with WebML and ECA rules to generate web applications. But the UX of web applications doesn't necessarily translate to DSML editors. Many other tools allow UX designers to configure interaction rules [**29**]. The paradigm of the interaction rules we propose is inspired by ECA rules [**39**] and Trigger-action rules [**121, 122**] which allow users to specify interactions visually using domain-specific concepts.

2.4. Model-based specification of interactions

The MDE community also provides some forms of UX sub-aspect specification.

2.4.1. Interaction Flow Management Language

The Interaction Flow Management Language (IFML) [**24**] by the Object Management Group (OMG) was born from works on modeling web applications [**32**]. Although UX was not intended to be within the scope of IFML and its precursors, we still consider it as related work since it covers some segments of the interaction portion of UX. Its goal is to allow software developers to specify user interactions, by describing its components, state objects, references to the underlying business logic and its data, and the logic that controls the distribution and triggering of the interactions. Despite this distancing from full UX, the accumulated knowledge present in IFML is still valuable to our research.

Approaches like IFML 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 DSML and the interface model. Like IFML, when specific operations must be executed in the effect of the rule, an Application Programming Interface (API) must be available to implement the operation. As, we will discuss in our solution we hide this from the designer, as to promote end-user development, as motivated in [**97**].

One should keep in mind that, in this case the end-user is a design expert. So, the end-user of the tools developed in this thesis. Not the end-user *of the editor these tools will produce.*

There are nevertheless several aspects of IFML that do not align with our goal. IFML describes User Interfaces (UI) as constituents without any specification of visual properties or design choices. This limits the description since these properties have a large influence on the UX and should be adapted accordingly. An example of this is how different screen sizes will have different layouts, and consequently different interactions and vice versa: a large screen can have all interactive elements such as buttons displayed at the same time, a smaller screen will make use of nested elements such as menus, and a smartphone screen will have its interactive elements positioned to facilitate gesture interactions. From a task perspective all of these examples could perform the same tasks, but the details of how interactions are used to perform such tasks is tightly connected to the interfaces themselves. So a development of interaction where the context of interface is present, should be beneficial.

IFML is targeted to Software Developers instead of UX designers that could use their domain knowledge to better adapt the software to its users. It promotes the combined use with other OMG standards, such as Class Diagrams and Business Process Models, but through declaration of specific function calls that effectively bind the user interaction to a specific implementation, obfuscating most of the abstraction effort. It also relegates complex interactions to be specified by the implementation framework instead of being an explicit part of the interaction model. Finally, our evaluation of IFML also concluded that it has some limitations in terms of scalability, where the complexity of the specification increases greatly with each user interface element that is introduced, and the use of modules is focused on implementation reuse rather than development simplification.

That is not to say that other approaches such as IFML are bad, simply that we chose to explore a different take on the problem. Furthermore, our proposed solution is still compatible with many of these approaches, where they can still be used in conjunction when best suited.

2.4.2. Other modeling techniques

The work in [**105**] 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 [**51**] define graph transformations that are triggered by user actions, such as scaling language elements, which, in turn, triggers GUI events.

In [**69**], the authors define a DSML on akin to the use of Flowcharts merged with a lighter version of Sketching to provide a specialized abstraction that brings the concepts used in

Flowcharts closer to an HCI perspective. However, this approach has its main focus on tasks, their workflow, and their goals, addressing limitations of other task specification approaches. We on the other hand chose to focus on the interactions themselves independently of task or goal outside the immediate effects of said interaction.

2.5. Relation to modeling editors

This of course begs the question, why not just use any number of the existing approaches? Although many of these approaches provide plenty of insight, and at a minimum we do try to be compatible with a number of them, this is a problem that spans two technical communities. The available approaches do not fill that cross community gap — that is, they do not present an abstraction level that fits both communities, or are not widely used to be known in both communities. Furthermore, none of these approaches provides a holistic approach to the specification of interaction. That is, provide a good specification of the artifacts that can be interacted with, the interactions that can occur and the effects of such actions, a well-defined semantic specification that provides guaranties about the good functioning of the environment and its evolution to increasingly meet user expectations, and also make use of model driven approaches and techniques to integrate with the existing expectation on automatically generated DSML editors.

2.6. Existing approaches to Statechart Refinement

Refinement [**52**] is the process of gradually building a model by making it more and more precise. During this process each refined model is verified to check that it satisfies the pre-existing abstract behavior. The advantage of having an abstract model as the starting point, is that important properties can be defined earlier and in a simple model. This approach is therefore less likely to contain mistakes. Refinements then introduce detail in steps which are guaranteed to preserve the important properties.

Several works in the literature have explored the concept of Statecharts refinement. Our work on refinement focuses on the preservation of pre-defined structure and reachability properties in the process of defining new details in the original model. Most of the works looked at the problem in the context of class inheritance, where a subclass can inherit from a superclass' statechart. We therefore compare how this Statecharts specialization is performed in contrast with the Statecharts refinement we propose. Table 2.1 summarizes the comparison of different refinement apporaches present in the literature with ours. Like in this work, most approaches propose specific refinement rules we briefly describe:

— Variant applied on: the Statecharts formalism variant the refinement approach is applied on in the referenced paper.

— General rules:

    — Element removal: removing states and transitions without restriction.
    — Garantee regression: presence of any mechanisms to verify the preservation of behavior.
— State refinement:
    — Add states: adding new (basic) states to a statechart without restriction or by connecting it to existing elements.
    — Convert to complex: refining a state into a more complex state, *i.e.,* Basic-to-OR, OR-to-AND and Basic-to-AND.
    — Add orthogonal: adding new orthogonal components, *i.e.,* AND-to-AND.
    — Preserve nesting: preserving the relation with a parent state when refining a state.
    — Preserve transitions: preserving all incoming and outgoing transitions when refining a state.
    — Split state: splitting a state and its components into multiple states.
    — Add actions: adding new entry and exit actions in a state.
    — Preserve actions: preservation of pre-existing entry and exit actions of a state.
    — Add variables: adding and using new variables without any restriction.
    — Refine pseudo-states: presence of rules specific to pseudo states, except their addition or removal.
— Transition refinement:
    — Add transitions: adding transitions between two arbitrary states without any restrictions.
    — Preserve source: preserving the source state when refining a transition.
    — Preserve target: preserving the target state when refining a transition.
    — Preserve trigger: preserving events that trigger an existing transition.
    — Preserve guards: preserving guards on existing transitions.
    — Preserve broadcasts: preserving broadcasting events on existing transitions.
    — Split transition: refining a transition into a path composed of multiple transitions and at least one intermediate state.

All the refinement approaches allow removing an element of the original model, except for refinements in Stateflow. [**100**] is the only one that provides static regression guaranties over the behavior. The approach in [**71**] only preserves behavioral regression on flat state machines. Furthermore, approaches based on μ-calculus, B, and as described in [**23**] require additional proofs or simulations. The remaining approaches do not integrate any such guaranties or analysis in the refinements. In the following, we provide further insight than reported in Table 2.1 on the particularities of each approach.

Table 2.1. Comparison of Statecharts refinement approaches along different rules and features

Refinement approach	[**23**]	[**110, 109**]	[**71**]	[**100**]	[**93**]	[**104**]	[**16**]
Variant applied on	Stateflow	μ-Charts	Flat state machines	Flat state machines	Flat state machines	UML-B	B Method
Element removal	N	R	Y	_	Y	_	_
Guarantee regression	R	R	Y	R	N	R	R
Add states	Y	R	Y	Y	Y	Y	Y
Convert to complex	Y	Y	n/a	n/a	n/a	Y	Y
Add orthogonal	Y	Y	n/a	n/a	n/a	Y	Y
Preserve nesting	N	N	n/a	n/a	n/a	_	Y
Preserve transitions	_	_	_	Y	_	_	_
Split state	_	_	Y	_	_	_	_
Add actions	_	_	_	Y	Y	_	_
Preserve actions	_	_	_	Y	N	_	_
Add variables	R	Y	_	Y	Y	Y	Y
Refine pseudo-states	_	_	_	_	_	_	_
Add transitions	Y	R	Y	R	Y	Y	Y
Preserve source	Y	_	_	_	_	_	_
Preserve target	_	_	_	_	_	_	_
Preserve trigger	Y	R	_	_	_	_	_
Preserve guards	Y	R	_	_	_	_	Y
Preserve broadcasts	Y	_	_	_	_	_	_
Split transition	_	_	_	_	_	R	_

Legend

Y	fully supported
N	explicitly not supported
R	restricted support under specific conditions
_	not defined
n/a	not applicable

2.6.1. Simulink/Stateflow

Simulink [**83**] uses a causal block diagram notation to model control systems. It provides blocks, which are similar to states in Statecharts, and ports, which are the input/output of the blocks. Ports are capable of having preconditions or postconditions defined on them. Stateflow charts [**84**], a variant of Harel's Statecharts, are introduced within Simulink models to explicitly model the behavior of the reactive system. In what concerns refinements for Stateflow charts only, most of the literature focuses on transforming Stateflow into formal languages that support refinement, such as in [**87**]. Nevertheless, Boström *et al.* [**23**] presented a refinement calculus for Simulink to provide additional capabilities for stepwise development, which is not included by default within Simulink. As with our approach, the purpose of the work from Boström *et al.* is to make common development practices more predictable and usable. Their refinements focus on two main goals: to create an implementation of a specification and to add features to an implementation, otherwise referred to as superposition refinement.

Specifications are similar to abstract classes, in that they simply define the overall structure for an implementation to follow. These specifications do not contain actual implementation details within them. During the refinement of a specification into a concrete implementation, several criteria must be met, summarized in Table 2.1. Superposition refinement aims at adding new behavior to a concrete model while preserving the original behavior of the model. New unconnected ports may be added to the refined model, however these ports cannot have preconditions attached to them. Concrete blocks may only be added and cannot be removed from the original model. This is similar to our requirement that states or transitions shall not be removed during refinement. Any block within the model that are marked virtual can be freely added or removed during refinement. We do not have this concept and do not allow for any states or transitions to be removed, as in our solution they may contain needed implementation details already.

2.6.2. Refinement Calculus for μ-Charts

The μ-Charts formalism [**110, 109**] was introduced as a variant of Statecharts that sought to provide additional expressive power. As with our solution, μ-Charts are based on Harel's Statemate semantics of Statecharts. In these papers, Scholz defines a calculus for refining μ-Charts as part of an incremental design process.

Transitions in μ-Charts can be added as long as the event trigger and guard associated with the new transition do not conflict with other transitions. In particular, the calculus requires that in composite states, any outgoing transition of the new substates does not broadcast events that are already being broadcast within that composite state (*i.e.,* broadcasts must be unique). This is somewhat similar to our Constraint 4. We do not make this stipulation, as there is no need to limit broadcasts to this extent to preserve structure and reachability. Instead, we require that the transitions with these new broadcasts be guarded.

Transitions can be removed only if there is another transition present that is enabled with the same event and guard evaluation as the deleted transition, and if the latter is never able to be triggered. Guards can be added through either conjunction or disjunction, as long as several formal conditions hold. Events and guards can also be removed if certain

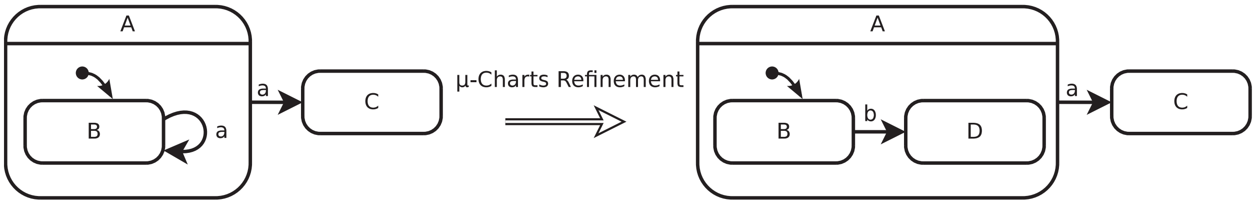


Figure 2.3. An example of a μ-Charts refinement that is not valid by our rules.

formal conditions hold. We do not allow for events or guards to be modified in our solution due to our focus on preserving the external behavior of the Statechart. Figure 2.3 shows an example where an internal transition with event a has been removed, since event a will always trigger the outer transition first. The figure also illustrates a refinement where an additional transition and path have been added out of state B. This would be valid with Rule 7 in our rules if transition a was preserved.

The μ-Charts rules allow removing default states, which is equivalent to removing a sub-statechart. Additional states can also be added, as long as they are plugged into the existing flow using new transitions. This is similar to Rules 5–8, as we require new states to be connected with new transitions or part of a valid statechart that is a component of a connected state.

2.6.3. Refinement on flat state machines

The authors in [**71**] define State Transition Diagrams with a semantics based on Statecharts, but where the focus is on asynchronous communicating systems such as in I/O-Automata. They present a refinement process where only the I/O interface of the machine is preserved. In contrast with our approach, this ignores variables and actions that may occur inside the states which allows us to guarantee reachability preservation statically. Furthermore, the authors define rules for addition and removal of states and transitions for flat state machines which would breach structure preservation in our approach.

In [**100**], the refinement is defined as an inclusion of the input/output trace between the refined and the original statechart. This is a black-box approach where internal events and the structure of the statechart are ignored. Since they allow non-determinism, refinements that remove states and transitions are defined to resolve the presence of chaotic behavior. Our approach does not treat the statechart as a black-box, and focuses instead on enforcing backward compatibility of the behavior, and the structure and internal events of the statechart.

Paech and Rumpe have introduced in [**93**] a type-centered technique where they equate automata with types. Specializing a type by adding attributes or operations to it is then reflected in the equivalent automaton in the form of adding data or transitions to it. The authors shown that such type specializations do preserve the fact that, regarding an environment (also defined using types), the supertype and the specialized subtype cannot be distinguished. Consequently, this implies a refinement relation between the corresponding automata. Besides presenting rules that do preserve the specialization/refinement relation, the authors also prove that such a relation is transitive, as we do in our work.

2.6.4. Refinement in UML-B

In [**104**], Event-B is used to refine UML class diagrams, and their semantic description defined in a state machine. This is done by translating the state machines into a state set representation or a state function representation. The refinement of the state machine is achieved in one of two ways. One is the refinement of transitions into multiple parallel transitions that specify subcases of the original transition. The other is the elaboration of a state with a nested state machine. This is similar to Rule 2, but without giving a concrete resolution to the situations where the state being refined already has a nested state machine, though not explicitly disallowing it. Most of the presented rules that guide these two refinement patterns deal with the preservation of the elements unaffected by the refinement, while we make use of a notion of identity, that helps us isolate this preservation of elements. While we aim at proving static verification of preservation of the refined statechart, the work in [**104**] executes an evaluation using theorem provers that, in conjunction with gluing expressions, checks the preservation of the original reachability of the state machine. Gluing expressions are used to provide the translation of the original conditions in terms of the refined elements. These can be manually defined or the evaluator can try and complete them using a discoverer, in accordance to the defined parameters and the free variables present. This leaves the possibility, although unlikely, of having gluing expressions of successive refinements that lead to a complete loss of the original semantics.

With the refinement of transitions, although different mechanisms of refinement are used, both provide similar results. In UML-B, the transitions are refined into subcases of the same transition as parallel transitions, while in our work we split the transition into a sequence of transitions where the subcases can be addressed between intermediate states of this path. This makes our approach more susceptible to the order of refinements, but with harder guaranties of preservation of the original conditions to enable the transition.

Another situation that is not approached in UML-B state machine refinement is the interference of orthogonal components and other sections of the statechart that is covered by our guards. Our approach is more restrictive, but covers more situations with respect to Statecharts semantics, while the approach presented for UML-B refinement allows for greater flexibility and evaluates guard expressions themselves, at the cost of having to prove reachability for every refinement.

2.6.5. B Method

The well-known B-method [**16**] proposes using the Abstract Machine Notation (AMN) to represent specifications throughout the software development process. A mathematical theory to refine AMN specifications allows one to add the necessary detail from requirements through implementation, while formally guaranteeing that relevant invariant properties of the system always hold. Each AMN machine contains a set of variables and a set of operations for manipulating those values and communicating with the environment. Although it also describes behavior, a B machine is different from a statechart in the sense that states are not explicitly modeled. Nevertheless, there are several resemblances between the notion of B refinement and our proposal of statechart refinement.

As in our approach, refinements in B aim at preserving the structure and reachability of a machine. Also, the behavior of a refined B machine can be more detailed than that of the original machine, *e.g.,* by eliminating non-determinism. As in our approach, sequences of events in the traces generated by the B machine can be interleaved by additional detail in the sequences of events in the traces of a refined B machine. Furthermore, the structural preservation condition in Definition 1 resembles the hiding principle of import rules in B refinement. The import rules ensure that the structure of the specification and the system's invariants are preserved throughout refinement. Finally, as in our case, the refinement relation in the B-method is also proved to be transitive.

There are also important differences between the two approaches. Firstly, refinements in B can be done over variables and/or the internal specification of operations. Our refinement relation concentrates on the notion of state, while abstracting from the data types manipulated by the entry or exit operations inside states. Our approach does nonetheless enforce certain syntactic constraints on the guards of the refined statechart transitions, thus implying limitations on the data flow possibilities through the refined statechart. Secondly, proof obligations are necessary in B to formally show and document that the invariants that were originally stated about a machine still hold. Proof obligations are generated on-the-fly for a given B refinement. This allows refinements of B machines to be very flexible, as long as the operations affected by the invariants remain structurally the same. However, the burden is on the developer to demonstrate that invariants hold manually or with automatic assistance. In our case, our refinements are less general and allow less flexibility because the rules are predefined and do not concern data types in general. Furthermore, our approach guarantees the preservation of structure and reachability of the original statechart, caveat the limitations discussed in Section 4.7 because of the tradeoff between theory and practice.

2.6.6. Refiniment for subtyping

Subtyping is related to the notion of refinement, yet it serves a different purpose. We analyze some Statecharts subtyping approaches for additional incites on Statecharts refinement.

2.6.6.1. Statemate

Harel *et al.*[**53**] address the concept of Statechart refinement through the use of object oriented inheritance, laying out several basic rules for states and transitions. As with our implementation, the focus for this approach is on creating functional and pragmatic rules to govern the refinement operations that can be performed on a statechart. To add new states, the developer can apply any of our rules (except Rule 1), but connectivity is mandatory to make sure the new states are reachable.

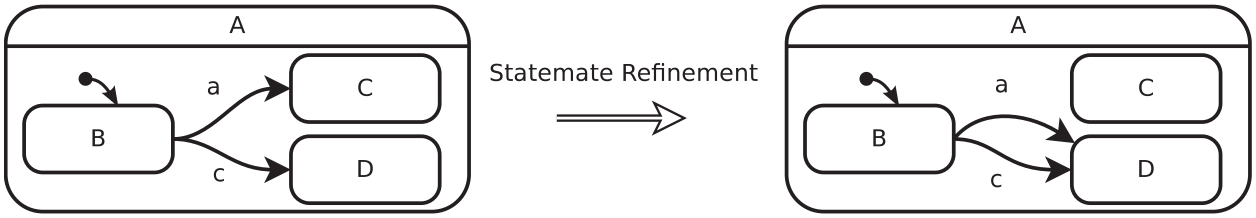


Figure 2.4. An example of a Statemate refinement that is not valid by our rules.

Harel *et al.* also provides several rules for transitions refinement. The target state and guard of a transition can be changed as desired. Such refinements can easily break the original structure, as demonstrated in Figure 2.4. By having stricter structural rules, we can provide a more consistent and predictable relationship between the original and the refined statecharts.

2.6.6.2. Rational Rhapsody

Rhapsody [**61**] defines three types of modifiers that can be annotated on a statechart: *inherited*, *overridden*, and *regular*. Elements marked as being inherited will allow for changes to the original statechart to propagate to the refined model. Elements marked as being overridden no longer accept general changes from the original statechart. However, deletions from the parent will still propagate to the child. Overridden statecharts have no relationship to the original statechart. In our approach, we do not provide the concept of overriding the original model. If the original structure and reachability is no longer desired, then the original-refined statecharts relation no longer holds and the new statechart should be developed independently from the original. If a new behavior is desired in the refined model, then that behavior must be added following the refinement rules.

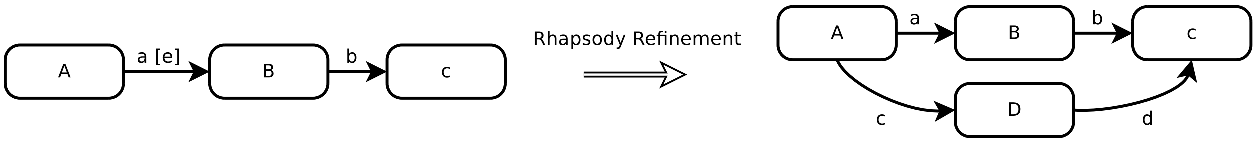


Figure 2.5. An example of a Rhapsody inheritance refinement that is not valid in our rules.

There are several Statecharts inheritance rules in Rhaspody, but are less restrictive than ours at the cost of not preserving neither the structure nor the behavior. For example, Figure 2.5 shows a valid refinement in Rhapsody, where state `D` and transitions `c` and `d` have been added. In contrast, in our approach, the removal of the existing guard is not permitted, but the new path can be created using Rule 8 with the added restriction that the initial transition `c` is deterministic and does not interfere with `a` applies.

2.6.6.3. UML State Machines

UML presents three sets of inheritance rules for State Machines [**50**]: *subtyping*, *strict inheritance*, and *general refinement*. We only focus on the former two, because the latter allows modifying the statechart almost arbitrarily.
Subtyping. As in our work, the focus of subtyping is to preserve the pre and postconditions of applying the events and actions of a state machine, thereby satisfying LSP. When refining with subtyping in UML, a refined transition must remain connected to its original source state, though it can have its target state altered. We do not allow for altering the source or the target state of transitions, as this would break the semantics of the original state machine. Guards are allowed to utilize conjunctions, which may invalidate the enabling of a transition that would have been enabled in the original. The specification in [**50**] argues that the states preserve the output of the statechart as a whole, while the transitions define how each state is related to other states. In our rules Condition 4.3.3 prevents altering existing guards from being changed. This decision was made so that we can only restrict the state space expected by any refined statechart, leaving us within the originally defined state space. In UML subtyping, it is possible to effectively remove the effect of a guard by disjuncting it with `true`.

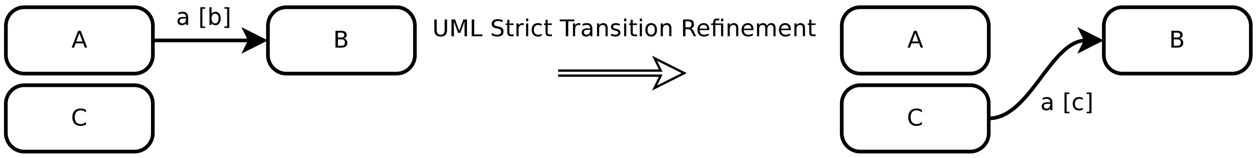


Figure 2.6. An example of UML transition strict inheritance that is not valid in our rules.

Strict inheritance. The focus of strict inheritance is to re-use a statechart associated with a parent class [**50**]. The rules are very permissive which hinders regression guarantees.

Figure 2.6 demonstrates a transition following the strict inheritance rules of UML that violates the preservation of the identity of the refinement definition as the path from A to B no longer exists. In our rules, we only permit the creation of a new path between C and B if there already exists a way of reaching B from C.

Strict inheritance and subtyping provide greater freedom than permitted by our rules. Our focus is to provide more predictability and clearly defined expectations of any refined statechart than the rules provided by UML.

2.6.7. Refinement in Other Behavior Modeling Formalisms

2.6.7.1. Interface Automata

Interface Automata [**35**] are an alternative formalism to Statecharts for designing reactive systems. This formalism utilizes, at its base, a component structure which exposes a set of expected inputs and outputs. Components can then contain a set of states and transitions which fire on specific inputs. It is assumed that the executing environment treats each component as black box.

Refinement of Interface Automata is concerned with following the principle of contravariance. This is closely related to our intention of preserving the external behavior of a Statechart during refinement. These refinements have complete freedom in terms of the creation, removal, or modification of existing states and transitions, so long as the refinement respects the state space of the input and output events, since this solution seeks to preserve aspects of the input and output events of the component being refined. We do not grant this freedom in our solution since we do not assume that components are black boxes from the environment.

2.6.7.2. Partial Behavioral Models with Uncertainty

The work by Chechik *et al.* [**42**] on partial modeling with uncertainty looks at behavioral models, such as Statecharts, and their refinements from three points of view. Typically, a statechart describes how the modeled system shall behave (positive set) and, consequently, how it should not behave, often defined by the complement of the traces output by all its possible execution (negative set). However, the approach by Chechik *et al.* allows the modeler to explicitly specify what parts of the positive or negative sets are deemed uncertain at earlier stages of the development. To achieve that, they annotate the elements of a model with three possible values: *true*, *false*, and *maybe*. The refinement of a state machine will reduce the uncertainty in the partial model, thereby replacing *maybe*'s by *true*'s and *false*'s. Unlike their approach where the modeler can freely modify a *maybe* element, we only allow for a limited set of modifications to refine the model. The advantage of our approach is that refinements are guaranteed to preserve the structure and reachability of the original model, while their approach requires to explicitly verify these properties as in the B-method.

2.6.7.3. Petri Nets

Petri nets are another formalism for modeling the behavior of systems. Since Petri net models are also developed incrementally, there is a lot of literature on guaranteeing the preservation of certain properties when refining models [**92, 91, 76, 60**]. In previous work [**81**] we have explored the conditions under which refinements of a Petri net model preserves invariant safety properties by construction. Place preservation and strengthening of guards on transitions are similar to our rules for Statecharts refinement.

2.6.7.4. Specification and Description Language

The Specification and Description Language (SDL) [**63**] is a hierarchical modeling language that is an extension of finite state machines. It defines reactive systems similar to those represented by Statecharts. Systems are the root elements, which contain blocks. Blocks contain processes. Processes contain procedures. The procedures contain the operational statements in the model, while the other layers contain different levels of abstraction for signal routing. The SDL specification allows for specialization/inheritance, however there is no specific definition for an SDL equivalent of our refinement rules.

SDL specialization results in a type-subtype relationship between original and specialized model elements. SDL elements can be marked as virtual, which allows for the element to be completely redefined. Elements that are not marked as virtual must be at least partially preserved. All properties must be preserved during specialization. Transitions can be redefined during specialization. Additionally, a subtype should only be expected to have one concrete super-type, though the specification allows for multiple interfaces to be implemented. Interfaces define signals, signal lists, or remote procedures that the implementation will require. This is similar to languages such as Java which allow for multiple interfaces to be implemented but only one class to be extended. This implementation is overall very similar to the inheritance implementation in Rhapsody.

2.6.7.5. Abstraction patterns

In [**111**], the author presents a series of structural, behavior and temporal abstraction patterns, with the aim of better help in reducing the discrepancies between design aim and actual implementation. Therefore, these patterns are related to our refinement rules when applying their inverse. Most of the patterns presented in the structural and behavioral categories rely on the composition or a variation of the Black Box and Cable patterns. In particular, the Summary State and the Summary Message patterns map closely to our Rule 2 and 8 respectively. However, to be applicable to Statecharts, the inverse of the Summary State pattern would require additional information to not break the semantics and structure of the original Statechart. Apart from the Group Transition pattern that allows one to fundamentally change the structure, all other patterns can be achieved by composing our rules. This reinforces the breadth of applicability of the possible refinements achievable with our approach.

2.6.7.6. Software behavior refinement

The FOCUS software development method from Broy and Fuchs [**25**] provides a thorough description of how distributed software can be developed in a component-based formal fashion. FOCUS encompasses the complete software development cycle, from requirements to code. In FOCUS, software development is achieved incrementally, via many different types of behavioral or structural refinements, adapted to the current model and methodological step in the development cycle. While FOCUS is a collection of mathematical models, concepts and rules, its tool counterpart AutoFOCUS [**18**] allows building hierarchical state machines as models of the behavior of individual components. The FOCUS theory provides a formal background for the integrated semantics of all types of models proposed in AutoFOCUS and also a set of refinement rules to be applied during the software lifecycle. Broy and Fuchs require that formal proofs following refinements are built, such it can be shown that the development steps are theoretically sound. In the AutoFOCUS tool these principles are loosely applied and, while soft principles of model correctness are applied to the models that are built (*e.g.,* conformance to a metamodel), a gap with the theory remains to allow for usability in practice.

2.7. Related Work on Incremental Model Transformation

The work presented also presents some common ground with incremental Model transformation. Tools such as VIATRA [**125**] and EMF-IncQuery [**120**], allow for exactly that. But the use of such approach would shift the complexity of the problem to how to express it as model transformation rules, and how to guarantee the correctness of the results as expressions of the transformations themselves. This would only provide tool by tool solutions. We on the other hand take inspiration from the declarative nature of Model transformation rules, but our rules define actions to be added to the model, not operations over the model. As such there is no direct match of our conditions in the model, and we do not apply rules over a portion of model, but instead to a single point (if we consider the orthogonal nature of the matched states as independent). We also rely on a RETE network, but customize the transformation engine to our purposes and kind of rules. So instead of this network codifying the full model to evaluate where rules can be applied, our use of the network codifies one point of the model, telling us what rules can be applied there.

There are of course more reasons to not use pre-existing model transformation tools. For one the rules presented may still further evolve away from the declarative structure of conditions and effects. Maintaining a specialized engine keeps this path more flexible, instead of having to redo the full transformation process at every iteration of the transformation rules structure.

Even considering that we can, in a reasonable manner, convert the interaction rules into transformation rules that perform the refinement of the statechart; we still have hurdles to the use of pre-existing transformation engines. These lie with the nature of generic Model transformations.

Model transformations for the purpose of this argument can be categorized into three types.

Outplace transformations where the effects of a transformation are applied on a different model instance. This would lead to a new model per iteration, and therefore a lot of temporary files, and we still need an external controller because we do not know how many iterations any given model will have before we run the transformation itself.

Inplace transformations where the matches are calculated before applying the effects. Would not have the many files problem of outplace transformations, but would still require an external controller to iterate over the model.

Inplace transformations where the applications of rules is recalculated after the application of each effect. Would potentially enter an infinite loop as rules become re-aplicable after small changes, or it would require a substantial re-encoding of the rules to have the required information to stop a rule from being re-applied wile preventing similar rule applications from being blocked. This would shift the complexity of making a custom transformation engine into the creation and maintenance of the rules themselves.

None of these options is ideal for our situation, as they still some level of custom control code or constant maintenance from project to project. So instead we built our own tool to do the refinement.

Chapter 3

Explicitly Modeling of Modeling Editor Interactions

Mixing one's wines may be a mistake,
but old and new wisdom mix
admirably.

Bertolt Brecht, The Singer, in The Caucasian Chalk Circle (1944)

We now present our proposal to specify user interaction following MDE principles. We first illustrate the practical implications of this approach with a running example.

3.1. Running Example

We have used the music editing example as a motivating example, because it provides a broad understanding of pre-existing interactions that are not your typical computer editor interactions. And we do indeed use this example in our tests of the process and platform presented in this thesis. But the definition of a music editor, even a simple one, relies on a large number of interaction rules, and at the same time these rules are not representative of a broad set of application patterns. So instead we turn to an example that can provide the illustration of a broad set of features and functionalities of the presented work, with a minimum number of rules.

As a running example, let us define a simple DSML editor to model a configuration of Conway's Game of Life (GoL) [**47**]. We assume DSML editors with a 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 `alive` status. In this example, we construct a DSML editor to model configurations of the cell grid. We consider that the operational semantic rules are specified and simulated using a model transformation.

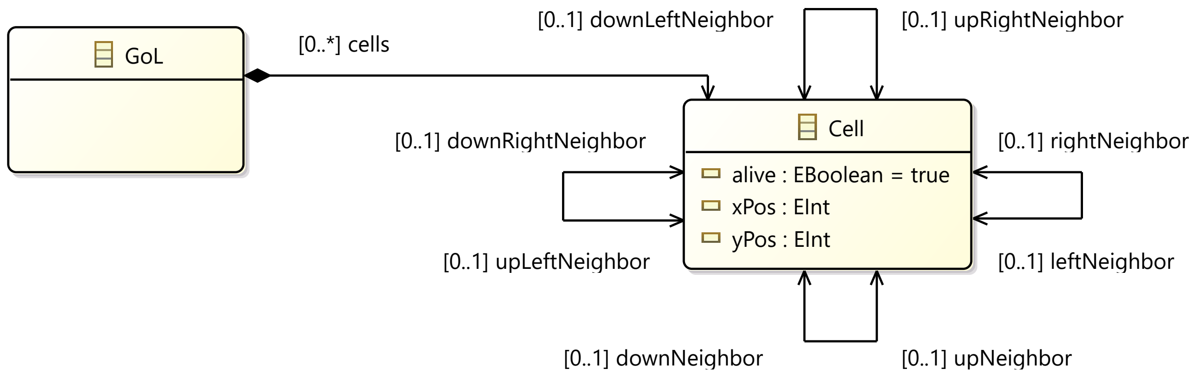


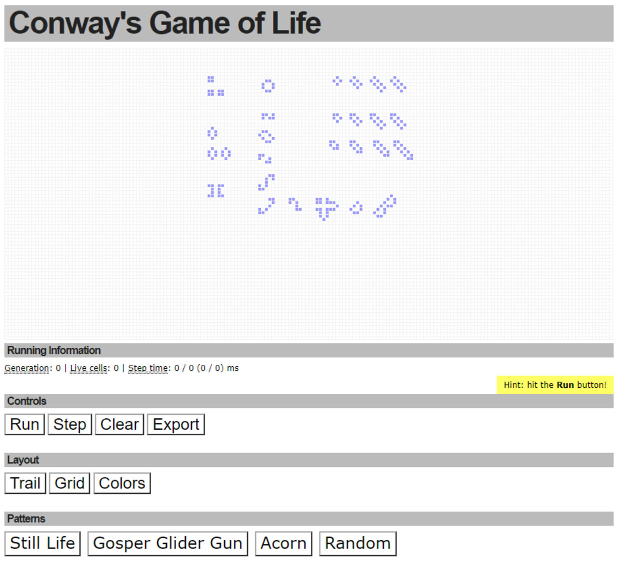
Figure 3.1. GoL Metamodel.

If we were to synthesize a graphical GoL editor with common MDE frameworks, we would first create a metamodel of GoL, as depicted in Figure 3.1. This metamodel consists of a single class `Cell` with a boolean attribute `alive`, two integer attributes `xPos` and `yPos` for the coordinates of the cell, and a set of `neighbor` associations to itself defining the eight possible neigbor positions. 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 configuration model, the user needs to manually instantiate each cell from the toolbar and set the `alive` property of the cells alive.

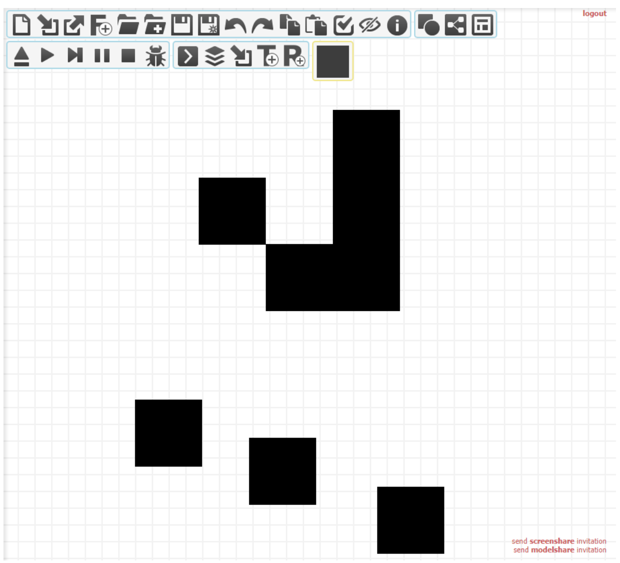
If we look in Figure 2(a), we have an example of a traditional GoL editor [1], where the interactions are reduced to a minimum. This helps to reduce accidental complexity, and it is what we are replicating with our example. In Figure 2(b) we can see how such an editor looks like using the standard approach to DSML editors.

Although it provides no direct impact on the interaction itself, GoL can be simulated by a three steps model transformation. In the first step normalizes the model. For this process, a rule connects missing adjacent `Cells` connections, another rule creates all the missing dead `Cells` around each alive `Cell`, and a final rule merges any overlapping dead `Cells`. The second step applies the operational semantics rules of GoL as described earlier. The final step reduces the size of the model to its essential data by removing all dead `Cells` with only dead neighbors. With this simulation the modeling process only needs to focus on live `Cells` and the specification of their more immediate neighboring `Cells`. This is still more work than the traditional editors. We could simplify the simulation steps, by explicitly edit

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



(a) Traditional GoL editor



(b) GoL editor in AToMPM

Figure 3.2. Examples of GoL editors

the model in a way to include dead cell. But would bring the interaction even further away from the essential provided by the traditional editors.

So ideally, the editor should only provide a single language element `Cell` with the attribute `alive` set to `true`, forgoing any language element selection toolbar or attribute view. The reliance on an underlying grid to instantiate the models, also means it is easier to find neighboring `Cells` without having to explicitly state them. Additionally, since the only other operation in GoL is to run the simulation, triggering its execution should be easily accessible, instead of configuring it and launching it from menus and wizards.

3.2. 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 DSML 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 DSML, 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.

3.2.1. 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 [**28**]. On top of providing designers the freedom to design the appearance of the editor, this model also affects user interaction. For example in Figure 3.3, 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.

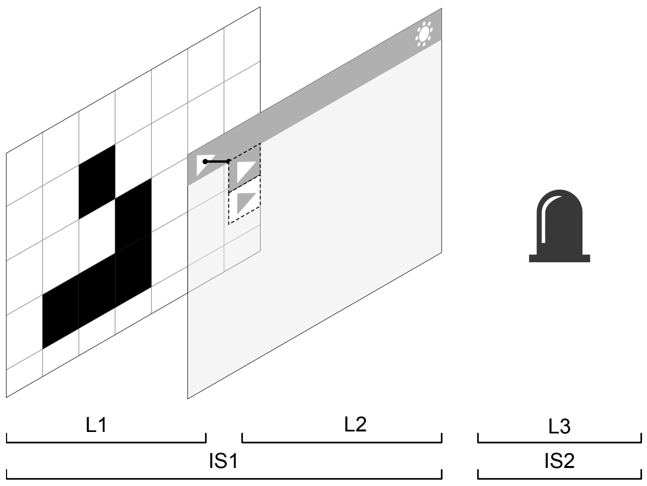


Figure 3.3. Desktop interface layers.

Figure 3.4 presents the basis of the metamodel of the interface modeling language. It is based on OMG's Diagram Definition standard [**49**]. The elements of the interface model are very different from DSML 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 3.3), we define one toolbar with two buttons. In a mobile version of our example, as depicted in Figure 3.5, we define two toolbars, one with one button and another with three buttons. This setup has the objective of using the first button as a toggle of the second toolbar, as to minimize the elements occupying the space in a small screen. Given that there is only one element type in this DSML, 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. The **canvas**, in which the user performs the modeling actions, is typically a grid where DSML 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 DSML 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

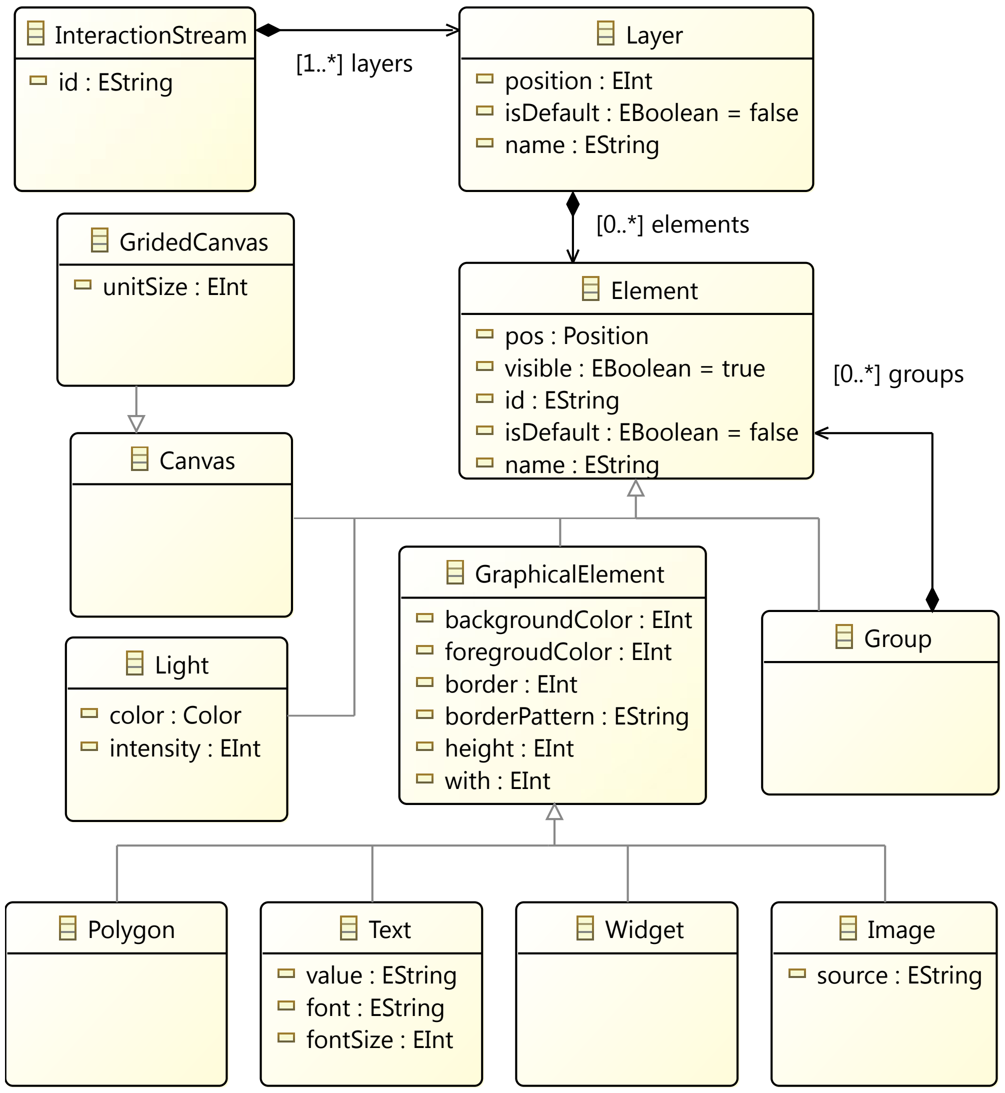


Figure 3.4. Interface metamodel

the canvas and the DSML as a black-box that connects the editor to any model-aware back-end. When prompted by any canvas interaction, the back-end responds with the updated model.

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. Another way to view layers is as a meta-group of elements that have the common interaction intentions within the same interaction stream. In Figure 3.3, 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. The third layer (L3) contains a **light** element that has no implication on overlay rendering of the interface; but still adheres to the second aspect of the use of layers, intent. In this perspective, the interactions performed on the canvas have a different intent from the ones over the toolbar

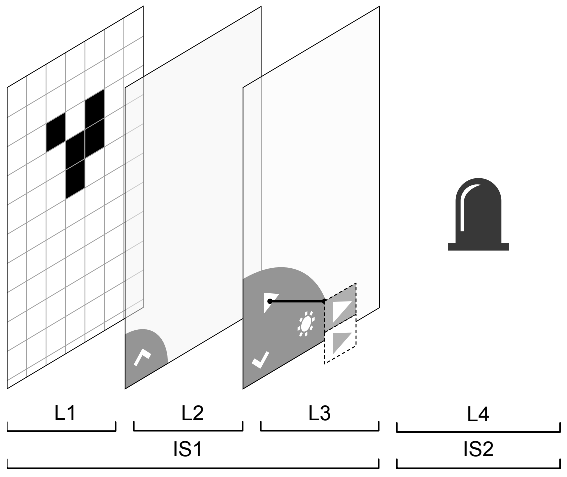


Figure 3.5. Mobile application interface layers.

or the light. The DSML editors we consider assume there is a layer dedicated to the canvas, where DSML instances reside.

Interaction Streams segment forms of interaction that are in different interaction spaces, *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.

Ultimately we refer to the metamodel in Figure 3.4 as a basis for interface specification, because it can always be extended to include new interface elements beyond this initial specification. This is accomplished by extending the Element class, as we can observe with the class Light that allow us to extend the interface beyond the screen into an actual physical light. Still, in this process of extending the interface metamodel, we need to be careful and not extend it to every possible interface element or device, as there will be new interface elements do add with new technological innovations, leading to a cumbersome metamodel, and its respective maintenance. Instead, the focus should be on specifying the generic aspects unique to interacting with these extended elements so that they can be reused as a category of elements instead of every single implementation of a new device. An example of this is that the class Light encompasses any form of external light, be it an embedded LED on a phone or table, an LED in an Arduino board, the actual lighting system of a room or any new system that provides an equivalent feedback.

3.2.2. 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 DSML are defined, the user can tailor the user interaction of the DSML editor to the target users.

As defined by its metamodel in Figure 3.6, an interaction model consists of a set of **interaction rules**, like the five needed for the GoL editor in Listing 3.1.These rules follow the formating specified in the grammar depicted in Listing 3.2.

Listing 3.1. Interaction rules for GoL

```
InteractionRule A : CSAddInstance
  Condition {
    focus Canvas {}
  }
  --- select -->
  Effect {
    Lang Cell {op = add}
  }

InteractionRule B : CSRemInstance
  Condition {
    focus Lang Cell{}
  }
  --- select -->
  Effect {
    Lang Cell {op = rem}
  }

InteractionRule C : RunTransfromation
  Condition {
    focus Interface playModelButton{}
  }
  --- press -->
  Effect {
    Interface light {value = "running"}
    Interface playModelButton {value = "active"}
    Operation runGoL {}
  }

InteractionRule D : EndTransformation
  Condition {
    Interface playModelButton {value = "active"}
  }
  --- _finish -->
  Effect {
    Interface light {value = "finished"}
    Interface playModelButton {value = "default"}
  }

InteractionRule E : TurnLightOff
  Condition {
    focus Interface lightButton {}
  }
```

```
--- press -->
Effect {
  Interface light {value = "off"}
}
```

Listing 3.2. Interaction rule fromat

```
InteractionRule <ruleName>
    Condition {
      ( (focus|not)
        ( Canvas {},
          Interface <elementName> {
            (value = "<valueString>")?
          },
          Lang <elementType> {
            (value = ("<valueString>"|<varName>))?
          },
          Var <varName> {
            (value = ("<valueString>"|<varName>))?
          } )
      )*
    }
    ( --- <eventName> -->
    Effect {
        ( Interface <elementName> {
            (value = "<valueString>")?
          },
          Lang <elementType> {
            op = (add|rem|move) (, value = [<varName>(,<varName>)*])?
          },
          Var <varName> {
            (value = ("<valueString>" | <varName>) )?
          },
          Operation <operationName> { }
        )+
    } )*
```

As commonly represented in UX using Trigger-Action rules [**122**], interaction rules represent declarative context-dependent effects on the editor: when an event occurs and the interface or the DSML satisfies a specific condition, the rule produces an effect on either of them. Semantically, an interaction model takes its root ECA rules [**39**].

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 DSML (`LangElement`). These are represented by `Interface` and `Lang` keywords respectively, in Listing 3.1. One of the elements in the condition can be the **context** of the rule, represented in Listing 3.1 with the keyword `focus`. The context is used to determine the exact point of interaction, when events from pointing devices, such mouse,

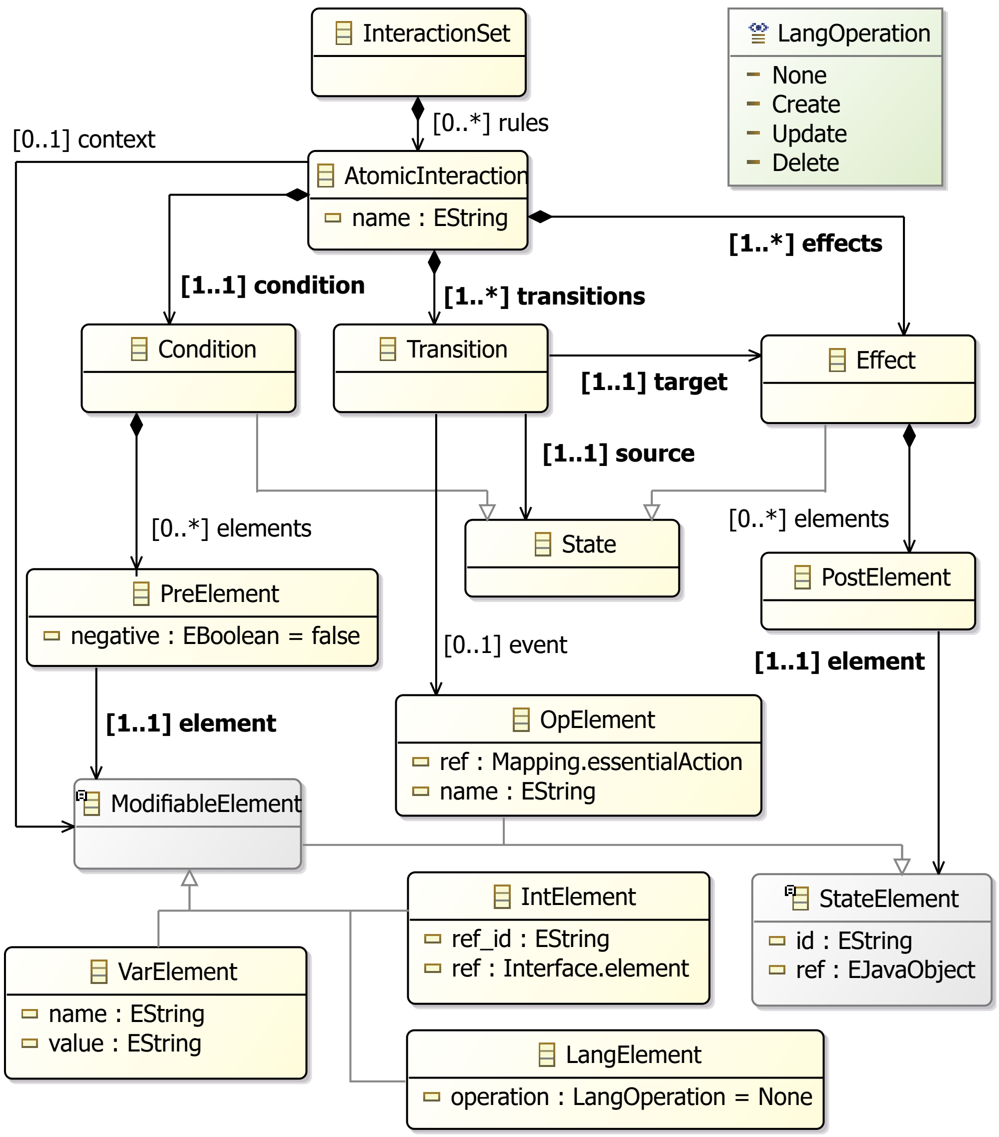


Figure 3.6. Interaction metamodel

touch screens or a light gun, are triggered and provide such information. For this reason the context is optional, as many events do not provide this information. Such as keyboard events or internal events. Nevertheless, it is advisable to use a context whenever an event provides one.

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 DSML it should apply. These operations can be set operation on interface model elements (`IntElement`), CRUD operations on DSML (`LangElement`) or backend function calls (`OpElement`).

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, 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 `Lang` 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`.

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 DSML, 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 3.6, `IntfElement` reuses classes from the interface metamodel, as we do for `LangElement`. The `ref_id` attribute ensures that the interface element corresponds to one that had been defined in the interface model. An effect may also call an external operation, defined in the event mapping model, such as `runGoL` 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 `runGoL` operation terminates. The rule then enables the play button and turns the notification light to green. This implies that execution of the `runGoL` operation does indeed raise an event when it terminates Note also how no context is needed with the internal event. This is because such event provides no interaction context, but still possible because the elements of this rule can be unambiguously selected in the condition, and updated in the effect, that such a rule doesn't require such an explicit context.

The interaction model expresses interactions at a level of abstraction that is closer to the intent than the implementation of the interaction within the DSML editor. For instance, rule C states that a button is pressed independently of whether it is a mouse click or a touch on the screen. Nevertheless, there are situations where just these constructs are not enough. For this, the interaction modeling languages allows for multiple effect blocks and variables.

There is no inherent order inside and effect block, as it describes a set of simultaneous actions. The use of multiple successive effect block allows the enforcing of an order in the rare occasions it is required. This is done by exhausting all actions in one effect block before starting the next block in a sequence. This is also done without any further events being triggers, as shown in rule `A` of Listing 3.3, thus preserving the atomicity of the rules.

`VarElements` allow the preservation of values beyond the scope of a single rule. This allows the coordination of rules and the sharing of data between rules, in a way similar to passing parameter values between rules, like pivots in model transformations [ref to T-Core]

Listing 3.3 extends the GoL rules shown in Listing 3.1 into more elaborate interactions to show examples of the use of variables to share data between rules and to coordinate the applicability of rules.

Listing 3.3. Interaction rules for GoL using variables

```
InteractionRule A : CSAddInstance
  Condition {
    focus Canvas {}
  }
  --- select -->
  Effect {
    Lang Cell {op = add}
  }
  ----->
  Effect {
    Var addedCell {value = focusElement}
  }

InteractionRule B : CSRemInstance
  Condition {
    focus Lang Cell {}
  }
  --- select -->
  Effect {
    Lang Cell {op = rem}
  }

InteractionRule C : RunTransfromation
  Condition {
    focus Interface playModelButton {}
    not Var Simulate {value = "running"}
  }
  --- press -->
  Effect {
    Interface light {value = "running"}
    Operation runGoL {}
    Interface playModelButton {value = "active"}
    Var Simulate {value = "running"}
  }

InteractionRule D : EndTransformation
  Condition { }
  --- _finish -->
  Effect {
    Interface light {value = "finished"}
    Interface playModelButton {value = "default"}
    Var Simulate {value = "null"}
  }

InteractionRule E : TurnLightOff
  Condition {
    focus Interface lightButton {}
  }
  --- press -->
  Effect {
    Interface light {value = "off"}
  }

InteractionRule F : UndoAdd
  Condition {
    Lang Cell {value = addedCell}
  }
  --- undo -->
  Effect {
    Lang Cell {op = rem}
    Var addedCell {value = "null"}
  }
```

In this extended example, we add rule `F` and extend rule `A`, to exemplify the use of variables for passing data and the use of multiple effects blocks. The new rule `F` simulates undoing the addition of a `Cell` outside the modeling backend. This rule does this, by looks at the last added `Cell` and removing it. But to know what was the last `Cell` added, we need to extend rule `A`. In this extended version of rule `A`, we copy the reference of the added `Cell` to the variable `addedCell`. Furthermore, we need to ensure that the variable `addedCell` is only updated after the new `Cell` was added. So we need to place this step in a new `effect` block that is called after the `effect` block that adds the `Cell` to our GoL layout.

In this extended example we update rules `C` and `D` to exemplify the use of variables for coordinating between rules. Rule `C` and `D` are now coordinated with the use of variable `Simulate`, so that rule `C` is only available if no simulation is already running. For this we have the negative condition over the value `running` on variable `Simulate`, and rule d as an effect removes the value `running` from the variable `Simulate`. This actually reflects a coordination pattern that should be generalized, instead of letting the user manually specify it and maintain it. To manage this we added a coordination model to our solution.

3.2.3. Coordination Model

When defining the interaction rules, all the rules are potentially applicable to create concurrent interactions. But sometimes during the process of designing the UX of a DSML editor, we may want to initially disable some interactions, or may need to resolve ambiguities in the rules that can lead to a lack of determinism. We have seen in the example in Listing 3.3 how these issues can be manually tackled with the use of variables. However, the manual use of variables is verbose and forces the designer to think like a programmer, diverting from the interactions themselves. This also means that any change in how the rules are coordinated implies having to re-factor all affected rules by hand. So we propose a coordination model that defines relations between rules that are applicable concurrently.

For this, we looked into CTT, for inspiration in identifying the pertinent coordination operators. The operators we have identified are *triggers*, *enables*, *disables* and *precedence*. Other operators may be identified with future work, but at present these are the ones we have found are needed, from our case studies. CTT is designed to coordinate tasks that can have multiple steps, whereas we are coordinating atomic interactions, and as such many of the operators in CTT are not applicable.

We have formalized how the operators we propose affect the interaction rules. Still, these patterns can and have been applied manually.

3.2.3.1. Formalization

Consider an interaction rule as $\alpha = \langle f, I, e, R, O\rangle$, the tuple of respectively the optional reference to the element that serves as the Focus of the interaction f, the set of initial predicates I from the condition of a rule, the triggering event e, a sequence of result predicates R from the effects of the interaction rule, and a set of operations O performed in the effects. Both sets of predicates I and R, are composed of predicates over Interface (i), Language (l) and Variable (v) elements. The set O is composed of any operational function such as raising an event.

Given any two interaction rules A and B, using the Plotkin notation [**99**], we define the effects of our coordination rules as follows:

A triggers B : Executing the interaction rule A triggers the execution of the interaction rule B. Where B is executed only if its conditions I_B are satisfied. We denote this coordination operation with the symbol `->`.

$$\frac{\begin{array}{c} A = \langle f_A, I_A, e_A, R_A, O_A\rangle \\ B = \langle f_B, I_B, e_B, R_B, O_B\rangle \end{array}}{\begin{array}{c} (A : A \texttt{ -> } B) = \langle f_A, I_A, e_A, R_A, O_A \cup e_B\rangle \\ (B : A \texttt{ -> } B) = \langle f_B, I_B, e_B, R_B, O_B\rangle \end{array}}$$

If we have two rules A and B that resolve to their respective tuple representation. Then A such that A triggers B resolves to the extended tuple where raising event e_B is added to the operations of A. B such that A triggers B remains unchanged. By placing a raise event e_B operation as an effect of rule A, whenever the interaction of rule A is executed, the system will attempt to trigger rule B.

A practical application of this coordination operator would be to include debugging outputs from the interactions. Instead of adding this effect to all pertinent rules, one would create a separate rule for debugging output, and coordinate all pertinent rules as triggering this new debugging rule. This would allow to add or remove this feature without risking introducing errors in the remaining rules.

A enables B : Rule B is applicable only after A has been applied at least once. Note that B is executed only if its conditions I_B are satisfied and its event e_B is triggered. We denote this coordination operation with the symbol |>.

$$\frac{\begin{array}{c} A = \langle f_A, I_A, e_A, R_A, O_A \rangle \\ B = \langle f_B, I_B, e_B, R_B, O_B \rangle \end{array}}{\begin{array}{c} (A : A \;|\!> B) = \langle f_A, I_A, e_A, R_A \cup \{v \leftarrow \texttt{true}\}, O_A \rangle \\ (B : A \;|\!> B) = \langle f_B, I_B \cup \{v \leftarrow \texttt{true}\}, e_B, R_B, O_B \rangle \end{array}}$$

If we have two rules A and B that resolve to their respective tuple representation, then A such that A enables B resolves to the extended tuple where a control variable v is added to the result predicates R_A with the boolean value `true`, and B such that A enables B resolves to the extended tuple where the same control variable v is added to the condition predicates I_B with the same boolean value `true`. By placing as a condition of rule B a variable that is affected by the effect of the application of rule A we give control to rule A about the applicability of rule B.

An example of the practicality of this coordination operator, is to start be only viewing the model, and having a button whose interaction enables editing of the model and all the interactions associated with that mode.

A disables B : Once A has been applied, B can no longer be applied, even if its conditions I_B are satisfied when e_B is triggered. We denote this coordination operation with the symbol >| .

$$\frac{\begin{array}{c} A = \langle f_A, I_A, e_A, R_A, O_A \rangle \\ B = \langle f_B, I_B, e_B, R_B, O_B \rangle \end{array}}{\begin{array}{c} (A : A >\!| \; B) = \langle f_A, I_A, e_A, R_A \cup \{v \leftarrow \texttt{true}\}, O_A \rangle \\ (B : A >\!| \; B) = \langle f_B, I_B \cup \{v \leftarrow \texttt{false}\}, e_B, R_B, O_B \rangle \end{array}}$$

If we have two rules A and B that resolve to their respective tuple representation, then A such that A disables B resolves to the extended tuple where a control variable v is added to the result predicates R_A with the boolean value `true`, and B such that A disables B resolves to the extended tuple where the same control variable v is added to the condition predicates I_B with the opposite boolean value `false`. The application of this coordination operator is similar to the **enables** operator, with the diference that instead of rule A seting and enabling value it is effect, it has a disabling one.

The practical application of this coordination operator, is situations as popup menus when active disable all other interactions not related to the popup itself.

A has **precedence** over **B** : If A and B are both applicable, always apply A. Note that B is only applicable if A is not. Here, we assume that for both rules to be applicable at the same time, then $e_A = e_B$ and $I_A \nsubseteq I_B$. We also follow the assumption that the negation of a set is the set disjunction of the negations of all its elements. We denote this coordination operation with the symbol $\dashv$.

$$\begin{array}{r}
A = \langle f_A, I_A, e, R_A, O_A \rangle \\
B = \langle f_B, I_B, e, R_B, O_B \rangle \\
\hline
(A : A \dashv B) = \langle f_A, I_A, e, R_A, O_A \rangle \\
(B : A \dashv B) = \langle f_B, I_B \cup \neg \left(I_A \setminus (I_A \cap I_B) \right), e, R_B, O_B \rangle
\end{array}$$

If we have two rules A and B that resolve to their respective tuple representation, then B such that A has precedence over B resolves to the extended tuple where the negation of the conditions of A that do not overlap with the conditions of B is added to the conditions of B, and A such that A has precedence over B remains unchanged. By adding the conditions of rule A as a negative condition of rule B we guarantee that rule B will not be applied if rule A can be. When doing this we make sure to exclude any overlapping conditions between rule A and rule B from the negative conditions as to nit disturb the original evaluation of rule B.

An example of the use of this coordination operator is when incrementally adding interactions to an existing set of interactions, the conditions of a rule application can overlap, specially if the designer is only focusing on that one rule. This operator, provides a way to resolve any ambiguities that can arise without having to check and potentially change every pre-existing rule, every time a new rule is introduced.

3.2.3.2. Inter-operator interaction

We saw in Listing 3.3 the use of variables manually defined using the principles of the coordination model. In Listing 3.4 we have the same rules, but instead of coordinating them manually, we specify what operators coordinate the rules in Listing 3.5.

Listing 3.4. Interaction rules for GoL relying on coordination operators

```
InteractionRule A : CSAddInstance
```

```
    Condition {
      focus Canvas {}
    }
    --- select -->
    Effect {
      Lang Cell {op = add}
    }
    ----->
    Effect {
      Var addedCell {value = focusElement}
    }

  InteractionRule B : CSRemInstance
    Condition {
      focus Lang Cell {}
    }
    --- select -->
    Effect {
      Lang Cell {op = rem}
    }

  InteractionRule C : RunTransfromation
    Condition {
      focus Interface playModelButton {}
    }
    --- press -->
    Effect {
      Interface light {value = "running"}
      Operation runGoL {}
      Interface playModelButton {value = "active"}
    }

  InteractionRule D : EndTransformation
    Condition { }
    --- _finish -->
    Effect {
      Interface light {value = "finished"}
      Interface playModelButton {value = "default"}
    }

  InteractionRule E : TurnLightOff
    Condition {
      focus Interface lightButton {}
    }
    --- press -->
    Effect {
      Interface light {value = "off"}
    }

  InteractionRule F : UndoAdd
    Condition {
      Lang Cell {value = addedCell}
    }
    --- undo -->
    Effect {
      Lang Cell {op = rem}
      Var addedCell {value = "null"}
    }
```

Even though, there are no big differences between Listings 3.3 and 3.4 because the GoL example requires a small number of rules. As the number of rules increases and to have a better understanding of how the rules relate in coordinating with each other, these will be better maintained by a global view of coordination as provided in Listing 3.5.

Listing 3.5. Coordniation rules for GoL

```
{
  A : CSAddInstance     |> F : UndoAdd
  B : CSRemInstance     >| F : UndoAdd
  C : RunTransfromation >| F : UndoAdd
  F : UndoAdd           >| F : UndoAdd
}
{
  D : EndTransformation |> C : RunTransfromation
  C : RunTransfromation >| C : RunTransfromation
}
```

In Listing 3.5 we can see that rule A (adding a new `Cell`) enables rule F (undo add `Cell`), and that rules B (remove `Cell`), C (run transformation) and F disable the rule F, so that the undo function is enabled when a `Cell` is added and disabled following any other model changing interaction. By grouping these rules together in a shared concern, we can make these rules share the same coordination variable. We can also see that rule D (finish transformation) enables rule C (run transformation) and rule C disables rule C, so that when a transformation is started, no new transformation can be started until the running transformation if finished. In these listings we can also observe that the sharing of data between rules still requires the manual use of variables, and this might be a good addition to the coordination rules at a future point.

Still, this also provides a proof that future practical implementations of the coordination model, will be possible without changes on the remaining framework. It also shows that there is space to improve the actual implementation of the coordination model, with further improvements such as optimization on the use of variables, and group declaration of coordinations.

3.2.4. Event Mapping Model

Interaction rules rely on domain-specific concepts, be it the elements of the model and interface, the events that trigger the rules or in the actions the rules perform. We call **essential actions** all actions, that compose the minimum set of actions needed in that domain-specific usage. These actions can be actions performed by the user on the editor through triggered events or actions the editor performs as part of the effects of its interaction rules. This allows us to keep the interaction rules independent of the implementation and solely focused on the intent of the designer. Nevertheless, the generated editor must rely on an implementation. To do so and keep the focus and independence of the interaction rules, we use an event mapping model that fills in the gap between design and implementation.

As presented in the metamodel in Figure 3.7, there are two types of essential actions, **Input Events** and **Output Events**. Input Events can be further subdivided into **User Input Events**, that represent interactions of the user on the editor (*e.g.,* pressing a button) or complex interactions using frameworks such as [**108**]; and **Internal Events**, that 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`. Output events, can be subdivided into **Operations**, **Interface Element Operations** and **Language Element Operations**. Language Element Operations, are the CRUD (Create, Read, Update and Delete) operations that need to be mapped to specific modeling frameworks, such as Eclipse or AToMPM. Interface Element Operations, are operations to be performed on an interface element. Specifically if we are connecting to pre-existing interfaces instead of generating its code alongside the interaction rules. Operations, correspond to function calls, *e.g.,* running the simulation, or saving a model.

The event mapping model provides a matrix that translates each essential action to at least one **system action**, a platform-specific function offered by the implementation of the system. Input Events get mapped to listener code that translates the system events into events of the controlling Statechart. Output events, get mapped into the system equivalent function calls. 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.

In Table 3.1, we have the representation of how this mapping is defined between the GoL editor specification and a framework that runs the editor on a desktop browser with a specific Statechart library (SCCD), for all the input events of the editor. In it, we can see

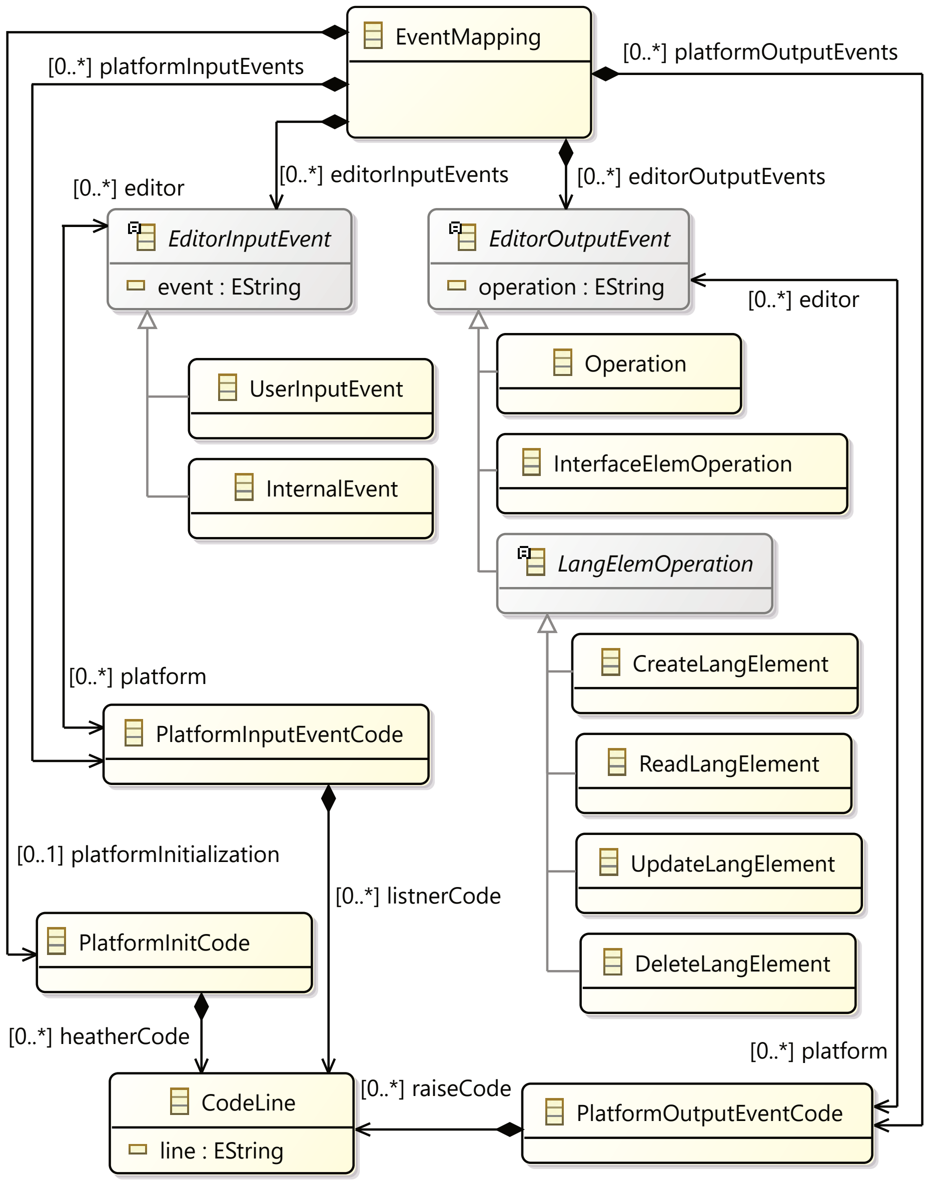


Figure 3.7. Event mapping metamodel

how multiple mappings from or in the shown case to the same event. Table 3.2 shows the representation of the mapping for the GoL output events, where the complexity of the actual operations are encapsulated in a library of functions, and in the mapping we only need to define the called function and any parameters it may require.

Because the event mapping model encapsulates the operationalization requirements for a specific platform, we also use the event mapping model to define any initialization code required by that specific platform to execute the defined mappings, *e.g.,* disabling the right mouse button behavior so it can be properly set by the interaction rules.

Editor Events				Platform Events
press	select	undo	_finish	
▨	▨			document.addEventListener("click", function() { GoL.Interaction.prototype.triggerEvent(event,event.target.getAttribute("data-type"), $EditorEventName$;} false);

		▨		document.addEventListener("keydown", function() {
				if(event.keyCode == 27 && !event.repeat){
				GoL.Interaction.prototype.triggerEvent(
				event,$EditorEventName$;} false)}};

			▨	document.addEventListener($SystemEventName$, function() {GoL.Interaction.prototype.triggerEvent(event,$EditorEventName$;} false);

Table 3.1. Input Events Mapping

Editor Events						Platform Events
Lifeadd	Liferem	LightSetRunning	LightSetFinished	LightSetOff	runGoL	
▨						addLife.call(this,"Canvas");
	▨					remLife.call(this,this.event.target);
		▨				lightset.call(this,"Running");
			▨			lightset.call(this,"Finished");
				▨		lightset.call(this,"Off");
					▨	simulateGoL.call(this);

Table 3.2. Output Events Mapping

3.3. Integration Steps

To bring these models together into functioning DSML editor with customized interactions, we address each model in a sequence of discrete steps, as illustrated in Figure 3.8. The discreteness of these steps allows us to tackle each individually. As the requirements to define a DSML editor may change, those changes and evolutions remain localized and the remaining process remains functional and valid. Additionally, Figure 3.8 also indicates the sections where the models and the steps to process them are discussed and detailed in this thesis.

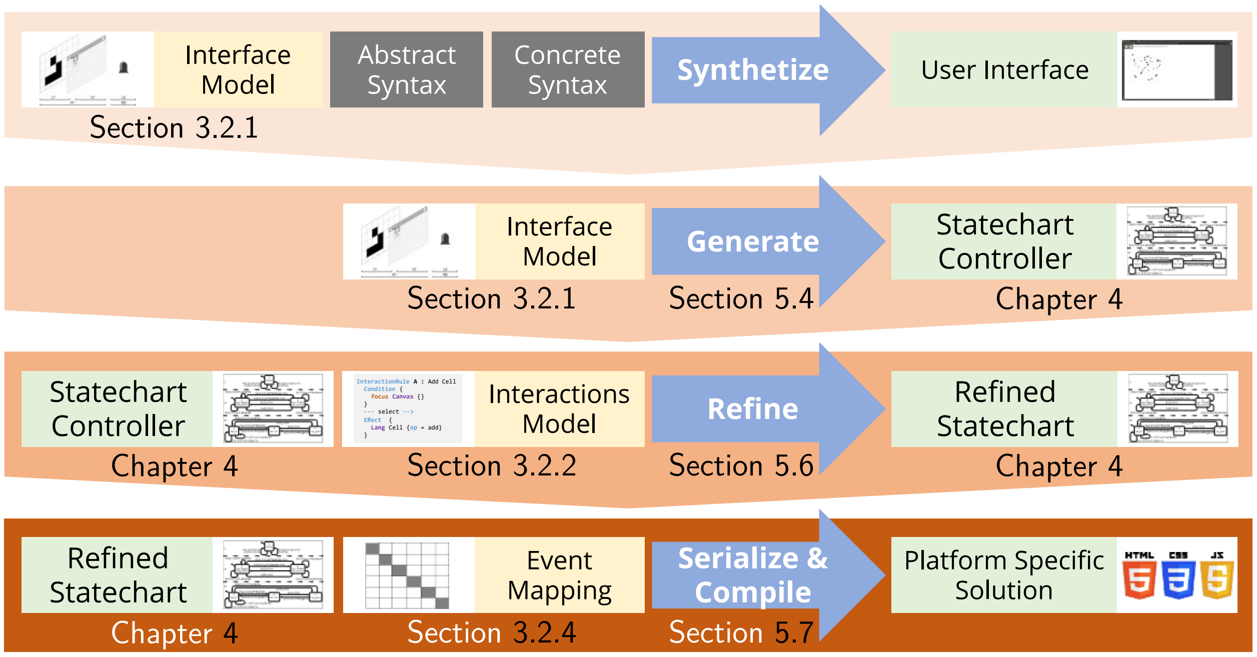


Figure 3.8. Operationalization Steps
and the chapters where they are described

Chapter 4

Statechart Refinement

Don't for heaven's sake, be afraid of talking nonsense! But you must pay attention to your nonsense.

Ludwig Wittgenstein (1947)

When defining how our proposed solution to interaction description will be executed, we will translate it to Statecharts, a formalism with a well defined semantics. The reactive nature of Statecharts, places it as the ideal formalism to do so. Instead of building the statechart that describes all the possible interactions in one big step, we will build it incrementally to promote reuse in our solution and to do so we will apply the notion of refinement.

The Statecharts formalism is considered a defacto standard for modeling reactive systems [**52**]. It describes how objects communicate and how they carry out their own internal behavior [**53**]. It is very often used for simulation, analysis, and code generation, making Statecharts modeling a critical activity in the development of software. These considerations, make it ideal for describing the reactive nature of a DSML editor, and integrate that description in an automated generation process. As in any modern software development, statecharts (models) are developed incrementally following *stepwise refinement* strategies. For example, a developer starts by modeling a high-level Statecharts model, typically with no composite state, and then incrementally adds more detailed behavior in each state or transition. The Statecharts formalism supports incremental modeling by design through, for instance, the addition of orthogonal components in an AND-state or the replacement of a basic state with an OR-state. Each incremental modification, within the bounds of the rules defined in [**116**] and presented in this thesis, can be viewed as a *refinement* of the previous version. Refinement is the process of building a model gradually by making it more and more precise, while verifying that each refined model preserves the abstract behavior [**104**].

The advantage of starting with a more abstract model is that important properties can be defined in a simple model, which is therefore less likely to contain mistakes. Refinements then introduce detail in steps which are guaranteed to preserve the important properties.

There is a considerable amount of Statecharts refinement approaches proposed in the literature where the main concern is to preserve the behavior of the original model [**104, 110, 54, 119, 23, 100**]. This is a necessary condition to refine a model. However, there is almost no guarantee that the structure of the original model is preserved after refinement. This reduces the practicality of applying refinements in systems where Statecharts models are implementation artifacts. Because our refinement process is applied through automated tools, the predictability of structure preservation is essential to maintain said automation with good results.

In this chapter, we present a set of rules that govern Statecharts refinement with the intention of preserving the *structure and reachability* of the original statechart. We allow refinement opportunities of Statecharts which permit the modeler to control refinements such that the resulting statechart is still compatible with the original. The refinements our approach allows must be useful in practice and enable the developer to control what refinements should be allowed in the future. Focusing on this kind of preservation has the potential to increase the predictability of the refinement steps, as well as respecting the design decisions made when it was initially developed. We consider the refinement process to be purely additive: *i.e.,* no removal of elements from the original is allowed. One major advantage of the refinement technique we propose is that the preservation of the structure and behavior of the original statechart is guaranteed by construction: no model checking technique is needed. Refinement is not meant as a replacement for arbitrary modification, which follows an entirely distinct process that should be performed on the original statechart rather than on the refined one. These rules must also be focused on providing realistic and usable boundaries, so as not to constrain the modeler too much. From a practical point of view, our rules are best suited for a stepwise refinement development methodology that we illustrate on a real example. A more general description and a brother analysis of the use of this approach was published in [**116**].

4.1. Formal Definition of Statecharts

As defined in [**52**], a statechart model is defined as a tuple $SC = \langle S,T,E,V,en,ex,in,I \rangle$. It consists of four sets of states $S = S_B \cup S_{OR} \cup S_{AND} \cup S_{pseudo}$, transitions T, events E, and variables V. E denotes both external events provided as stimuli from the context outside the statechart (the queue) and signals of internal events generated by the statechart. We

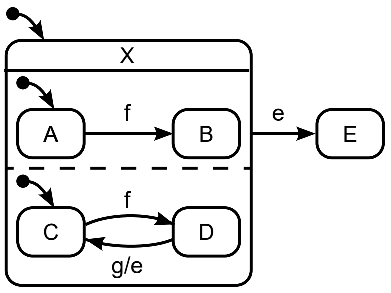


Figure 4.1. An example of a statechart

consider time events and the null event φ just like any other event in E. States can be basic-states (S_B), OR-states (S_{OR}), AND-states (S_{AND}), or pseudo-states[1] (choice, fork, join, and history) $S_{pseudo} = S_{choice} \cup S_{fork} \cup S_{join} \cup S_{history}$. Basic states are primitive states that model an atomic execution. The system stays in a basic state until an outgoing transition is triggered. OR-states are composite states that encapsulate a substatechart. AND-states define orthogonal components where substatecharts can be executed in parallel, *i.e.,* more than one state can be active at a time. Pseudo-states are control-flow nodes that are transient and that augment the expressiveness of transitions.

The relation $in \subseteq S \times S$ denotes the acyclic containment relationship which must be non-empty for OR-states and AND-states. Furthermore, history states are only contained in OR-states, *i.e.,* $\forall s \in S$, $h \in S_{history}$ such that $(s,h) \in in$, then $s \in S_{OR}$. To illustrate the definition, consider the hierarchical statechart represented graphically in Figure 4.1. At the root, it contains the OR-state X and basic state E. X contains two AND-states, one containing basic states A and B, and the other C and D.

A transition is a tuple $t = (s,e,x,g,s')$ where $s, s' \in S$ are the source and target states of the transition, $e,x \in E$ are the triggering and output events respectively and $g : \Xi \rightarrow \{true,false\}$ is a guard predicate for t that acts over the set of substitutions Ξ of variables V into their values. For convenience we write $var(g)$ to denote the variables of V that are used by guard g. Let $src,tar : T \rightarrow S$ denote the source and target state of a transition[2] respectively. Additionally, if $src(t) \in S_{fork}$ or $tar(t) \in S_{join}$, then $t = (src(t),\varphi,\varphi,true,tar(t))$ is unguarded *i.e.,* $\forall \xi \in \Xi, g(\xi) = true$, and does not broadcast any event. Finally if $src(t) \in S_{choice}$, then t must be mutually exclusive with other transitions leaving $src(t)$. In Figure 4.1, the transition from D to C is unguarded, triggered by event g, and it outputs event e.

As defined in [**55**], a configuration of a statechart is a maximally legal snapshot before or after a step. We formally define a configuration of SC as $(\bar{S},\xi,\eta) \in \mathsf{CONF}_{SC}$, where $\bar{S} \subseteq S$, $\xi : V \rightarrow Value \in \Xi$ is a substitution function assigning values to the variables

1. Although they are part of Harel Statecharts, we do not consider junctions and conditionals since they are simply syntactic shortcuts to reduce the number of transitions with similar enabling events and guards.
2. We do not consider hyperedges for transitions because their semantics is emulated with fork and join pseudo-states.

of the statechart, and η is the set of the last states that were active in all OR-states with history. $I = (S_\odot,\xi_\odot,\eta_\odot) \in \mathsf{CONF}_{SC}$ is the initial configuration representing the default states, initial value of variables, and initial history which consists of the default states of all OR-states containing a history pseudo-state. States can define entry and exit actions $en,ex : S \times \Xi \rightarrow \Xi$ which can modify the previous value of variables in V. In Figure 4.1, $S_\odot = \{\mathsf{X},\mathsf{A},\mathsf{C}\}$. Although not illustrated in the figure, we suppose that each basic state has an entry action that prints the name of the state and an empty exit action. In this case, no variables are used.

We say that a transition $t = \left(s,e,x,g,s'\right)$ is *enabled* for a configuration $(\bar{S},\xi,\eta)$ where $s \in \bar{S}$, if $g(\xi) = true$ and e is the next event in the event queue. In case $e = \varphi$, only the fact that $g(\xi) = true$ enables t. When transition t is enabled for a configuration $(\bar{S},\xi,\eta)$, we write $enabled(t,(\bar{S},\xi,\eta))$. For example given the initial configuration in Figure 4.1, if f is the first event in the queue, then the transitions outgoing from A and C are enabled. For our work, we rely on the Statecharts *outer-first step semantics* as defined in [**55**]: if two transitions are in conflict (enabled at the same time, but not in different orthogonal components of the same OR-state), then the transition outgoing from the outermost state of the currently active state hierarchy is the one enabled. In the example, the two transitions labeled with f are not in conflict. However, if the transition labeled with e were triggered by event f as well, then the three transitions would be in conflict. With outer-first semantics, only the latter would be enabled.

We also require the notion of sequential application of steps that consumes a list of events from a queue, which we denote *bigstep*. More formally, assume a *step* relation for statechart SC is defined such that $\xrightarrow{step} \subseteq \mathsf{CONF}_{SC} \times E \times \mathsf{CONF}_{SC}$. Let also Q_{SC} denote the set of all queues consisting of events from SC. We define the *bigstep* relation for SC as $\xrightarrow{bigstep} \subseteq \mathsf{CONF}_{SC} \times \mathsf{Q}_{SC} \times \mathsf{CONF}_{SC}$ to be the smallest bigstep relation satisfying:

$$\frac{}{(\bar{S},\xi,\eta),\mathsf{Q}_\emptyset \xrightarrow{bigstep} (\bar{S},\xi,\eta)}$$

$$\frac{(\bar{S},\xi,\eta),e \xrightarrow{step} (\bar{S}',\xi',\eta'), \quad (\bar{S}',\xi',\eta'),Q \xrightarrow{bigstep} (\bar{S},\xi,\eta)}{(\bar{S},\xi,\eta),e :: Q \xrightarrow{bigstep} (\bar{S},\xi,\eta)}$$

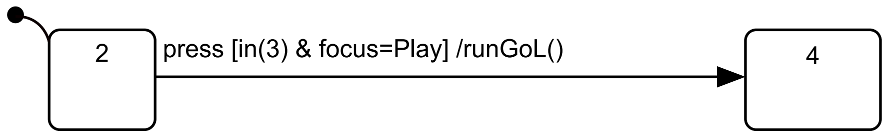
In this notation, $e :: Q \in \mathsf{Q}_{SC}$ is a queue having as first element $e \in E$. Also, $\mathsf{Q}_\emptyset$ denotes the empty queue. When executing a statechart, its input is the sequence of external events from the queue and the output consists of the signals produced by the entry and exit actions of states. New external events received are appended to the end of the queue. The input events the statechart reacts to and the output it produces in its actions are often referred to as the *external interface* of the statechart [**100**].

We briefly describe the $\xrightarrow{step} \subseteq \mathsf{CONF}_{SC} \times E \times \mathsf{CONF}_{SC}$ relation for a statechart SC, following the definition in [**55**]. When an event e is the first in the queue, it is removed from that queue and processed by the statechart. Assuming SC is in a configuration $c = (\bar{S},\xi,\eta) \in \mathsf{CONF}_{SC}$, all non-conflicting transitions $t \in T$ enabled by the configuration are executed simultaneously. The execution of a transition $t = (s,e,x,g,s')$ entails the execution of the exit action of state s under substitution ξ, leading to an intermediate updated configuration $c'' = (\bar{S},\xi'',\eta)$. The new state s' is entered leading to the execution of the entry actions of state s' which, in turn, updates configuration c'' to $c' = (\bar{S}',\xi',\eta')$ – where $S' = S''\backslash\{s\} \cup s'$, ξ' is a new substitution resulting from executing the exit actions of s' under substitution ξ' and η' is the updated history of the last exited states.

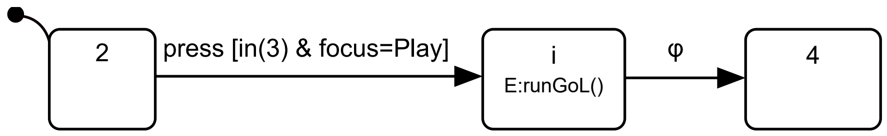
We illustrate the step semantics with the example in Figure 4.1. Suppose that we provide the queue $Q = \langle \mathsf{f},\mathsf{g} \rangle$. The initial configuration can produce the string AC after the execution of the entry actions of the default states. f being the first event in the queue, the transitions outgoing from A and C are enabled. The first *bigstep* completes by outputting the string BD. The queue now only consists of the event g, which initiates the second *step*. The statechart then produces the string C and event e is prepended to the queue. This entails a third *step* to produce the string E, which completes the second *bigstep*.

4.1.1. Actions in states vs. transitions

Although, in our formalization, actions are encoded in states as entry or exit actions, in practice we perform actions on the transitions. This is possible if we consider transitions as a shorthand for a static pattern of triggering transition into a state that encodes the actions with an outgoing transition triggered by an empty event. We can see the differences of these



(a) GoL transition with effect applying actions



(b) GoL transition with effects as entry or exit actions of states

Figure 4.2. Equivalence between encoding actions in transitions and in states.

two approaches in practice in Figure 4.2, where we show how we implement a transition in Figure 2(a) and how it should be interpreted for the formal framework in Figure 2(b). With this example we can see that both paths, although different, they will produce the same operational results from a symbolic point of view. This only applies if we restrict the imaginary state i to only having exactly one incoming and one outgoing transitions, and the exit transition is triggered by the empty event (φ) without any guards. In this configuration the actions can be either in the entry or exit actions of state i, as long as whatever option is chosen is consistent throughout the Statechart. This is possible, because the state will be immediately exited, as soon as the entry actions are performed, and if the entry actions are empty, the execution of the exit actions is immediate.

So it comes to reason that there is a static, behavior preserving refactoring of Statecharts that bridges the gap between implementation and formalization, that ensures the compatibility of the implementation with the formalization. In this way we can keep the simplicity of encoding the actions in the transitions and still be compatible with our formalization, while keeping the formalization focused and generic. As the main reason to only formalize Statecharts with only actions in states in the first place was to reduce the size of the formalization while knowing that any statechart outside the formalized format has an equivalent in the formalized format.

4.2. Preservation of Structure and Reachability

The refinement approach we propose abides to basic Object Oriented design principles, such as the open-closed principle (OCP) stating that the model should be "open for extension, but closed for modification" [**85**], the Liskov substitutability principle (LSP) [**77**], and the principle of covariance and contra-variance [**30**]. Our premise is that the refined Statecharts model should utterly preserve the external interface and behavior as perceived by the entities it is collaborating with. All inputs accepted and all outputs produced from these inputs by the original model should be accepted and produced in the same order by the refined model. This means that external systems should still be able to send the same events to the refined statechart when it is in specific configurations. They also expect that during the *bigstep* execution, the refined statechart produces signals as expected from the original. To satisfy this post-condition, the reachability of configurations should be preserved. Masiero *et al.* [**82**] defined the notion of reachability for Statecharts in a way similar to Petri nets. A configuration is reachable if there is a sequence of events, starting from the initial configuration, that enables transitions between intermediate state configurations. Informally, **reachability preservation** means that, given a sequence of events in the external stimuli queue, every configuration reached in the original statechart shall be reached in the same order in the refined statecharts, where substates are allowed. Consequently, the refined statechart can at least receive the same queues of events and output at least the same signals as the original would. This is analogous to the principles of covariance and contra-variance between overridden methods in sub-classes in Object Oriented design, which is considered well-formed and safe. Reachability preservation is needed to make sure that any system interacting with the original statechart can still interact with the refined one, given the same queue, without requiring any modification, to conform to the LSP.

Furthermore, we also require that there is a mapping from the refined statechart to the original one by preserving states, transitions, and state hierarchy. This is needed because, although two statecharts may be behaviorally equivalent without sharing the same structure [**78**], the aim of the refinements we propose is tied to the practicality and usability in development, maintenance, and debugging activities. Informally, **structural preservation** means that any state in the original statechart must be present in the refined statechart, where the refined state may be composite. Also, for any transition connecting two states in the original statechart, there must be a path of transitions connecting the corresponding states in the refined statechart. Structure preservation abides to the OCP, which reduces future developments efforts on the model. Therefore, refinements must preserve the structure of the original statechart and the reachability of its configurations.

4.3. Refinement Relation

Statechart refinement is a restricted form of modification of the original model such that design decisions are preserved in the refined model. Given a statechart SC, we use the notational abuse $t = s \xrightarrow{e[g]/x} s' \in SC$ to refer to a transition $t = \left(s,e,x,g,s'\right)$ of SC. We also use the notation $s \xrightarrow{+} s' \in SC^+$ to refer to a path of at least one transition in the transitive closure of the transitions of SC. In the following definitions $SC_o = \langle S_o,T_o,E_o,V_o,en_o,ex_o,in_o,I_o\rangle$ and $SC_r = \langle S_r,T_r,E_r,V_r,en_r,ex_r,in_r,I_r\rangle$ denote the original and the refined statecharts, respectively.

Definition 1 (Statecharts Refinement)**.** A statechart refinement is a triple (SC_o,R,SC_r), denoted $SC_o \overset{R}{\preceq} SC_r$, where $R \subseteq (S_o \cup T_o) \times (S_r \cup T_r)$ is a binary relation subject to the conditions that follow. The set of all Statecharts refinements in called $\mathfrak{R}$.

State and Transition Preservation:

$$\forall st_r \in ST_r, \exists! st_o \in ST_o : (st_o,st_r) \in R$$
$$\text{where } ST_i = S_{B_i} \cup S_{OR_i} \cup S_{AND_i} \cup S_{pseudo_i} \cup T_i \text{ for } i = o,r \tag{4.3.1}$$

Hierarchy Preservation:

$$\forall (s_o,s_r), (s'_o,s'_r) \in R,\ (s_o,s'_o) \in in_o \Rightarrow (s_r,s'_r) \in in_r^+$$
$$\text{where } in_r^+ \text{ is the transitive closure of } in \tag{4.3.2}$$

Event and Variable Preservation:

$$E_o \subseteq E_r \wedge V_o \subseteq V_r \tag{4.3.3}$$

Connectivity Preservation:

$$\text{if } \exists s_o \xrightarrow{e[g]/x} s' \in SC \text{ and } (s_o,s_r),(s'_o,s'_r) \in R \text{ then either } \exists s_r \xrightarrow{e[g]/x} s'_r \in SC_r$$
$$\text{or } \exists s_r \xrightarrow{e[g]} s''_r \xrightarrow{\varphi[true]/x} s'_r \in SC_r \text{ or } \exists s_r \xrightarrow{e[g]} s''_r \xrightarrow{+} s'''_r \xrightarrow{\varphi[true]/x} s'_r \in SC_r^+,$$
$$\text{where } (t_o,t_r) \in R \text{ and } t_o \in T_o, t_r \in T_r \tag{4.3.4}$$

State Reachability Preservation:

$$\forall Q_o, \in \mathsf{Q}_{SC_o}, \text{ if } I,Q_o \xrightarrow{bigstep} (\bar{S},\xi,\eta)$$
$$\text{then } \exists Q_r \in \mathsf{Q}_{SC_r} \text{ such that } Q_o \overset{R}{\preceq} Q_r \text{ and } I_r,Q_r \xrightarrow{bigstep} (\bar{S}_r,\xi_r,\eta_r),$$
$$\text{where } \forall s \in \bar{S}, s_r \in \bar{S}_r, (s,s_r) \in R \tag{4.3.5}$$

In the notation used above, we define $Q_o \overset{R}{\preceq} Q_r$ to mean that Q_o is a subsequence of Q_r, imposing that the order of the events in Q_o is preserved in Q_r.

Let us now informally describe the refinement relation above. Condition (4.3.1) ensures that any state or transition in SC_r is mapped back to exactly one state or transition in SC_o. Therefore the inverse refinement R^{-1} is a surjection. Condition (4.3.2) ensures that state hierarchy is preserved. Condition (4.3.3) ensures that all original events and variables must be preserved in the refined statechart. A consequence of Condition (4.3.4) is that if there is a transition from s_o to s'_o in SC_o, then the refinement can either preserve this transition or there must be a SC_r, where s_r is the refined version of s_o and s'_r is the refined version of s'_o. In this path, the transition outgoing from s_r must preserve the original event and guard, and the transition incoming to s'_r must have no guard or event and output the original output event. Through transitivity we can preserve the connectivity of any path defined in SC_o. Condition (4.3.5) guarantees that states that are reachable during execution can still be reachable after refinement. When the above conditions are met, we say that SC_r preserves the *structure and reachability* of SC_o.

Symbolic analysis of entry and exit actions is outside the scope of the work presented in [**116**] and this thesis, therefore we do not enforce a constraint between ξ_o and ξ_r. Nevertheless, because we rely on the identity to refine all elements of the statechart that are not explicitly refined in a rule, we assume that the original entry and exit actions of every state in the original statechart are preserved in the refined states. With this assumption, the sequence of all outputs produced by the original statechart shall be preserved by the refined statechart.

Consider the statechart in Figure 4.3 that is a possible refinement of the one in Figure 4.1. We notice that every state and transition path is preserved. For instance, state E remains identical. Basic state A is still present but is now an OR-state containing two substates. OR-state X now contains an additional AND-state. States D and C are still connected by a

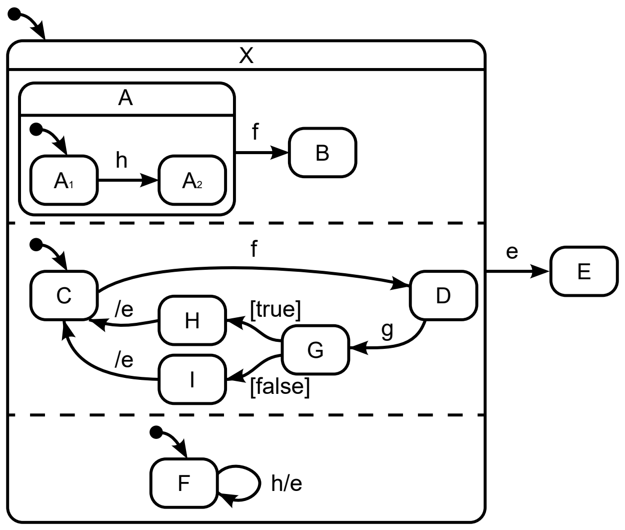


Figure 4.3. A statechart refining the one in Figure 4.1 according to Definition 1

cyclic path of transitions. It is clear that Conditions 4.3.1–4.3.4 are satisfied, therefore the structure of the original statechart is preserved.

To verify the reachability preservation of the refined statechart, we can see that, given any queue, the trace output by the original statechart (the names of basic states entered) is a subsequence of the one output by the refined statechart, where substates are allowed. For example, given the queue $Q = \langle f,g \rangle$, the original statechart outputs the sequence AC BD C E and the refined statechart outputs A_1CF BD G H C E. The output of the original statechart is then a subsequence of the output of the refined statechart, a constraint that is implied[3] by Condition 4.3.5. Nonetheless, this statechart allows for new behaviors that were not possible in the original statechart. For instance, if we provide the queue $Q = \langle h \rangle$ to the refined statechart, it will output: A_1CF A_2CF E. This shows that the refined statechart allows for more behaviors than the original, while preserving the original behavior. Consequently, the Statecharts refinement defined in Definition 1 allows for the following possible modifications: new events can be interjected between events from the original (*i.e.,* new steps are possible), new states can be added, and new transitions can be added. However, these additions are subject to specific constraint rules in order to reflect the intention of predictive refinement in the context of incremental modeling.

3. To fully demonstrate that Condition 4.3.5 is preserved by the refinement we would have to extensively show the condition holds for all input queues. This is impossible due to the fact that queues can be arbitrarily large. Rather, we demonstrate formally in A that if the modeler only makes use of the refinement rules we propose in Section 4.4, then Condition 4.3.5 always holds for any refinement and input queue.

4.4. Refinement Rules for Statecharts

In the previous section, we laid out the requirements for refinement rules. In this section, we define the rules that the developer is bound to, so he can produce a well-formed refined statechart.

In the definitions that follow we assume an original statechart to be refined $SC = \langle S,T,E,V,en,ex,in,I\rangle$ and the statechart resulting from the refinement $SC_r = \langle S_r,T_r,E_r ,V_r,en_r,ex_r,in_r,I_r\rangle$. Refinement rules are additive: SC_r is obtained by introducing a new statechart $\hat{SC} = \langle \hat{S},\hat{T},\hat{E},\hat{V},\hat{en},\hat{ex},\hat{in},\hat{I}\rangle$ in SC. For each refinement rule, $\hat{SC}$ is subject to constraints that serve as precondition before applying a specific rule. All rules are intended to be statically verified, while constraints specify assumptions on the dynamic behavior of the statechart.

4.4.1. Rules for State Refinement

Constraint 1 (Constraint on Actions)**.** $\forall s_r \in S_r, \forall \xi \in \Xi$, let $\xi' = en_r(s_r,\xi)$. Then if $\forall t = (s,e,x,g,s') \in T, \exists \xi" \in \Xi$ such that $g(\xi") = false$ where $s,s' \in S$ and $e,x \in E$, then $\forall v \in var(g)$ we have that $\xi'(v) = \xi(v)$. The same holds for $\xi' = ex_r(s_r,\xi)$.

This constraint states that the new entry and exit actions of states in SC_r cannot modify the values of any variable $v \in V$ if v is used to restrict a guard of a transition in SC. Nevertheless, refined entry and exit actions can modify newly introduced variables $v_r \in V_r \backslash V$. This is needed to ensure the refinement does not prevent the enabling of an already existing transition.

Rule 1 (Action Rule)**.** Let SC be a statechart and $s \in S_B \cup S_{OR} \cup S_{AND}$ be a state of SC. Let also $\hat{SC} = \langle \{s\},\emptyset,\emptyset,\hat{V},\hat{en},\hat{ex},\emptyset,\emptyset\rangle$ be another statechart that satisfies Constraint 1. Refining entry and exit actions of state s produces a statechart SC_r such that: $S_r = S$, $T_r = T$, $E_r = E$, $V_r = V \cup \hat{V}$, $en_r = en \cup \hat{en}$, $ex_r = ex \cup \hat{ex}$, $in_r = in$ and $I_r = I$.

This rule adds further action in the entry and exit actions of s, possibly modifying values of newly introduced variables. As seen in Figure 4.4, the statechart remains structurally the same, but the refined state has an additional action that does not manipulate a variable used in an existing guard. The refinement keeps a mapping between the original state and the updated state, as denoted by the red doted line.

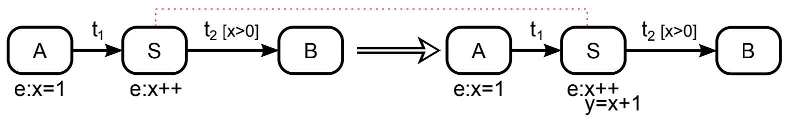


Figure 4.4. Refinement of a state into a state

Constraint 2 (Constraint on Broadcast)**.** $\forall t_r = (s_r,e_r,x_r,g_r,s'_r) \in T_r \setminus T$, if $x_r \neq \varphi$ then either $e_r \neq \varphi$ or $\exists v_r \in V_r, \exists \xi \in \Xi$ such that $g_r(\xi(v_r)) = false$.

This constraints states that newly introduced transitions cannot broadcast an event unless they require a non-null event to enable them or have a guard that may disable them for some configuration. This is needed in an outer-first semantics setting in order to prevent existing states to be unreachable in SC_r that were previously reachable in SC. To illustrate this situation, suppose that, in the statechart SC_r in Figure 4.3, the transition between A_1 and A_2, enabled by event h, outputs the event e. If we provide the queue $Q = Q_r = \langle f,e \rangle$, SC outputs BD E and SC_r outputs BD E. Thus B is still reachable after refinement. However, if the transition between A_1 and A_2 is now enabled by φ, B is no longer reachable with any Q_r.

Rule 2 (Basic-to-OR State Rule)**.** Let SC be a statechart, $s \in S_B$ be a basic state of SC, and $\hat{SC}$ be another statechart where all of its components are disjoint from the corresponding ones in SC. Assume $\hat{SC}$ satisfies Constraints 1–2. Refining state s into an OR-state containing statechart $\hat{SC}$ produces a statechart SC_r such that:

(1) $S_{r_B} = S_B \setminus \{s\} \cup \hat{S}_B$, $S_{r_{OR}} = S_{OR} \cup \{s\} \cup \hat{S}_{OR}$, $S_{r_{AND}} = S_{AND} \cup \hat{S}_{AND}$ and $S_{r_{pseudo}} = S_{pseudo} \cup \hat{S}_{pseudo}$

(2) $T_r = T \cup \hat{T}$, $E_r = E \cup \hat{E}$, $V_r = V \cup \hat{V}$, $en_r = en \cup \hat{en}$, $ex_r = ex \cup \hat{ex}$

(3) $in_r = in \cup \bigcup_{\hat{s} \in \hat{S}} (s,\hat{s}) \cup \hat{in}$ where $\hat{s}$ is not contained in any other state

(4) $I_r = \begin{cases} (S_\odot \cup \hat{S}_\odot, \xi_\odot \uplus \hat{\xi}_\odot, \eta_\odot \cup \hat{\eta}_\odot) & \text{if } s \in S_\odot \\ (S_\odot, \xi_\odot \cup \hat{\xi}_\odot, \eta_\odot) & \text{otherwise} \end{cases}$ where $I = (S_\odot,\xi_\odot,\eta_\odot)$ and $\hat{I} = (\hat{S}_\odot,\hat{\xi}_\odot,\hat{\eta}_\odot)$ [4]

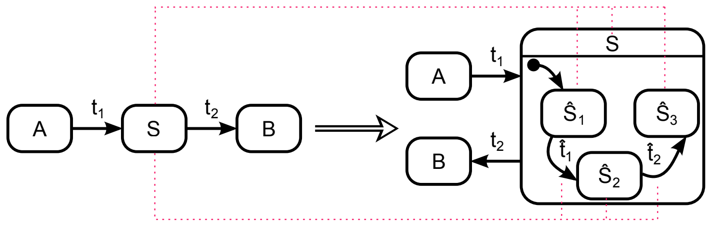


Figure 4.5. Refinement of a basic state into an OR-state

Figure 4.5 illustrates the refinement of a basic state into an OR-state as per Rule 2. The statechart SC is on the left-hand side and the SC_r is on the right-hand side. The refined state is removed from the set of basic states and added to the set of OR-states of the original

4. The $\uplus$ operator enforces that values coming from the variable substitutions on the right of the operator override their counterparts on the left of the operator, thus guaranteeing the resulting substitution is still a function.

statechart. All the states of statechart $\hat{SC}$ become part of the newly created OR-state. The default states of $\hat{SC}$ become part of the default state of the original statechart SC whenever state s being refined is default. A consequence of having $\hat{SC}$ disjoint from SC is that new states in $\hat{S}$ and transitions in $\hat{T}$ only work on local variables in $\hat{V}$, *i.e.,* the new ones. This also creates a mapping R, between the corresponding elements of SC and SC_r, as per Definition Rule 1. The state s is mapped to the new OR-state s, along with all elements of $\hat{SC}$, as they are introduced in the refinement of s. In Figure 4.5 the dotted lines only shows the refinement mapping R for the new elements in SC_r, since all elements of SC_r are mapped by their identity in SC.

Rule 3 (OR-to-AND State Rule)**.** Let SC be a statechart, $s \in S_{OR}$ be an OR-state of SC, and $\hat{SC}$ be another statechart where all of its components are disjoint from the corresponding ones in SC. Assume $\hat{SC}$ satisfies Constraints 1–2. Let $T_{in} = \{t \mid src(t) = s\}$ and $T_{out} = \{t \mid tar(t) = s\}$. Refining state $s \in S_{OR}$ into an AND-state containing statechart $\hat{SC}$ produces a statechart SC_r such that:

(1) $S_{r_B} = S_B \cup \hat{S}_B$, $S_{r_{OR}} = S_{OR} \cup \hat{S}_{OR} \cup \{s_{or}\}$, $S_{r_{AND}} = S_{AND} \cup \hat{S}_{AND} \cup \{s_{and}\}$ and $S_{r_{pseudo}} = S_{pseudo} \cup \hat{S}_{pseudo}$ such that $s_{or} \notin S_{OR} \cup \hat{S}_{OR}$ and $s_{and} \notin S_{AND} \cup \hat{S}_{AND}$

(2) $T_r = T \cup \hat{T} \cup \{(s_{and},\hat{e},\hat{x},s') \mid \exists(s,e,x,s') \in T_{in}$ where $\hat{e} = e,\hat{x} = x$ for some $s' \in S\} \cup \{(s',\hat{e},\hat{x},s_{and}) \mid \exists(s',e,x,s) \in T_{out}$ where $\hat{e} = e,\hat{x} = x$ for some $s' \in S\} \setminus T_{in} \setminus T_{out}$

(3) $E_r = E \cup \hat{E}$, $V_r = V \cup \hat{V}$, $en_r = en \cup \hat{en}$, $ex_r = ex \cup \hat{ex}$

(4) $in_r = in \setminus \{(p,s)\} \cup \{(s_{and},s),(s_{and},s_{or}),(p,s_{and})\} \cup \bigcup_{\hat{s}\in\hat{S}}(s_{or},\hat{s}) \cup \hat{in}$ where $\hat{s}$ is not contained in any other state and $p \in S$

(5) $I_r = \begin{cases} (S_\odot \cup \hat{S}_\odot,\xi_\odot \uplus \hat{\xi}_\odot,\eta_\odot \cup \hat{\eta}_\odot) & \text{if } s_{or} \in S_\odot \\ (S_\odot,\xi_\odot \cup \hat{\xi}_\odot,\eta_\odot) & \text{otherwise} \end{cases}$ where $I = (S_\odot,\xi_\odot,\eta_\odot)$ and $\hat{I} = (\hat{S}_\odot,\hat{\xi}_\odot,\hat{\eta}_\odot)$

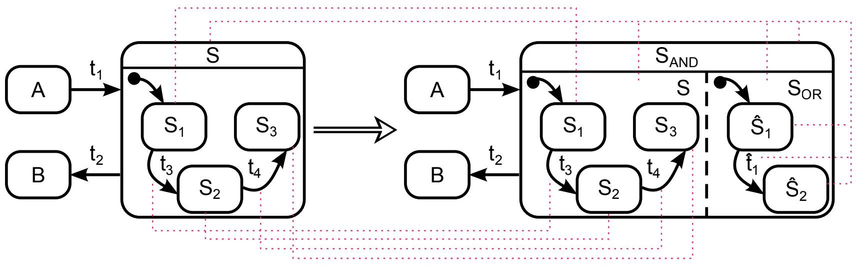


Figure 4.6. Refinement of an OR-state into an AND-state

Figure 4.6 illustrates the refinement of an OR-state into an AND-state as specified in Rule 3. In SC_r on the right-hand side, there is a new AND-state S_{AND} that contains all the original OR-states as well as a new OR-state containing $\hat{SC}$. Note that step 2 makes sure all transitions incoming to s (*e.g.*, t_1) and all transitions outgoing from s (*e.g.*, t_2) are rerouted so that S_{AND} is their source or target, respectively. The remaining elements of SC_r remains unchanged otherwise. In this case the R in Figure 4.6 maps $\hat{SC}$, the created AND-state and OR-state alongside $\hat{SC}$ and s_r to state s.

Rule 4 (AND-State Rule)**.** Let SC be a statechart, $s \in S_{AND}$ be an AND-state of SC, and $\hat{SC}$ be another statechart where all of its components are disjoint from the corresponding ones in SC. Assume $\hat{SC}$ satisfies Constraints 1–2. Refining state $s \in S_{AND}$ into an AND-state containing statechart $\hat{SC}$ in an additional region produces a statechart SC_r such that:

(1) $S_{r_B} = S_B \cup \hat{S}_B$, $S_{r_{OR}} = S_{OR} \cup \hat{S}_{OR} \cup \{s_{or}\}$, $S_{r_{AND}} = S_{AND} \cup \hat{S}_{AND}$ and $S_{r_{pseudo}} = S_{pseudo} \cup \hat{S}_{pseudo}$ such that $s_{or} \notin S_{OR} \cup \hat{S}_{OR}$

(2) $T_r = T \cup \hat{T}$, $E_r = E \cup \hat{E}$, $V_r = V \cup \hat{V}$, $en_r = en \cup \hat{en}$, $ex_r = ex \cup \hat{ex}$

(3) $in_r = in \cup \{(s,s_{or})\} \cup \bigcup_{\hat{s} \in \hat{S}}(s_{or},\hat{s}) \cup \hat{in}$ where $\hat{s}$ is not contained in any other state

(4) $I_r = \begin{cases} (S_\odot \cup \hat{S}_\odot, \xi_\odot \uplus \hat{\xi}_\odot, \eta_\odot \cup \hat{\eta}_\odot) & \text{if } s \in S_\odot \\ (S_\odot, \xi_\odot \cup \hat{\xi}_\odot, \eta_\odot) & \text{otherwise} \end{cases}$ where $I = (S_\odot, \xi_\odot, \eta_\odot)$ and $\hat{I} = (\hat{S}_\odot, \hat{\xi}_\odot, \hat{\eta}_\odot)$

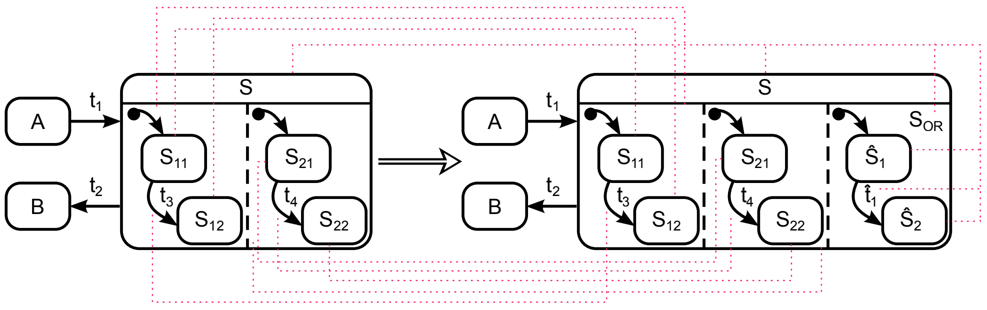


Figure 4.7. Refinement of an AND-state into an AND-state

Figure 4.7 illustrates the refinement of an AND-state into an AND-state as per Rule 4. The new elements of $\hat{SC}$ are encapsulated in a new OR-state s_{or} in the refined statechart SC_r. R maps s_{or} and all of $\hat{SC}$ to s, the original state to refine.

4.4.2. Rules for Transition Refinement

In the following, we denote, for the new statechart to introduce $\hat{SC}$, $\hat{S}_\odot \subseteq \hat{S}$ as the set of states that are not contained in any other state and $\hat{s}_\odot \in \hat{S}_\odot \cap \hat{S}^I$, where $(\hat{S}^I, \hat{\xi}^I, \hat{\eta}^I) = \hat{I}$. We also denote $\hat{S}_{end} \subseteq \hat{S}$ where any $\hat{s}_{end} \in \hat{S}_{end}$ is not the source of any transition in $\hat{T}$ and is not contained in any other state.

Constraint 3 (Constraint on Paths)**.** $\exists Q \in \mathsf{Q}_{\hat{SC}}, \hat{s}_{end} \in \hat{S}_{end}$ such that $\hat{I}, Q \xrightarrow{bigstep} (\{\hat{s}_{end}\}, \hat{\xi}, \hat{\eta})$.

This constraint states that the newly introduced statechart $\hat{SC}$ must have at least one state $\hat{s}_{end}$ without outgoing transitions that must be reachable from the default state. An example is depicted in Figure 4.8. The constraint ensures that s' will still be reachable. Although this is a strong constraint, this is only needed for transition refinement rules.

Rule 5 (Transition Rule)**.** Let SC be a statechart containing a transition $t = s \xrightarrow{e[g]/x} s' \in T$ where $s \in S \setminus (S_{fork} \cup S_{history})$ and $s' \in S \setminus (S_{fork} \cup S_{join})$. Let $\hat{SC}$ be another statechart where all of its components are disjoint from the corresponding ones in SC and that satisfies Constraints 1–3. Refining transition t into statechart $\hat{SC}$ produces a statechart SC_r such that:

(1) $S_r = S \cup \hat{S}$

(2) $T_r = T \setminus \{t\} \cup \hat{T} \cup \{s \xrightarrow{e[g]/\varphi} \hat{s}_\odot\} \cup \bigcup_{\hat{s}_{end} \in \hat{S}_{end}} \{\hat{s}_{end} \xrightarrow{\varphi[true]/x} s'\}$

(3) $in_r = \begin{cases} in \cup \hat{in} & \text{if } s \text{ and } s' \text{ do not have any parent} \\ in \cup \bigcup_{\hat{s} \in \hat{S}_\odot} (LCP(s,s'), \hat{s}) \cup \hat{in} & \text{otherwise} \end{cases}$
where $LCP(s,s')$ is the least common parent state containing s and s'

(4) $E_r = E \cup \hat{E}$, $V_r = V \cup \hat{V}$, $en_r = en \cup \hat{en}$, $ex_r = ex \cup \hat{ex}$, $I_r = I$

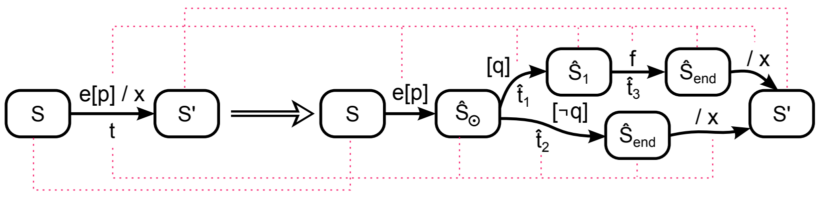


Figure 4.8. Refinement of a transition

Figure 4.8 illustrates the refinement of a transition t from s to s' into a set of states and transitions. The first transition outgoing from s retains the original event and guard in the refined statechart. All incoming transitions to s' must broadcast the original event x. All intermediate states in $\hat{SC}$ must be connected such that there is always a path from s to s'. R maps all these states and transitions to t. Regarding the containment of the states added by the refinement, if either the source s or target s' of t is not contained in any other state, then nor are the states of $\hat{SC}$. If s or s' are contained in the same state, then all states of $\hat{SC}$ are as well. Otherwise, s or s' have a different parent, so the states of $\hat{SC}$ are contained in the least common parent of the s and s'.

Rule 6 (Fork Rule)**.** Let SC be a statechart containing a transition $t = s \xrightarrow{e[g]/x} s' \in T$ where $s \in S \setminus (S_{fork} \cup S_{history})$ and $s' \in S_{fork}$. Let $\hat{SC}$ be another statechart where all of its components are disjoint from the corresponding ones in SC and that satisfies Constraints 1–3. Refining transition t into statechart $\hat{SC}$ produces a statechart SC_r such that:

(1) $S_{r_B} = S_B \cup \hat{S}_B \cup \{s_{fork}\}$, $S_{r_{OR}} = S_{OR} \cup \hat{S}_{OR}$, $S_{r_{AND}} = S_{AND} \cup \hat{S}_{AND}$ and $S_{r_{pseudo}} = S_{pseudo} \cup \hat{S}_{pseudo}$ such that $s_{fork} \notin S_B \cup \hat{S}_B$

(2) $T_r = T \setminus \{t\} \cup \hat{T} \cup \{s \xrightarrow{e[g]/\varphi} \hat{s}_\odot\} \cup \bigcup_{\hat{s}_{end} \in \hat{S}_{end}} \hat{s}_{end} \xrightarrow{\varphi[true]} s_{fork} \cup \{s_{fork} \xrightarrow{\varphi[true]/x} s'\}$

(3) $in_r = \begin{cases} in \cup \hat{in} & \text{if } s \text{ and } s' \text{ do not have any parent} \\ in \cup \bigcup_{\hat{s} \in \hat{S}_\odot} (LCP(s,s'),\hat{s}) \cup \hat{in} & \text{otherwise} \end{cases}$

(4) $E_r = E \cup \hat{E}$, $V_r = V \cup \hat{V}$, $en_r = en \cup \hat{en}$, $ex_r = ex \cup \hat{ex}$, $I_r = I$

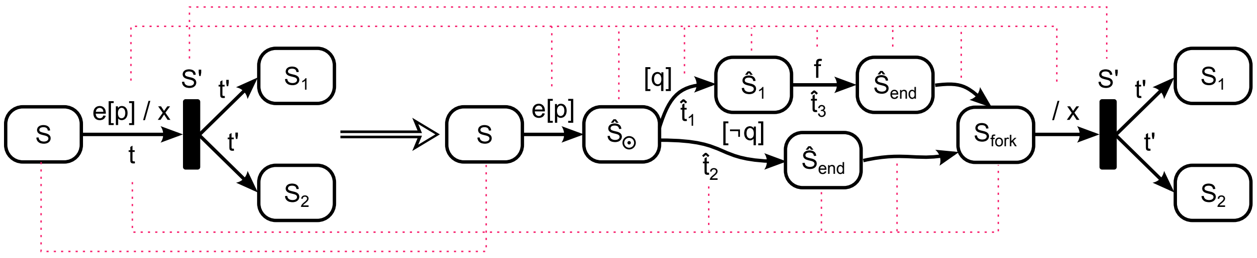


Figure 4.9. Refinement of a transition to a fork

Figure 4.9 illustrates the refinement of a transition as per Rule 6. This is the same rule as for transition refinement in Rule 5, with the following additions. We add an unguarded transition with no output from each ending state in $\hat{S}_{end}$ to a new state s_{fork}. We also add an unguarded transition outputting the same output x as t from s_{fork} to the fork. s_{fork} is needed since a fork can only have one incoming transition. R defines the same mapping as in Rule 5.

4.4.3. Rules for Refinement by Extension

Constraint 4 (Constraint on Transitions)**.** Let $\hat{t}_{start} = (s,\hat{e},\hat{x},\hat{g},\hat{s}_\odot)$ be a transition with $s \in S$, $\hat{e},\hat{x} \in \hat{E}$ and $\hat{s}_\odot \in \hat{S}_\odot \cap \hat{I}$ as in Section 4.4.2. If $\exists s' \in S\ \forall t = (s^+,e,x,g,s') \in T$ where $s^+ \in s \cup in^+(s)$ and $\hat{e} = e$, then $\forall \xi \in \Xi, g(\xi) \rightarrow \neg\hat{g}(\xi)$.

This constraint states that for a given configuration ξ, $\hat{t}_{start}$ cannot be enabled if there is another transition with the same source state that can be enabled with ξ, such as t in Figure 4.10. This is necessary to ensure that the resulting statechart SC_r is still deterministic. Furthermore, this restriction also applies to any transitions inside s when s is an OR-state or an AND-state. This ensures that pre existing segments of the statechart don't become inaccessible as a result of the outer first policy.

Rule 7 (State Extension Rule)**.** Let SC be the statechart we are refining. Let $\hat{SC}$ be another statechart where all of its components are disjoint from the corresponding ones in SC. Let $s \in S \setminus (S_{join} \cup S_{history})$ be the state we want to refine and $\hat{t}_{start} = (s,\hat{e},\hat{x},\hat{g},\hat{s}_{\odot})$ be a new transition restricted by Constraint 4. Refining s by extending it with statechart $\hat{SC}$ produces a statechart SC_r such that:

(1) $S_r = S \cup \hat{S}$

(2) $T_r = T \cup \hat{T} \cup \{\hat{t}_{start}\}$

(3) $in_r = \begin{cases} in \cup \bigcup_{\hat{s}\in\hat{S}_{\odot}}(p,\hat{s}) \cup \hat{in} & \text{if } \exists p \in S \mid (p,s) \in in \\ in \cup \hat{in} & \text{otherwise} \end{cases}$

(4) $E_r = E \cup \hat{E},\ V_r = V \cup \hat{V},\ en_r = en \cup \hat{en},\ ex_r = ex \cup \hat{ex},\ I_r = I$

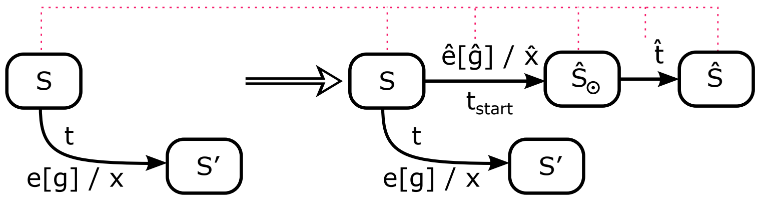


Figure 4.10. Refinement by extending a state with a statechart

Figure 4.10 illustrates a state extension refinement rule as per Rule 7. This rule allows one to create a new transition $\hat{t}_{start}$ from an existing state s to a new statechart $\hat{SC}$ with states that were not in SC. Constraint 4 guarantees that $\hat{t}_{start}$ does not conflict with another transition to keep SC_r deterministic. We then define the mapping R, from all elements of $\hat{SC}$ and the new transition to s.

Rule 8 (Path Refinement Rule)**.** Let SC be the statechart we are refining. Let $\hat{SC}$ be another statechart where all of its components are disjoint from the corresponding ones in SC. Let $s \xrightarrow{+} s' \in SC^+$ be the path starting from the state $s \in S \setminus (S_{join} \cup S_{history})$ we want to refine and $\hat{t}_{start} = (s,\hat{e},\hat{x},\hat{g},\hat{s}_{\odot})$ be a new transition restricted by Constraint 4. We also define another new transition $\hat{t}_{end} = (\hat{s},\varphi,x,g,s')$ for some $\hat{s} \in \hat{S}$, such that the path $\exists \hat{s}_{\odot} \xrightarrow{+} \hat{s} \in \hat{SC}^+$. Refining the path from s to s' with statechart $\hat{SC}$ produces a statechart SC_r such that:

(1) $S_r = S \cup \hat{S}$

(2) $T_r = T \cup \hat{T} \cup \{\hat{t}_{start}\} \cup \{\hat{t}_{end}\}$

(3) $in_r = \begin{cases} in \cup \hat{in} & \text{if } s \text{ and } s' \text{ do not have any parent} \\ in \cup \bigcup_{\hat{s}\in\hat{S}_{\odot}}(LCP(s,s'),\hat{s}) \cup \hat{in} & \text{otherwise} \end{cases}$

(4) $E_r = E \cup \hat{E},\ V_r = V \cup \hat{V},\ en_r = en \cup \hat{en},\ ex_r = ex \cup \hat{ex},\ I_r = I$

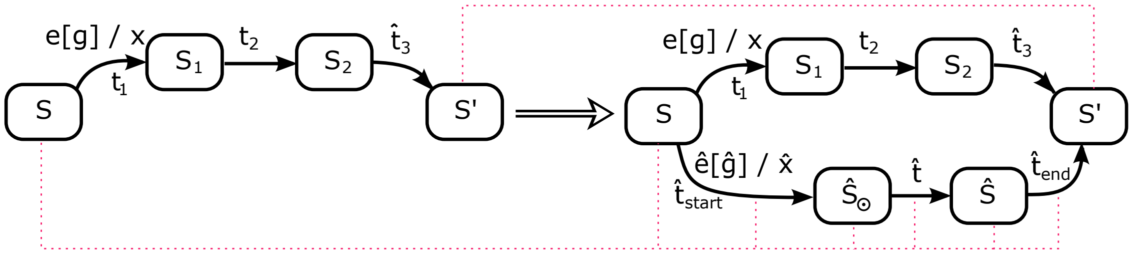


Figure 4.11. Refinement by extending a new path between two states

Figure 4.11 illustrates a path refinement as per Rule 8. In this refinement we extend the ways we can reach state s' from state s without affecting the applicability of the preexisting paths between these two states. For this purpose, the transition t_{start} from s to the starting state $\hat{s}_{\odot}$ of $\hat{SC}$, should not conflict with any outgoing transition from s such as t_1, as in the rule of state extension in Rule 7. The purpose of adding t_{end} from a state in $\hat{SC}$ to s' is to ensure that s' is reachable from this new path. R maps all new elements in SC_r that were not in SC to s.

4.4.4. Refinement of pseudo-states

All pseudo-state refinements are the identity, *i.e.,* they cannot be refined. This is also the case for transitions outgoing from a history state or a fork, and incoming to a join.

4.5. Formal Rule Application

We are interested in applying sequences of refinements to an initial statechart, in order to reach a final design. It is thus of the utmost importance to be able to show that sequences including applications of *refinement rules*, defined in Section 4.4 are statechart refinements, as formally stated in Definition 1. Refinement rules can be applied in isolation in any order.

This section provides the formal foundation and necessary proofs of the soundness of our approach. The reader who is more interested by the implementation and application of our approach may safely skip this section without any prejudice to the understanding of the remaining of the thesis. In what follows, we will only present the headings of our formal definitions and properties, and refer the reader to A for the mathematical proofs. More precisely, the main results of this section are the proofs that refinement rules are formally statechart refinements, as per Definition 1. We will then go on to formally prove that sequences of refinements are also refinements.

Let us then start by formally demonstrating that the refinement rules from Section 4.4 are refinements. We start by building *rule applications* consisting of a triple including only the part of the statechart being refined by the rule, the part of the statechart created by the rule, and a traceability relation between the two. The notion of *local rule application* captures only how the part of the statechart affected by these rules is changed.

Definition 2 (Local Rule Application)**.** Let statechart SC_r be the result of applying a *refinement rule* to a statechart SC.

An application of a state refinement rule (Rules 1–4) or a state extention rule (Rule 7) to a state s of SC is a triple (SC,R,SC_r), where R is a binary relation between s and each new state or transition created by the state refinement rule in SC_r. The five kinds of state refinement rules imply local rule applications as follows:

— *Basic State* (Rule 1): the local rule application (SC,R,SC_r) is such that $R = \{(s,s)\}$.

— *Basic-to-OR State* (Rule 2): the local rule application (SC,R,SC_r) is such that $R = \{(s,s)\} \cup \Big\{(s,s_r) \,|\, s_r \in S_r \setminus S\Big\} \cup \Big\{(s,t_r) \,|\, t_r \in T_r \setminus T\Big\}$.

— *OR-to-AND State* (Rule 3), *AND-to-AND State* (Rule 4) and *State Extension* (Rule 7), the local rule application (SC,R,SC_r) is such that $R = \{(s,s)\} \cup \Big\{(s,s_r) \,|\, s_r \in S_r \setminus S\Big\} \cup \Big\{(s,t_r) \,|\, t_r \in T_r \setminus T\Big\} \cup R_{lid}$ where R_{lid} is the identity relation for states contained in s united with the identity relation for transitions between states contained in s.

A local application of a transition refinement rule (Rules 5–6) on a transition t of SC is a triple (SC,R,SC_r), where $R = \Big\{(t,s_r) \,|\, s_r \in S_r \setminus S\Big\} \cup \Big\{(t,t_r) \,|\, t_r \in T_r \setminus T\Big\} \cup R_{lid}$ and R_{lid} is the identity relation for the source and target states of t united with all states and transitions contained within them.

The local application of the *Path Refinement Rule* (Rule 8) on $s \xrightarrow{*} s'' \in SC^*$, is a triple (SC,R,SC_r), with $R = \Big\{(s,s_r) \,|\, s_r \in S_r \setminus S\Big\} \cup \Big\{(s,t_r) \,|\, t_r \in T_r \setminus T\Big\} \cup R_{lid}$ where R_{lid} is the identity relation of the s and s'' states united with all states and transitions contained within them.

Figures 4.5–4.11 depict examples of state, transition, and path refinement local rule applications. The respective mapping R for each rule is depicted by the red dotted lines.

4.5.1. Soundness

The following lemma ensures that the changes made locally by state, transition, and path refinement rules are formal refinements that satisfy the conditions of Definition 1.

Lemma 1 (Soundness: Local Rule Application is a (local) Refinement)**.** *Refinement rules* create (local) refinements restricted to the elements of the statechart being refined and satisfy the five refinement conditions. More formally, given a statechart SC, and a local rule application (SC_l,R,SC_{lr}), we have that $SC_l \overset{R}{\preceq} SC_{lr} \in \Re$.

Intuitively, a statechart can be refined to itself. The following property ensures that the refinement relation is reflexive. This is an important result we will use subsequently.

Property 1 (Refinement Reflexivity)**.** For any statechart SC we have that $SC \overset{R_{id}}{\preceq} SC \in \Re$, where R_{id} is the union of the identity relations for states and transitions of SC.

It follows from Property 1 that the *copy* operation of a statechart is a refinement: this is the identity refinement. The following defines that the application of a refinement rule is a local rule application *merged* with the identity refinement. This allows us to add to the local refinement all elements of the statechart untouched by the rule and the remaining traceability relations, thus rebuilding the complete application of the rules. Note that the merge operation "glues" the local rule application on top of the identity refinement, while removing the replaced elements resulting from the identity refinement.

Definition 3 (Rule Application)**.** The application of a *state refinement* rule or a *state extension* rule to a statechart SC on state s consists of a merge of the identity refinement $id = SC \overset{R_{id}}{\preceq} SC$ and a local refinement $loc = SC \overset{R_{loc}}{\preceq} SC_r$ of a state s of SC. This is denoted $merge(id, loc) = SC \overset{R}{\preceq} SC_r$ where $R = R_{loc} \cup R_{id} \setminus \{(s,s)\}$ and SC_r is the statechart obtained from applying the refinement rule.
Similarly, the application of a *transition refinement* rule to a statechart SC on a transition t is also a merge of the identity refinement and a local refinement of a transition t of SC, denoted $merge(id, loc) = SC \overset{R}{\preceq} SC_r$ where $R = R_{loc} \cup R_{id} \setminus \{(t,t)\}$.
Finally, the application of a *path refinement* rule to a statechart SC on a path between two states s and s'' is also a merge of the identity refinement and a local refinement of a transition t of SC, denoted $merge(id, loc) = SC \overset{R}{\preceq} SC_r$ where $R = R_{loc} \cup R_{id} \setminus \{(s,s),(s'',s'')\}$.

The following finally shows that a rule application is a refinement by proving that the result of merging an identity refinement with a local rule application refinement of a statechart as done in Definition 3 is still a refinement.

Lemma 2 (Rule Application is a Refinement)**.** Let id be the identity refinement and loc be a local rule application of a *refinement rule.* Then, the rule application $merge(id, loc) = SC \overset{R}{\preceq} SC_r \in \Re$ is a refinement.

Having shown that applying state, transition, or path refinement rules result in formal refinements, we now show that sequences of formal refinements are refinements. We do so by showing the transitivity of statechart refinement in Property 2, which naturally leads to our final result in Property 3.

Property 2 (Refinement Transitivity)**.** If $SC \overset{R_1}{\preceq} SC_{r_1} \in \Re$ and $SC_{r_1} \overset{R_2}{\preceq} SC_{r_2} \in \Re$, then there exists a refinement $SC \overset{R_3}{\preceq} SC_{r_2} \in \Re$.

Property 3 (Sequence of Rule Applications is a Refinement)**.** The composition of two or more rule applications $SC_1 \overset{R_1}{\preceq} SC_2 \in \Re, \ldots, SC_{n-1} \overset{R_{n-1}}{\preceq} SC_n \in \Re, \forall n \geq 3$, denoted $SC_1 \overset{R_1}{\preceq} SC_2 \overset{R_2}{\preceq} \ldots \overset{R_{n-1}}{\preceq} SC_n$ is a refinement.

Therefore, conceptually, every rule application is a valid refinement and a sequence of rule applications is also a refinement. This enables modelers to define multi-step refinements and hence allows for incrementally refining statecharts.

4.5.2. Completeness

Thanks to Property 3, it is possible to define new rules that encapsulate specific combinations of some of the eight rules presented, thus allowing further expressiveness for refinement. For example, it is possible to connect two states that are not (transitively) connected by first applying Rule 7 followed by Rule 8. Additional rules can be added to the eight we propose, as long as they satisfy the refinement definition and can not be achieved by composing existing rules or overlap with them. We discovered this specific set of rules from the ground up, trying to achieve refinements on several case studies. This set of rules is not complete, but provides guidelines to perform refinements that are useful in practice while preserving the structure and behavior.

4.6. Comparison with Existing Refinements

Updating the comparison presented in Table 2.1 with our own approach, we can observe in Table 4.1 how our approach compares to the remaining existing approaches. Compared to the other approaches, our approach is the only that preserves all properties. The limitations being on the refinement of pseudo-states and the splitting of states and the removal of states. Apart from the history states, most pseudo-states can have their behavior represented in the statechart in alternatives compatible with our refinement, thus we chose to not immediately address their refinement. Like Stateflow we do not support removal of elements, as this may break the preservation of existing behavior. Similarly, splitting states may also hinder the preservation of existing behavior. The restrictions that apply on supporting adding states, transitions and splitting transitions, are also tied to the ways these refinements can be applied while still preserving the existing behavior. Finlay, like [**100**] we also provide static regression guaranties over the behavior.

4.7. Discussion

All our decisions were taken with the intention of making the proposed Statecharts refinement usable in practice and so that the refinement relation can be statically verified between two statecharts. The refinement relation is defined so that the structure of the

Table 4.1. Updated comparison of Statecharts refinement approaches along different rules and features

Refinement approach	**Ours**	[23]	[110, 109]	[71]	[100]	[93]	[104]	[16]
Variant applied on	**Statemate**	Stateflow	μ-Charts	Flat state machines	Flat state machines	Flat state machines	UML-B	B Method
Element removal	**N**	N	R	Y	_	Y	_	_
Guarantee regression	**Y**	R	R	Y	R	N	R	R
Add states	**R**	Y	R	Y	Y	Y	Y	Y
Convert to complex	**Y**	Y	Y	n/a	n/a	n/a	Y	Y
Add orthogonal	**Y**	Y	Y	n/a	n/a	n/a	Y	Y
Preserve nesting	**Y**	N	N	n/a	n/a	n/a	_	Y
Preserve transitions	**Y**	_	_	_	Y	_	_	_
Split state	**N**	_	_	Y	_	_	_	_
Add actions	**Y**	_	_	_	Y	Y	_	_
Preserve actions	**Y**	_	_	_	Y	N	_	_
Add variables	**Y**	R	Y	_	Y	Y	Y	Y
Refine pseudo-states	**N**	_	_	_	_	_	_	_
Add transitions	**R**	Y	R	Y	R	Y	Y	Y
Preserve source	**Y**	Y	_	_	_	_	_	_
Preserve target	**Y**	_	_	_	_	_	_	_
Preserve trigger	**Y**	Y	R	_	_	_	_ _	
Preserve guards	**Y**	Y	R	_	_	_	_ Y	
Preserve broadcasts	**Y**	Y	_	_	_	_ _	_	
Split transition	**R**	_	_	_	_	_	R	_

Legend

Y	fully supported
N	explicitly not supported
R	restricted support under specific conditions
_	not defined
n/a	not applicable

original statechart is preserved. Given this, we are able to preserve the reachability as much as possible.

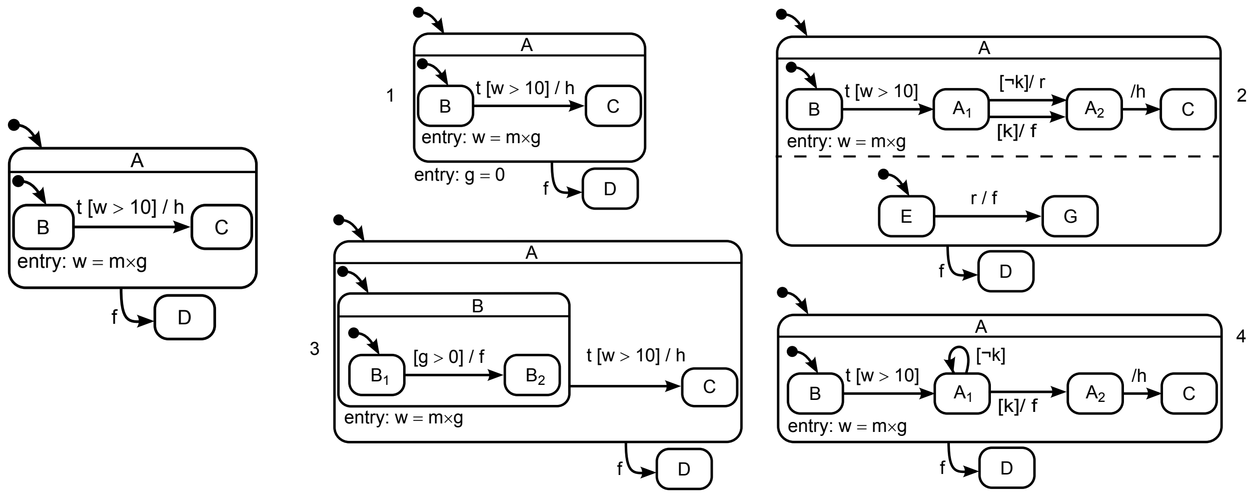
Here we present the original discution points taken into account in the presented work.

4.7.1. Static verification of refinement rule application

All rules are intended to be statically verified, while constraints specify assumptions on the dynamic behavior of the statechart. This means that, ideally, a dynamic analysis of the statechart, actions, and guards would be necessary.

4.7.1.1. Constraint on actions

In our implementation, we assume that all actions are specified as a sequence of variable assignments. Therefore, Constraint 1 can be statically verified by checking that the variables used in new entry or exit actions of a particular refinement are not part of the set of variables already in use by existing guards. When more complex operations are performed within entry and exit actions, the burden is on the developer to make sure they satisfy Constraint 1. These are the situations where a `Validator` (as defined in [**116**]) can return `WARN`, as a reminder to the developer to make sure of their validity.



(a) Original SC (b) Refinements

Figure 4.12. Set of permissible refinements of (a) that do not preserve the reachability of state `C`

Nevertheless, only statically checking if the literal variable name used in a guard is not used in an action, is in practice not sufficient. For example, consider refinement 1 in Figure 4.12(b). The original guard is on variable `w`, so we cannot perform any further operation over `w`. The new entry action in the refined statechart modifies variable `g`. As `w` depends on `g`, the guard will never evaluate to `true`, and state `C` is no longer reachable with any queue. A symbolic verification of all dependent variables may resolve this issue as done in [**104**]. However, in practice, this may be very complex when external operations are invoked and the analysis needs to go beyond the scope of the statechart. To mitigate this threat, our implementation reminds the user with a warning message every time an entry or exit action is refined.

4.7.1.2. Constraint on paths

To check Constraint 4, symbolic analysis of guards is required. Our implementation enforces mutually exclusive guards on transitions having the same event with the same source state (or substate). This restriction makes it possible to verify statically the satisfaction of Constraint 4. It avoids the necessity of determining whether a guard is not a tautology (or contradiction), *i.e.,* given an input queue, there exist two possible configurations such that one makes the guard evaluate to `true` and the other to `false`. Introducing mutually exclusive guards makes sure that the resulting statechart is still deterministic. It also makes sure that the target state of the original path is still reachable. Nevertheless, this still leaves the statechart open to other refinements that break reachability. Take for instance refinement 2 in Figure 4.12(b) where the mutually exclusive guards ensure that A_2 is always reachable, but because both paths generate a sequence of events broadcasting that ultimately trigger transition `f`, `C` is never reached, as imposed by the original statechart. Detecting this situation requires a deep analysis of the possible chains of *steps* that internal events may generate.

4.7.1.3. Constraint on broadcast

Verifying the satisfaction of Constraint 2 also requires a dynamic analysis of guard evaluation. Our implementation warns the user of this threat when he decides to apply a refinement rule requiring this constraint. However, enforcing a falsifiable guard in new transitions with a broadcast may not be sufficient to ensure reachability. For example, in refinement 3 of Figure 4.12(b), state `B` is refined such that it broadcasts an event that prevents `C` from being reached.

Constraint 2 cannot be statically verified in all circumstances. The burden of well-formedness is on the user to create guards that make sense in the context of a particular statechart. As a result, this constraint acts only as a guideline for the semantics of the broadcast and guards used, so that if followed, we can have some pre-established assumptions on the behavior of the modeled statechart. Note that our suggested procedure to delay the inclusion of guards helps in maintaining awareness of the semantic context involved, as well as maintaining simpler guards that are more easily manageable to conform to this constraint.

4.7.1.4. Constraint on transitions

Constraint 3 can have the presence of $\hat{s}_{end}$ states verified, but we cannot statically verify that they are reachable. This occurs due to guard expressions which may not be statically verifiable, like for Constraint 2. Furthermore, the statechart structure itself may contain loops that make the $\hat{s}_{end}$ states unreachable. An example of this is illustrated in refinement 4 of Figure 4.12(b), where a transition refinement introduces a self-loop. Although this self-loop is guarded, we cannot statically check that this guard is not a tautology, in which case the statechart can reach starvation, making state `C` no longer reachable: the only two possible paths are to remain in state A_1 or to exit state A altogether.

Similarly, we do not support verification of entry and exit functions of basic states to avoid the interpretation of arbitrarily complex action code.

4.7.1.5. Restrictiveness of our approach

In some regards, the refinement rules we propose may appear too restrictive. With our rules, it is not possible, *e.g.,* to delete an element of the statechart, split states, or modify higher hierarchies by encapsulating existing elements in an OR-state, and refinements must satisfy the four constraints (*e.g.,* preserve existing actions). Nevertheless, the rules guarantee with static verification that the structure and external behavior the original statechart is preserved, leading to safer refinements with respect to regression. Most of the rules allow for a complete statechart model to be introduced in the original, and thus provide extensive freedom to the developer. Furthermore, the restrictions imposed by our approach make it possible to assist developers through proper tooling.

As mentioned in Section 4.5.2, the set of rules we propose is not meant to be complete. For example, the developer may desire other refinements, such as merging two states into one, splitting a basic state into two, or refinements to additional pseudo-states (*e.g.,* conditionals, junctions). It is possible to add rules for those as long as they preserve structure and reachability. In practice, the eight rules we propose permit most statechart refinements. However, they may require more attention to the order the different aspects of the statechart are approached, following the recommendations presented here and in [**116**]. They may also require in a first instance, less intuitive refinements to reach the desired results.

4.7.2. Refinement vs. modification of Statecharts

In this article we assume a clear difference between modifying and refining Statecharts. Modifying encompasses any operation that changes a Statechart, be it adding new elements, removing them or changing them. In this work, we focus on refinement operations that provide an increment to the Statecharts such that the structure and reachability are preserved. With a stepwise refinement perspective, the developer initiates the statechart development with a more abstract and general behavior to which he keeps adding new information in the form of new Statecharts elements (states, transitions, actions, events, and guards). This assumes the correctness of the original Statechart, so any removal of elements, renaming, rerouting transitions, placing a subset of the statechart inside a composite state, or any other form of modification of the Statechart should be performed on the original Statechart. The refinement approach we propose revolves around the incremental development of Statecharts. This means that at each new refinement step adds new behavior. We do not consider as refinement any operation that may disrupt the pre-existing behavior.

Nevertheless, to be applicable in practice, the refinements we propose should complement refactoring a statechart model. Assume developers started with a first version of the statechart SC_0 and refined it into SC_n after n refinement steps. At this point, a developer wishes to perform changes that will not preserve the structure, such as refactoring. He proceeds by modifying SC_n leading to SC_{n+1}, with no by-construction static preservation guarantees from any SC_i for $0 \leq i \leq n$. He may be interested in only preserving the behavior of SC_i. Other refinement techniques discussed in Section 2.6 can then be applied or he should employ other verification techniques (*e.g.,* reusing an existing test suite of SC_i). From then on, he may resume using the refinement rules to preserve properties of SC_{n+1}. In this case, SC_{n+1} becomes the root of a new refinement evolution tree, distinct from the one rooted at SC_0.

Chapter 5

Operationalization of User Interactions

Becoming sufficiently familiar with something is a substitute for understanding it.

John Conway, Knots and Numbers, Tangles and Bangles (1996)

5.1. Synthesis of DSML editor

Figure 5.1 illustrates the process to produce custom DSML 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 DSML, 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 DSML 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 DSML editor. We model this behavior with a Statecharts model, which serves as semantic anchoring [**33**] of the interaction model. Statecharts are a common formalism to control the behavior of GUIs [**58**]. 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.

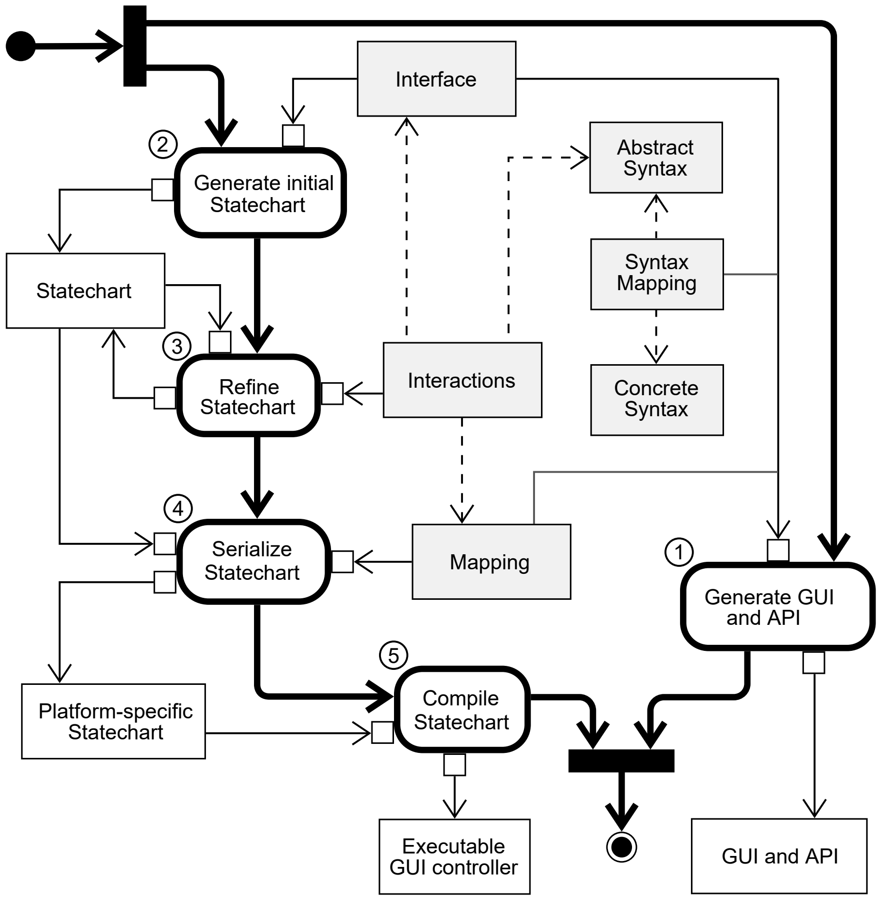


Figure 5.1. Process to synthesize DSML editors

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.

5.2. Background on RETE networks

To address the re-evaluation of state configurations in our implementation we use a RETE based algorithm [**43**]. The **RETE algorithm**, takes a series of **facts**, in our case the status of the modeling environment , and evaluates what **rules** are applicable to these facts; in our case the Interactions to be refined into the statechart. This is done through two sets of nodes connected in a directed network.

The first set, called **α-nodes** evaluate and filter the facts to the values required by the rules. In our case each α-node contains a reference to an Interface model Element required by the rules, the required status value, and a set of links to the nodes that depend on the evaluation encoded in this α-node. With the example of Listing 3.1, each pairing of element and its status present in the Interaction is encoded in a different α-node, such as the play button, in interactions C and D, will be encoded in an α-node for the ready value of the button, and an α-node for the playing value of the button.

The second set, called **β-nodes**, composes the result of the α-nodes evaluations into expressions encoded inside each β-node, merging the different evaluations. This structure of merges cascades into a final set of nodes that provide the selection for each rule. In our case this is the conjunction of all the α-nodes that contain the evaluation of the pre-conditions of a specific Interaction, by accumulating the positive evaluations of the α-nodes it depends on, and a linking it to the Interaction the evaluation corresponds to.

This evaluation process is done through token propagation, where a **token** stores the result of a node's evaluation and if considered a positive evaluation, *i.e.,* the evaluation meets the criteria encoded in the node, it is then propagated to the next connected nodes of the network. This process is done until all tokens are propagated as far as possible and potentially into the β-nodes that select a rule, therefore triggering it. The great advantage of this process is that if there is a change in the facts, either by external sources or by the application of the rules themselves, we do not require to re-evaluate all nodes. Because the configuration of tokens through the network is preserved, we then only need to evaluate what changed in the facts and propagate the respective tokens. In our case, if the status of interface elements changes, such as the button being pressed, we only need to re-evaluate the nodes that correspond to that change.

This is particularly pertinent if we have a large set of facts, but incur in very small changes. Which should be the case when applying changes to the interface and execution of a Modeling IDE, interaction by interaction. In our example, the change of the play button, from ready to playing, triggers the re-evaluation of the α-nodes that reference the play button, and change their tokens to false and true respectively. This token is then propagated through the β-nodes, until interaction D is selected to be applied and interaction C is deselected.

In practice, the status of the interface model elements associated to a state do not change. However, if we consider the differences in the status of the interface model elements between two adjacent states, we can view this difference as an update to the facts used to evaluate the RETE network. Because a previously evaluated state can have multiple neighbors, instead of just updating the token values of the RETE network to the next state, we copy them and preform the evaluation on the copy. It is then possible to use the RETE algorithm to build the statechart, as we can now have multiple branching evaluations.

5.3. Semantic Anchoring

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 DSML 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 per layer as in Figure 5.2. Since we assume all graphical DSML editors have a canvas, there is always at least the one layer dedicated to it.

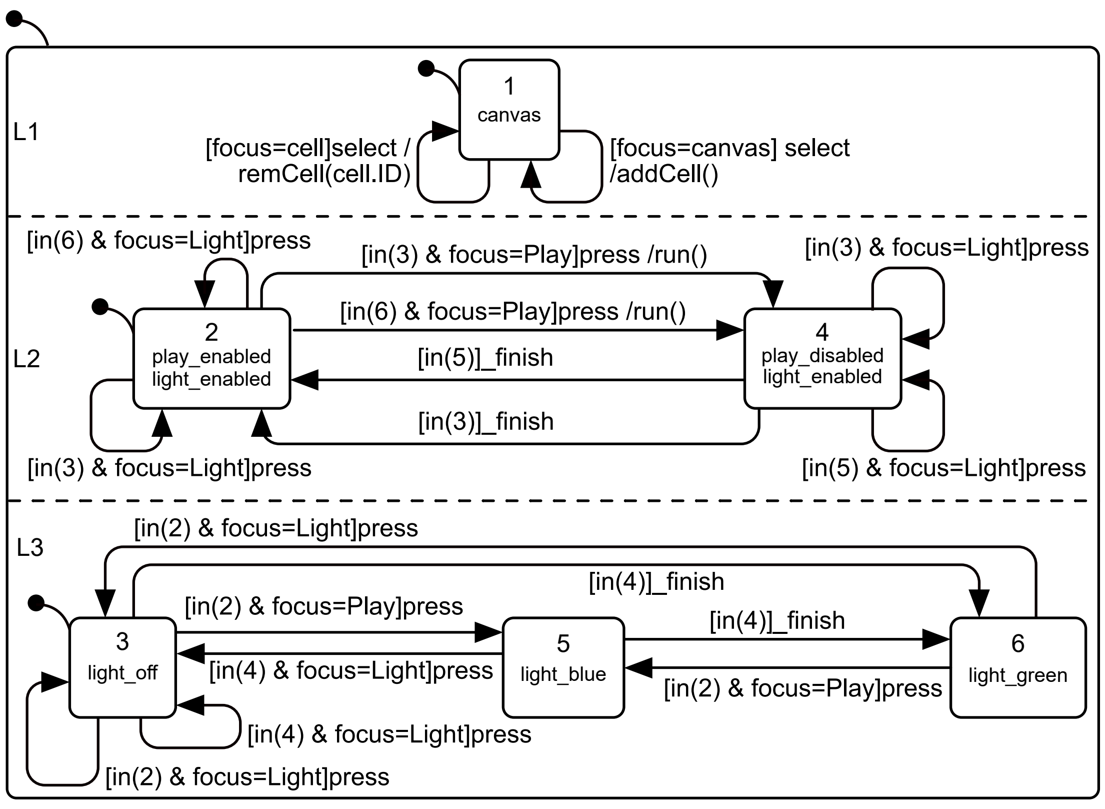

L1
1
canvas
[focus=cell]select /
remCell(cell.ID)
[focus=canvas] select
/addCell()
L2
[in(6) & focus=Light]press
[in(3) & focus=Play]press /run()
[in(3) & focus=Light]press
2
play_enabled
light_enabled
[in(6) & focus=Play]press /run()
4
play_disabled
light_enabled
[in(5)]_finish
[in(3)]_finish
[in(3)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

Figure 5.2. The resulting statechart after refinements according to the five rules

As we established in Section 4.1 we define a **Statechart** as $SC = \langle S,T,E,V,en,ex,in,I \rangle$. But because we restrict ourselves to a very specific structuring when building Statecharts, we refine this representation to $SC = \langle L \cup S,T,E,V,\emptyset,\emptyset,in,I \rangle$. Where L is the set of OR-states corresponding to the interaction layers, S is the set of basic states that encode the status of the interface, and the function $in : L \rightarrow S$ identifies the layer of each basic state. Let $l \in L$ and $s \in S$, $l = layer(s) \Leftrightarrow (l,s) \in in$. This, allow us to better highlight how the interaction rules are refined into the statechart.

Because the structure of the statechart stays constant with a main AND-state encapsulating all of the OR-states, we factor it out of the statechart representation and its configurations.

Because we simplify the statechart with actions in the transitions, we update the representation of a transition $t \in T$ to $t = (s,e,\wp(A),g,s')$. Furthermore, t connects states within the same layer where $s,s' \in S$ such that $in(s) = in(s')$. They perform a set of actions A and have a guard g as triggering condition.

Basic **states** within a layer represent the status of all interface elements within that layer at a given point of interaction. For example in GoL, layer L3 contains only the light. Therefore, there must 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 5.2). 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 the reachable configuration of the canvas.

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`.

Because we do not rely on Statechart history, and the rule application is independent of variable values, instead of using $(\bar{S},\xi,\eta) \in \mathsf{CONF}_{SC}$ to denote a configuration, we use a notation that better helps understand the interaction refinement process. In this context, we denote a **configuration** by $config : S \rightarrow \wp(ModifiableElement)$ 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 $in(s)$. ME encodes the attribute values of every interface element in that layer corresponding to s, possibly with instances of the metamodel of the DSML. 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.

5.4. Generate Initial Statechart

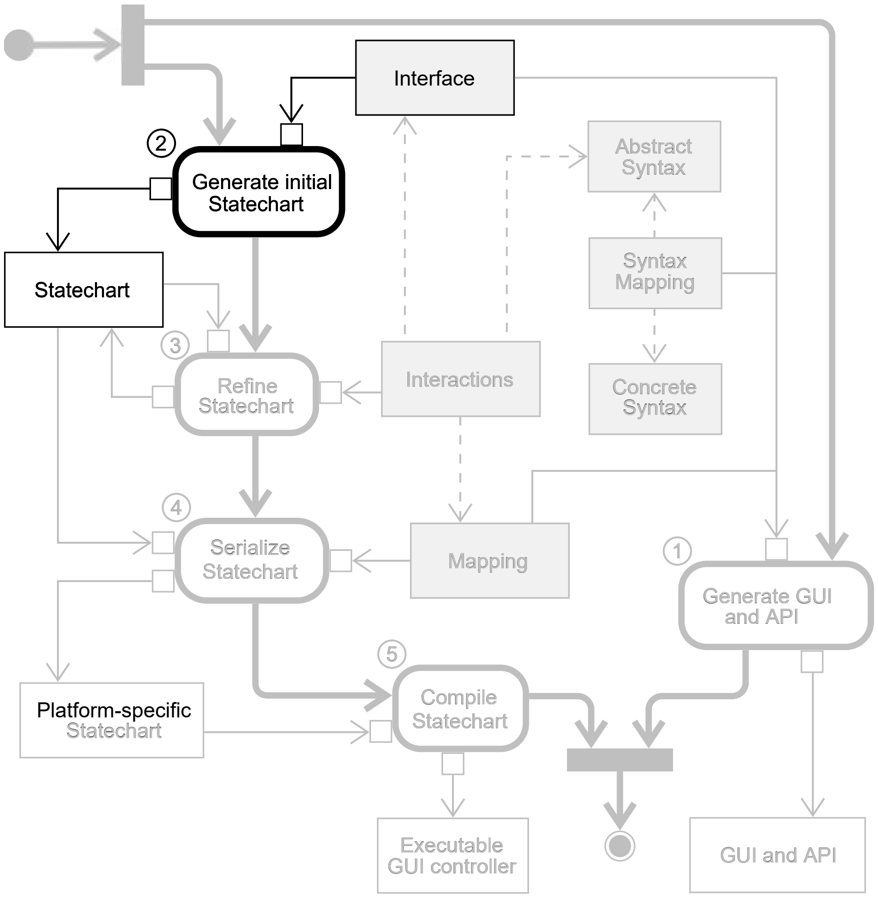


Figure 5.3. Synthesis process: generating the initial Statechart

We use the declarative model-to-model transformation language Epsilon Transformation Language (ETL) [**73**] for the transformation *Generate Initial Statechart* (highlighted in Figure 5.3). The dependencies of this transformation and how it integrates to the remaining process is highlighted in Figure 5.3. This transformation 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. The result of this step when applied to our GoL example is is the Statechart in Figure 5.4

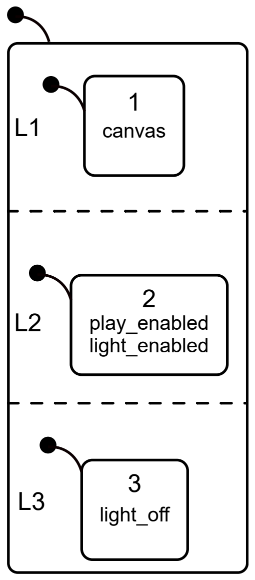


Figure 5.4. Initial GoL Statechart

5.5. Challenge of Incremental Refinement

The initial statechart (as shown in Figure 5.4) consists of one orthogonal component per layer, 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 5.2.

One major challenge of this transformation is that we do not know how many states and configurations will be evaluated a priori. 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* [**125**]. Inspired by these approaches, we define an incremental refinement procedure to construct the statechart, while reusing evaluations of previous state configurations.

5.6. Refine Statechart

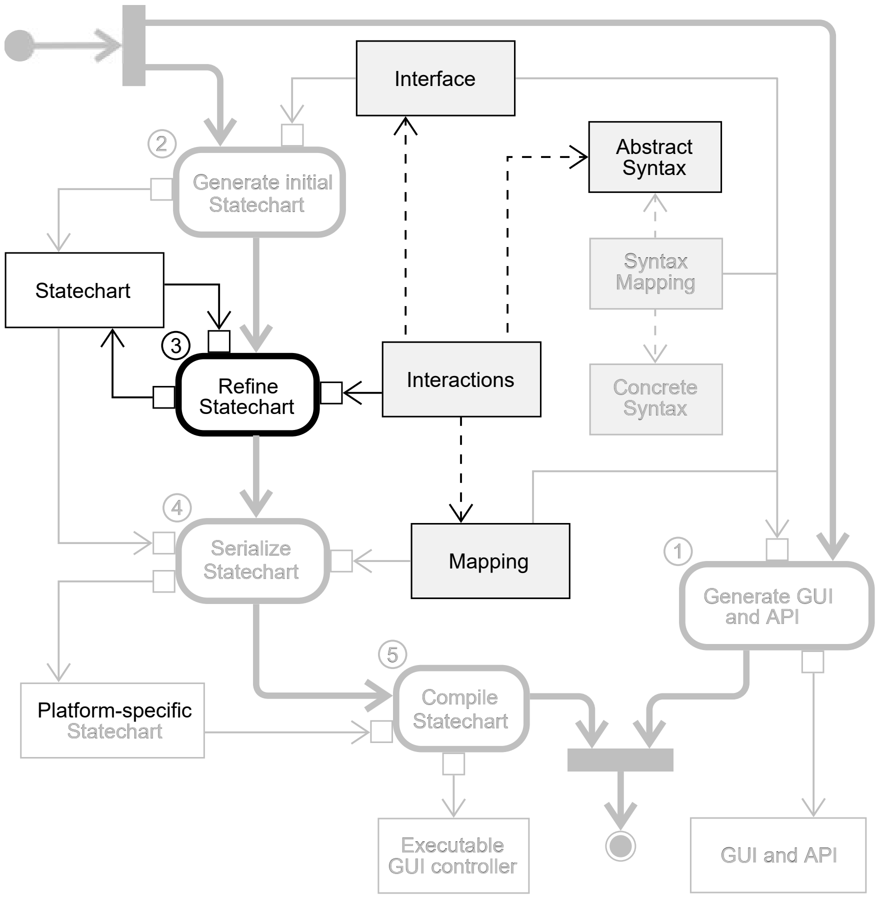


Figure 5.5. Synthesis process: refining the Statechart

We now look in detail at the steps of the Statechart refinement, to generate a Statechart that controls the behavior of the DSML editor. The different steps of this refinement where implemented in Epsilon Object Language (EOL) [**72**]. The dependencies of this refinement and how it integrates to the remaining process is highlighted in Figure 5.5.

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 5.2. Then, on line 7, it evaluates the network as described in Section 5.6.2 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 5.6.3), 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 }
```

In this algorithm, we have an evaluation queue (**evalQueue**) where the state configurations are kept for future evaluation. Initially, it is set to the initial state configuration ({(1,$ME1$),(2,$ME2$),(3,$ME3$)} as seen in Figure 5.4). Then, while this queue is not empty, we evaluate its contents. To do so, first we apply the evaluation of the RETE algorithm, that will return us the list of rules (**RuleList**) that apply to this state configuration, and the updated token values of all nodes inside the RETE network (Algorithm 1, line 7). Then for each rule made active by the RETE algorithm, we apply its Statechart refinement (Algorithm 1, line 8). The result of this is a potentially new configuration of active states, where some of the states may not previously exist. If this new state configuration is indeed new — in the sense that it has never been reached by the application of previous refinements — it is then associated with the token configuration that triggered the rule that created it and placed in the evaluation queue (Algorithm 1, line 11). If it is a configuration of active states, that encapsulates a set of Element values, that already exists in the Statechart, and therefore part of the evaluation queue's history, it is ignored because it will already be evaluated.

These steps, are enough if we are transforming to flat Statecharts. However, because we are organizing the semantics, of the different interface components the user can interact with, in different orthogonal components, this means that what we evaluate is not a single state, but a set of states, one from each orthogonal component. This set of simultaneously active states define a reachable configuration in the Statechart. Because a new state configuration may reuse existing states in the orthogonal components. In these situations, rules that depend only on the unchanged components, do not need to be reapplied as they have already been applied in a previous iteration. Take for instances the state configurations $\{(1,ME1),(2,ME2),(3,ME3)\}$ and $\{(1,ME1),(2,ME2),(6,ME3)\}$, in both these cases the result of applying Interaction A only depends on state $(1,ME1)$ and will have the same results, so we only need to apply it on one of these state configurations — but we still evaluate both resulting state configurations. On the other hand the application of Interaction C only depends on state $(2,ME2)$ (Play Button), so it may appear to be in a similar situation as Interaction A. However, unlike A, Interaction C has effects on layers outside its conditions (Light Layer), so in fact the application on these two state configurations produce different results (as seen in Appendix B.2.3 and Appendix B.4.1). Therefore, we must guaranty that it is applied to both. To do this, we add a new set of *tokenData*to the network configuration, that keeps track if an orthogonal component has been changed or not between the already evaluated configuration and the new one. If the original tokens all needed to be positively evaluated to trigger the application of an interaction rule, we only need one *change token* in addition to the regular evaluation of the nodes to propagate the token evaluation.

Because the depth and structure of the network remains regular we can optimize the evaluation by doing it in a fixed number of stages.

All steps of applying this refinement on the GoL example are illustrated in Appendix B.

The procedure keeps track of the configurations already processed; therefore each configuration is processed only 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 (as we can see in the steps presented in Appendix B there are only four such state combinations that are reachable for the GoL example). In the worst case, a change in a modifiable element triggers the reevaluation of all α-node. 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 values, like buttons and lights.

5.6.1. Construction of the Network

The first step of this approach (Algorithm 1, line 1) is to build the RETE network structure. This **RETE network** is composed of $\langle A, B_\vee, B_\wedge, E, elem,\ layer, rule\rangle$. And these constituents of the network are derived from the Interface and Language elements in our Interaction rules, alongside the organizational structure of these elements as provided by the Interface model.

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 $elem : A \to ME$ 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, $layer : B_\vee \to L$ assigns a layer to a $\beta_\vee$-node and $rule : B_\wedge \to IR$ assigns an interaction rule to a $\beta_\wedge$-node.

Edges in E connect nodes according to their relation in the rules and in the interface models. An edge connects α to $\beta_\vee$ if $elem(\alpha)$ pertains to $layer(\beta_\vee)$, as specified in the interface model. 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 $rule(\beta_\wedge)$.

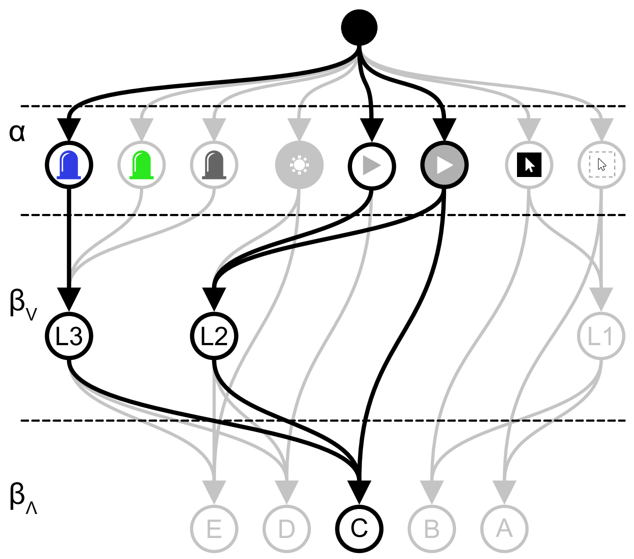


Figure 5.6. RETE network highlight of the nodes required to evaluate rule C

For example, if we take rule C, to encode it 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, as highlighted in Figure 5.6. The first comes from the rule's application conditions, the remaining come from the rule's effects. The play button belong to layer $L2$, so the α-nodes that encode its different states are connected to the same $\beta_\vee$ $L2$, whereas the light belongs to $L3$, so it is connected to $\beta_\vee$ $L3$. 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 the C $\beta_\wedge$-node.

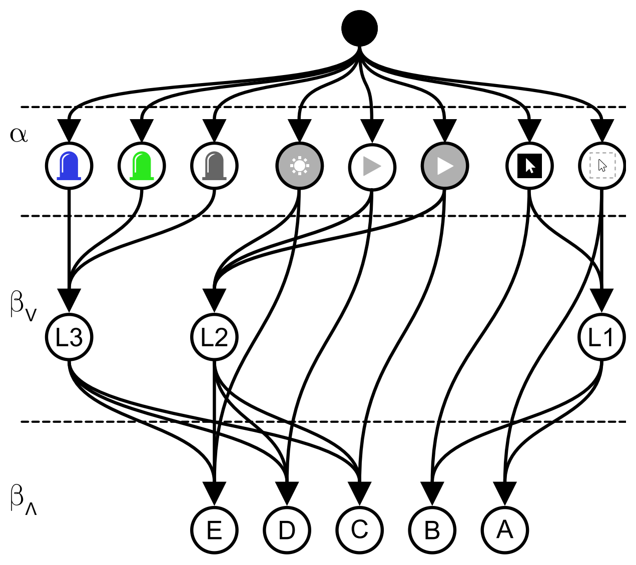


Figure 5.7. Structure of the RETE network for GoL

This process is repeated for every rule and reusing any node that is already present in the network, until all rules are encoded. The result of this is a network as sown in Figure 5.7

5.6.2. 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 $elem(\alpha) \in ME$. That is, it is set to *true* if the element value encoded in that α-node is present in the configuration being evaluated. The second boolean is the **change token** representing if the presence token has changed between the previous configuration and the current configuration. This is the previous configuration whose evaluation has led to the current configuration. This is needed in our implementation because unlike classical RETE networks, changes are not immediately evaluated due to the branching nature of our evaluations. So we need to keep track of those changes so that we can come back to previous configurations and evaluate the alternative branches. 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 layers involved in an interaction rule be changed to potentially trigger it, $\beta_\vee$-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 the disjunction from its incoming $\beta_\vee$-nodes (this is illustrated in the RETE evaluation steps shown in Appendix B).

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 5.7, 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.

With these adaptations, the evaluation performed in Algorithm 1 (line 7) adheres to Algorithm 2. This algorithm takes as input a state configuration to be evaluated and the token values of the state configuration that lead to the configuration being evaluated.

In the first stage (Algorithm 2, lines 1-9) of this algorithm, for every α-node, we search for its element in the currently reachable configuration of states and check if its status is equal or different from what is required by the α-node for a positive evaluation of the rule. We then compare the result of this evaluation with the previous token value. If it changed, we propagate that change. To the $\beta_\wedge$-nodes, we propagate the result of the evaluation. To the $\beta_\vee$-nodes, we propagate *true*, to signal that there was a change in the orthogonal component. This terminates the evaluation of α-nodes, so we proceed to the next stage.

Algorithm 2: EvaluateRETE(config, token)

```
Input: The current Statechart configuration and the token values from the configuration
Output: An evaluation of what rules should be applied to the current configuration
1  for $\alpha$ in RETENetwork.alphas do
2      element ← config.get($\alpha$.element)
3      isEq ← equalValues (element, $\alpha$.element)
4      if token[$\alpha$] $\neq$ isEq then
5          token[$\alpha$] ← isEq
6          for $\beta$ in $\alpha$.betas$_L$ do
7              token[$\beta$] ← true
8          for $\beta$ in $\alpha$.betas$_\wedge$ do
9              token[$\beta$][$\alpha$ ] ← token[$\alpha$]
10 for $\beta_L$ in RETENetwork.betas$_L$ do
11     if token[$\beta_L$] = true then
12         for $\beta_\wedge$ in $\beta_L$.betas do
13             token[$\beta_\wedge$][0] ← true
14         token[$\beta_L$] ← false
15 RuleList ← $\varnothing$
16 for $\beta_\wedge$ in RETENetwork.betas$_\wedge$ do
17     if $\bigwedge_i$token[$\beta_\wedge$][i] then
18         RuleList ← RuleList ∪ $\beta_\wedge$.rule
19     token[$\beta_\wedge$][0] ← false
20 return (RuleList, token)
```

In the second stage (Algorithm 2, lines 10-16), we check every $\beta_\vee$-node. Each $\beta_\vee$-node that is set to *true*, we propagate its value to the connected $\beta_\wedge$-nodes, and we reset the $\beta_\vee$-node being checked back to `false`, so that it is ready for the next run of evaluations.

In the last stage (Algorithm 2, lines 16-19), we check every $\beta_\wedge$-node if their α-node requirements are met and if they depend on at least one positive $\beta_\vee$-node that was updated during the α-node evaluation. If so, we add its corresponding rule to the list of rules that are applicable to this state configuration.

So, for example, the initial state configuration $\{(1,ME1),(2,ME2),(3,ME3)\}$ will propagate through the RETE network as shown in Figure 5.8. It will evaluate the facts for the play and the light buttons as enabled, the light off and because we treat the canvas as a black box, it also evaluates the focus of an action on the canvas and on a cell as *true*. In turn, this will evaluate L1, L2 and L3 to *true*, and then the $\beta_\wedge$-node that triggers the Interactions A, B, C and E will also evaluate to *true*.

If we then follow the refinement of Interaction C into the Statechart we get a new state configuration that has the token propagation shown in Figure 5.9. With the dashed connections showing unchanged values that do not need to be propagated, we can see that because

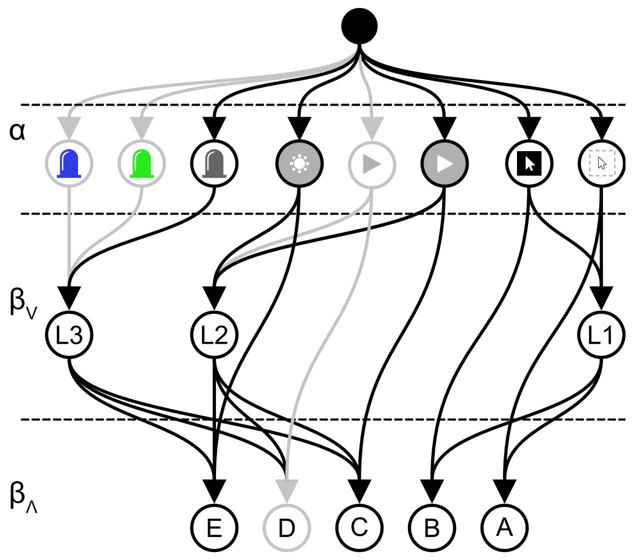


Figure 5.8. Propagation of values from the initial state configuration.

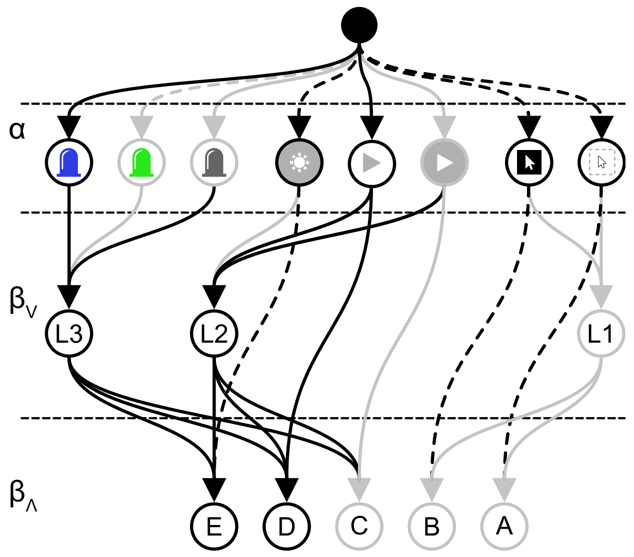


Figure 5.9. Propagation of values after the application of Interaction rule C.

the availability of Cells and the Canvas did not change from the previous configuration, their values do not need to be propagated to $\beta_\wedge$-node A and B, but on the other hand because of this lack of change $\beta_\vee$-node L1 is now disabled, which in turn gets propagated to $\beta_\wedge$-node A and B, disabling them. With this mechanism, we avoid the over application of Interaction rules A and B. A similar situation could be present with the light button, as it too did not change, but because there were other changes in the dependencies of $\beta_\vee$-node L2 and L3, these still propagate as *true* keeping Interaction E enabled to be applied on this new configuration.

5.6.3. Construction of the Statechart

Algorithm 3 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 Section 4.

Algorithm 3: 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  $L \leftarrow$ rule.getAllLayers() // from condition and effect
2  ActiveStates $\leftarrow \{s \mid (s, ME) \in$ config $\wedge$ in($s$) $\cap L \neq \varnothing\}$
3  config' $\leftarrow$ clone(config)
4  event $\leftarrow$ rule.getEventName()
5  varGuard $\leftarrow$ rule.condition.getVarElements()
6  context $\leftarrow$ rule.getContextName()
7  for effect $\in$ rule.effects do
8      actions $\leftarrow$ effect.getOperationElems()
9      actions $\leftarrow$ actions $\cup$ effect.getVarElements()
10     for $s \in$ ActiveStates do
11         guard $\leftarrow \langle$ ActiveStates $\setminus\{s\}$, context, varGuard $\rangle$
12         for $e \in$ effect.getModifElems(in($s$)) do
13             actions $\leftarrow$ actions $\cup\{e\}$
14             config'[$s$].replace($e$)
15         target = NULL
16         if effect.next = NULL then
17             target $\leftarrow$ configHist.findState(config'[$s$],in($s$))
18         if target = NULL then
19             target $\leftarrow$ SC.newState(in($s$))
20         SC.newTransition($s$, target, event, guard, actions)
21         config[target] $\leftarrow$ config'[$s$]
22         $s \leftarrow$ target
23         actions $\leftarrow \varnothing$
24     event $\leftarrow \varphi$
25     varGuard, context $\leftarrow \varnothing$
26 return config
```

Recall that a state s in the Statechart represents a particular state of all interface elements of the layer corresponding to 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 11–13). 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 18), be it a new state, or an existing state. If this target state is a new state, we adhere to Rule 7 of our Statechart refinement, if it not, we adhere to Rule 8.

The transition is initially triggered by the event that corresponds to the event of the rule (line 4). If the rule has multiple effect steps, all remaining steps are triggered by the empty event (line 21). Similarly, we collect the VarElements to be used as guards on the first step of the rule (line 5), and remove it for any subsequent steps (line 22). The set of actions of a transition is defined by the attribute values of interface elements (line 12), the CRUD operations on DSML elements (line 12), value assignment of variables (line 8), and the external operations to invoke (line 7). The function `getModifElems` returns language elements (*e.g.,* deleted cell in Interaction Rule B) or interface elements (*e.g.,* unlit light in Interaction Rule E) of a specific layer in the effect of the rule. With Interaction Rule C, the function `getOperationElems` returns the `RUN` operation with its parameters.

Eventually it will be possible to set some of the variables to affect the state configuration because with the use of the coordination language they will have a limited set of values and thus possible to encode in the RETE network. The consequence of this would be a reduction on the relience of guards to verify variables and coordinate rules. But the use of variables to share values between Interaction Rules, namely DSML related values, make these unbound, and therefore we must rely on guards to verify them. The consequence of this is that we now have a set of actions that are bound the constraints in Section 4.7.1.1. Because we are using an automated refinement instead of performing the refinements one by one, we preemptively know what guards will occur. We could postpone their application until after all refinements are applied. But because all refinements that are affected by the same variable are all defined at the same time by the Interaction Rules, we can apply their guards as the rules are applied instead of at the end and still result in the same Statechart. It is the fact that we do achieve the same Statechart that allows us to do this small optimization without jeopardizing the semantics of the final Statechart. Still, we must keep in mind the remaining constraints, as there may be values that cannot be statically verified, further motivating the direct use of Variables to only exceptional cases.

The condition and effects of a rule may involve elements from different layers, such as Interaction Rule C. Therefore, as stated in the broad view of the refinement step in Section 5.6, we must synchronize the transitions across layers to satisfy the condition of the rule. On line 8, 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 $\{(1, ME1), (4, ME2), (3, ME3)\}$ and receive a `press` event on the play button in Figure 5.2.

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 DSML 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). The function `findState` searches through the refined Statechart for a state corresponding to the set of modifiable elements in the new configuration within the layer of s. Then, this state should be the target of the new transition. The original states before the application of the rule is marked as source. And then source and target are connected by any number of intermediate states and transitions required to make the system react correctly to the interaction. Also, to note that in any case it is possible that both the source and the target are actually the same state. We can see this in the Appendix B, in the applications of rule E when configuration being refined already has state 3.

In our example of applying rule C to the state configuration $\{(1, ME1),\ (2, ME2),\ (3, ME3)\}$. It will first collect states 2 and 3. Then for state 2 we collect the action `run`, the event `press` plus the context `Play`, and guard `in(3)`. We calculate the effects of the rule, creating state 4. We search for state 6 in the Statechart, and if it does not already exist, we add it to the Statechart. We then add a transition from 2 to 4 with guard `in(3) & focus = Play`, event `press`, and action `run()`, and add 4 to the new state configuration. We then process state 3, where we calculate state 5 and create the transition from 3 to 5 with guard `[2]`, and event `press`. Because state 3 does not include the context, it performs no actions. We add state 5 to the new state configuration. These effects of the refinement process are illustrated in Figure 5.10. The result of this process then returns the state configuration $\{(1, ME1), (4, ME2), (5, ME3)\}$.

The successive application of refinements of the Statechart ends when Algorithm 1 is completed, resulting in the Statechart in Figure 5.2.

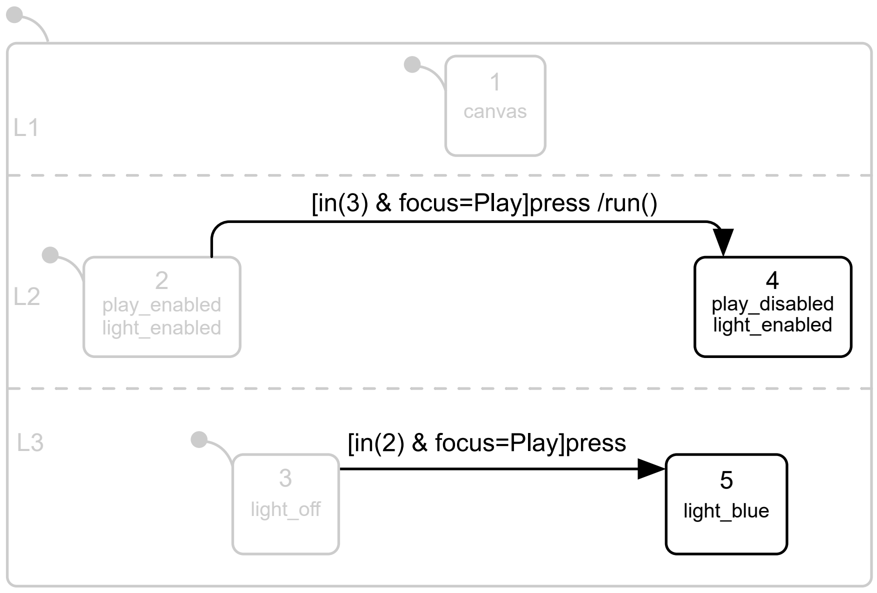


Figure 5.10. Effect of refining rule C into the initial Statechart configuration

5.6.4. Use of Statechart refinement rules

Throughout this implementation, the refinement process was based on the previously established state chart refinement rules, namely path refinement and extension rules. Nevertheless, the implementation itself takes some liberties in district or their refinement defined with those rules. An example of this is the inclusion variable guards throughout the refinement process, instead of refining these guards at the end, as stated in the original specification. These liberties were taken because this application state-chartered refinements is done in a very specific scope and controlled environment. That said we can ensure that the implemented refinement, has the same final result as if we strictly adhered to the refinement guidelines. The point of these liberties was therefore to simplify the algorithms and the rule application itself. The future inclusion of other refinement rules with the expansion for the coordination language or if we stop considering the canvas as a black box, may require us to revise this implementation strategy. But a full adherence to the state chart refinement guidelines should not impact the core of the algorithm itself only the number of iterations with some additional control data.

5.7. Adaptation to a specific platform

The serialization process involves two aspects. We translate the final statechart into an executable format, such as SCXML. In this case we used the Epsilon Generator Language (EGL) a template based model-to-text generation language. During this process, we discard references from states to interface elements. We also adapt the events and actions of every transition to be compatible with the target platform. Two APIs must be readily

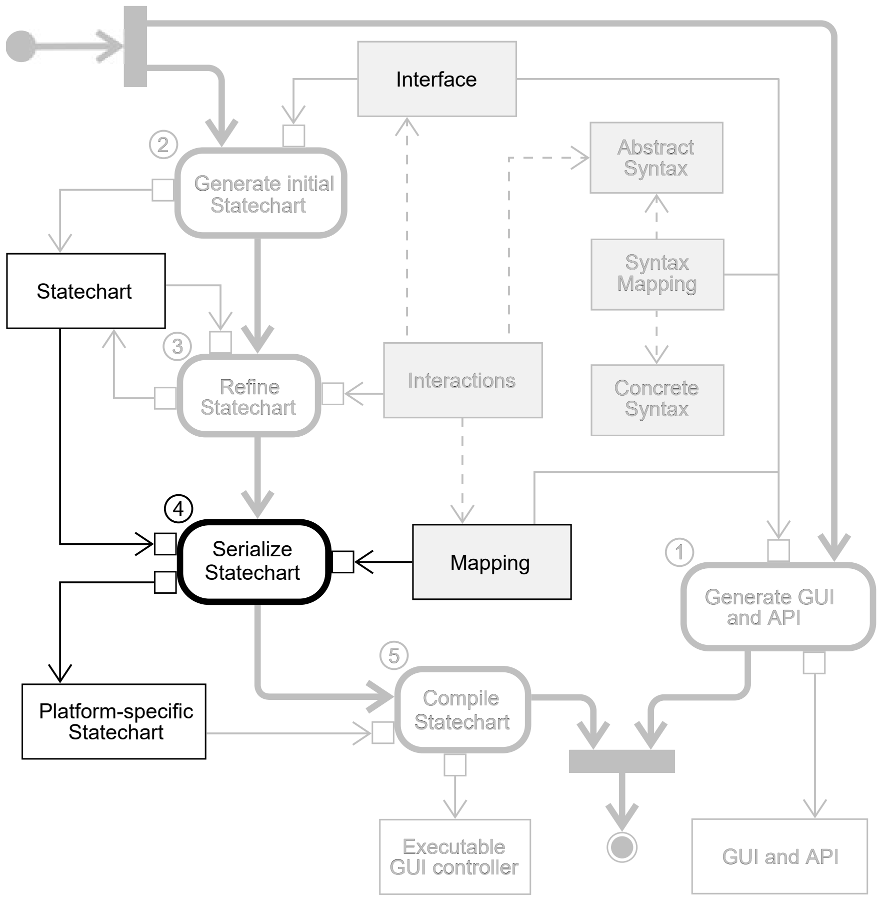


Figure 5.11. Synthesis process: serializing the Statechart

available for the platform: one to manipulate DSML elements and one to operate interface elements. The event mapping model defines the required translations. It provides the listeners 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 `press`. Actions on transitions in the canvas layer are translated into the CRUD operations of the modeling framework API. The second API must be configured 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. The dependencies of the serialization step and how it integrates to the remaining process is highlighted in Figure 5.11.

5.8. Implementation

To demonstrate the feasibility of our approach, we implemented a prototype. For this prototype, we defined the input modeling languages defined in Section 3.2, in EMF using

Ecore metamodels and deployed as plugins. We also 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. The Statecharts Compiler is an out-of-the-box solution that can be switched to adapt to different Statechart implementations. For this thesis, we use the SCCD [**123**] format and compiler, which is compatible with Harel's semantics [**55**]. The remaining steps in the process shown in Figure 5.1 done using the Epsilon tool suite [**72**] and deployed both as Eclipse plugins and as a stand-alone executable. We implemented all transformations using this framework because it integrates different types of transformations seamlessly and allow us to run them in headless mode outside of Eclipse.

This prototype generates web-based graphical DSML editors. Therefore, we generate the GUI in a combination of HTML, CSS and JavaScript. We also provide a JavaScript API for the statechart to control the interaction with the GUI and bridge the connection with the modeling framework.

Due to the heterogeneous nature of the models produced at the early stages of development, we tested the development of each stage of our approach by first creating a simple example. During development, we tried to approximate the semantics of that example.

As this was achieved, we built these examples towards the GoL example. With a fully functioning editor, we introduced variations to explore additional issues. If an interaction could be expressed in different ways, new test examples variations where created for each of them, to ensure compatibility and long term testing of their validity.

Because some issues still required changes in the specification models definitions, requiring changes in their meta-models and syntax, automated testings was not possible. Still, on each fix, all previous tests where retested and compared to previous solutions, to ensure the consistency of the implementation.

To facilitate this process we created an Ant script to automate the generation and cleanup of the test models. This also allowed us to execute all the tests in an automated manner.

For the detection of issues themselves we relied on the output models, including the serialization of the RETE network execution data, effectively providing us a high level trace of the evaluations, alongside traditional tracing techniques.

Chapter 6

Implementation Evaluation

Regard it as just as desirable to build
a chicken house as to build a cathedral.

Frank Lloyd Wright, Two Lectures on
Architecture (1931)

In this chapter, we evaluate the generation process of modeling editors produced with our approach.

6.1. Objectives

For this evaluation, we focus on the following two objectives:

Objective One. *(O1) Verify that our approach is expressive enough to mimic defining user interactions found in existing DSML editors.*

Objective Two. *(O2) Verify if the process of generation of the editor scale with the increasing complexity of user interactions.*

To answer these questions, we perform two experiments, one to address each question. For (O1), we qualitatively assess the user interaction with the generated editors compared to existing ones. We try to replicate the distinguishing interaction features of these editors, and assess the complexity of achieving such results. For (O2), we measure the time performance and memory usage of the transformation process with varying complexities of interaction models.

6.2. Reproducing existing editors

To evaluate that the declarative specification of interactions results in a correct interaction environment, beyond the GoL running example shown in Section 3.1, we implemented two

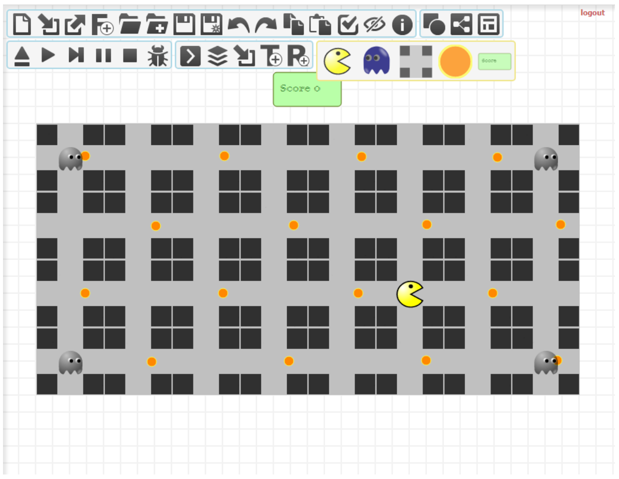
Number of	GoL	Pac-Man	Music
Interaction Rules	5	37	66
Interface elements	6	13	32
Layers	3	3	4
α-nodes	8	38	12
States	6	12	16
Transitions	18	77	406

Table 6.1. Characteristics of each case study

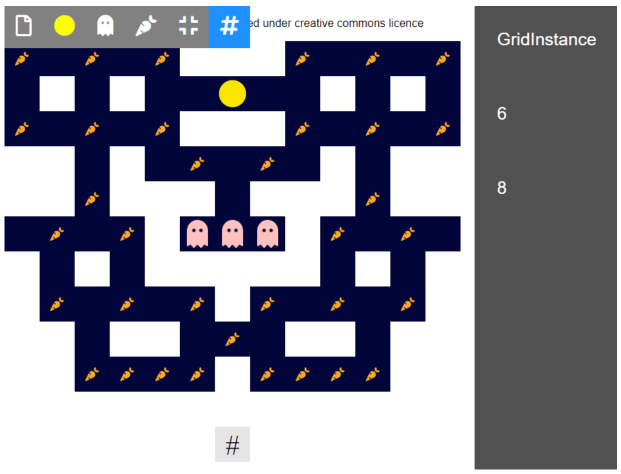
additional editors that replicate the main interaction aspects of existing editors. Table 6.1 outlines the number of interaction rules, interface model elements, and layers required to generate each editor. It also depicts the size of the generated network and statechart.

6.2.1. Pac-Man game configuration editor

The goal of this case is to replicate a DSML editor generated inside a modeling language workbench. The Pac-Man editor present in AToMPM [**118**] (which has similar behavior to a GMF editor in Eclipse) allows the modeler to define game configurations where Pac-Man 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 DSML elements to instantiate and the ability to drag-and-drop elements on the canvas. As reported in Table 6.1, 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.



(a) Interface of AToMPM Pac-Man editor



(b) Interface of our generated Pac-Man editor

Figure 6.1. Comparison of Pac-Man modeling editors

The main challenge, in this case, is to enable dragging DSML elements. Each element type (*e.g.,* Pac-Man, 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 DSML types on the toolbar to make sure exactly one type is selected when creating an element on the canvas. This is due to the fact that this version of the editor was implemented without the use of variables and therefor, the interaction rules were not capable of sharing information.

A comparison of the interface of the editor we produce with the one produced by an exiting DSML workbench is shown in Figure 6.1.

6.2.2. Music Modeling editor

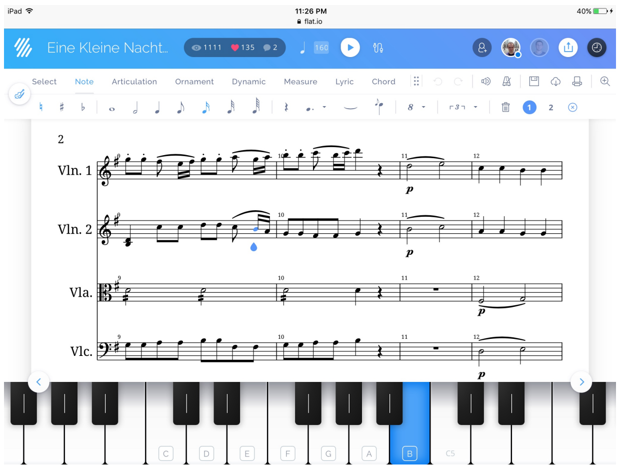
To illustrate our proposal we use a case of a music modeling language akin to MusicXML [**48**], and the subsequent specification of the interaction and artifacts of this modeling environment. We choose the domain of music writing because its interactions are diverse and complex due to the large work outside computational environments, as well as because it removes the bias of staying in a domain close to computing, which is the predominant domain of application in the MDE literature. This is inspired from the MuseScore modeling environment [**112**].

The language provides a form for writing music sheets. For this thesis, we focus only on a small number of interactions: various ways of placing notes on the sheet, playing a note, and selecting a note length (half note, quarter note ...). The goal here is to demonstrate how we address multiple interaction requirements of a music modeling language, so that the final modeling environment is properly adapted to the music domain expert and the environment he uses.

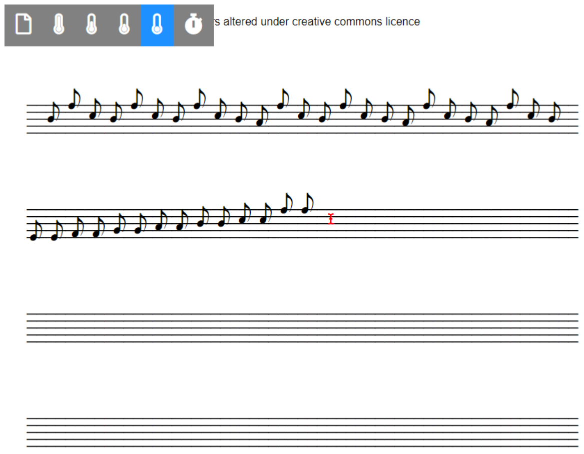
Software tools targeted at non-programmers involve a variety of interactions [**103**]. For example, Flat [1] allows musicians to compose music sheets by playing on a MIDI keyboard. To replicate some of its unique methods of interaction, we generated a music sheet editor where the user is presented with 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

1. https://flat.io/

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.



(a) Interface of flat.io music editor



(b) Interface of our test music editor

Figure 6.2. Comparison of Music editors

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.

A comparison of the interface of the editor we produce with an exiting dedicated editor is shown in Figure 6.2.

6.3. Evaluation of the performance of the generation

From a performance point of view, the most important phase in the generation process of the DSML editor is the construction of the statechart. As described in Section 5.1, 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, as the remaining steps rely on pre-existing tools.

6.3.1. Experiment setup

In Section 5.6, 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 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 DSML to contain 10 language elements. 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.

6.3.2. 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.

6.3.3. 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 6.3 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 Java Virtual Machine. 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

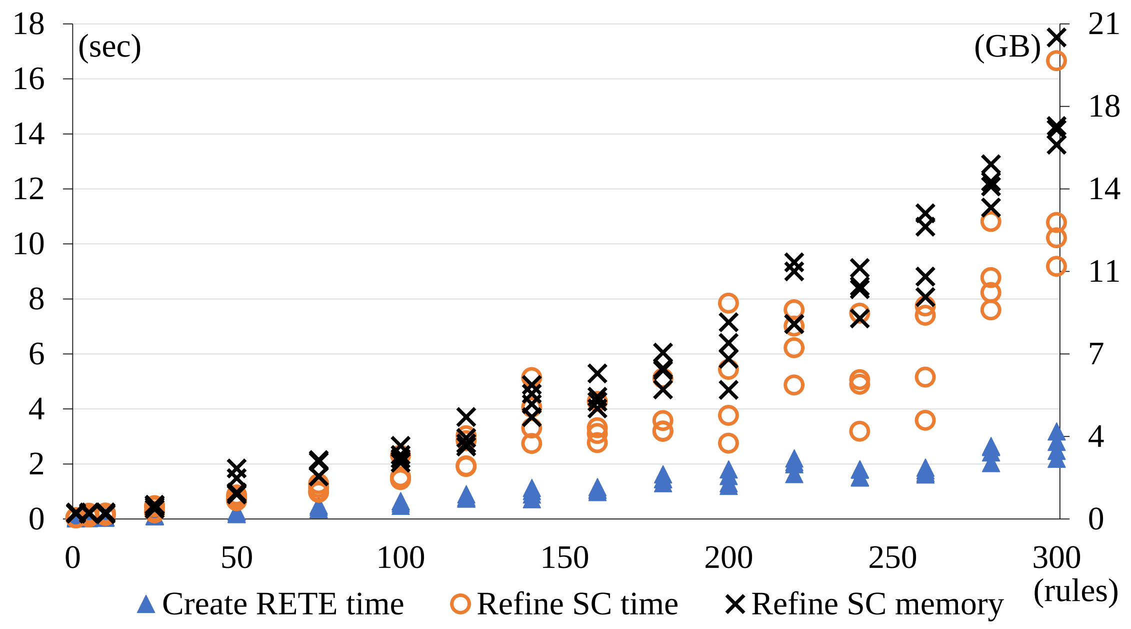


Figure 6.3. Time and memory usage of the `RefineStatechart` procedure in seconds and gigabytes per number of rules

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 6.3 shows that memory usage follows the same trend as time performance. Compared to the example DSML editors we modeled (Section 3.1, Section 6.2.2 and Section 6.2.1), generating an editor with 200 rules takes less than 10 seconds and under 8GB of memory.

6.3.4. 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 DSML 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.

Chapter 7

User Evaluation

No plan survives contact with the enemy.

Helmuth von Moltke the Elder, as paraphrased by Correlli Barnettin (1963)

When building a system, such as the one we present in this thesis that aims to be of practical use in generating DSML editors with custom interactions, we need to know how usable in practice such system is. In particular, at an early stage, we want to evaluate how much our approach allows for the creative aspects of explicitly defining the interaction of DSML editors. To investigate this aspect of our work, we conducted the presented user study.

7.1. Objectives

With this user study, we want to analyze our approach for the purpose of creating interactive DSML editor with respect to the creativity, from the point of view of MDE developers and UX designers in the context of defining the interactions using our proposed approach.

For this goal, we propose a series of research questions:
[RQ1] Does the process of defining interactions in a declarative manner with the proposed interaction concepts, restricts the defined interactions to a specific way of interacting with the generated editors?

Our true aim is of course to have an approach that provides as broad as possible variety of possible interaction. But because there is no effective upper bound on such variety, we take a null hypothesis approach and instead test how restrictive this approach is. Connected to this point, we also try to gain insight into how *much this approach allows and promotes the exploration of new interactions, be it by simplifying the definition of interactions, or by providing a quick development cycle, that allows for the exploration of alternatives.*

[RQ2] Is the provided process and level of abstraction appropriate to the developer/UX designer?

There is a fine balance between how flexible and how restrictive the approach is, so that the provided tools allow for full expressiveness of interaction ideas, without being overwhelming.

[RQ3] How effective is the approach in providing a complete editor?

This may seem redundant with the previous study, the difference here is that, here we are not replicating an existing editor as we develop the tool, but instead let others develop an editor from the perspective of a finished tool. The aim is then to see if there is a clear boundary between the editing specification and the modeling back-end at an engineering level. This also opens observations on the possible limits of exploratory interaction design versus starting with requirements analysis.

7.2. Experiment setup

To perform this experiment, we built a virtual machine (running LUBUNTU 19.10) with the developed software installed, support software such as a browser (Chromium), a PDF reader and remote access software (AnyDesk). The virtual machine provides an environment that will be exactly the same to all participants, by having all requirements pre-installed, using the same versions of software and providing a setup snapshot that allows for the effective reset of the environment between users. The participants connect to the virtual machine through virtual desktop software. This allows for a broader access to participants, without them having to be physically present, having to install the software themselves, and it enables us to record the sessions.

The experiment itself is divided into three stages. The first stage provides an introduction to the tools and process through an optional step-by-step tutorial, recreating an extended version of the GoL editor. We chose this example as it is simple to understand, yet illustrates all aspects of using our approach.

The second stage is a free-from exercise where the participants use the tool however they want for the purpose of completing the given tasks of designing a modeling editor for the given Mind Map language. We chose the Mind Map as the target modeling language of the editor, as it is simple to understand, there is a good chance the participants will already be familiar with the concepts used in Mind Map, and as a tool to organize thoughts, the choices of how to interact with such a language should lean more into personal preferences, rather than pre-established interactions. It also provides a range good range of tasks, including the instantiation and further manipulation of multiple instance types (`Topic` and `SubTopic`), with settable attributes and the use of links to connect topics, leading to the participants needing to make their differentiation in how interactions occur.

In the third and final stage the participants answer a post-experiment questionnaire to evaluate their assessment of the provided tools and approach.

During these sessions, interventions were deemed acceptable if they provided no information towards a specific solution to the tasks involved, and were only technical in nature, *e.g.,* helping the compilation run the first time, point out syntactic error when prompted to do so, etc. That is even if assistance was provided, they did not receive any assistance for issues related to designing their solution.

7.2.1. Participant selection

To enroll participants for this study, we sent a call for volunteers, emphasizing on knowledge in MDE or UI and UX design is required. We followed a convenience sampling, by sending invitations to research groups and industry of our knowledge.

From these, eight volunteers responded to the call and participated in this study. Appendix F presents the profile of the participants. One is a designer in the UX industry, the remaining participants are graduate students or postdoctoral fellows.

From their self reported experience, most of the participants reported being knowledgeable or experts in computer science. Half reported 5-6 years of experience, 2 reported 9 or more years, and one reported no experience at all. Most of them reported intermediate experience in MDE (1 ot 6 years), one reported being an expert (more than 10 years, and one reported no experience. With respect to experience in UX and UI development, most reported being intermediate or knowledgeable, while the remaining two are novice or have no experience. Due to the small number of participants we cannot draw clear conclusions in terms of inflation of self-reported expertise, though this aspect may influence the conclusions.

7.2.2. Tasks

During the first stage of the study, the only task for the participants is to be acquainted with the tools and process of generating a DSML editor with our approach. In the tutorial, they learn all the steps to build a DSMLeditor, how to define interaction rules, and how to use them to call output events, use variables, and in general express interactions and their effects in the DSMLeditor and model. For this purpose four variations of the GoL example where provided along with a GoL project where only the interaction definitions where undefined, in the same manner as the final exercise, to allow the participants to follow along the tutorial as it shows the construction of a GoL editor.

For the second stage, the participants were given a short description of a simple form of Mind Map.For the purpose of this task, to not overload the learning process and to reduce the amount of time necessary during these tests, four pre-made layout models where provided, alongside a pre-made event mapping model, modeling back-end and corresponding Meta-model. The layout models define, an editor layout with only the canvas (`Blank`), relying solely on user inputs, an editor layout with two buttons in the upper left corner (`Simple Toolbar`), an editor layout with eight buttons in the upper left corner (`Large Toolbar`), and an editor layout with a button on each corner (`Corner Toolbar`). This allows for a variety of editor usages without the participants requiring to customize the layout model themselves. The event mapping model provides a large number of pre-mapped events, including all mouse usage combinations and a selection keys. Thus, the variability in this exercise was focused solely on the specification of interactions. This was done to provide variability in the possible solutions, while simplifying the exercise, and focusing it on a single task. This task was to define the interactions for the Mind Map editor. Again, to reduce any influence on the reached solutions, no functional requirements were defined, leaving this aspect of the editor to the discretion of the participants to foster the creativity of the participants' solutions. Appendix D presents the documentation provided to the participants for this stage, including the full description of the Mind Map language and all pre-made models.

At the end of the test session, the users where asked to answer the questionnaire presented in Appendix E. This questionnaire is composed of 20 questions using either a list of classifications answers (*e.g.,* age brackets or education background), or a five point Likert scale [**65**] of agreement (strongly agree, agree, indifferent, disagree, strongly disagree) or expertise (none, novice, intermediate, knowledgeable, expert). The first 11 questions are used to profile the population of participants, questions 12 to 15 test for the initial mindset of the participants, questions 16 to 20 ask directly how the participants feel about the approach. A final open-ended question is made at the end to allow the participants to express any additional thought that may provide insightful.

7.2.3. Data collection

After the confirmation of consent from the participant, the sessions where recorded using CamStudio, capturing the video of the virtual machine at 4 frames per second. This included all elements of the virtual desktop environment including the Eclipse editor used to edit the interaction models, the command line terminal to generate the editor, the browser to test the generated editor, and the documentation. No audio of these sessions was collected. No direct user data was collected. These settings where deemed enough granularity for the data being collected. From this data, we measured the number of times the participants compiled and tried their generated editor for how it was behaving, the number of times they got errors during the generation process, and the time spent on the exercice itself.

Simultaneously an audio conference call was held with each of the participants during their session, to clarify any questions and provide general tool support.

From the exercises, the interaction specification produced by the participants was also collected for analysis. From the set solution interaction models, we reverse engineer a feature diagram that composes the solutions provided, by their commonalities in providing DSML editor features. In this feature diagram we can apply common analysis techniques [**21**] and measure the feature coverage and the number of answers per feature.

At the end of the session we collected the answers to the user questionnaire, to provide us a contextualization of the directly collected data.

With this data, we can answer our questions in the following manner. [RQ1] is answered by analyzing the set of solutions provided in how similar they are across the different participants. [RQ2] is answered by how the questionnaire answers fall in the provided Likert scale and confirmed with the recorded data. [RQ3] is answered by the coverage to the provided solutions, their level of functionality, and the impact of errors during the development process.

7.3. Results

Let us then look into the results provided by the collected data.

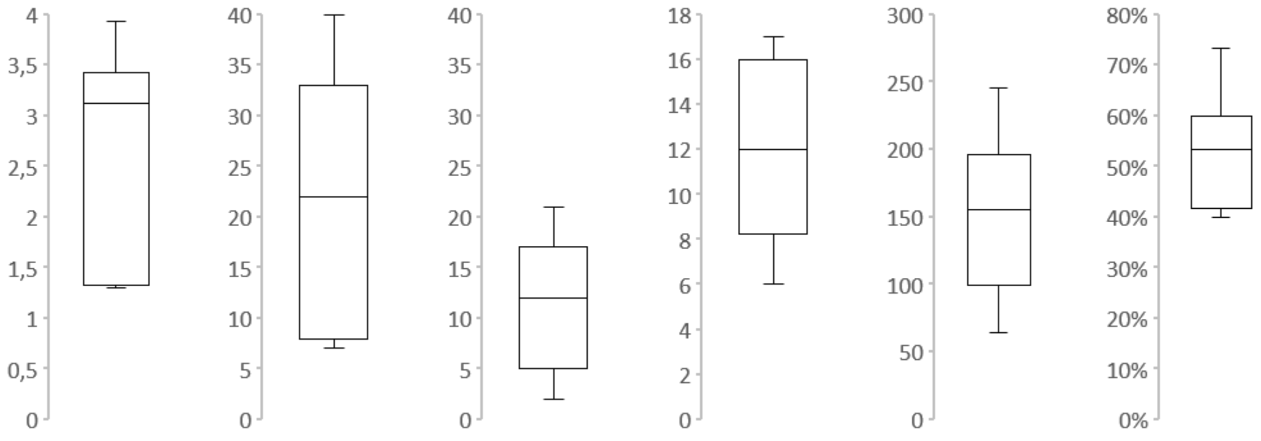
7.3.1. Recorded Sessions

Table 7.1 presents the measurements of the performed sessions, with Figure 7.1 summarizing it with box plots. In Figure 7.1(a), we can see that on average the sessions took around four hours. Of these two and a half, or over half the session, was dedicated to the Mind Map exercise, while the remaining time was spent with getting acquainted with the tools through the tutorial, or answering the questionnaire.

During the exercise, as shown in Figure 7.1(b), the participants took an average of 22 iterations to produce an editor to their satisfaction. Of these, around 11 iterations where in direct response to errors in the resulting generated editor, as shown in Figure 7.1(c). These errors ranged from Javascript crashes — usually due to bad references in the declaration of the interactions — to the generated editor behaving differently than intended — but not from what was declared in the interaction rules from a technical point of view. An example of the former type of error would be to simply misspell the name of an interface element or a back end function. An example of the latter, was when specifying how to add an instance when a specific button was set to active, without placing the status of said button as a

Table 7.1. Descriptive results of the second stage of the study

Participant	Total Test Time (h)	Exercise Time (h)	Exercise Iterations	Exercise Errors	Denied Solutions	Number of Rules	Lines of Code	Coverage	Interface Layout
P1	6.63	3.93	25	14	1	11	193	73.3%	Blank
P2	4.00	1.33	22	12	0	8	105	53.3%	Blank
P3	2.42	1.41	16	5	2	17	197	53.3%	Blank
P4	5.00	3.12	33	21	0	9	98	60.0%	Blank
P5	3.00	1.30	8	5	0	6	64	40.0%	Blank
P6	4.17	3.26	7	2	0	16	161	46.7%	Corner T.
P7	3.83	3.42	40	17	0	16	246	60.0%	Large T.
P8	3.75				0	13	149	40.0%	Blank
Mean	4.10	2.54	21.57	10.86	0.38	12.00	151.63	50.5%	
Median	3.92	3.13	22.00	12.00	0.00	12.00	155.00	53.0%	
Std. Dev.	1.28	1.14	12.30	7.06	0.74	4.14	60.36	11.3%	



(a) Exercise Time (h) (b) Exercise Iterations (c) Exercise Errors (d) Number of Rules (e) Lines of Code (f) Coverage

Figure 7.1. Synthesis process: refining the Statechart

condition, leading to a discrepancy between the intent and what was declared, that was then detected when trying out the generated editor.

Due to the corruption of one session's recording, it was not possible to extract all values for participant P8. Nevertheless, their provided solution model and questionnaire were unaffected and therefore used in the remaining results.

We also had three occasions — one from participant P1 and two from participant P3, as shown in column *Denied Solutions* in Table 7.1 — where the participant tried to use the tools in a way or intent not currently supported. Examples of these were having a sequence of events in a single rule as a form of chaining effects, instead of using rule conditions to coordinate them, or trying to directly call Javascript functions instead of relying on the provided abstraction.

7.3.2. Modeled Solutions

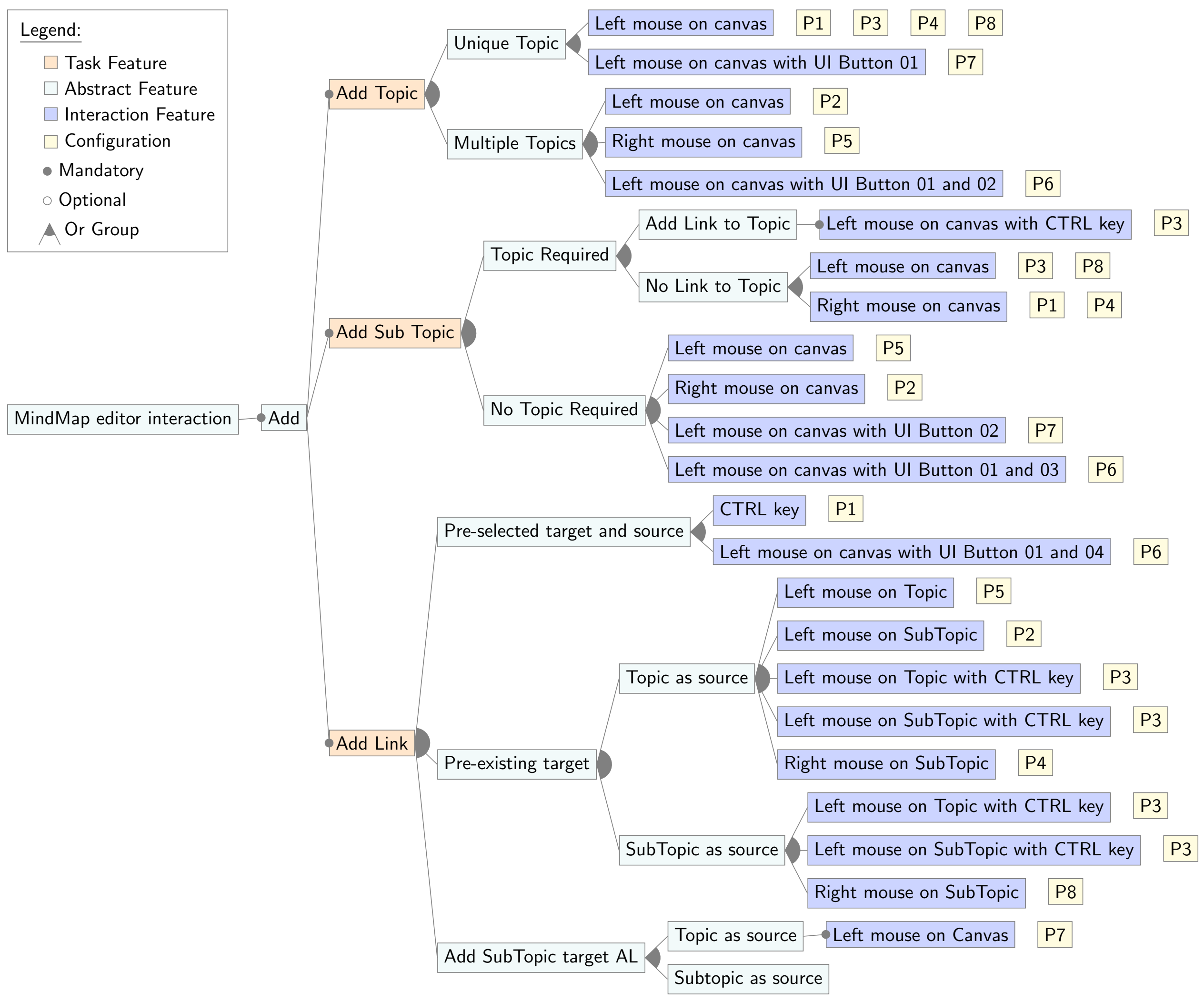


Figure 7.2. Feature diagram resulting from the Mind Map interaction rules classification.

From the collected modeled solutions data, we reverse engineered the feature diagram presented in Figures 7.2 and 7.3. The reverse engineering process was done by first building a base feature diagram to encode the interactions of the expected tasks, leading to the set of features from the root up to the task features. From the interaction model files,

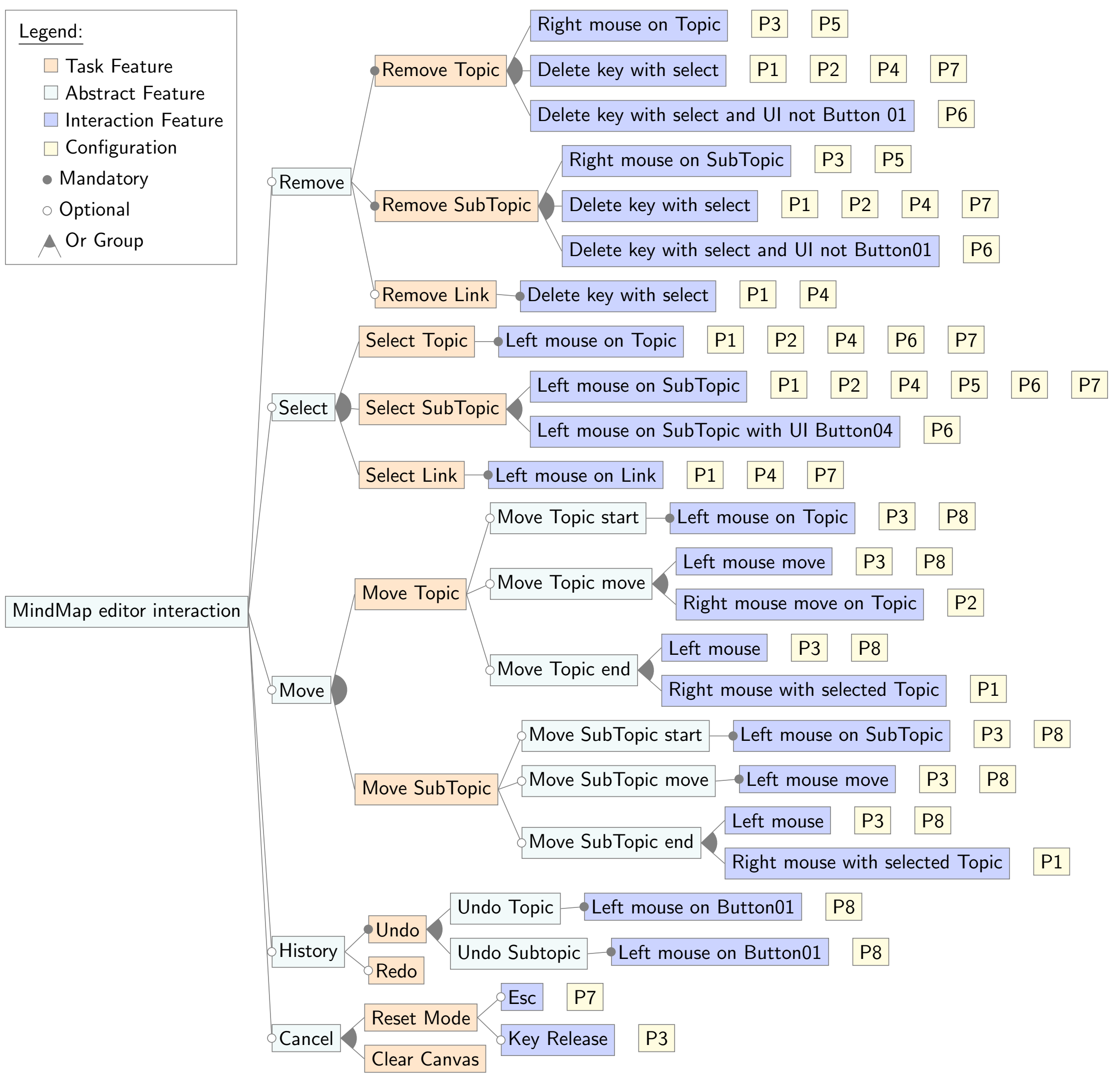


Figure 7.3. Remaining feature diagram from the Mind Map interaction rules classification.

we categorized each rule by their end user interaction and placed them as children of the appropriate task feature. For instance, participant P1 defined a rule that adds a Topic when a click from the left mouse button is made on the editor's canvas. Then, this rule's interaction category is added as a child feature of the task feature `Add Topic`. Because further effects and conditions may be present, when needed, sub-categorizations were added. This is the case with `Add Topic`, as some rules accounted for only allowing one Topic to be instantiated, while others left such evaluation to the back-end. When no differences were present from a usage point of view, these where aggregated as different configurations that made use of the same concrete interaction feature. So that when looking at the feature diagram in Figures 7.2 and 7.3 we read it as: *there is at least a rule with a specific interaction feature that provides the functionality of its ancestor task feature, and each annotated configuration* *is a participant's solution with a concrete rule that implements this task with this specific interaction.*

By assigning each participant's solution to their specific features, we can track how many of the expected modeling tasks each solution satisfies. We can then calculate the coverage as the percentage of said modeling tasks, that where satisfied by a particular participant's solution, as shown in Figure 7.1(f). On average this was a coverage of 50.5% with a standard deviation of 11.3. No solution provided interactions for all such modeling tasks, with the highest solution providing a coverage of 73.3%. On the other side of the spectrum all solutions fulfilled interactions for at least 40% of the expected modeling tasks.

To have an overview of the distribution of rules provided by the participants along the different tasks, and their variability, we compiled this distribution in the form of a histogram. Figure 7.4 shows this distribution, where each bar shows the number of rule instances that contribute to a specific task feature. The blue section of the bar is the number of distinctly unique rules and the striped orange section accounts for the number of submitted rules that are functionally indistinguishable from any of remaining distinct rules of that task. In this figure, we can easily see that in some tasks all rules were distinct between the participants, such as `Add Link`. This would show us that the approach does not restrict the expressed interactions to a strict form of interacting with the editor. But as we can see in the figure there are also tasks that show us a very small number of unique interactions with a large number of rules repeating such interactions, as is the case with `Select Sub Topic`. So it becomes difficult to have a clear assessment by looking at it task by task, and know which case may be an outlier if at all.

We can then globally quantify this restrictiveness by following the variability factor of a feature diagram, as described in [**21**], we calculate the variability of the modeling tasks and the variability of the interaction features that are present in the implementation of those tasks. The variability factor is a calculation of the ratio between the number of possible products and 2^n where n is the number of considered features, most commonly the number of leaves. So for the variability of the modeling tasks — that is considering the feature model from the root only until the `Task Features`— we get 1152 product configurations and 15 `Task Features`. This results in a variability score of $1\,152 \div 2^{15} = 0.035$. For the variability with the interaction features, we use the complete feature model, and we get $3.211E + 13$ product configurations — considering that there are no conflicts between rules of the same task and that the same task can effectively have more than one interaction — and 47 `Interaction Features`. This results in a variability score of $3.211E + 13 \div 2^{47} = 0.228$. Comparing the two variability values obtained from our results we conclude that the choices made by the participants on how to interact with the editor increased the variability by a factor of 6.49.

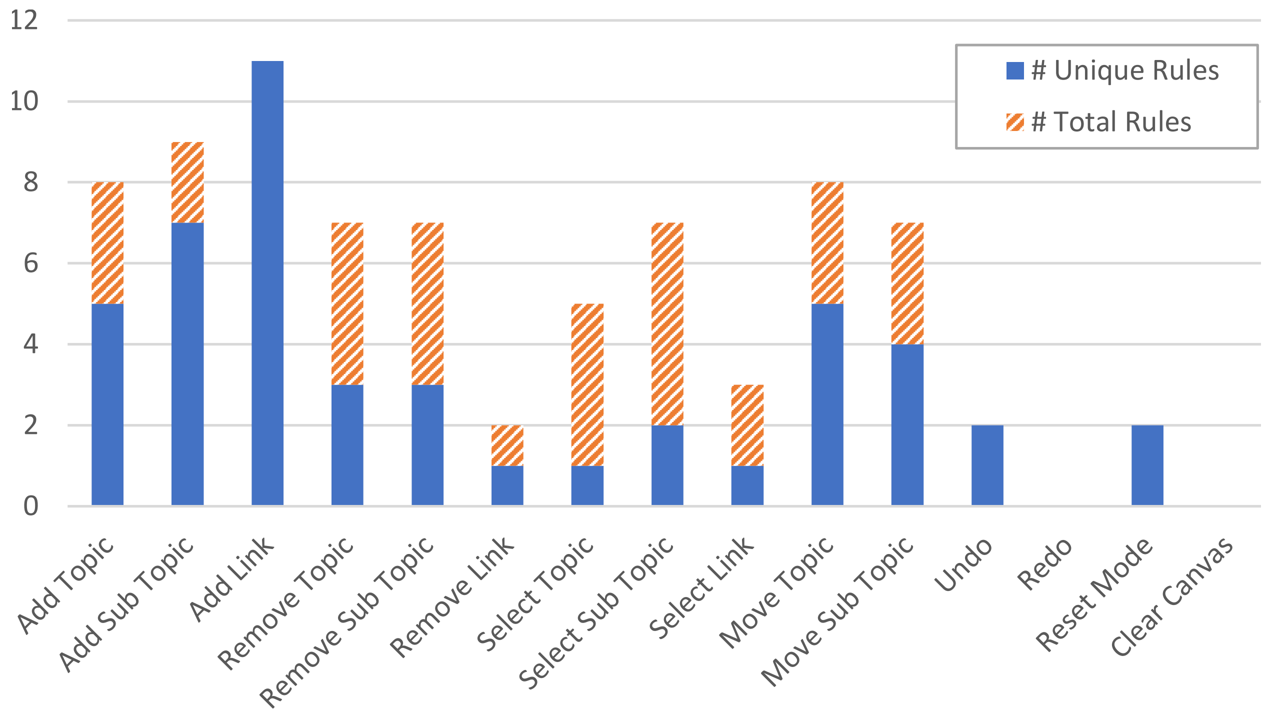


Figure 7.4. Diversity of rule implementations per task feature

7.3.3. Questionnaire

We finally interpret the answers to our questionnaire.

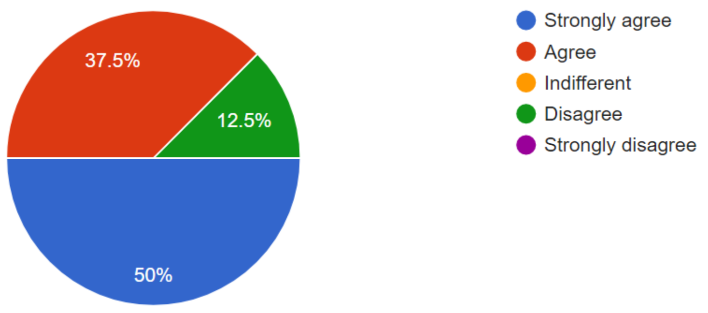


Figure 7.5. Answer distribution to Question 17

The most direct of the questions is Question 17: *It was easy for me to apprehend the concepts required by this approach.* While — as shown in Figure 7.5 — 87.5% agreed or strongly agreed, the one participant without a computer science background answered this question with `Disagree`. This tells us that, at the moment, we might have some bias towards having the abstraction geared to programmers. The final comments of this particular participant, clarify that their biggest hurdle was the textual concrete syntax. If it is the case that this bias is only at the level of the apprehension of the concrete syntax, then it will be easier to correct this aspect in the future without requiring a full rework of the presented modeling languages and tools.

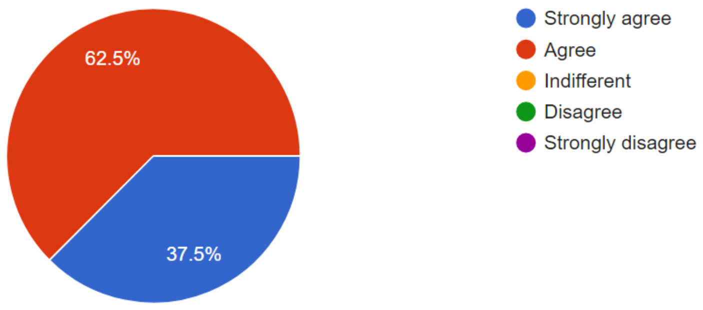


Figure 7.6. Answer distribution to Question 16

Taking this into account, we look at other aspects impacted by the abstraction level, such as expression of interaction intents. Question 16: *The presented approach allowed me to express the interactions I wished*, addresses this aspect. For this question — as shown in Figure 7.6 — all participants agreed or strongly agreed, indicating that once learned the description of the interaction rules did not pose a barrier in of itself.

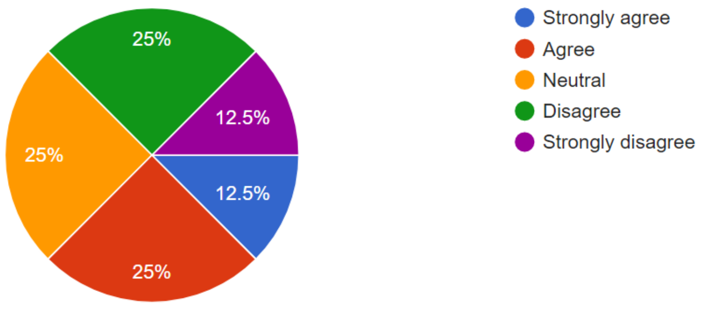


Figure 7.7. Answer distribution to Question 19

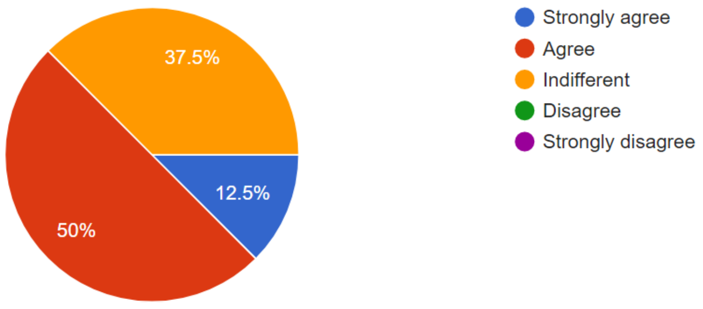


Figure 7.8. Answer distribution to Question 20

The use of the tools and approach were also inquired through Questions 19: *When using this approach, I spent most of my time fixing errors or trying to make the interactions work* *as I intended* and 20: *When using this approach, I tried new ways to express interactions or explored new interactions*. The answers to question 19 — as shown in Figure 7.7 — appear consistent with a normal distribution, where, if we considered that all participants had just learned to use the tools and process proposed, gives us leverage to interpret this aspect in favor of the overall evaluation. With Question 20 the answers — as shown in Figure 7.8 — 62.5% agreed or strongly agreed, where the remaining answers were indifferent. This tells us that even if, on average, close to half of the iterations were done to correct errors, the perception was that the approach and tools provided an efficient recovery to not make them a hurdle, contributing to a more exploratory feel, when developing the interactions.

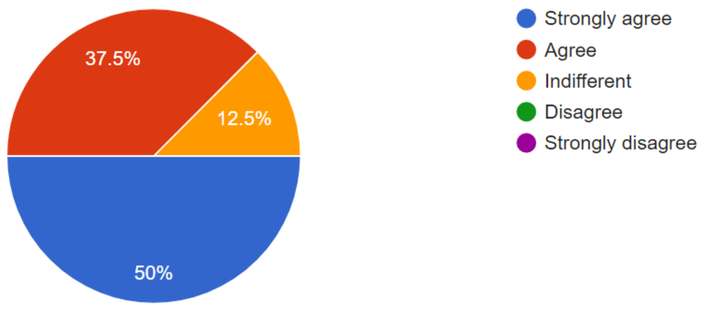


Figure 7.9. Answer distribution to Question 18

Finally, with Question 18: *I would recommend this approach to a colleague who needs to design a modeling editor with custom interactions*, 87.5% of participants agreed or strongly agreed — as shown in Figure 7.9. This is consistent with some of the participants expressing interest in knowing about future developments of the tested tools.

The remaining results of the questionnaire were used to identify any possible biases and misrepresentations from the participants sampling. All of these results are presented in Appendix F.

7.4. Discussion

We now look into what we can interpret from these result to answer our research questions, and wat limitations these results and consequent interpretations may have.

7.4.1. Does our proposed approach enforce specific interactions ?

To answer [RQ1] we looked at the results from the provided solutions shown in Section 7.3.2, to see how much they converged in implementation.

When we compared the variability values of the expected modeling tasks with the interactions present in the implementations, we observed a variability increase by a factor of 6.49. This shows us that the used tools and process do not restrict the produced editors to a specific way of interacting. If it were the case, the variability factor ratio would be much smaller, at an extreme even close to 1 — that is each modeling task would have only one possible interaction to trigger that task, preserving the number leaves and the number of product configurations. Of course this only tells us that the approach does not restrict the interactions expressed, work must still be done to properly explore that potential.

7.4.2. Are the process and level of abstraction appropriate ?

For [RQ2] only looking at the data collected during the exercise is not enough. The proportion of iterations and errors, or the time spent on the exercise do not provide a clear measurement of how adjusted the level of abstraction is. Instead, we look into the questionnaire results shown in Section 7.3.3 to clarify our data. Given these results we can conclude that although improvements are needed at the level of the concrete syntax and apprehension, of the interaction modeling language, the underlying concepts and the overall approach provide a good enough level of abstraction to allow a positive use of the proposed models, tools and process.

7.4.3. How effective is the approach at providing a complete editor?

With [RQ3] we first look at feature coverage of the provided solution. As presented in Table 7.1, and in Figure 7.1(f), the coverage is on average 50.5% with a maximum of 73.3% and no smaller than 40%. It largely differed from participant to participant, but we must also be aware there were occasions where the participants were worried on how long it would take to perform the exercise, focusing on speed and minimum features instead of completeness. Still, all strived to have what they considered a minimum of functionality, this being mostly adding and connecting `Topics` and `Subtopics`. Finally, even if no singular participant provided all aspects of functionality of an editor, their combined response made it clear that the tools can certainly provide it.

> *Where only two of the `Task Features` had no implementation, and even these would be implementable only using the interaction patterns expressed in the remaining features.*

Although for the most part the participants were satisfied with the provided back-end functionalities, there were occasions where more or different functionalities were requested. With a more thorough and robust set of back-end libraries the separation between interaction and back-end can be reinforced, so that a full editor can be achieved with only the provided modeling languages, and without having to tweak or update the back-end for every editor.

To further understand the effectiveness of our approach we also look into the answers of Questions 19 and 20. From these answers we concluded that even with some instances of relative high number of errors, they were not perceived as a hurdle. Still, such number of errors should be addressed. But we require further investigation to verify if these improve with more time to learn and get acquainted with the tools and approach, as it was a new development paradigm for the participants, or if further improvements to the tools are needed.

So we conclude that the provided approach and tools, allow for the development of complete editors. Although more time to learn the approach and to develop such editors may be required. Furthermore, we would need a direct comparison with traditional development and customization of an editor, for a full assessment of any development improvements. But if we extrapolate from what was achieved from on average three hours of development and 152 lines of code, we can have some confidence that that comparison would be positive enough, and that the proposed approach is a viable way of developing DSML editors.

7.4.4. Limitations

Due to cultural/educational components, it is possible to have different participants naturally provide the same answer; this will lower the variability score of the `Interaction Features` without it being the fault of the tools or approach being tested. We can see the possible presence of this limitation in the `Select Topic` feature, where all provided solutions were functionally similar, and in the choice of the layout model, where 75% of participants chose the `Blank` layout model. Still, the remaining factors are diverse enough to not appear to be heavily impacted by this limitation.

Because of the limitations of the experimental setup, both in terms of time to learn and use the tool being limited and the number of potential different interactions being limited to mouse and keyboard to facilitate the execution of the tests — instead of being open to further interactions such as touch, audio recognition or haptic feedback — the proper assessment of diversity may be limited.

Some care with the overall positive responses must also be taken, as some acquiescence bias may be present, due to the form of participant selection and their number.

Another limitation of this study was the time dedicated to the evaluation. We designed this study to be performed in four hours, as to strike a balance between giving enough time to the participants to get acquainted with the tools and approach, and be small enough that volunteers could take time in their schedules to participate. Given more time with each participant would have allowed for more in depth testing — such as editing more than just the interaction model, or having a richer set of potential tasks — and a better exploration of the capabilities of the approach.

But the most limiting factor is the number of participants. This affects our ability to establish statistical correlations — for example the difference between the participants with a background in UI and UX from those with a background in MDE. A major factor to the lower number of participants is the time requirements of the study. Even if we reduced as much as possible the time needed for the evaluation, this was still a large hurdle in recruiting the participants, especially from the industry. Still, in [**90**] a case is made for user testing with small groups of participants, as low as five, where no statistical significance can be established, but the UIs and UX of tools can be qualitatively evaluated, helping in identifying problems earlier during development. This approach is very much in line with other software development approaches, such as agile development. So considering that there is still much to be developed and researched as a continuation of this thesis, there is still plenty of value in such a study even with a small number of participants, as we can still draw conclusions that will help us guide these future developments.

Chapter 8

Conclusion

And to stand at either end and to look out off the end of the pier only, hoping out in that direction is the complete understanding, is a mistake.

Richard Feynman, Lecture 5 — The Distinction of Past and Future (1964)

8.1. Summary

In this thesis, we have addressed the explicit inclusion of user interaction when automatically generating DSML editors. For this, we defined four modeling languages to address the different aspects of user interaction implementation. We chose to use different modeling languages for each aspect as a way of tapping into the different sets of expertise required. Alongside these modeling languages, we have developed a set of tools that take the models of the different aspects of user interaction implementation and automatically generates fully functional DSML editors.

The execution semantics of these editors is anchored in the well-established Statecharts formalism. This allows us to reuse existing Statecharts tools and techniques, to analyze and improve the modeling editors generation and execution. This includes the Statecharts refinement technique presented in Chapter 4 that provides us semantic preservation guarantees when incrementally building Statecharts; and therefore providing us semantic preservation guarantees when incrementally building our modeling editors.

We also performed evaluations to both the efficiency of the tools and process developed, and its usability. From these we concluded that we can efficiently produce working DSML editors, but further improvements can be made, specifically at the level of the defined modeling languages concrete syntax.

In the end, we were successful in delivering a process of generating graphical DSML editors grounded on a set of modeling languages and associated tools, that allows for the focused development of interaction, promoting its further exploration. Through this exploration, interaction patterns and base models can be defined and reused across different editors. This also opens the door to new forms of interacting with graphical DSML editors, further expanding their use and accessibility.

8.2. Outlook

There is much that can be done to progress, improve, or leverage the present work; both from a technical standpoint and scientific research. This would encompass the full implementation of the coordination language, with assisted concrete syntax and associated operations on the interface model. These operations can be implemented through the use of declarative model transformation.

Further work can be done to refine and develop the usage of variables, make their use accessible and intuitive to non-programmers, *e.g.,* UX designers. These may entail the identification of variable usage patterns when sharing data between rules. Implementing variables as part of states instead of constraints, may also be investigated to improve refinement flexibility and possibly performance. This is only possible when variables are used to coordinate rules, as all possible values are known beforehand, unlike when variables are used for data sharing. So to implement such approach, both cases may have to be treated differently.

Improvements can be made to the concrete syntaxes of the proposed modeling languages. Namely, further distance the interaction language from programming paradigms and into more visual or by-example [**124, 66, 45**] specification of the interactions. Conversely, the inclusion of further abstraction mechanisms such as rule inheritance, element alias and other abstractions may also prove fruitful. Improvements to the mapping language concrete syntax to make use of recurring patterns, possibly through the use of templates [**86**].

Further flesh out the layout language syntax to make it closer to common design tools and to include the specification of topological constraints. Developments can also be made so that when specifying the concrete syntax of graphical DSMLs they also include elements and information to make them fully intractable. Such specifications would then be refined into the canvas layer, further leveraging the presented Statecharts refinement rules.

These improvements may also imply back-end development. Such as further development of the model rendering engine to facilitate model usage and editor development. This can be achieved making use of more performant rendering techniques and libraries, these would decrease the reliance on hard-coded concrete syntax element rendering and would allow for the inclusion of more dynamic constraints in the layout of elements of the editor. These improvements should also include robust usage of connectivity libraries to fully fledged modeling back-ends and the development of mapping libraries to be used and referenced in the mapping model. This would also open the interaction rules to properly use model querying as part of the conditions of rule application.

The developed tools can also be taken into new directions, such as usage of other Statecharts implementations such as XState [**14**] and Yakindu [**15**], or take it even further and generate offline editors in Python (as producible by SCCD) or other languages. The consequence of this is that each particular Statecharts implementation has its specific limitations leading to slightly different approach to their construction, and if all are taken into account, a more universal implementation of the proposed approach and tools.

On a more exploratory point of view, research can be done on interaction patterns and the process of providing feedback to the interaction design. These would entail the detection of conflicting interaction rules, both from implementation conflicts and best practices. These best practice interactions can also be further investigated, specially as we combine uncommon forms of interaction, akin to the work on concrete syntax presented in [**88**], but encompassing more mediums of interaction.

Combining the presented work with example-based learning such as presented in [**19**], methods of generating representative interaction test cases for stress testing or the automated creation of editor variations for best usage testing can be developed.

Explorations can also be made on non-standard interaction methods such as the usage of Augmented and Virtual Reality in modeling environments, smart devices or custom-built interaction solutions.

We can also explore how to extend the proposed approach to support multi-user input for collaborative modeling. This would require investigation on how to specify different users their focus and triggering events, on remote collaboration or in the same shared physical space.

Other avenues to further develop include expanding the proposed approach for multi-language support. This would entail the possibility of multiple canvases, and dynamic loading and unloading of interface elements and their corresponding Statecharts components.

Another point of development would be to extend the generation algorithm to be applicable on a pre-existing fully functioning statechart, instead of relying on the newly built one every time. This would imply saving the RETE network state for further use, extending it with new alpha and beta notes, and recalculate rule applicability without triggering the rule application of rules already refined in the base Statecharts being extended.

The creation of a library of common notable interactions, the automatic generation of baseline language interactions based on the concrete syntax to be customized by the interaction designer or the generation of templates based on pre-existing interactions would also facilitate the use of the tool.

Finally, generalizing the proposed approach to other applications outside DSML editors could also be pursued.

Références bibliographiques

[1] EMFText. `www.emftext.org`, 2007.

[2] Graphiti. `https://eclipse.org/graphiti/`, 2011.

[3] Concur Task Trees (CTT). `https://www.w3.org/2012/02/ctt/`, février 2012.

[4] Graphical Modeling Framework (GMF). `http://www.eclipse.org/gmf-tooling/`, 2014.

[5] Adobe Experience Design CC. `http://www.adobe.com/ca_fr/products/experience-design.html`, 2016.

[6] Fluid UI. `https://www.fluidui.com/`, 2016.

[7] MAX. `https://cycling74.com/products/max/`, 2016.

[8] Microsoft Visio. `https://products.office.com/fr-ca/visio/flowchart-software`, 2016.

[9] *MODELS '16: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, New York, NY, USA, 2016. ACM.

[10] Pidoco. `https://pidoco.com`, 2016.

[11] Sirius. `http://www.eclipse.org/sirius/`, 2016.

[12] UXPin. `https://www.uxpin.com/`, 2016.

[13] wireframe.cc. `https://wireframe.cc/`, 2016.

[14] XState. `https://xstate.js.org`, 2020.

[15] Yakindu. `https://www.itemis.com/en/yakindu/state-machine/`, 2020.

[16] Jean-Raymond Abrial, Matthew K O Lee, David S Neilson, P N Scharbach et Ib Holm Sørensen : The B-method. *In Formal Software Development Methods*, volume 552 de *LNCS*, pages 398–405. Springer, 1991.

[17] Fernando Alonso, José L. Fuertes, Ángel L. González et Loïc Martínez : User-Interface Modelling for Blind Users. *In Computers Helping People with Special Needs*, volume 5105 de *LNCS*, pages 789–796. Springer, 2008.

[18] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl et Bernhard Schätz : Autofocus 3: Tooling concepts for seamless, model-based development of embedded systems. *In Workshop on Model-based Architecting of Cyber-physical and Embedded Systems*, pages 19–26, 2015.

[19] Edouard Batot : From examples to knowledge in model-driven engineering : a holistic and pragmatic approach. 2019.

[20] Gregory Z. Bedny, Waldemar Karwowski et Inna Bedny : *Applying Systemic-Structural Activity Theory to Design of Human-Computer Interaction Systems*. CRC Press, Inc., Boca Raton, FL, USA, 1st édition, 2014.

[21] David Benavides, Sergio Segura et Antonio Ruiz-Cortés : Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35:615–636, 09 2010.

[22] Grady Booch, Ivar Jacobson et Jim Rumbaugh : Omg unified modeling language specification. *Object Management Group*, 1034:15–44, 2000.

[23] Pontus Boström, Lionel Morel et Marina Waldén : Stepwise Development of Simulink Models Using the Refinement Calculus Framework. *In Theoretical Aspects of Computing*, volume 4711 de *LNCS*, pages 79–93. Springer, 2007.

[24] Marco Brambilla et Piero Fraternali : *Interaction Flow Modeling Language*. OMG, février 2015. Version 1.0.

[25] Manfred Broy et Max Fuchs : The design of distributed systems - an introduction to focus. Rapport technique, Institut für Informatik – Technische Universität München, 1992.

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

[27] Judee K Burgoon, Joseph A Bonito, Bjorn Bengtsson, Carl Cederberg, Magnus Lundeberg et L Allspach : Interactivity in human–computer interaction: A study of credibility, understanding, and influence. *Computers in human behavior*, 16(6):553–574, 2000.

[28] Bill Buxton : *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[29] Danilo Caivano, Daniela Fogli, Rosa Lanzilotti, Antonio Piccinno et Fabio Cassano : Supporting end users to control their smart home: design implications from a literature review and an empirical investigation. *Journal of Systems and Software*, 144:295–313, 2018.

[30] Giuseppe Castagna : Covariance and Contravariance: Conflict Without a Cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, mai 1995.

[31] Stefano Ceri, Florian Daniel, Maristella Matera et Federico M. Facca : Model-driven development of context-aware web applications. *ACM Transactions Internet Technology*, 7(1), 2007.

[32] Stefano Ceri, Piero Fraternali et Aldo Bongio : Web modeling language (webml): A modeling language for designing web sites. *Comput. Netw.*, 33(1-6):137–157, 2000.

[33] Kai Chen, Janos Sztipanovits, Sherif Abdelwalhed et Ethan Jackson : Semantic anchoring with model transformations. *In* Alan Hartman et David Kreische, éditeurs : *Model Driven Architecture – Foundations and Applications*, pages 115–129, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[34] Krzysztof Czarnecki et Simon Helsen : Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[35] Luca de Alfaro et Thomas A. Henzinger : Interface automata. *SIGSOFT Software Engineering Notes*, 26(5):109–120, sep 2001.

[36] Romuald Deshayes et Tom Mens : Statechart modelling of interactive gesture-based applications. *In Proc. First International Workshop on Combining Design and Engineering of Interactive Systems through Models and Tools (ComDeis-Moto),. Lisbon, Portugal (September 2011), iNTERACT*, 2011.

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

[38] Andrew DILLON : Beyond usability: process, outcome and affect in human-computer interactions. *Canadian Journal of Library and Information Science*, 2002.

[39] Klaus R. DITTRICH, Stella GATZIU et Andreas GEPPERT : The active database management system manifesto: A rulebase of adbms features. *In Rules in Database Systems*, pages 1–17. Springer, 1995.

[40] Alan DIX, Janet E. FINLAY, Gregory D. ABOWD et Russell BEALE : *Human-Computer Interaction (3rd Edition)*. Prentice-Hall, Inc., USA, 2003.

[41] Sebastian ERDWEG, Tijs van der STORM, Markus VÖLTER, Meinte BOERSMA, Remi BOSMAN, William R. COOK, Albert GERRITSEN, Angelo HULSHOUT, Steven KELLY, Alex LOH, Gabriël D. P. KONAT, Pedro J. MOLINA, Martin PALATNIK, Risto POHJONEN, Eugen SCHINDLER, Klemens SCHINDLER, Riccardo SOLMI, Vlad A. VERGU, Eelco VISSER, Kevin van der VLIST, Guido H. WACHSMUTH et Jimi van der WONING : The state of the art in language workbenches. *In* Martin ERWIG, Richard F. PAIGE et Eric VAN WYK, éditeurs : *Software Language Engineering*, pages 197–217, Cham, 2013. Springer International Publishing.

[42] Michalis FAMELIS, Rick SALAY et Marsha CHECHIK : Partial Models: Towards Modeling and Reasoning with Uncertainty. *In International Conference on Software Engineering*, pages 573–583. IEEE Press, 2012.

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

[44] Robert FRANCE et Bernhard RUMPE : Model-driven Development of Complex Software: A Research Roadmap. *In Future of Software Engineering*, pages 37–54, Minneapolis, mai 2007. IEEE Computer Society.

[45] Thomas FUNKHOUSER, Michael KAZHDAN, Philip SHILANE, Patrick MIN, William KIEFER, Ayellet TAL, Szymon RUSINKIEWICZ et David DOBKIN : Modeling by example. *In ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, page 652–663, New York, NY, USA, 2004. Association for Computing Machinery.

[46] Miguel Andrés GAMBOA et Eugene SYRIANI : Automating activities in mde tools. *In Model-Driven Engineering and Software Development*, pages 123–133. SciTePress. Rome, 2016.

[47] Martin GARDNER : Mathematical Games: The fantastic combinations of John Conway's new solitaire game 'life'. *Scientific American*, 223(4):120–123, 1970.

[48] Michael GOOD : MusicXML for Notation and Analysis. *In The Virtual Score: Representation, Retrieval, Restoration*, volume 12 de *Computing in Musicology*, pages 113–124. MIT Press, Cambridge MA, 2001.

[49] Object Management GROUP : *Diagram Definition, Version 1.1*, 2015.

[50] Object Management GROUP : *Unified Modeling Language Superstructure*, 2.5.1 édition, dec 2017.

[51] Esther GUERRA et Juan de LARA : Event-Driven Grammars: Towards the Integration of Metamodelling and Graph Transformation. *In International Conference on Graph Transformation*, volume 3256 de *LNCS*, pages 54–69. Springer, 2004.

[52] David HAREL : Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, juin 1987.

[53] David HAREL et Eran GERY : Executable Object Modeling with Statecharts. *Computer*, 30(7):31–42, juillet 1997.

[54] David HAREL et Orna KUPFERMAN : On Object Systems and Behavioral Inheritance. *IEEE Transactions on Software Engineering*, 28(9):889–903, 2002.

[55] David HAREL et Amnon NAAMAD : The STATEMATE semantics of statecharts. *Transactions on Software Engineering and Methodology*, 5(4):293–333, octobre 1996.

[56] David HAREL et Bernhard RUMPE : Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, 2004.

[57] Thomas T. HEWETT, Ronald BAECKER, Stuart CARD, Tom CAREY, Jean GASEN, Marilyn MANTEI, Gary PERLMAN, Gary STRONG et William VERPLANK : Acm sigchi curricula for human-computer interaction. Rapport technique, ACM, New York, NY, USA, 1992.

[58] Ian HORROCKS : *Constructing the User Interface with Statecharts.* Addison-Wesley, 1999.

[59] Siw Elisabeth HOVE et Bente ANDA : Experiences from Conducting Semi-structured Interviews in Empirical Software Engineering Research. *In Proc. 11th IEEE Int. Software Metrics Symposium*, METRICS, pages 23–. IEEE Computer Society, 2005.

[60] Hejiao HUANG et Li JIAO : *Property-Preserving Petri Net Process Algebra in Software Engineering.* World Scientific Pub., 2012.

[61] IBM : *Rational Rhapsody*, 8.0 édition, 2012.

[62] Information Processing - Documentation Symbols And Conventions For Data, Program And Systems Flowcharts, Program Network Charts, And System Resources Chart. Standard, International Organization for Standardization, Geneva, CH, février 1985.

[63] ITU-T : *Specification and Description Language*, Z.100 édition, 2016.

[64] Kenneth E. IVERSON : Notation as a tool of thought. *Commun. ACM*, 23(8):444–465, 1980.

[65] Ankur JOSHI, Saket KALE, Satish CHANDEL et Dinesh PAL : Likert scale: Explored and explained. *British Journal of Applied Science & Technology*, 7:396–403, 01 2015.

[66] Gerti KAPPEL, Philip LANGER, Werner RETSCHITZEGGER, Wieland SCHWINGER et Manuel WIMMER : *Model Transformation By-Example: A Survey of the First Wave*, pages 197–215. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[67] Steven KELLY, Kalle LYYTINEN et Matti ROSSI : MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. *In* Juhani IIVARI, Kalle LYYTINEN et Matti ROSSI, éditeurs : *Conference on Advanced Information Systems Engineering*, volume 1080 de *LNCS*, pages 1–21, Crete, mai 1996. Springer-Verlag.

[68] Steven KELLY et Juha-Pekka TOLVANEN : *Domain-Specific Modeling: Enabling Full Code Generation.* John Wiley & Sons, 2008.

[69] Iyad KHADDAM, Nesrine MEZHOUDI et Jean VANDERDONCKT : Towards task-based linguistic modeling for designing guis. *In Proceedings of the 27th Conference on l'Interaction Homme-Machine*, IHM '15, New York, NY, USA, 2015. Association for Computing Machinery.

[70] Kyungdoh KIM, Robert W PROCTOR et Gavriel SALVENDY : The relation between usability and product success in cell phones. *Behaviour & Information Technology*, 31(10):969–982, 2012.

[71] Cornel KLEIN, Christian PREHOFER et Bernhard RUMPE : Feature specification and refinement with state transition diagrams. *In Workshop on Feature Interactions in Telecommunications Networks and Distributed System*, pages 284–297. IOS Press, 1997.

[72] Dimitrios Kolovos, Richard Paige et Fiona A. C. Polack : The epsilon object language (eol). *In Model Driven Architecture – Foundations and Applications*, pages 128–142. Springer, 2006.

[73] Dimitrios Kolovos, Richard Paige et Fiona A. C. Polack : The epsilon transformation language. *In Theory and Practice of Model Transformations*, pages 46–60. Springer, 2008.

[74] Dimitrios Kolovos, Louis Rose, Richard Paige et A Garcia-Dominguez : The epsilon book. *Structure*, 178:1–10, 2010.

[75] Thomas Kühne : Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, décembre 2006.

[76] Glenn Anthony Lewis : *Incremental Specification and Analysis in the Context of Coloured Petri Nets*. Thèse de doctorat, University of Tasmania, 2002.

[77] Barbara Liskov : Data Abstraction and Hierarchy. *SIGPLAN Notices*, 23(5):17–34, janvier 1987.

[78] Quan Long, Zongyan Qiu et Shengchao Qin : The Equivalence of Statecharts. *In* JinSong Dong et Jim Woodcock, éditeurs : *Formal Methods and Software Engineering*, volume 2885 de *LNCS*, pages 125–143. Springer, 2003.

[79] Juan Pablo López-Grao, José Merseguer et Javier Campos : From uml activity diagrams to stochastic petri nets: application to software performance engineering. *ACM SIGSOFT software engineering notes*, 29(1):25–36, 2004.

[80] Levi Lúcio, Moussa Amrani, Jürgen Dingel, Leen Lambers, Rick Salay, Gehan MK Selim, Eugene Syriani et Manuel Wimmer : Model transformation intents and their properties. *Software & systems modeling*, pages 1–38, 2014.

[81] Levi Lúcio, Eugene Syriani, Moussa Amrani et Qin Zhang : Invariant Preservation In Iterative Modeling. *In Model Evolution*. ACM, 2012.

[82] Paulo Cesar Masiero, José Carlos Maldonado et I G Boaventura : A reachability tree for statecharts and analysis of some properties. *Information and Software Technology*, 36(10):615–624, 1994.

[83] MathWorks : *Simulink User's Guide*, r2013b édition, septembre 2013.

[84] MathWorks : *Stateflow User's Guide*, r2013b édition, septembre 2013.

[85] Bertrand Meyer : *Reusable Software: the base object-oriented component libraries*. Prentice-Hall, 1994.

[86] Bart Meyers, Antonio Cicchetti, Esther Guerra et Juan de Lara : Composing textual modelling languages in practice. *In Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, MPM '12, page 31–36, New York, NY, USA, 2012. Association for Computing Machinery.

[87] Alvaro Miyazawa et Ana Cavalcanti : Refinement-oriented models of Stateflow charts. *Science of Computer Programming*, 77(10-11):1151–1177, 2012.

[88] Daniel Moody : The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *Transactions on Software Engineering*, 35(6):756–779, novembre 2009.

[89] Isaac Nassi et Ben Shneiderman : Flowchart techniques for structured programming. *ACM Sigplan Notices*, 8(8):12–26, 1973.

[90] Jakob Nielsen : Usability engineering at a discount. *In Proceedings of the Third International Conference on Human-Computer Interaction on Designing and Using Human-Computer Interfaces and Knowledge Based Systems (2nd Ed.)*, page 394–401, USA, 1989. Elsevier Science Inc.

[91] Julia Padberg, Magdalena Gajewsky et Claudia Ermel : Rule-Based Refinement of High-Level Nets Preserving Safety Properties. *Science of Computer Programming*, 40(1):97–118, 2001.

[92] Julia Padberg et Milan Urbášek : Rule-Based Refinement of Petri Nets: A Survey. *In Petri Net Technology for Communication-Based Systems*, volume 2472 de *LNCS*, pages 161–196. Springer, 2003.

[93] Barbara Paech et Bernhard Rumpe : A new concept of refinement used for behaviour modelling with automata. *In FME '94: Industrial Benefit of Formal Methods, Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 24-18, 1994, Proceedings*, pages 154–174, 1994.

[94] David Paquette et Kevin A. Schneider : *Task Model Simulation Using Interaction Templates*, pages 78–89. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[95] Fabio Paternò : Model-based design of interactive applications. *intelligence*, 11(4):26–38, 2000.

[96] Fabio Paternò : Concurtasktrees: An engineered notation for task models. *The Handbook of Task Analysis for Human-Computer Interaction*, page 483, 2003.

[97] Fabio Paternò : End user development: Survey of an emerging field for empowering people. *ISRN Software Engineering*, 2013, 2013.

[98] Fabio Paternò et Carmen Santoro : A logical framework for multi-device user interfaces. 06 2012.

[99] Gordon Plotkin : A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 07 2004.

[100] Christian Prehofer : Behavioral refinement and compatibility of statechart extensions. *Electronic Notes in Theoretical Computer Science*, 295:65–78, 2013.

[101] Christian Prehofer, Andreas Wagner et Yucheng Jin : A model-based approach for multi-device user interactions. *In Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, pages 13–23, New York, NY, USA, 2016. ACM.

[102] Louis M Rose, Nicholas Matragkas, Dimitrios S Kolovos et Richard F Paige : A feature model for model-to-text transformation languages. *In Proceedings of the 4th International Workshop on Modeling in Software Engineering*, pages 57–63. IEEE Press, 2012.

[103] Jean Michel Rouly, Jonathan D Orbeck et Eugene Syriani : Usability and suitability survey of features in visual ides for non-programmers. *In Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 31–42. ACM, 2014.

[104] Mar Yah Said, Michael Butler et Colin Snook : A method of refinement in UML-B. *Software & Systems Modeling*, 14(4):1557–1580, 2015.

[105] Arvind Satyanarayan, Kanit Wongsuphasawat et Jeffrey Heer : Declarative interaction design for data visualization. *In Symposium on User Interface Software and Technology*, pages 669–678. ACM, 2014.

[106] Anthony Savidis et Constantine Stephanidis : Developing Dual User Interfaces for Integrating Blind and Sighted Users : the HOMER UIMS. *In CHI'95 Proceedings*. ACM, 1995.

[107] Douglas C. Schmidt : Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[108] Christophe Scholliers, Lode Hoste, Beat Signer et Wolfgang De Meuter : Midas: A declarative multi-touch interaction framework. *In International Conference on Tangible, Embedded, and Embodied Interaction*, pages 49–56. ACM, 2011.

[109] Peter Scholz : A refinement calculus for statecharts. *In Fundamental Approaches to Software Engineering*, volume 1382 de *LNCS*, pages 285–301. Springer, 1998.

[110] Peter SCHOLZ : Incremental design of statechart specifications. *Science of Computer Programming*, 40(1):119–145, 2001.

[111] Bran SELIC : A short catalogue of abstraction patterns for model-based software engineering. *International Journal of Software and Informatics*, 5(2):313–334, 01 2011.

[112] Maxwell SHINN : *Instant MuseScore*. Packt Publishing Ltd, 2013.

[113] Thomas STAHL, Markus VOELTER et Krzysztof CZARNECKI : *Model-Driven Software Development – Technology, Engineering, Management*. John Wiley & Sons, 2006.

[114] Dave STEINBERG, Frank BUDINSKY, Marcelo PATERNOSTRO et Ed MERKS : *EMF: Eclipse Modeling Framework*. Addison Wesley Professional, 2nd édition, 2008.

[115] Alan B. STERNECKERT : *Critical Incident Management*. Taylor & Francis, 2003.

[116] Egene SYRIANI, Levi LÚCIO et Vasco SOUSA : Structure and behavior preserving statecharts refinements. *Science of Computer Programming*, 170:45–79, 2019.

[117] Eugene SYRIANI et Hans VANGHELUWE : De-/re-constructing model transformation languages. *Electronic Communications of the EASST*, 29, 2010.

[118] Eugene SYRIANI, Hans VANGHELUWE, Raphael MANNADIAR, Conner HANSEN, Simon VAN MIERLO et Hüseyin ERGIN : Atompm: A web-based modeling environment. *In Demos/Posters/StudentResearch@ MoDELS*, pages 21–25. Citeseer, 2013.

[119] Nora SZASZ et Pedro VILANOVA : Behavioral refinements of UML-Statecharts. Rapport technique RT 10-13, Universidad de la República Montevideo, 2010.

[120] Zoltán UJHELYI, Gábor BERGMANN, Ábel HEGEDÜS, Ákos HORVÁTH, Benedek IZSÓ, István RÁTH, Zoltán SZATMÁRI et Dániel VARRÓ : Emf-incquery: An integrated development environment for live model queries. *Science of Computer Programming*, 98:80–99, 2015.

[121] Blase UR, Elyse MCMANUS, Melwyn PAK YONG HO et Michael L. LITTMAN : Practical trigger-action programming in the smart home. *In SIGCHI Conference on Human Factors in Computing Systems*, pages 803–812. ACM, 2014.

[122] Blase UR, Melwyn PAK YONG HO, Stephen BRAWNER, Jiyun LEE, Sarah MENNICKEN, Noah PICARD, Diane SCHULZE et Michael L. LITTMAN : Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes. *In SIGCHI Conference on Human Factors in Computing Systems*, pages 3227–3231. ACM, 2016.

[123] Simon VAN MIERLO, Yentl VAN TENDELOO, Bart MEYERS, Joeri EXELMANS et Hans VANGHELUWE : SCCD: SCXML extended with class diagrams. *In Workshop on Engineering Interactive Systems with SCXML, part of EICS 2016*, 2016.

[124] Dániel VARRÓ : Model transformation by example. *In* Oscar NIERSTRASZ, Jon WHITTLE, David HAREL et Gianna REGGIO, éditeurs : *Model Driven Engineering Languages and Systems*, pages 410–424, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[125] Dániel VARRÓ, Gábor BERGMANN, Ábel HEGEDÜS, Ákos HORVÁTH, István RÁTH et Zoltán UJHELYI : Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software and System Modeling*, 15(3):609–629, 2016.

[126] Vlatko ČERIC : Hierarchical abilities of diagrammatic representations of discrete event simulation models. *In Proceedings of the 26th Conference on Winter Simulation*, WSC '94, pages 589–594, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[127] John Vines : The failure of designers thinking about how we think: The problem of human-computer interaction. *In Transtechnology Research Reader 2010*, pages 98–105. Plymouth, University of Plymouth, 2010.

[128] Markus Voelter, Janet Siegmund, Thorsten Berger et Bernd Kolb : Towards user-friendly projectional editors. *In* Benoît Combemale, David J. Pearce, Olivier Barais et Jurgen J. Vinju, éditeurs : *Software Language Engineering*, pages 41–61, Cham, 2014. Springer International Publishing.

[129] Andreas Wagner et Christian Prehofer : Translating task models to state machines. *In Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, pages 201–208, 2016.

[130] Jos B Warmer et Anneke G Kleppe : The object constraint language: Precise modeling with uml (addison-wesley object technology series). 1998.

[131] Dustin Wüest, Norbert Seyff et Martin Glinz : Flexisketch: A mobile sketching tool for software modeling. *In International Conference on Mobile Computing, Applications, and Services*, pages 225–244. Springer, 2012.

Appendix A

Statechart Refinement Proof

A.1. Lemma 1: Local Rule Application is a (Local) Refinement

Refinement rules create (local) refinements restricted to the elements of the statechart being refined and satisfy the five refinement conditions in Definition 1. Formally, given a statechart SC, and a local rule application (SC,R,SC_r) , we have that $SC_{lo} \overset{R}{\preceq} SC_{lr} \in \Re$ where: SC_{lo} is the statechart that defines the local scope of application of the rule, as stated in Section 4.1 ; and SC_{lr} is the statechart composed of only the states and transitions of SC_r that are mapped by R onto SC.

Proof. Condition (4.3.1): From Definition 2, regardless of the refinement rule used, every state or transition in SC_{lr} is mapped to exactly one state or transition in SC_{lo}, hence R^{-1} is surjective.

Condition (4.3.2): Trivially true for *Basic-state to Basic-state* and *Basic-state to OR-state* refinement rules, because no state hierarchy exists in the state s being refined. When an *OR-state to AND-state* refinement rule is applied to a state s (Rule 3), for any state p such that $(p,s) \in in_l$ we have that $(p_r,s_r) \in in^*_{lr}$, where (p,p_r),$(s,s_r) \in R$ given p remains indirectly attached to s via the newly created AND-state. When an *AND-state to AND-state*, *transition*, *state extention* or *path* refinement rule is applied (Rule 4, 5, 7 and 8), all the state hierarchy is preserved given that by Rule 4 $in_l \subseteq in_{lr}$ and that by Definition 2 we have $(s,s) \in R$ for any $s \in S$ and $(t,t) \in R$ for any $t \in T$.

Condition (4.3.3) is satisfied by design since, regardless of the refinement rule used, new variables can be introduced in the entry/exit actions of the new states in S_{lr} and new events can be introduced on the new transitions in T_{lr}. All refinement rules Rules 1–8 guarantee this condition by construction.

Condition (4.3.4): Trivially true for any state refinement rule (Rules 1–4, 7 and 8), because the transitions going inside and outside of the state being refined are not touched by the refinement, and in particular in Rules 7–8, by construction any added transition does not interfere with pre-existing transitions. For the transition refinement rule (Rules 5–6), the source and target states of a refined transition t are preserved as well as all transitions adjacent to them. A new transition is built from the source state of t into the initial state of the statechart used as a parameter for the refinement. Another new transition is built from the parameter statechart end states into the target state of t, thus satisfying the condition.

Condition (4.3.5):Reachability preservation is guaranteed in three ways. First, since we only modify new variables, any state reachability that depends on $en(s)$ and $ex(s)$ is preserved in $en_r(s)$ and of $ex_r(s)$. This means that we have $\forall v \in V \cap V_r, g(\xi(v)) \rightarrow g_r(\xi_r(v))$ and any action on variables in $V_r \setminus V$ does not impact reachability. Second, through Constraint 2, we only allow a new event broadcast if there exists a queue where its transition is not enabled. These new broadcasts could trigger events in such an order that would render some states of the statechart inaccessible. By only allowing these new broadcasts with the presence of guards, we establish a range of executions where the new broadcast is not triggered, resulting in the preservation of the original reachability conditions. Third, with Constraint 3, we establish that the application of Rules 5–6 will not introduce dead-ends where previously there was a path.

□

A.2. Property 1: Refinement Reflexivity

The identity is a refinement $SC \overset{R_{id}}{\preceq} SC \in \Re$.

Proof. To demonstrate the reflexivity of $\Re$, we need to show that each refinement condition allows for the identity and consequently from Lemma 1, each refinement rule allows for the identity.

Condition (4.3.1): Since the identity mapping is a bijection, then R^{-1} is surjective because $id^{-1} = id$.

Condition (4.3.2): Holds because $(s,s') \in in$ implies $(s,s') \in in_r$ and $(s,s),(s',s') \in R$.

Condition (4.3.3): Satisfied because $\subseteq$ is reflexive.

Condition (4.3.4): We have that $\forall (s,s),(s',s') \in R$, if $s \xrightarrow{e[p]/x} s' \in T$ then $s \xrightarrow{e[p]/x} s' \in T_r$, and thus all connections in the statechart are maintained.

Condition (4.3.5): By definition of the identity, all transition guards, events and broadcast are the same for any t and t_r where $(t,t_r) \in R$. Also, all entry and exit actions of states are the same for any s and s_r where $(s, s_r) \in R$. Therefore, if a transition t is enabled in SC, its corresponding refined transition t_r will be enabled in SC_r. Hence if a state s is reachable in SC, its corresponding refined state s_r will be enabled in SC_r. □

A.3. Lemma 2: Rule Application is a Refinement

The application of a *state refinement* of a *transition refinement* rule is a refinement $merge(id, loc) = SC \overset{R}{\preceq} SC_r \in \Re$ where $id = SC \overset{R_{id}}{\preceq} SC$ and $loc = SC \overset{R_{loc}}{\preceq} SC_r$.

Proof. We can always separate the merge of R_{id} with a local refinement R_{loc} in three parts: $R_1 = R_{id} \setminus R_{loc}$, $R_2 = R_{loc} \setminus R_{id}$ and $R_3 = R_{id} \cap R_{loc}$.

Condition (4.3.1): $R_1 \cup R_3$ is inversely surjective because it is a subset of R_{id}. From Lemma 1, we have that R_{loc}^{-1} is surjective. Furthermore, note that the domain of R_{loc}^{-1} has no common elements with the domain of R_{id}^{-1}. Therefore R_1, R_2, and R_3 are each inversely surjective, hence $merge(id, loc) = R_1 \cup R_2 \cup R_3$ is inversely surjective.

Condition (4.3.2): Because $R_1 \cup R_3$ are a subset of R_{id}, and by definition of identity all containment relations are kept unchanged. By construction Rule 3 ensures that the altered containments relation maintain a path to the originally contained state, and all remaining Rules are only able to add to the hierarchy, so R_2 preserves its hierarchy. Because hierarchy relations are unique the $merge(id,loc) = R_1 \cup R_2 \cup R_3$ maintains the hierarchies unchanged.

Condition (4.3.3): Like before, because $R_1 \cup R_3$ is a subset of R_{id}, they keep all events and variables unaltered. Because the refinement rules can not alter or remove events and variables, only create new ones, we have that R_2 also preserves this condition. Because we can only add events and variables, the $merge(id,loc) = R_1 \cup R_2 \cup R_3$, will also only add events and variables, making it still preserve the condition.

Condition (4.3.4): From Lemma 1, we have that R_{loc} is a refinement, so $R_2 \cup R_3$ conforms to this condition. By definition, R_{id} also preserves the connectivity between states. Because the connectivity in $R_1 \cup R_3$ and in $R_2 \cup R_3$ and there is no path that connects between states of R_1 and R_2 without passing through R_3, the connectivity of $merge(id,loc)$ is also preserved.

Condition (4.3.5): Similarly, by definition of identity, $R_1 \cup R_3$ maintains its reachability, and as shown in Lemma 1, $R_2 \cup R_3$ also holds this condition, guarantying that it cannot introduce new actions that affect pre-existing variables, including variables that exist in $R_1 \cup R_3$. Because pre-existing action cannot affect new Variables and we guard against new actions affecting pre-existing variables, $merge(id,loc)$ will preserve the reachability of all its states. □

A.4. Property 2: Refinement Transitivity

If $SC \overset{R_1}{\preceq} SC_{r_1} \in \Re$ and $SC_{r_1} \overset{R_2}{\preceq} SC_{r_2} \in \Re$, then there exists a refinement $SC \overset{R_3}{\preceq} SC_{r_2} \in \Re$.

Proof. To demonstrate the transitivity of $\Re$, we need to show that each refinement condition is transitive.

Condition (4.3.1): Since R_1^{-1} and R_2^{-1} are surjective, then $R_3^{-1} = R_1^{-1} \circ R_2^{-1}$ is surjective.

Condition (4.3.2) is satisfied because *in* is transitive.

Condition (4.3.3): Satisfied because $\subseteq$ is transitive.

Condition (4.3.4): Suppose we have $s \xrightarrow{e[p]/x} s' \in SC$. If we name the resulting connectivity results presented in Condition (4.3.4) as (1) $\exists s_r \xrightarrow{e[g]/x} s'_r \in SC_r$; (2) $\exists s_r \xrightarrow{e[g]} s''_r \xrightarrow{\varphi[true]/x} s'_r \in SC_r$; and (3) $\exists s_r \xrightarrow{e[g]} s''_r \xrightarrow{+} s'''_r \xrightarrow{\varphi[true]/x} s'_r \in SC_r^+$. The application of $SC \overset{R_1}{\preceq} SC_{r_1}$ will produce (1), (2) or (3). We can apply R_2 to any of these result, where if R_1 resulted in (1) the application of $SC_{r_1} \overset{R_2}{\preceq} SC_{r_2}$ to it will produce results of type (1), (2) or (3). If R_2 resulted in (2) the application of R_2 will result in (2) or (3). And if R_1 results in (3), R_2 will result in (3). In any of these cases they remain within the the results (1), (2) and (3) that preserve the connectivity of the Statechart.

Condition (4.3.5): Knowing that R_1 and R_2 are refinements. If we assume that $R_3 = R_1 \circ R_2$ is not a refinement, implying that $\nexists Q_{r_2}$ that allows $I_{r_2}, Q_{r_2} \xrightarrow{bigstep} (\bar{S}_{r_2}, \xi_{r_2}, \eta_{r_2})$. Then as R_3 produces the same refinement as $R_1 \circ R_2$ this means that either the application of R_1 or R_2 is preventing the existence of Q_{r_2}, implying that one of them is not a refinement, leading to a contradiction. Therefore there must be at least one Q_{r_2}, proving that R_3 must be refinement.

□

A.5. Property 3: Sequence of Rule Applications is a Refinement

$\forall n \geq 3$, the composition of rule applications $SC_1 \overset{R_1}{\preceq} SC_2 \overset{R_2}{\preceq} \ldots \overset{R_{n-1}}{\preceq} SC_n$ is a refinement.

Proof. The proof follows from Lemma 2 and Property 2. □

Appendix B

GoL Statechart refinement steps

In this appendix, we show all the steps for the Statechart refinmenet GoL example. In these steps we show the evaluation of the RETE network, with positive values in black, negative values in grey and unchanged values that do not need to be propagated as dashed lines; the state of the statechart after each refinement step; and the status of the evaluation queue, with the set of active states, presented from left to right as corresponding to layers L1 to L3, and the set of RETE token values of each of the queued configurations, presented from left to right in the same left to right orther as the RETE nodes presented in the figures, with first the eight pairs of tokens for the α-nodes, folowed by the set of three tokens for the $\beta_{\vee}$-nodes.

B.1. Initial Configuration

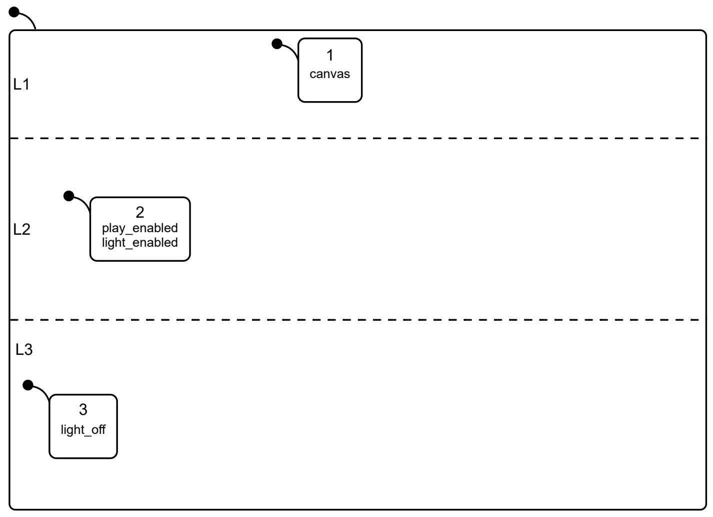


Figure B.1. Initial GoL Statechart

evalQueue = $\{\langle\{(1,ME1),(2,ME2),(3,ME3)\}$, $\{(false,true),(false,true),(true,true),$ $(true,true),(false,true),(true,true),(true,true),(true,true)\}$, $\{(false),(false),(false)\}\rangle\}$

B.2. RETE evaluation 1

evalQueue = $\{\langle\{(1,ME1),(2,ME2),(3,ME3)\}, \{(false,true),(false,true),(true,true),(true,true),(false,true),(true,true),(true,true),(true,true)\}, \{(false),(false),(false)\}\rangle\}$

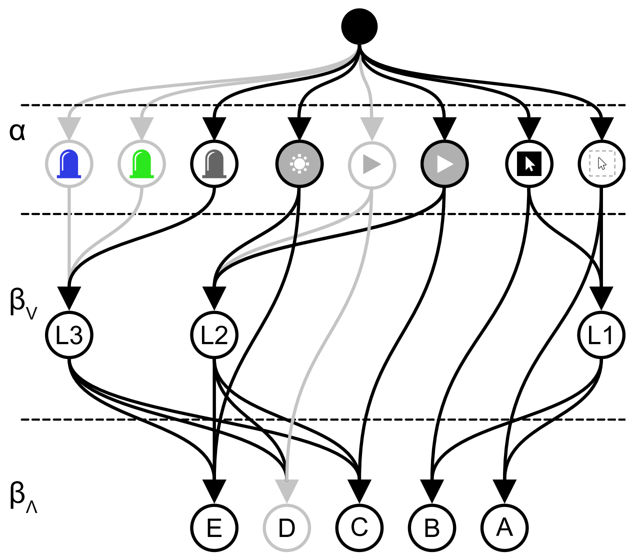


Figure B.2. RETE evaluation for configuration $\{(1,ME1),(2,ME2),(3,ME3)\}$

B.2.1. Rule A refinement

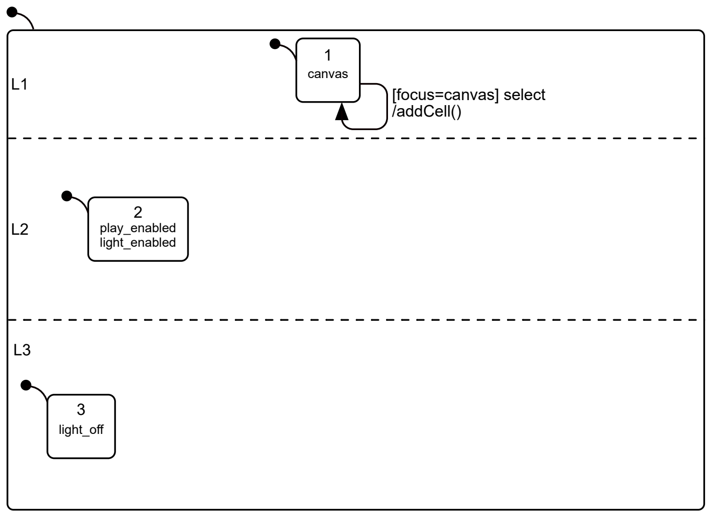
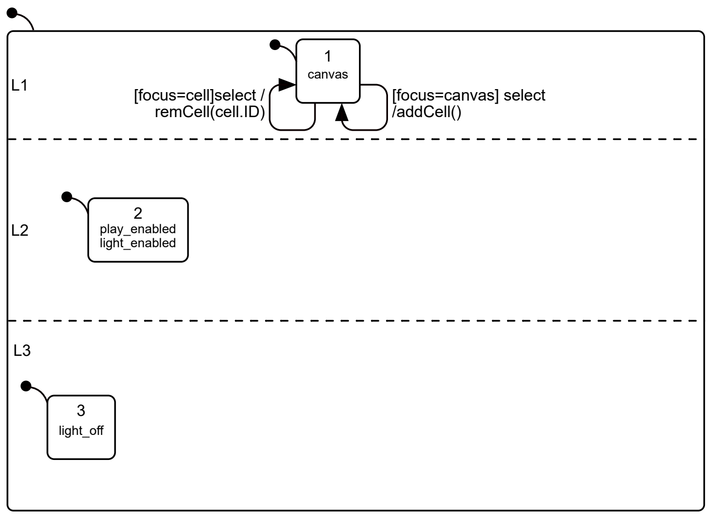


Figure B.3. Statechart after rule A is refined in configuration $\{(1,ME1),(2,ME2),(3,ME3)\}$

evalQueue = $\{\}$

B.2.2. Rule B refinement

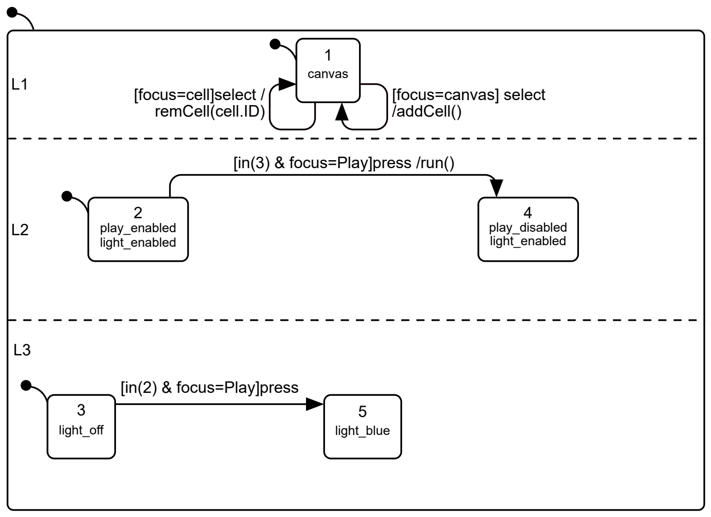


L1
1
canvas
[focus=cell]select / remCell(cell.ID)
[focus=canvas] select /addCell()
L2
2
play_enabled
light_enabled
L3
3
light_off

Figure B.4. Statechart after rule B is refined in configuration $\{(1,ME1),(2,ME2),(3,ME3)\}$

evalQueue $= \{\}$

B.2.3. Rule C refinement

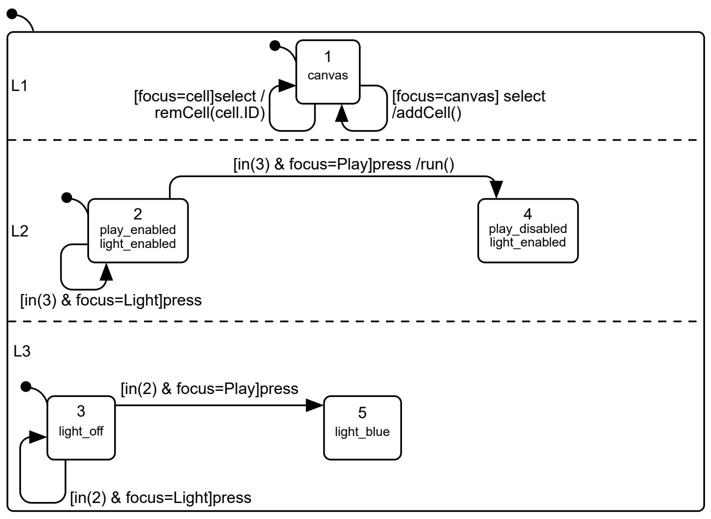


L1
1
canvas
[focus=cell]select / remCell(cell.ID)
[focus=canvas] select /addCell()
[in(3) & focus=Play]press /run()
L2
2
play_enabled
light_enabled
4
play_disabled
light_enabled
L3
[in(2) & focus=Play]press
3
light_off
5
light_blue

Figure B.5. Statechart after rule C is refined in configuration $\{(1,ME1),(2,ME2),(3,ME3)\}$

evalQueue $= \{\langle\{(1,ME1),(4,ME2),(5,ME3)\},\ \{(true,true),(false,false),(false,true),$ $(true,true),(true,true),(false,true),(true,false),(true,false)\},\ \{(false),(false),(false)\}\rangle\}$

B.2.4. Rule E refinement



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

Figure B.6. Statechart after rule E is refined in configuration $\{(1{,}ME1){,}(2{,}ME2){,}(3{,}ME3)\}$

evalQueue $= \{\langle\{(1{,}ME1){,}(4{,}ME2){,}(5{,}ME3)\}{,}\ \{(true{,}true){,}(false{,}false){,}(false{,}true){,}$ $(true{,}true){,}(true{,}true){,}(false{,}true){,}(true{,}false){,}(true{,}false)\}{,}\ \{(false){,}(false){,}(false)\}\rangle\}$

B.3. RETE evaluation 2

evalQueue $= \{\langle\{(1{,}ME1){,}(4{,}ME2){,}(5{,}ME3)\}{,}\ \{(true{,}true){,}(false{,}false){,}(false{,}true){,}$ $(true{,}true){,}(true{,}true){,}(false{,}true){,}(true{,}false){,}(true{,}false)\}{,}\ \{(false){,}(false){,}(false)\}\rangle\}$

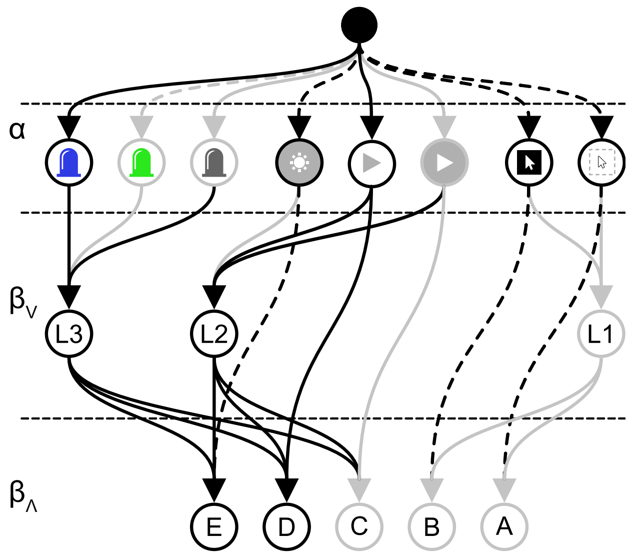
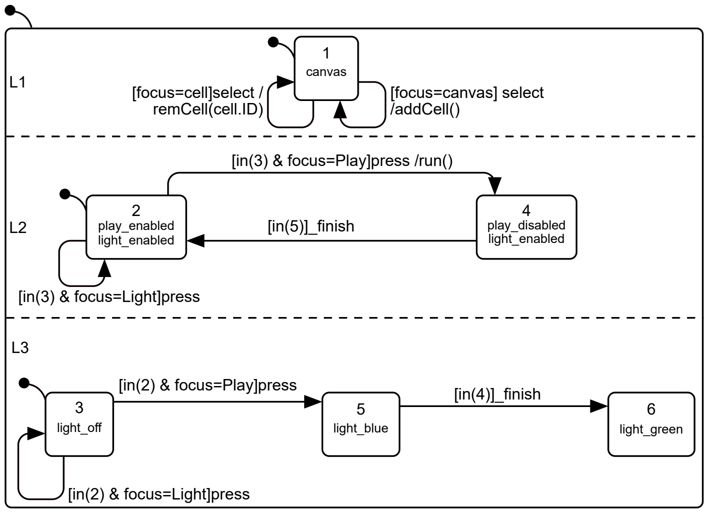


Figure B.7. RETE evaluation for configuration $\{(1{,}ME1){,}(4{,}ME2){,}(5{,}ME3)\}$

B.3.1. Rule D refinement

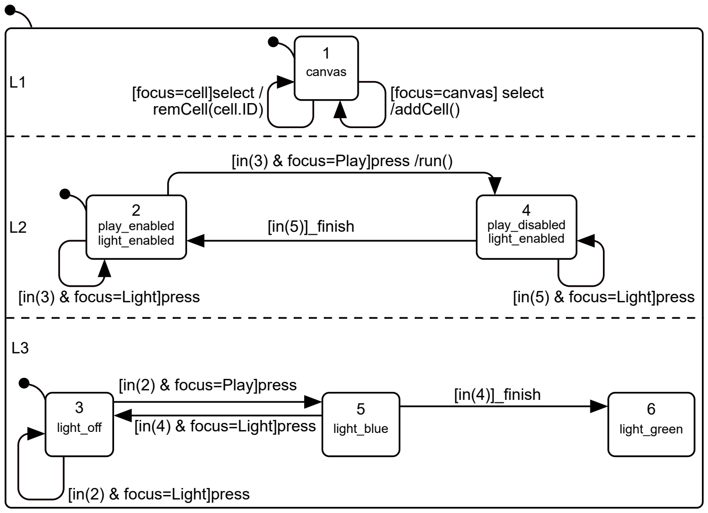


L1
1
canvas
[focus=cell]select / remCell(cell.ID)
[focus=canvas] select /addCell()
L2
[in(3) & focus=Play]press /run()
2
play_enabled
light_enabled
[in(5)]_finish
4
play_disabled
light_enabled
[in(3) & focus=Light]press
L3
[in(2) & focus=Play]press
3
light_off
5
light_blue
[in(4)]_finish
6
light_green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

Figure B.8. Statechart after rule D is refined in configuration $\{(1,ME1),(4,ME2),(5,ME3)\}$

evalQueue $= \{\langle\{(1,ME1),(2,ME2),(6,ME3)\},\ \{(false,true),(true,true),(false,false),(true,false),(false,true),(ture,true),(true,false),(true,false)\},\ \{(false),(false),(false)\}\rangle\}$

B.3.2. Rule E refinement



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

Figure B.9. Statechart after rule E is refined in configuration $\{(1,ME1),(4,ME2),(5,ME3)\}$

evalQueue $= \{\langle\{(1,ME1),(2,ME2),(6,ME3)\},\ \{(false,true),(true,true),(false,false),(true,false),(false,true),(ture,true),(true,false),(true,false)\},\ \{(false),(false),(false)\}\rangle,$ $\langle\{(1,ME1),(4,ME2),(3,ME3)\},\{(false,true),(false,false),(true,true),(true,false),(true,false),(false,false),(true,false),(true,false)\},\ \{(false),(false),(false)\}\rangle\}$

B.4. RETE evaluation 3

evalQueue $= \{\langle\{(1,ME1),(2,ME2),(6,ME3)\}, \{(false,true),(true,true),(false,false),(true,false),(false,true),(ture,true),(true,false),(true,false)\}, \{(false),(false),(false)\}\rangle, \langle\{(1,ME1),(4,ME2),(3,ME3)\}, \{(false,true),(false,false),(true,true),(true,false),(true,false),(false,false),(true,false),(true,false)\}, \{(false),(false),(false)\}\rangle\}$

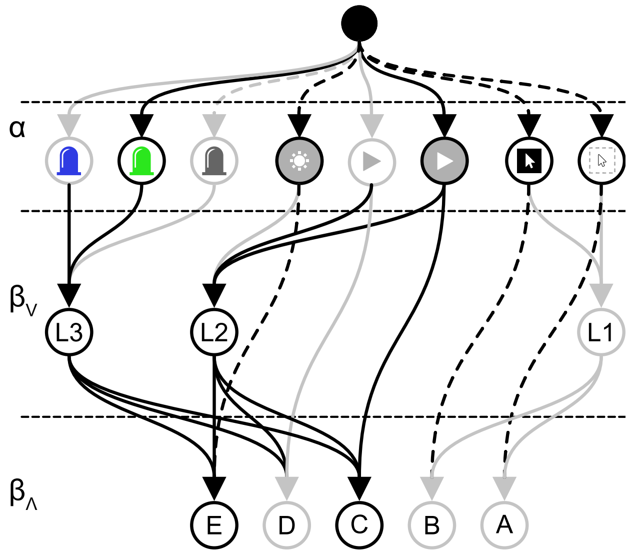


Figure B.10. RETE evaluation for configuration $\{(1,ME1),(2,ME2),(6,ME3)\}$

B.4.1. Rule C refinement

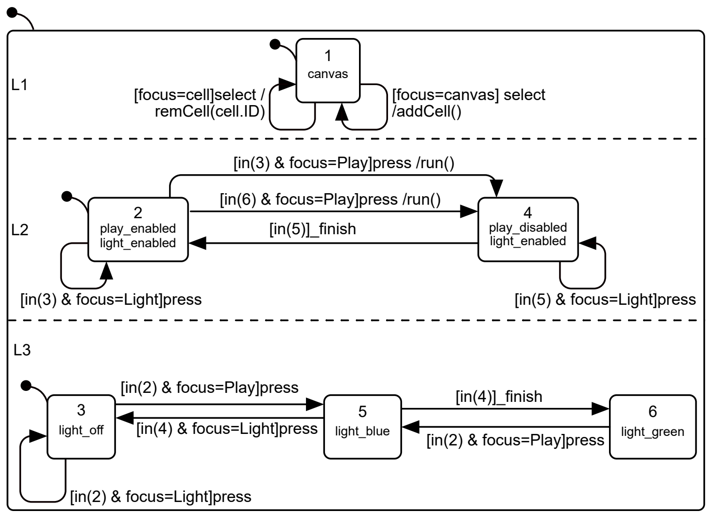
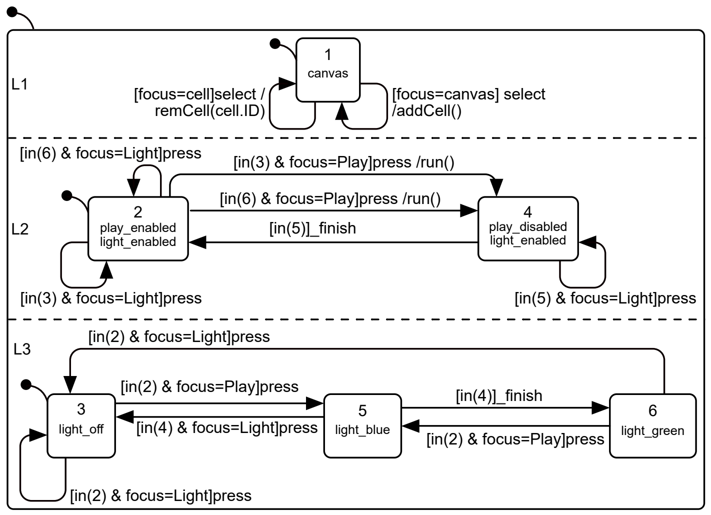

L1
1
canvas
[focus=cell]select /
remCell(cell.ID)
[focus=canvas] select
/addCell()
L2
[in(3) & focus=Play]press /run()
[in(6) & focus=Play]press /run()
[in(5)]_finish
2
play_enabled
light_enabled
4
play_disabled
light_enabled
[in(3) & focus=Light]press
[in(5) & focus=Light]press
L3
[in(2) & focus=Play]press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

Figure B.11. Statechart after rule C is refined in configuration $\{(1,ME1),(2,ME2),(6,ME3)\}$

evalQueue $= \{\langle\{(1,ME1),(4,ME2),(3,ME3)\}, \{(false,true),(false,false),(true,true),(true,false),(true,false),(false,false),(true,false),(true,false)\}, \{(false),(false),(false)\}\rangle\}$

B.4.2. Rule E refinement



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(6) & focus=Play]press /run()
2
play_enabled
light_enabled
4
play_disabled
light_enabled
[in(5)]_finish
[in(3) & focus=Light]press
[in(5) & focus=Light]press
L3
[in(2) & focus=Light]press
[in(2) & focus=Play]press
3
light_off
5
light_blue
[in(4)]_finish
6
light_green
[in(4) & focus=Light]press
[in(2) & focus=Play]press
[in(2) & focus=Light]press

Figure B.12. Statechart after rule E is refined in configuration $\{(1,ME1),(2,ME2),(6,ME3)\}$

evalQueue $= \{\langle\{(1,ME1),(4,ME2),(3,ME3)\}, \{(false,true),(false,false),(true,true),(true,false),(true,false),(false,false),(true,false),(true,false)\}, \{(false),(false),(false)\}\rangle\}$

B.5. RETE evaluation 4

evalQueue $= \{\langle\{(1,ME1),(4,ME2),(3,ME3)\}, \{(false,true),(false,false),(true,true),(true,false),(true,false),(false,false),(true,false),(true,false)\}, \{(false),(false),(false)\}\rangle\}$

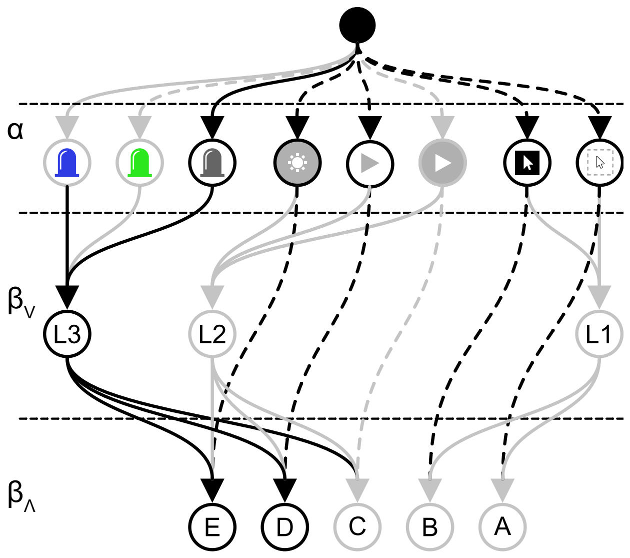


Figure B.13. RETE evaluation for configuration $\{(1,ME1),(4,ME2),(3,ME3)\}$

B.5.1. Rule D refinement

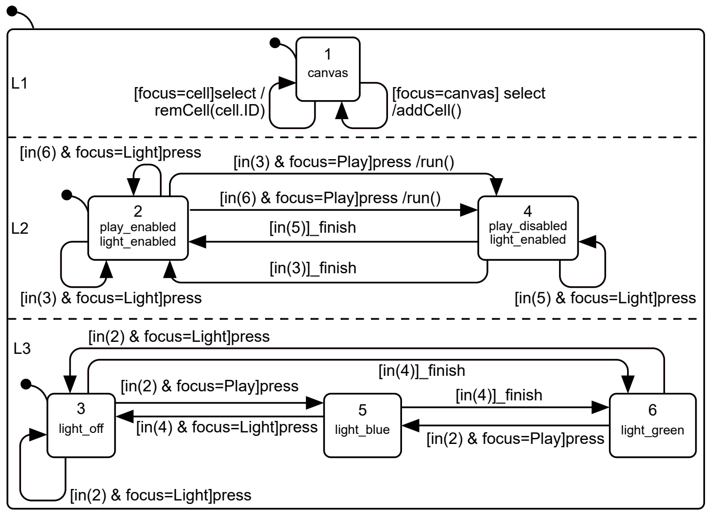

L1
1
canvas
[focus=cell]select / remCell(cell.ID)
[focus=canvas] select /addCell()
L2
[in(6) & focus=Light]press
[in(3) & focus=Play]press /run()
[in(6) & focus=Play]press /run()
2
play_enabled
light_enabled
4
play_disabled
light_enabled
[in(5)]_finish
[in(3)]_finish
[in(3) & focus=Light]press
[in(5)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

Figure B.14. Statechart after rule D is refined in configuration $\{(1,ME1),(4,ME2),(3,ME3)\}$

evalQueue $= \{\}$

B.5.2. Rule E refinement

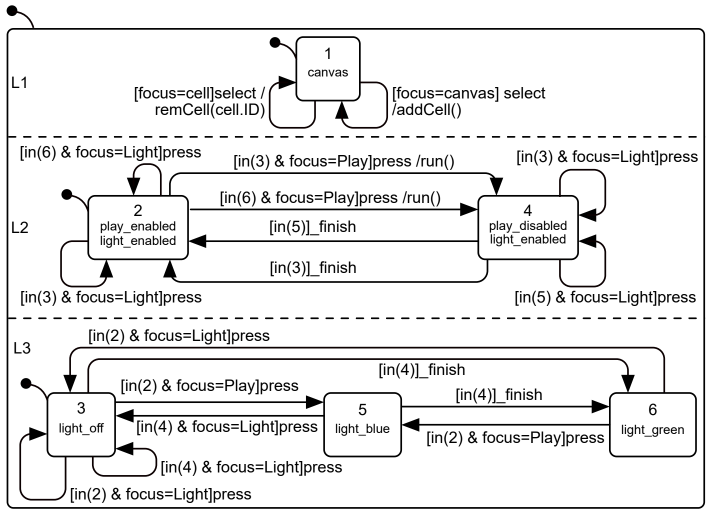

L1
1
canvas
[focus=cell]select / remCell(cell.ID)
[focus=canvas] select /addCell()
L2
[in(6) & focus=Light]press
[in(3) & focus=Play]press /run()
[in(3) & focus=Light]press
[in(6) & focus=Play]press /run()
2
play_enabled
light_enabled
4
play_disabled
light_enabled
[in(5)]_finish
[in(3)]_finish
[in(3) & 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 B.15. Statechart after rule E is refined in configuration $\{(1,ME1),(4,ME2),(3,ME3)\}$

evalQueue $= \{\}$

Appendix C

Example presented as a tutorial during the user evaluation

As an example, that you can follow along and try for yourself, let us define a simple Domain-Specific Modeling editor to model a configuration of Conway's GoL.

The Game of Life consists of a cell grid. Each cell can be either alive (represented by a black grid square) or dead (represented by a white grid square). The cells are subject to a series of rules that change their life status. If a cell is alive and has more than three neighbors, or fewer than two, it dies. If a dead cell has exactly three neighbors, it comes back to life. Semantically, it is a cellular automaton with the same expressiveness as a Turing machine. As this is a simple DSML, we want to build an equally simple editor to manipulate the Game of Life models and make it easy to use.

To define such an editor, we specify what the user can interact with (the **Interface** model), what interactions the user can perform (the **Interaction** model), and how these interactions translate to a specific platform implementation (the **Mapping** model). In this user study we are focusing our evaluation only on the **Interaction** model.

C.1. Setup

For this exercise, we are using `GoLSingleV` from our `/Examples` folder. In its `/InModels` folder we have complete **Interface** and **Mapping** models, and an empty **Interaction** model to be filled out during this exercise. These files can be opened and edited in the provided Eclipse bundle.

At any point in this exercise, the editor can be generated from the command line by running the following command in the `/Examples` folder to test how the editor behaves.

```
./RunAnt.sh --Project GoLSingleV
```

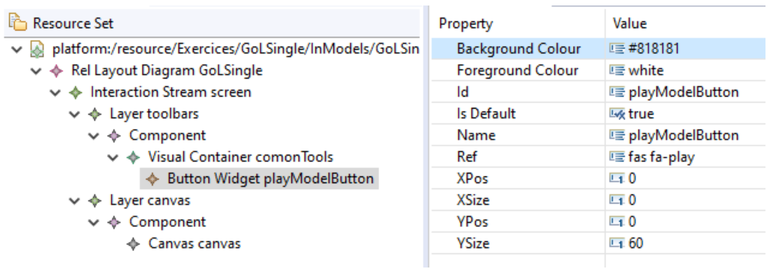
Additional ANT parameters can be passed at the end of the command line, such as `clean`, to remove all the generated files for a new generation. As a quick help the following command can be run, and it will provide the information on the command line usage and its parameters.

```
./RunAnt.sh --help
```

To run the examples, just open the `GoLSingleV.html` file in the `OutModels` folder, or from the browser bookmark provided.

C.2. Interface Model

The interface specifies what elements (visual and non-visual) the user can interact with, their representation, and their distribution in the interaction space. This model takes its cues from approaches like sketching, and organizes all its elements in a hierarchical structure, as shown in Figure C.1. The elements of the interface model are very different from DSML



```
Resource Set
  platform:/resource/Exercices/GoLSingle/InModels/GoLSin
    Rel Layout Diagram GoLSingle
      Interaction Stream screen
        Layer toolbars
          Component
            Visual Container comonTools
              Button Widget playModelButton
        Layer canvas
          Component
            Canvas canvas
```

Property	Value
Background Colour	#818181
Foreground Colour	white
Id	playModelButton
Is Default	true
Name	playModelButton
Ref	fas fa-play
XPos	0
XSize	0
YPos	0
YSize	60

Figure C.1. Interface hierarchy.

elements. So we look into more detail what these are, from higher in the hierarchy to lower.

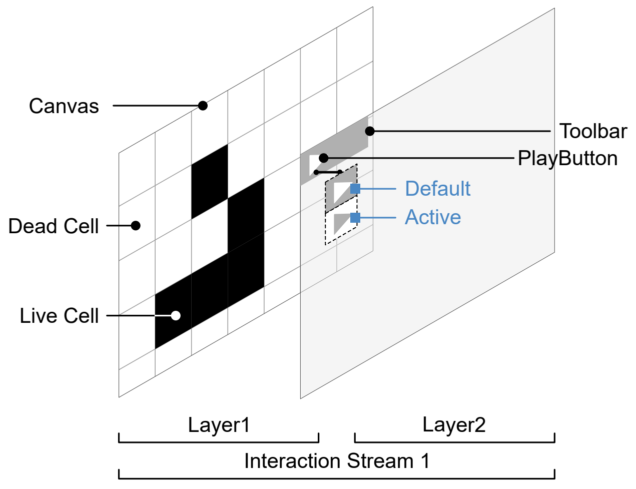


Figure C.2. Desktop interface layers.

Interaction Streams, separate the interactions with the modeling environment by their fundamental nature, (*e.g.,* screen elements, keyboard inputs, sound, motion capture) to make them modular and because their interaction elements are nor representable across different streams. In this study we will only focus on a single interaction stream, still this is part of the interface hierarchy. So when looking into GoL layout examples, a single interaction stream will appear. As the use of multiple interaction streams, is the focus of other studies.

We add the concept of **layer** to segment the interface by related elements. For visual representations, this also implies that the elements in one layer are rendered overlaying the elements of lower layers. In Figure C.2, 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.

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 C.2), we define one toolbar with a single buttons to simulate the GoL execution. Given that there is only one element type in the GoL DSML, 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.

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

For this experiment, you are not required to modify the layout models. This is for your information only.

C.3. Interaction Model

The Interaction modeling language allows for the definition of how the elements of the interface can be interacted with. Once the interface model and abstract/concrete syntax of the DSML are defined, the user can tailor the user interaction of the DSML editor to the target users.

The interaction model is composed by a set of interaction rules that represent declarative context-dependent effects on the editor: when an event occurs, and the interface or DSML satisfies a specific set of conditions, the rule produces an effect on the interface, the DSML or the system at large. A rule has a **condition** that describes the required configuration of the editor for the interaction to occur. The condition can comprise elements from the interface model (`Interface`), the DSML (`Lang`), or control elements that have no “physical” representation (`Var`). A context for the rule may be specified with the keyword `focus`. The `focus` is an abstraction on pointing devices like a mouse so that they don’t have to be explicitly added to the layout model. The events raised by such pointing devices usually reference an element of the environment, so we can just state that that element is the focus of the interaction. In the presented study only one element can successfully be set as `focus`. Additionally, a predefined `focusElement Var` will hold the target element of an event’s focus and a `focusEvent Var` will hold the full event information. Rules also accept negative application conditions with the keyword `not`. This will just reverse the evaluation of the element in the rule to being true unless it has the value provided. The `not` keyword still has some limitations. It cannot be applied to `Langs`, due to the black box nature of the canvas, as the support of this feature is dependent on the backend platform. In the same way `focus` and `not` cannot be used on the same element at the same time. The postcondition of a rule is a set of **actions** that describe what effects the rule can produce. These effects can be over any `Interface`, `Lang` or `Var` and additionally it can also trigger external executions with operation elements (`Operation`). For `Interface` these will be set operation, in particular for screen elements this set operation affects the CSS class of the element. For `Lang` this can be `add` or `remove` operations as well as move for updating the element’s position on the screen. For `Var` these will be a set operation by default.

An interaction rule must be explicitly triggered by an event, typically as a result of the user using an input device. Listing C.1 shows the syntax of an interaction rule.

Listing C.1. Interaction rule fromat

```
InteractionRule <ruleName>
    Condition {
      ( (focus|not)
        ( Canvas {},
          Interface <elementName> {
            (value = "<valueString>")?
          },
          Lang <elementType> {
            (value = ("<valueString>"|<varName>))?
          },
          Var <varName> {
            (value = ("<valueString>"|<varName>))?
          } )
      )*
    }
    ( --- <eventName> -->
    Effect {
        ( Interface <elementName> {
            (value = "<valueString>")?
          },
          Lang <elementType> {
            op = (add|rem|move) (, value = [<varName>(,<varName>)*])?
          },
          Var <varName> {
            (value = ("<valueString>" | <varName>) )?
          },
          Operation <operationName> { }
        )+
    } )*
```

For our first example of an interaction rule, we define the interaction that results in the creation of a cell. For this interaction, when the user clicks or taps on the canvas of our modeling environment, a cell is added to the canvas at the point of where the click or tap event occurred. In this simple example, we do not depend on anything else other than the canvas. So the context of our interaction is the canvas itself. Because canvas is also the element where the interaction occurs, it is also the focus of our interaction. The interaction occurs when the user clicks or taps the canvas; these are different specific events, that we can encapsulate with a higher level intent keyword, such as select; we will map these specific events with the higher-level intent with the Mapping model. These system specific events also provide the coordinates of where the interaction occurred so that the modeling environment can instantiate the cell at that point on the canvas. The effect of our interaction is the creation of a `Cell`. Listing C.2 shows how to write this interaction rule in our notation. Here, the `Lang` refers to the `Cell` class in the GoL metamodel. Lang elements can be added or removed from the canvas with the value `add` or `rem` respectively.

Listing C.2. Rule Add Cell

```
InteractionRule CSAddInstance
    Condition {
        focus Canvas {}
    }
    --- select -->
    Effect {
        Lang Cell {op = add}
    }
```

We then look at how to remove a `Cell`. For this, we want to select the cell to be removed by clicking or tapping it. So the context of our interaction is a preexisting `cell`, which will also be the focus of the interaction. The interaction will trigger when a `select` occurs. As a result, the `cell` will be removed from the model. Listing C.3 shows how to write this interaction rule in our notation.

Listing C.3. Rule Remove Cell

```
InteractionRule CSRemInstance
    Condition {
        focus Lang Cell{}
    }
    --- select -->
    Effect {
        Lang Cell {op = rem}
    }
```

With these two rules show how we can interact with a DSML. Now let us look at how to interact with the rest of the interface. For this GoL example, we want to be able to call its simulation at a push of a button. For this, having a suitable element in the interface (`playModelButton` in Figure C.1) as our focus, we can press it to run the simulation. So for this interaction our focus context is the `playModelButton`. When a press event occurs, the run simulation action is called. Listing C.4 shows how to write this interaction rule in our notation.

Listing C.4. Run Game of Life

```
InteractionRule RunTransfromation
    Condition {
        focus Interface playModelButton{}
    }
    --- press -->
    Effect {
        Operation runGoL {}
    }
```

With the `Interface`, we reference the elements from the interface model by their `id` attribute, as shown in Figure C.1. This ensures that the interface element corresponds to an existing element in the interface model. With the `Operation`, we just state the name of the operation to be called as an effect. In this case `runGoL`. This will later be mapped to a system specific function call from our code libraries (these libraries can be extended to accommodate new functions adjusted to different DSMLs). Because the system specific functions have access to the DSML and any environment variables, any data requirements of the operation are defined internally and on a system specific basis. This also means that in the interaction rules we cannot define parameters as these would be system specific.

But from an interaction point of view, this is not enough, as there are occasions where running the simulation produces no visible results in the model, so we do not know if the simulation is still running or not. To overcome this, we add `Var` to keep track of the state of the simulation. `Var` can be used as condition or effects in our rules, but they have no concrete representation in the interface. As such they can be used as control variables or to pass values between rules. Listing C.5 shows the revised rule that provides some interface feedback, by changing the state of the play button, and retains the knowledge that the simulation is running.

Listing C.5. Revised rule of Run Game of Life

```
InteractionRule RunTransfromation
    Condition {
        focus Interface playModelButton{}
    }
    --- press -->
    Effect {
        Interface playModelButton {value = "active"}
        Operation runGoL {}
        Var Simulate {value = "running"}
    }
```

Now that we have a way of being sure the transformation is running, we need to update the editor when the simulation is finished. For this, we can create a context-free rule that if there is some triggering event from the system itself stating the simulation has terminated, it updates our environment. So we established that we have no prerequisites for this rule. As a triggering event, we want an event that is raised when the GoL simulation is finished. This can be achieved by making the function mapped to the `runGoL` operation raise an event such as `_finish` right before it terminates. As an effect, we update our `Simulate Var` back to its initial state. This would result in a rule, shown in Listing C.6, where we reset the value of `Var Simulate` and bring the `playModelButton` back to its default value.

Listing C.6. Finish Game of Life

```
InteractionRule EndTransformation
    Condition {
    }
    --- _finish -->
    Effect {
        Var Simulate {value = "null"}
        Interface playModelButton{value = "default"}
    }
```

If we want, we can be more restrictive in our conditions to make sure we only react as the simulation is finished, if the simulation was running in the first place. For that we can add the value of `Var Simulate` as a condition, without it being necessarily the focus of the interaction. This results in the rule in Listing C.7.

Listing C.7. Restricted Finish Game of Life

```
InteractionRule EndTransformation
    Condition {
        Var Simulate {value = "running"}
    }
    --- _finish -->
    Effect {
        Var Simulate {value = "null"}
        Interface playModelButton{value = "default"}
    }
```

In the same way we can add a negative condition to make sure we only start the simulation if it is not already running. For that we add the extra condition on the Var that monitors the running status to the rule in Listing C.5, resulting in the rule in Listing C.8.

Listing C.8. Restricted rule of Run Game of Life

```
InteractionRule RunTransfromation
    Condition {
        focus Interface playModelButton{}
        not Var Simulate {value = "running"}
    }
    --- press -->
    Effect {
        Interface playModelButton {value = "active"}
        Operation runGoL {}
        Var Simulate {value = "running"}
    }
```

To finish demonstrating the features of the interaction language, let us pretend that an undo functionality shouldn't require proper transactions implemented in the modeling back end. For this we want a rule that when we hit the undo key (we will map this to the Esc key) we remove the just added `Cell`. But if we create a rule like the one in Listing C.9 we do not know what `Cell` to remove (this would not be the case if the modeling backend provided the undo functionality).

Listing C.9. Undo adding a cell

```
InteractionRule UndoAdd
    Condition {
    }
    --- undo -->
    Effect {
        Lang Cell {op = rem}
    }
```

This means that when we add a `Cell` we need to keep track of it for a potential undo. Because adding a model element places that newly created element as the new focus, we can then access the `focusElement var` to keep that value in a control variable. But if we do it as in Listing C.10, we might get inconsistent results as there is no guarantee in the order of execution within an `Effect` block, as all actions on an `Effect` are simultaneous.

Listing C.10. Extended rule Add Cell

```
InteractionRule CSAddInstance
    Condition {
        focus Canvas {}
    }
    --- select -->
    Effect {
        Lang Cell {op = add}
        Var addedCell {value = focusElement}
        Var LastAction {value = "adding"}
    }
```

Instead, we can create multiple `Effect` blocks, to establish precedence and order. So if instead we write our rule as in Listing C.11, we guarantee that the `Cell` is created and placed in `focusElement` before we retrieve the value from `focusElement`.

Listing C.11. Extended rule Add Cell

```
InteractionRule CSAddInstance
    Condition {
        focus Canvas {}
    }
    --- select -->
    Effect {
        Lang Cell {op = add}
    }
    ------>
    Effect {
        Var addedCell {value = focusElement}
        Var LastAction {value = "adding"}
    }
```

Now that we have saved the `Cell` to be undone in the `Var addedCell` we can use it in our undo rule. We also added the `Var LastAction` to controll that indeed the value in `addedCell` came from the add rule and it can be safely undone. To retrieve the `Cell` to be undone we just need to query a `Lang Cell` that is equal to content of our `Var addedCell`. This condition will only return true if such a `Cell` is found, and as a side effect if also places that `Cell` as the `focusElement` so operations can be performed on it in the effect block. If we also set the values of `addedCell` and `LastAction` with the effects of the undo action, our final rule will look like Listing C.12.

Listing C.12. Undo adding a cell

```
InteractionRule UndoAdd
   Condition {
      Var LastAction {value = "adding"}
      Lang Cell {value = addedCell}
   }
   --- undo -->
   Effect {
      Lang Cell {op = rem}
      Var LastAction {value = "undo"}
      Var addedCell {value = "null"}
   }
```

C.4. Event Mapping Model

As we saw in the Interaction Model, the interaction rules define a high-level event that triggers the interaction. This event represents the intent and has no direct representation in an actual system. What we do then is a mapping between this conceptual event to the events available on a specific platform.

This allows us to reuse interaction descriptions in different platforms and to associate the events described in the Interaction Rules to multiple events on a specific system, providing us various forms of performing the same conceptual interaction. An example of this would be to map the `select` event in our rules to both a mouse click and a touch screen system events.

This mapping is made with a simple text file that defines all mapping pairs used, where each line defines a pairing. This pairing is formatted, as shown in Listing C.13.

Listing C.13. Mapping Model language format

```
<MappingType> # <Event|Operation> # <Expression>
```

In this format, `MappingType` can either be `I` or `O` for mapping Input and Output events respectively. `Event` is the event name used in the Interaction rules. Alternatively, `Operation` is used for Output mappings, it is then the operations used in the Interaction rules, such as `runGoL` or in the case of operations on `Langs`, a composition of the element's class name with the operation perfomred, such as `Liferem` for the operation `rem` on an `Lang Life`. The `Expression` is the platform-specific code that translates the event. In this example we are using the JavaScript expression `document.getElementById(<Element Name>).addEventListener("<System Event Name>", function(){<ProjectName>.Interaction.prototype.triggerEvent( event,<Element Name>,'<Rule Event Name>');}, false);` to map input events.

For our rules, this results in the mapping in Listing C.14.

Listing C.14. Event Mapping Model of GoL

```
I # press # document.getElementById("playModelButton").
    addEventListener("click", function() {GoLSingle.Interaction.
    prototype.triggerEvent(event,"playModelButton",'press');}, false);
I # press # document.getElementById("lightButton").addEventListener("
    click", function() {GoLSingleV.Interaction.prototype.triggerEvent(
    event,"lightButton",'press');}, false);
I # select # document.getElementById("23").addEventListener("click",
    function() {GoLSingleV.Interaction.prototype.triggerEvent(event,
    event.target.id,'select');}, false);
I # undo # document.addEventListener("keydown", function() {if(event.
    keyCode == 27 && !event.repeat){GoLSingleV.Interaction.prototype.
    triggerEvent(event,"",'undo');}, false)}})

O # Lifeadd # addLife.call(this,"23");
O # Liferem # remLife.call(this,this.event.target);

O # runGoL # simulateGoL.call(this);
```

For this experiment, like the layout models, you are not required to modify the mapping models. This is for your information only.

C.5. Additional Examples

Two additional versions of Game of Life are available; these are provided to illustrate some additional interaction patterns beyond the initial example and to show that there are many possibilities of how to implement the interaction of a modeling editor.

The first additional example is **GoLToolbar**. It shows how an editor would be implemented in the traditional modeling editor way, with a language toolbar that allows the selection of the different language elements. In this example, an additional toolbar with two elements appears on the screen: one for the cell, that when selected the interactions on the canvas will instantiate cells, the other an empty cell, that when selected the interactions on the canvas remove the cell at that point of the canvas.

The second additional example is **GoLDrag**. It shows a common feature of GoL editors, that once you initiated an add or remove cell operation, the user can keep adding or removing by dragging the mouse over the canvas, thus reducing the number of clicks required on more complex cell configurations.

To compile these examples, just use the following command line.

```
./RunAnt.bat --Project <example_name>
```

To run the examples, just open the HTML file in the project's respective `OutModels` folder, or for the bookmark available in the provided browser.

Appendix D

Exercise performed during the user evaluation

For this exercise, we ask you to define the interactions of a simplified Mind Map modeling editor. You can specify any interactions that you think are of interest to the modeling process.

The focus of this exercise is the interaction specification. Thus, a series of artifacts are already provided to you, so you only need to edit the Interaction Model.

Because you can do as many interaction rules as you want, there is no need to try to figure out all interaction rules in one go, as that can become overwhelming. Instead, we suggest you tackle the interaction rules one use case at a time.

D.1. Mind Map

For this exercise, we are configuring an editor for a Mind Map DSL. Mind maps are a methodology to help layout ideas and thoughts around a topic and establish relations between the different subtopics. An example of this can be seen in Figure D.1, where a year is a central topic, and the seasons and the months are subtopics of the year, and organized into their inner relation between seasons and months.

D.1.1. Mind Map Language

For this exercise, we will focus on a language with only two elements: a Topic and a SubTopic. These elements follow the following rules. Only one Topic should exist in the model. Links can exist between a Topic and a SubTopic or between two SubTopics. All elements have a class name, an ID, and a Name.

We show the metamodel for this language in Figure D.2.

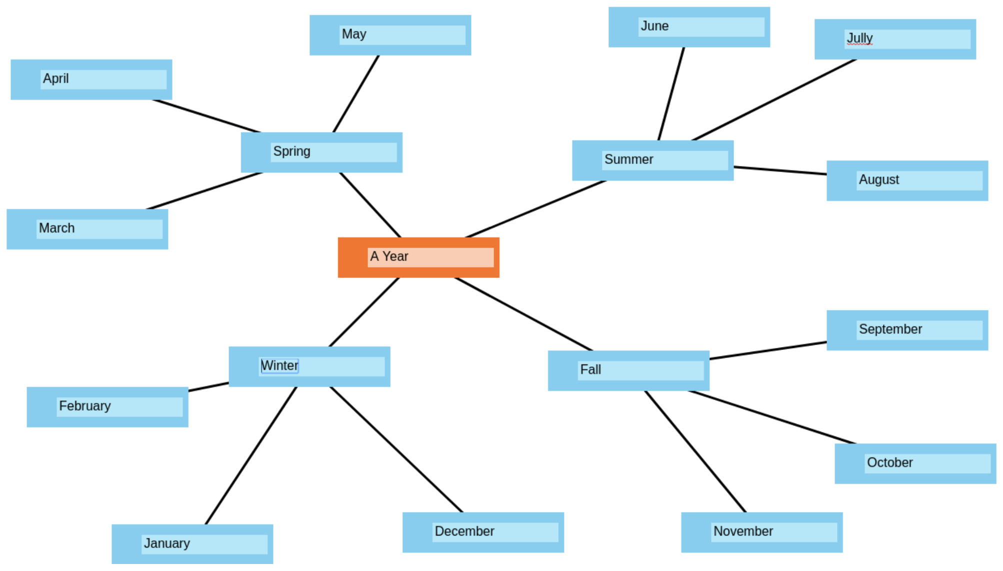


Figure D.1. Example of a Mind Map.

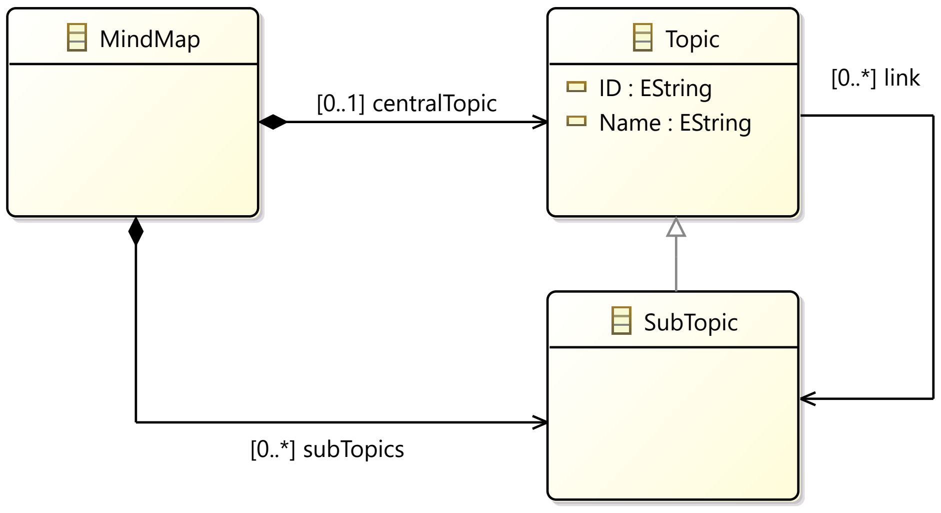


Figure D.2. Mind Map metamodel.

D.2. Artifacts

To help focus this exercise on designing interactions, we provide a set of support artifacts, so that any remaining models required to produce an editor are already setup while providing enough flexibility in building the interactions.

D.2.1. Layout Models

Four layouts are provided. The user can choose any of the models by opening the `HTML` corresponding to the desired layout in the `OutModels` folder.

The provided models are:

D.2.1.1. Minimal Layout

This model only provides a canvas and will rely solely on user interactions to perform the different tasks required by the language. To use this layout open the file named `MindMapBlank.html`.

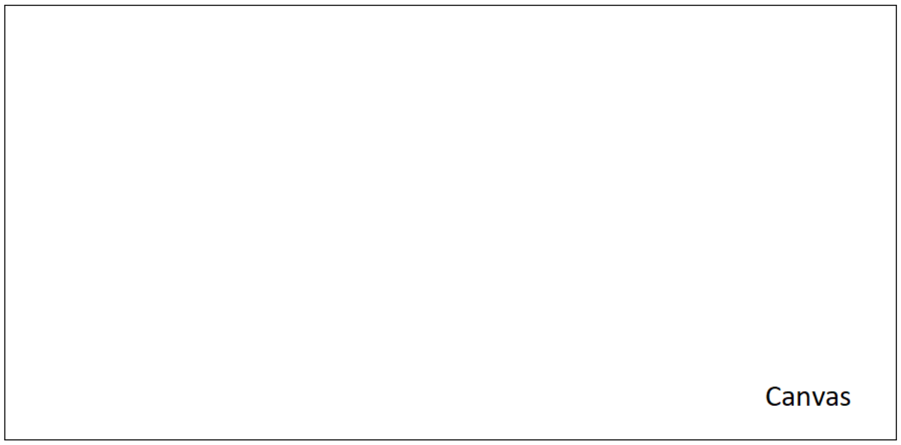


Figure D.3. Minimal Layout.

D.2.1.2. Simple Toolbar Layout

This model provides a small toolbar with two buttons in the top left corner of the screen, mimicking more traditional editors. The main difference still present in the provided layout it that the icons provided are completely abstract and it is up to you do decide and define their meaning. To use this layout open the file named `MindMapToolbar.html`.

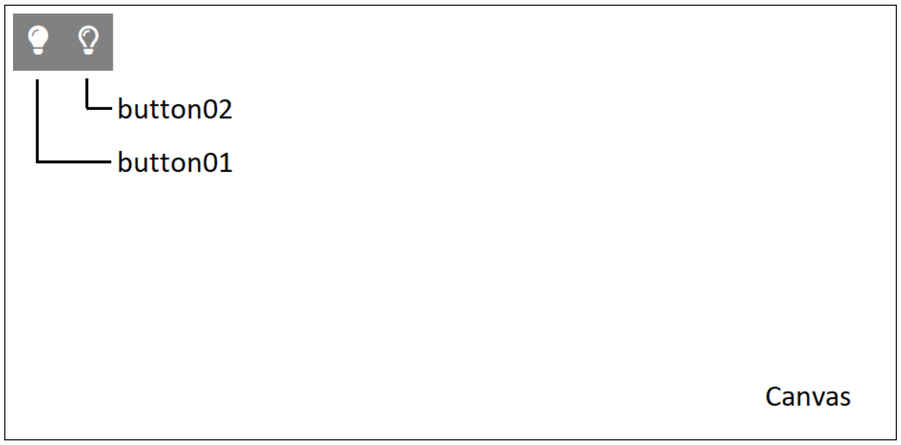


Figure D.4. Simple Toolbar Layout.

D.2.1.3. Large Toolbar Layout

This model provides a larger toolbar with eight elements at the top of the screen. These will provide more options if necessary. Like before the icons in the provided layout are completely abstract and it is up to you do decide and define their meaning. To use this layout open the file named `MindMapToolbarLarge.html`.

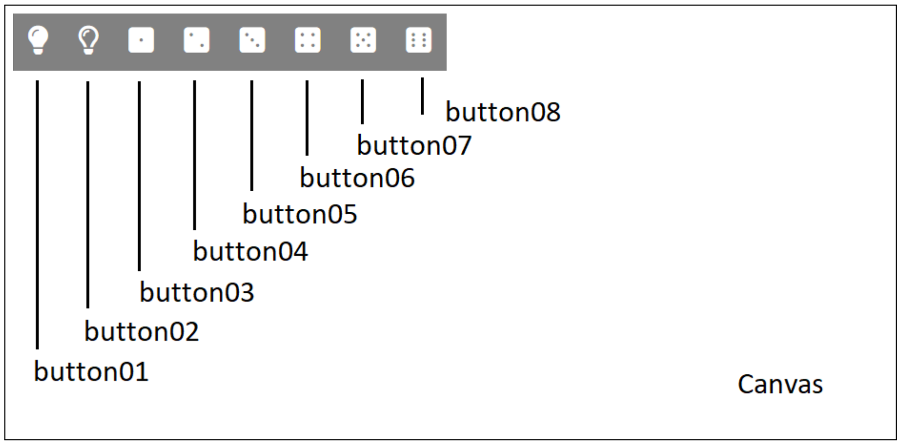


Figure D.5. Large Toolbar Layout.

D.2.1.4. Corner Buttons Layout

This model provides four buttons, one in each corner of the screen. Like before the icons in the provided layout are completely abstract and it is up to you do decide and define their meaning. To use this layout open the file named `MindMapCorner.html`.

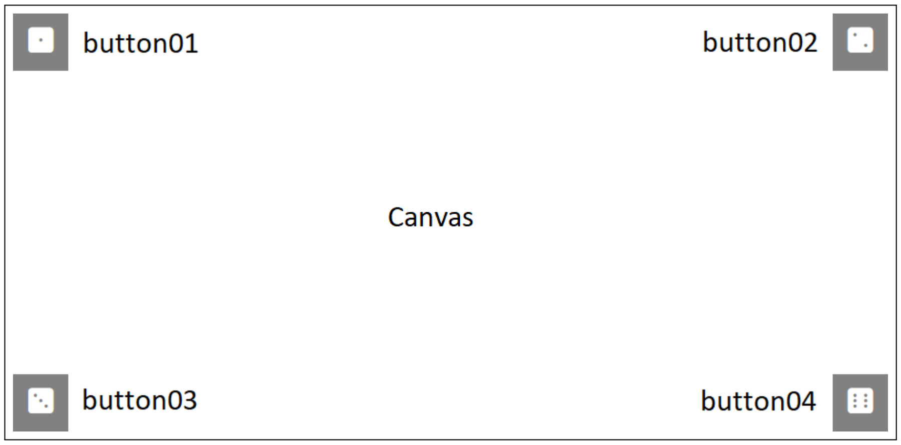


Figure D.6. Corner Buttons Layout.

Any model can be used, and a user is not obliged to make use of all elements of a layout. They are there just in case, but their use is entirely optional.

D.2.2. Mapping Model

A mapping between user actions and system events is provided, mapping pre-made action names to different system events such as mouse clicks, motion, and keyboard shortcuts. Listing D.1 shows the provided events and operations, and their respective mappings, present in the file `MindMap.mappingtxt`.

Listing D.1. Event Mapping Model for Mind Map

```
I # leftmousedown # left mouse button pressed
I # leftmousemove # mouse movement with left mouse button pressed
I # leftmouseup # left mouse button released
I # rightmousedown # right mouse button pressed
I # rightmousemove # mouse movement with right mouse button pressed
I # rightmouseup # right mouse button released
I # backspacekey # backspace key pressed
I # deletekey # delete key pressed
I # tabkey # tab key pressed
I # enterkey # enter key pressed
I # shiftkey # shift key pressed
I # ctrlkey # control key pressed
I # altkey # alt key pressed
I # escapekey # escape key pressed
I # spacekey # space bar pressed
I # keyrelease # any key is released

O # add # adds a Topic, SubTopic or Link to the model, where adding a
    Link, will take two parameters in the form 'value=[varName,varName
    ]' to define the start and end points of a Link.
O # rem # removes a Topic, SubTopic or Link to the model
O # move # moves a Topic, SubTopic to the current event coordinates
```

The user can use the pre-made action names or rename them to the appropriate names used in the interaction rules. Like the layout model, the user is not obliged to make use of all of the mapping; they are there just to provide flexibility and are expected to be more than the necessary mappings.

Appendix E

Questionnaire of the user evaluation

General information

(1) What is your age interval ?

(2) To which gender do you identify yourself ?

(3) To which ethnic identity do you identify?
[South Asian] [Chinese] [Black] [Filipino] [Latin American] [Arab]
[Southeast Asian] [West Asian] [Korean] [Japanese] [Caucasian]
[Aboriginal] [Polynesian] [First Nations]
[Minority not included elsewhere]

(4) What is your academic level
[None] [Technical] [Bachelor] [MSc] [PhD]

(5) Which of the following describes you best for the purpose of this study ?
[Student] [Professor] [Industry] [Researcher] [Other]

(6) How would you rate your expertise in Computer Science
[None] [Novice] [Intermediate] [Knowledgeable] [Expert]

(7) How many years of experience do you have in Computer Science

(8) How would you rate your expertise in Model-Driven Development
[None] [Novice] [Intermediate] [Knowledgeable] [Expert]

(9) How many years of experience do you have in Model-Driven Development

(10) How would you rate your expertise in UX and UI development
[None] [Novice] [Intermediate] [Knowledgeable] [Expert]

(11) How many years of experience do you have in UX or UI development

State how much you agree with the following sentences, or complete them to you agreement.

(12) Modeling environments should have their interactions explicitly expressed.
[Strongly agree] [Agree] [Indifferent] [Disagree] [Strongly disagree]

(13) The person best suited to model the Layout Model is a _________.
[UX/UI Designer] [Stakeholder] [Final User] [Domain Expert] [Software Engineer]

(14) The person best suited to model the Interaction Model is a _________.
[UX/UI Designer] [Stakeholder] [Final User] [Domain Expert] [Software Engineer]

(15) The person best suited to model the Mapping Model is a _________.
[UX/UI Designer] [Stakeholder] [Final User] [Domain Expert] [Software Engineer]

(16) The presented approach allowed me to express the interactions I wished.
[Strongly agree] [Agree] [Indifferent] [Disagree] [Strongly disagree]

(17) It was easy for me to apprehend the concepts required by this approach.
[Strongly agree] [Agree] [Indifferent] [Disagree] [Strongly disagree]

(18) I would recommend this approach to a colleague who needs to design a modeling editor with custom interactions.
[Strongly agree] [Agree] [Indifferent] [Disagree] [Strongly disagree]

(19) When using this approach, I spent most of my time fixing errors or trying to make the interactions work as I intended.
[Strongly agree] [Agree] [Indifferent] [Disagree] [Strongly disagree]

(20) When using this approach, I tried new ways to express interactions or explored new interactions.
[Strongly agree] [Agree] [Indifferent] [Disagree] [Strongly disagree]

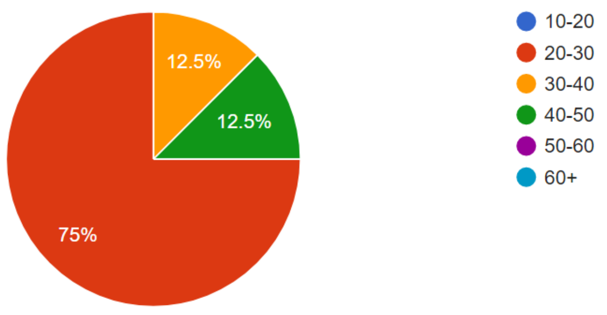
Additional comments.

Appendix F

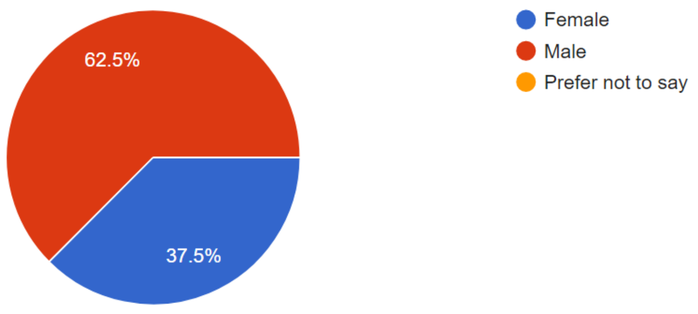
Results to the questionnaire of the user evaluation

General information

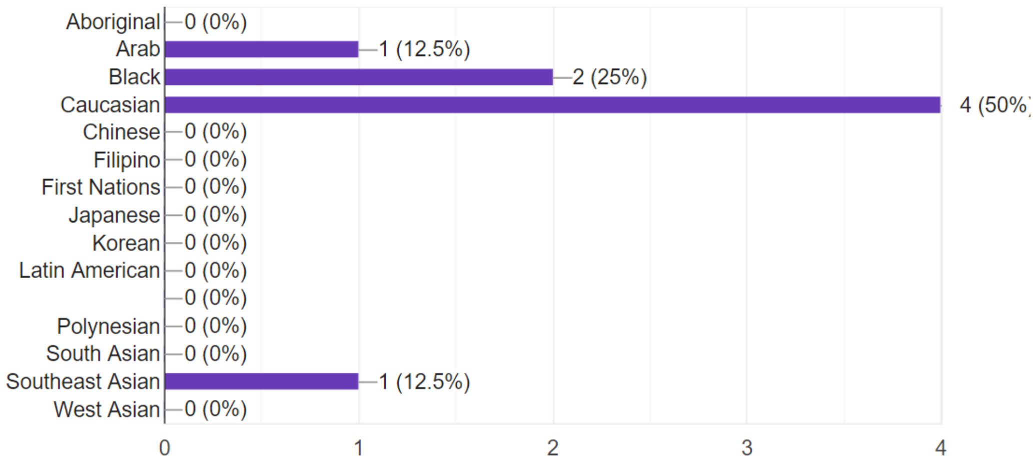
(1) What is your age interval ?



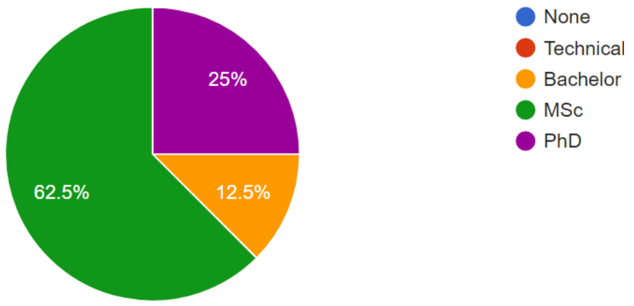
(2) To which gender do you identify yourself ?



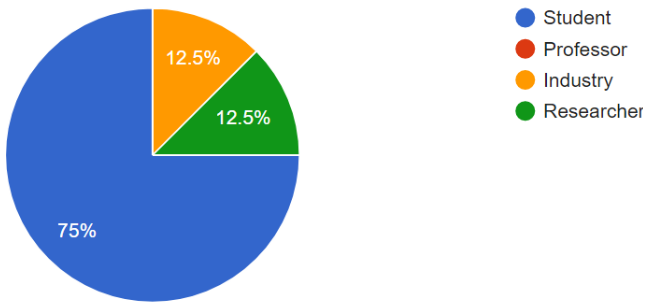
(3) To which ethnic identity do you identify?



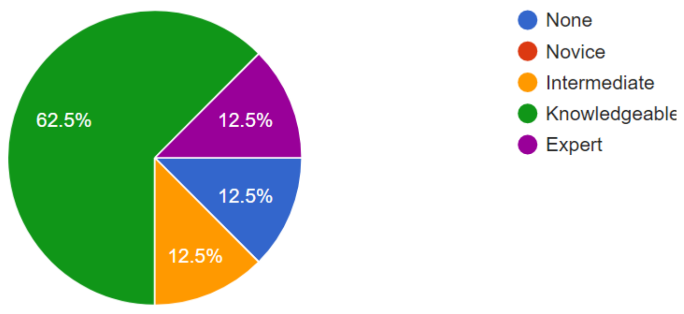
(4) What is your academic level



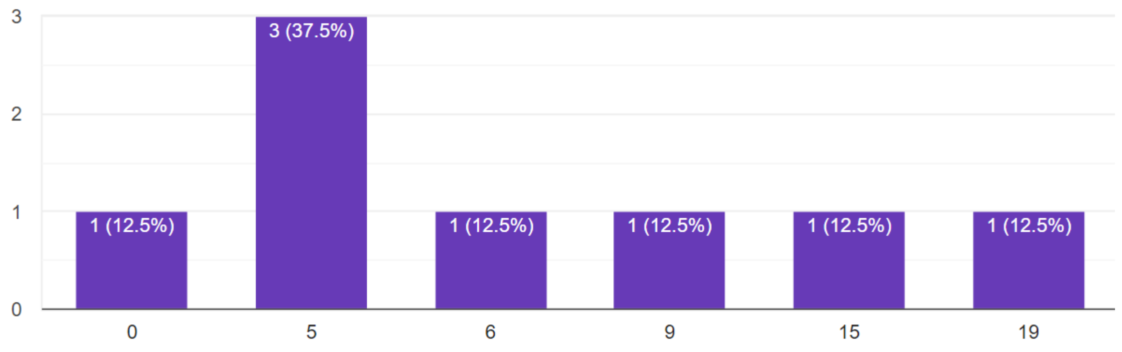
(5) Which of the following describes you best for the purpose of this study ?



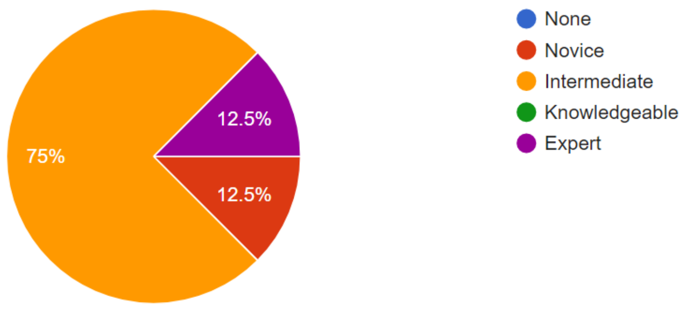
(6) How would you rate your expertise in Computer Science



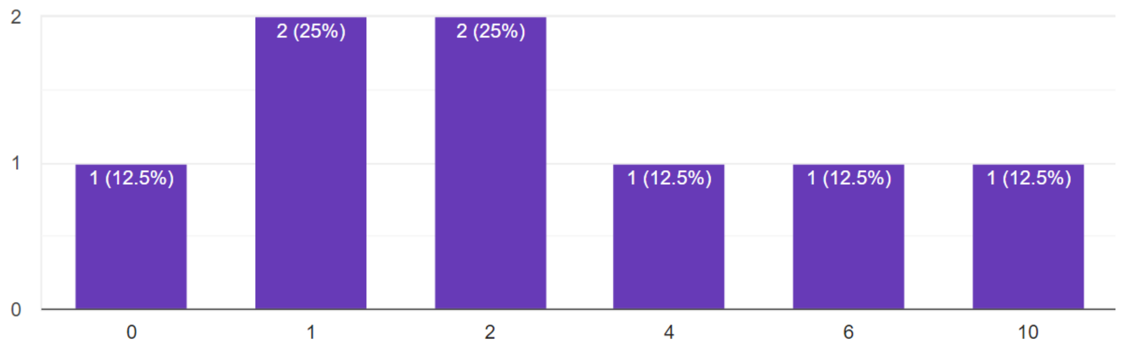
(7) How many years of experience do you have in Computer Science



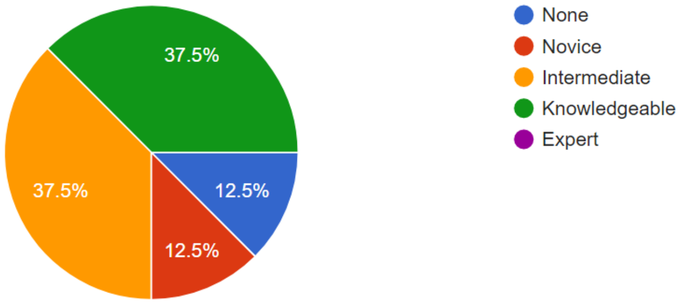
(8) How would you rate your expertise in Model-Driven Development



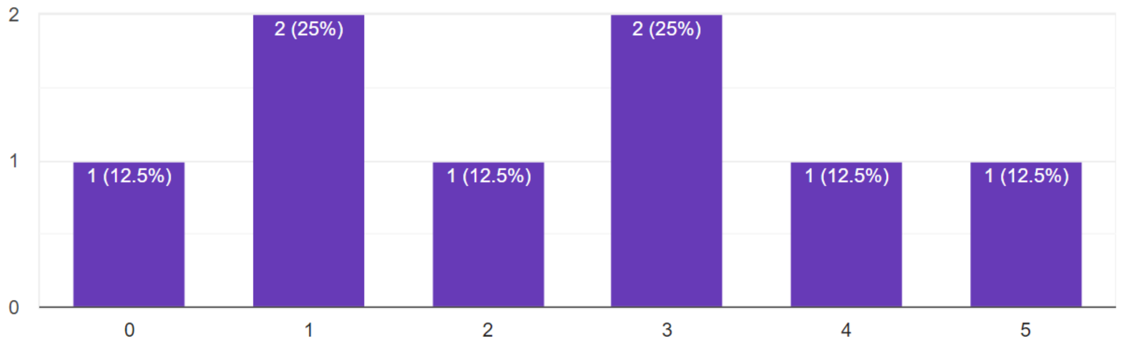
(9) How many years of experience do you have in Model-Driven Development



(10) How would you rate your expertise in UX and UI development

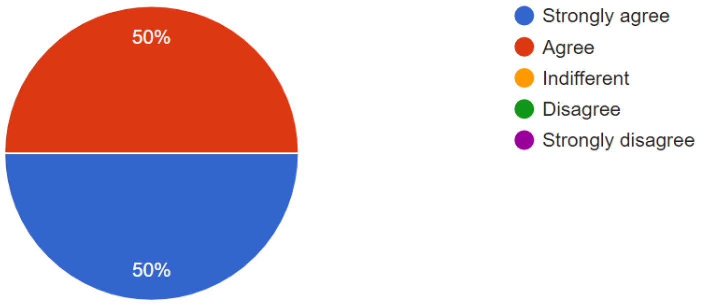


(11) How many years of experience do you have in UX or UI development

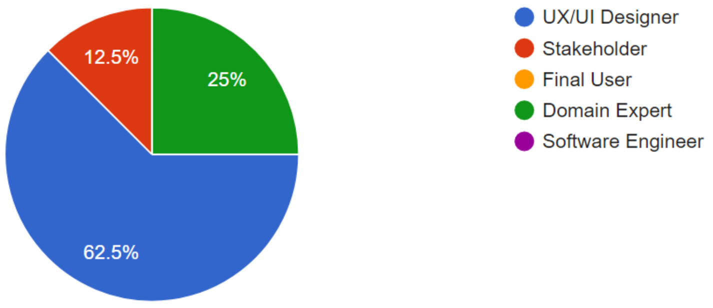


State how much you agree with the following sentences, or complete them to you agreement.

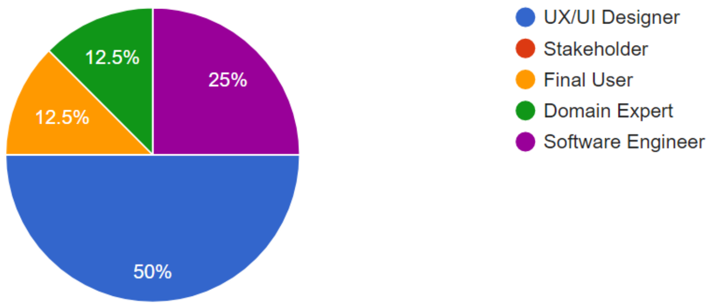
(12) Modeling environments should have their interactions explicitly expressed.



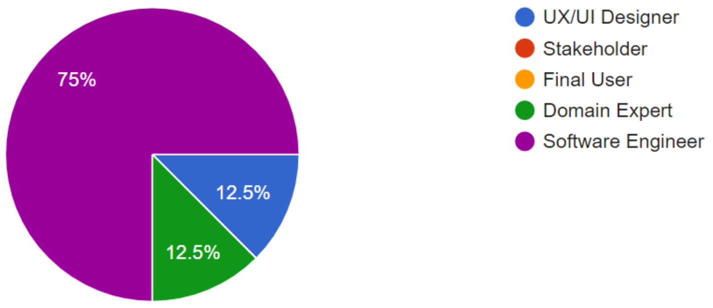
(13) The person best suited to model the Layout Model is a __________.



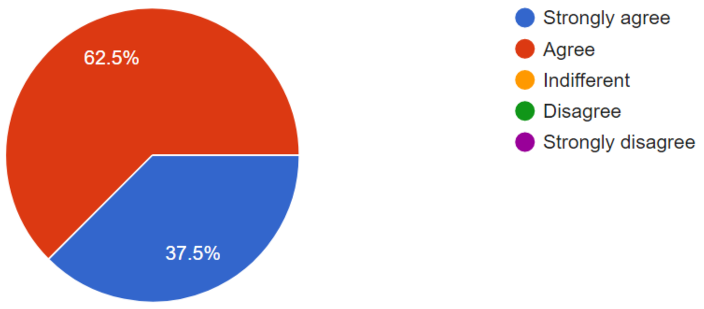
(14) The person best suited to model the Interaction Model is a __________.



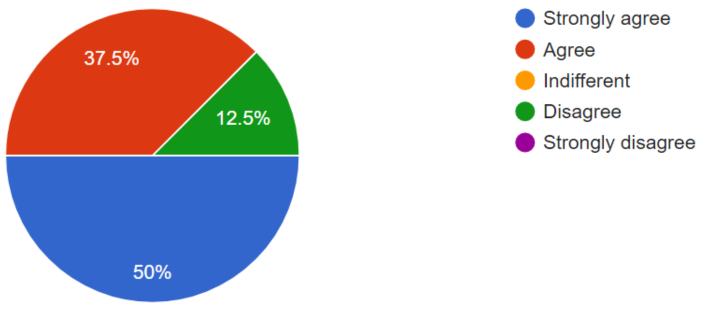
(15) The person best suited to model the Mapping Model is a __________.



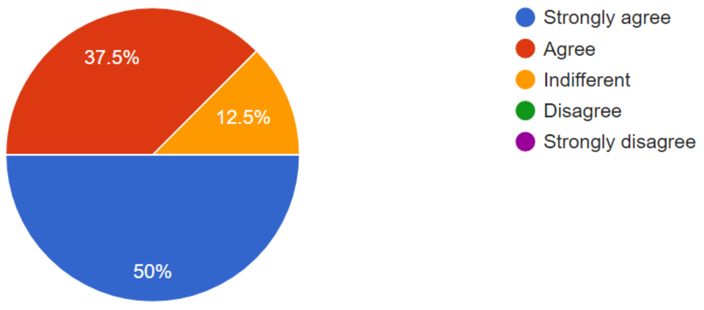
(16) The presented approach allowed me to express the interactions I wished.



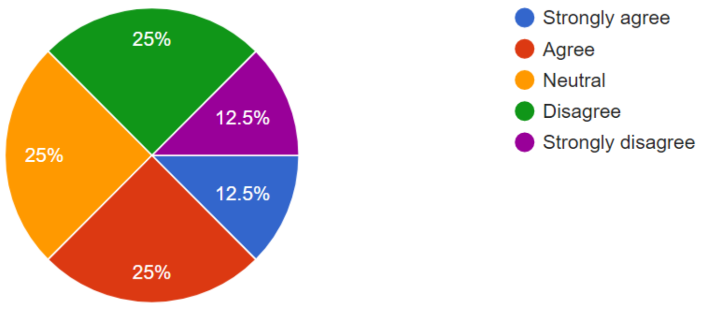
(17) It was easy for me to apprehend the concepts required by this approach.



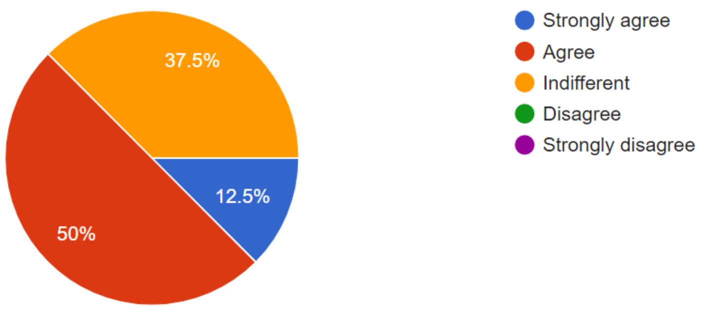
(18) I would recommend this approach to a colleague who needs to design a modeling editor with custom interactions.



(19) When using this approach, I spent most of my time fixing errors or trying to make the interactions work as I intended.



(20) When using this approach, I tried new ways to express interactions or explored new interactions.



Additional comments.

I had trouble grasping the way to use the MIDEAS because of the language. I am a very visual person and coding is not one of my strong suits unfortunately. Once I did get things done, it felt very empowering otherwise. I am curious of the direction this tool will take!

I was a bit confused when I wanted to add a "control click" event, mainly because I am used to java events where you can get informations on a mouseclick event like isCtrlDown etc...

Good work i liked the idea.