UNIVERSITY OF OSLO
Department of Informatics

Modeling Editing Behavior for Editors of Graphical Languages

Master thesis
60 credits

Rayner Ron Vintervoll

February 1st, 2010

ABSTRACT

This thesis presents a framework that expands upon the idea of a fully model-driven approach to editor development for Graphical Domain Specific Languages (DSL), originally put forth by the Graphical Modeling Framework (GMF). The framework's main component consists of a language for the declarative definition of editing behavior for said editors. We define the *Behavioral Definition Language* (BDL), and the execution semantics of a BDL-instance, *Behavioral Definitions* (BD).

Inconsistent DSL-instances are not desired when modeling them using modern editors. However, during user-interaction with the editor, edits may be attempted that would, if permitted, create inconsistent models. Instead of denying such edits we propose a different approach: to commit the edit to a separate model capable of representing the *result* of an inconsistency-creating edit. Upon this model we use *editing behaviors* to resolve the inconsistencies before committing any alterations to the DSL-instances. To simplify the complexity of reasoning about what editing behaviors may be applied, we present a method for presenting editing behaviors to a user for selection. Letting editing behaviors focus on resolving small fragments of inconsistency, while letting the user select the appropriate set of behaviors to ultimately create a DSL-consistent model.

The method presented for defining editing behaviors is based on graph transformation; we use graph transformation rules and patterns therein, to pattern-match rules against models capable of representing inconsistent DSL-instances ("models of inconsistency"). This to determine *when* and for *what* inconsistencies we may present editing behaviors to the user for selection. Using comprehensive examples, we argue for the validity of our approach to the definition and applicability of editing behaviors defined in such a manner.

.

ACKNOWLEDGEMENTS

Firstly, I would like to express my deepest gratitude to my supervisor, Dr. Øystein Haugen at SINTEF. For all the help, encouragement and guidance I have received from you over the years. I am forever in your debt.

To my girlfriend, Cathrine Skandsen, thank you so much for being as kind and understanding as you have been to me throughout all these years. Without your love and support I would have never made it this far.

I also would like to thank my mother and father. Thank you from the bottom of my heart, for always being there and supporting me in all my endeavors.

Rayner Ron Vintervoll,

1st of February, 2010

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

1.1 MOTIVATION

Before the introduction of the Graphical Modeling Framework (GMF) [1] many developers had undertaken the task of binding the Graphical Editing Framework (GEF) [2]to Eclipse Modeling Framework (EMF) [3] models, to create editors for models of graphical languages, like UML [4]. Among these editors one has a particular relevance to the motivation behind this thesis; the Papyrus UML Editor[5]. The author of this thesis has partaken in the process of developing the Sequence Diagram editor component of the Papyrus UML editor, which was based on an editor developed as part of a master's thesis at the University of Oslo [6]. During the development we were presented with several challenges regarding the definition of the editor's *behavior* when interacted with by users. We found the process of programmatically defining and incorporating automatic inconsistency resolutions to inconsistency creating edits, to be a daunting task. Quickly resulting in never-ending cascading behaviors, inconsistencies within the models the editor created, and in the code. Leading at times to crashes or to the editor behaving non-deterministically.

Some of these problems we believe to be related to the following issues; (1) the complexity of defining editing behavior consistently using a purely programmatic approach. (2) the lack of a formal method for defining concrete graphical syntax and enforcing the constraints it defines. (3) the special nature of UML Sequence Diagram's concrete graphical syntax; it does not match well with typical node-arrow-only type languages, resulting in added complexities since common layout-algorithms for those kinds of languages, like XYLayout and ToolbarLayout as presented in [7] by IBM for defining the layout of elements in a graphical syntax, quickly become to primitive to guarantee well-formed sequence diagrams.

GMF with its model-driven-development approach rectifies some of these issues by applying an abstract (implementation-distant) and formal modeling approach to the development of editors for graphical languages. Especially GMFs notation meta-model,

and subsequently IBM's GMF inspired Diagram Definition proposal [7] attempts to rectify issue (2).

However, in our view there still lacks a component able to rectify (1) and (3); a form of editing behavior definition that uses a DSL's own concepts to express the editing behavior needed to produce consistent and well-formed diagrams and models. This while supporting on an implementational level not merely the constriction and denial of edits resulting in inconsistencies, but rather support giving users inconsistency solutions to choose from during editing. Thereby leveraging the user's own knowledge about the diagram and model to restore consistency, instead of programmatically trying to enforce it at all times.

1.2 GOALS OF THE THESIS

The goal of this thesis is to present our findings regarding the relationship between graphical domain specific languages and editing behavior for editors of said languages. Our findings stem for work done on a prototype editor behavior subsystem and from work on this thesis. We will show how we may, when modeling DSLs, not only model their abstract and concrete graphical syntax, but also model their *editing behaviors* on an abstract level. We will define our Behavioral Definition Language (BDL), and show a Behavioral Definition (BD) for an extensively examined example of editing behaviors in response to an inconsistency creating edit. We will give the execution semantics of a generic Behavioral Definition and show how an executing Behavioral Definition may be integrated into the workflow of editor development. We will lay the formal foundation, upon which BDL and its supporting framework, depends upon. Since several of the components needed to create an executing Behavioral Definition are conceptual, we will explore in detail the features that we require of these conceptual components.

1.3 THESIS STRUCTURE

Chapter 2 Background: Domain Specific Languages This chapter focuses on what a Domain Specific Language (DSL) is. We will give a short presentation of the difference between abstract and concrete syntax, and then explain the concept of *graphical* languages and concrete *graphical* syntax. Will give some examples of languages capable of expressing

this syntax in different ways. We will also define a conceptual graphical definition language (GDL) based on the presented languages and that we will rely upon for the rest of the thesis.

Chapter 3 Background: Editors For Graphical Languages This chapter focuses on defining what an editor for a graphical languages is, and how they differ from common textual editors for programming languages. We will show the advantages and disadvantages of having completely syntax strict editors, in both the textual and graphical domain. We will also show how we may model editors for graphical languages using a model-driven development approach, and show the relationships between the models involved. Lastly we will examine how we *customize* modeled editors in a model-driven-development process.

Chapter 4 Editing Behavior This chapter focuses on defining what editing behavior is. We will give examples of *helpers* in a current state-of-the-art editor, laying the foundation for our motivation of giving editor users a *choice* of editing behavior instead of automatically implementing them. We will also see how we may define editing behaviors on an abstract level, and how we define editing behaviors on an more implementation specific level. We draw parallels to the realm of *inconsistency management* and show how we may represent *inconsistent* DSL-instances *consistently* using a non-constrained variant of the DSL in question.

Chapter 5 Behavioral Framework This chapter focuses on explaining our framework. We will present the components, our meta-modeling architecture, and present our language BDL along with an small example of it in use. We will explain a generic Behavioral Definition's execution semantics, and give the motivations behind the elements in BDL and behind the elements in the execution semantics.

Chapter 6 Example: Problem 1 Deals with a concrete example of a problematic edit on a sequence diagram that results in inconsistencies in the *diagram model*. We will show how we may find a set of editing behaviors capable of solving the inconsistency by examining and constricting the solution-space. We show how we naturally end up with concepts similar to concepts from the field of graph transformation when reasoning about editing behavior. We then proceed with formally defining editing behaviors, capable of solving the problem, as graph transformation rules, using the methods presented in Chapter 4. We will also show what we require of the special model presented in Chapter 4 to be able to reason

about the applicability of editing behaviors. We then give these rules in a Behavioral Definition.

Chapter 7 Behavioral System Prototype In this chapter we will talk about the prototype that was developed, what was examined during its development, and what we found. We will also talk about problems with the tools used to create the prototype, and with the prototype itself. We will also give concrete suggestions for future prototypes within this field of study.

Chapter 8 Conclusion and Further work In this chapter we will conclude with our findings, and give examples of further work within the field of editing behavior for editors of graphical languages, based on the findings presented in this thesis.

1.4 GOALS OF THE FRAMEWORK

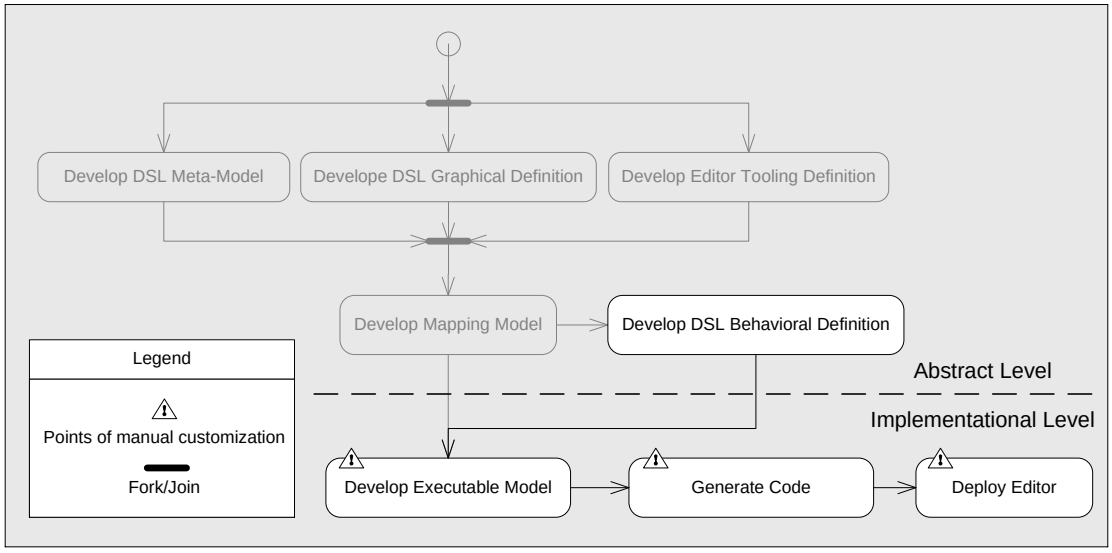

Figure 1-1 Reducing the amount of manual customization by modeling editors even more precisely with Behavioral Definitions

The goals of the framework is to reduce the amount of customization (in the figure: ⚠) not done on an abstract level when defining editing behavior in editors for graphical languages, and to make such customizations more consistent. We also want to be able to provide a different approach to when and how an editor executes editing behavior, in that

we do not automatically alter or deny edits performed by the user when they result in inconsistent models, but give the user a choice of which editing behavior to implement to resolve the inconsistency. We do this by capturing what we call a *Behavioral Definition* in its own model, which focuses on the *definition of* relevant behaviors, and by defining an underlying execution semantics for Behavioral Definitions that focuses on *finding* relevant behaviors. We show in Figure 1-1 the insertion of a Behavioral Definition in a GMF workflow (which will be explained in 3.2), but argue that such a definition is in fact more closely related to the process of developing a DSL than to the process of editor development, as we use will only use concepts from the DSL and BDL to create the Behavioral Definition, remote from any editor implementation specific concepts.

2 BACKGROUND: DOMAIN SPECIFIC LANGUAGES

> *"Language serves not only to express thought, but to make possible thoughts which may not exist without it" – Bertrand Russell*

2.1 DSL

Domain Specific Language, or DSL, is a term used by many and but perhaps not consistently. As put in [8] by Gronback et. al. "much has been written on the general topic of DSLs, with the domain-specific aspect being the most controversial and reminiscent of discussions regarding "meta-ness"".

The concept of using specialized languages to express concepts and relationships in a domain is not new, with COBOL (**CO**mmon **B**usiness-Oriented **L**angauge) being one of the more famous as a language actually acknowledged of "domain-specific" [9]. But the problem with using the term *doman-specific* as a qualifier for the proceeding term *language* is that the term is relative. One may argue, as Gronback in [8] that for some UML [4] is a language a consisting of several other languages describing domains such as; state machines, use cases, interactions and so on. Others may consider UML as a language that describes the domain of software development, not viewing it as a language for describing abstract concepts like state machines, but a language for describing software.

We also find those who differentiate between Domain-Specific-Languages and Domain-Specific-*Modeling* Languages[10], or DSML for short. A DSML may be thought of as a language described using meta-models. A meta-model is, to put it short, a model *of* a model. From Haugen in [11] we find the following citations from [12] :

"... meta-power, that is the power to change the rules of the game, the matrix of actions and interaction possibilities and their outcomes ... "

" ... meta-power as a relational control, i.e. control over social relationships and structures ..."

The citations are from a sociological text, but are clearly possible to translate into "software" terms. In fact we may think of meta-models as models that *define* legal constructs of other models. In "Matters of (meta-) modeling" [13], Kühne gives a quite concise example of what *meta* is:

"... a discussion of how to conduct a discussion is a "*meta-discussion*"".

Problems arise however if we start "discussing how conduct the discussion *of the* discussion" (a meta-meta-discussion?).

To summarize, we have a term, Domain-Specific-Language, that uses a somewhat relative term; *domain-specific*. We also have the concept of creating them using meta-models, also a somewhat relative term as the degree of *meta* can be fluctuating. E.g. is a language defined only by its meta-model, or is it also defined by the meta-model's *meta-model*? We will not delve deeper into the implications of meta-ness, and/or domain-specificity in this thesis, but to move forward we need a definition of what a DSL is in this thesis. Ruscio et. al. defines it as the following:

"DSLs are languages able to raise the level of abstraction beyond coding by specifying programs using domain concepts. In particular, by means of DSLs, the development of systems can be realized by considering only abstractions and knowledge from the domain of interest."

All computer languages consists of a concrete and abstract syntax, DSLs included [14]. Abstract syntaxes are the *backbone* of the language as they define how the language views information from an *internal* view-point. This in contrast to the *concrete syntax* which defines how a *user* of the language views the information. For instance a language for graph manipulation would contain the concepts of *nodes* and *edges*. From an internal view-point these concepts are merely objects, things of an abstract nature, or more precisely just *data*. From an external view-point however they are not just *data*, they are actual nodes and edges; boxes, lines, circles etc. This defines concrete syntax.

For textual programming languages these distinctions are easy to make; the abstract syntax defines how a compiler would view the language, the concrete syntax how *programmers* would view the language [15]. For instance; **x := 5** is an expression following an imagined

concrete syntax, the abstract syntax representation of it may however be: **assignStmt(ConstExp(x), ConstExp(5))**.

The step from going from the above abstract syntax representation of the expression, to a model representation is not hard; assignStmt as a element with 2 children-elements with two (typed) attributes x and 5. For the concrete syntax expression a model representation is also possible: An concrete syntax assignment-element has an attribute defining a string ":=" and two relationships, one *left* and one *right*. *left* points to the character *x* and *right* points to the character *5*.

2.2 GRAPHICAL LANGUAGES

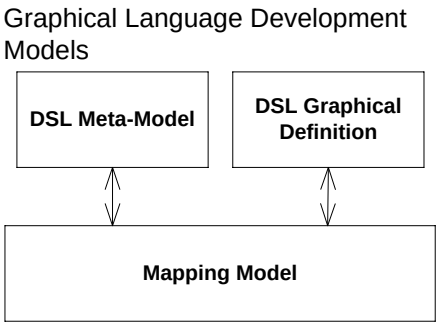

Figure 2-1 Develop and map abstract and concrete syntax

Graphical languages have concrete *graphical* syntaxes, either in addition to or in-place-of a regular concrete *textual* syntax. Among the challenges in graphical language development we have those of how to formally represent and define the graphical syntax. We also find challenges relating to how to create a mapping model capable of *binding* the abstract and concrete graphical syntax together in a coherent definition of a DSL. Consequently the challenge of creating a such a mapping model also becomes a challenge of defining a mapping *meta*-model [7]. We will in the following sub-chapters focus on the challenge of defining concrete graphical syntax, and assume for the remainder of the thesis that the mapping model is implicit between abstract and concrete syntax; that the challenge of mapping has been resolved. In Figure 2-1 we show the abstract and concrete graphical syntax as a DSL meta-model and DSL graphical-definition respectively. These are mapped

to each other via a mapping model to define the relationships between elements in the models.

2.2.1 CONCRETE GRAPHICAL SYNTAX

In this chapter we will present some of the attempts to create a language capable of defining concrete graphical syntax, and also present an interesting parallel to the field of Geographical Information Systems.

2.2.1.1 Espe's Graphical Description Language (E-GDL)

Type	Attribute	Description
Geometrical	*lx*	left-most x-position
	rx	right-most x-position
	by	lowest y-position
	ty	top-most y-position
Direction	*tail*	a line's start point
	head	a line's end point
Visualization	*appearance*	how the symbol looks

Table 1 GDL symbol attributes

GDL (Graphical Description Language, we refer to it E-GDL from now on, to separate it from another GDL later in the thesis) [16], was tested on a subset of the *Unified Modeling Language* [4]. This language specifies valid concrete syntactic constructions of a visual language using schemata. It relies on *predicates* and concepts from topology (*intersects, in, touch* etc.) to specify the spatial relationship between the various graphical elements of the language. Viewing graphical **symbols** (atomic entities in the syntax) as point-sets and exposing symbol attributes, allows E-GDL to specify predicates not so easily described with topology in a simple way.

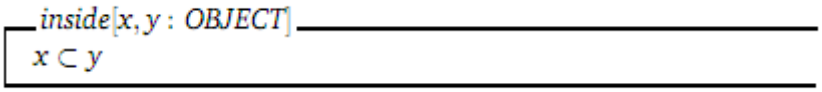

Figure 2-2 Examples of E-GDL-predicate: " x is inside y", courtesy of [16]

Figure 2-1 shows the inside relationship in the E-GDL-notation. It defines the following predicate; OBJECT x is *inside* y if x is a subset of y where x and y are point-sets on a 2-

dimensional plane. E-GDL lacks the capability of defining elements on a 3- or 2,5-dimensional plane, so as to define what is commonly called the *z-order* between elements.

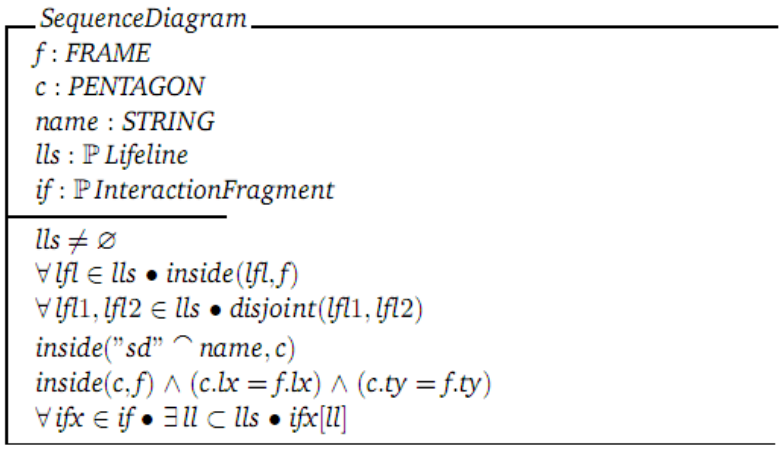

Figure 2-3 Example of using E-GDL to define a Sequence Diagram Symbol

Figure 2-3 shows a E-GDL-instance of a SequenceDiagram symbol. Defining constraints that use predicates, symbols and sets of symbols, to define valid SequenceDiagrams.

2.2.1.2 Diagram Interchange (DI), Diagram Definition (DD) and Diagram Graphics (DG) by IBM

IBM's Diagram Interchange (DI) (subsuming OMG DI), Diagram Definition and Diagram Graphics [7] are part of a response to an OMG Request For Proposal (RFP) on Diagram Definition [17]. The proposal is inspired by the mapping that GMF provides between EMF Ecore [18] models, notational elements and their tooling.

They split the concept of concrete graphical syntax into three parts: a language for persisting diagrams and interchanging them (DI), a language for the concrete syntax of DI with respect to a DSL (DD) and a language for expressing graphical syntax and mapping it to the abstract syntax (DG).

DI is a language used to persist and interchange diagrams between applications. DD defines valid usage of the elements that DI consists of for a given target domain; a definition that defines constraints upon instances of DI, for a given domain.

Essentially we may think of DD as defining the valid usage of elements in DI, when DI is to be used as the diagram persistence/interchange language of a DSL. Implying that DI has differing concrete syntaxes depending on the DSL in question. Therefore requiring an separate Diagram Definition model to define valid usages of DI for a given source DSL. They calls this *diagram syntax*, different from concrete graphical syntax. Diagram syntax in this manner is actually a form of abstract syntax, which when combined with the abstract syntax from the DSL, is capable of representing valid DSL-specific diagrams in a model.

The third language regards the concrete *graphical* syntax, DG. This language deals with how to define how graphical elements should be *rendered* on screen in an abstract manner; not defining *painting* logic itself, but the attributes and relationships needed to paint a *building block*. E.g. attributes like *position, size, color, line-style* . DG also is capable of defining how one may access attribute values in the underlying models (DSL meta-model instances, and DI-instances) to populate attributes in a DG-model. DG therefore consists of two parts; one part for the declarative description of *graphical syntax* and another part for the declarative descriptions of *mappings* from the model (abstract syntax *and* diagram syntax) to graphics.

The complexity of the proposal from IBM, and the large amount of meta-models involved prohibit us describe them in detail here. However, we are able to simplify our explanations by viewing the entire set of languages as a language for the definition of concrete graphical syntax. This since all of the meta-models may define constraints and values that affect how the concrete graphical syntax is *presented to the user* for interaction. We feel this is an important point to make; that a DSLs concrete graphical syntax is in many ways the net result of all the constraints defined upon the models used in its definition. The proposal segments the definition into multiple languages, but adhere to the basic notion of using *constraints* defined on instances of the languages to define valid syntax. DD-instances place constraints on loosely constrained DI-instances. While DG-instances have constraints defined on elements within it, elements that reference DI-instance elements. The sum of these constraints are, in our view, all constraints *on* the concrete graphical syntax, as it is the sum of the constraints that define the syntax ultimately need to interact with and adhere to.

Some of these constraints can be user-settable. DI defines a concept called StyleSheets that may contain appearance properties like colors, and layout constraints. DD defines valid styles for a given DSL so as to constrict the realm of possible stylesheets a user may

choose from. DG does not have user-settable constraints (as far as we can see) but does define a concept of Layout that DG-elements may enforce upon child elements according to some layout constraints.

2.2.1.2.1 GIS Extended OCL

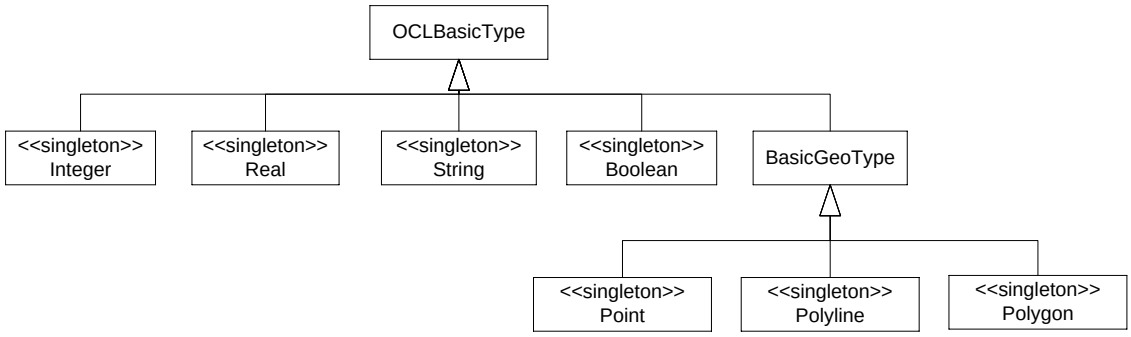

Figure 2-4 Extending OCL with a new type: BasicGeoType

Pinet et. al. [19] propose an extension to OCL [20] to allow for spatial constraint definition in response to the complexities of modeling *Geographical Information Systems* (GIS). Systems that require heavy use logical reasoning with spatial information [21]. UML [4] and MOF [22] only allow for topological constraint representation via relationships (e.g. Building ↔ disjoint ↔ River, where disjoint is a relationship). This approach is however lacking in its expressiveness, as complexities arise when trying to express more complex constraints (IF .. THEN .. the constraint is applied ELSE …).

Especially when examining spatial constraints. This is why they propose an extended OCL adjust for spatial reasoning. They do this by first; defining a new OCLBasicType called BasicGeoType (alongside Integer, Real, Boolean and String) which is the super type for 3 fundamental geographic types; Point, Polyline and Polygon. They further define that any element to be evaluated spatially in an OCL constraint to have a *geometry* attribute. This *geometry* attribute is a collection of elements were each element in the collection has a BasicGeoType. The *geometry* attribute is viewed by OCL to be equivalent to a OCL-collection, allowing for the use of OCL collection operations like forAll, select and size. The spatial operations they defined are either equivalent or similar to E-GDLs predefined predicates, so we will not reiterate them here. The example below shows how we may define an invariant on a *Diagram*:

```
context Diagram inv:
    self.geometry -> forAll(p1, p2 | p1<>p2 implies p1 -> disjoint(p2))
```

This constraint states that all elements `p1` and `p2` in the collection must be spatial disjoint as long as they are not equal.

Although this extension of OCL was not intended to *define a concrete graphical syntax* we find the resemblance between it and E-GDL striking. This extension shows that it is possible to use OCL to express spatial constraints, making OCL translations of E-GDL definitions possible.

2.2.1.3 Our conceptual language: Graphical Definition Language (GDL)

The concept of having a language purely for the formal definition of concrete graphical syntax is not new [23]. Unfortunately no OMG-supported standard has emerged yet, although as we saw in the previous chapter, an OMG RFP (Request for Proposal) and actual proposals for such a standard are under consideration.

For our purposes in this thesis we must create a conceptual language that allows us to reason about the effect of the concrete graphical syntax combined with the abstract syntax on how an editor *behaves*. We will however *not* attempt to create a GDL meta-model and represent it here, as this is not the purpose of the thesis. Rather we will find instances of an conceptual language that fits the problems we will examine in this thesis, and that hopefully are general enough to match a wide range of future meta-models for graphical definition.

We will simplify the ideas from E-GDL and IBMs Diagram Definition proposal, and use those ideas together with GIS extended OCL constraints, to allow us to create simple constrained models representing concrete graphical syntax for a DSL. Ignoring aspects such as persistence and interchangeability, and focusing on creating elements that are as closely related to the DSL abstract syntax as possible with respect to naming.

In our definitions using OCL we will employ E-GDL's predefined set of predicates using syntax derived from GIS extended OCL. We will also use the concept of E-GDL's Symbols to gain access to attributes, such as (x,y) coordinates; we assume that all Symbols have a collection akin to the *geometry* attribute in the previous chapter (and like Symbols and

point-sets in E-GDL) but that this attribute does not need to be explicitly stated or referenced, and is automatically inferred during constraint evaluation.

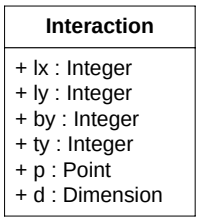

Figure 2-5 All Symbols have position and size attributes

In our conceptual GDL a Symbol is also not only a spatial entity consisting of constraints and other Symbols, but may also include model references to its domain meta-model instance. This allows us to just refer to a single Symbol when talking about both an elements graphical properties and model properties, shortening our statements.

Our GDL needs a form of diagrammatic concrete graphical syntax so that we may easily depict the situations we will be examining. There exists however no such diagrammatic graphical notation for either E-GDL or DG, yet. We will therefore imagine one: Given the E-GDL Symbol for SequenceDiagrams in Figure 2-3 we may draw a simplified class-diagram excluding the PENTAGON, NAME, InteractionFragment, and focusing only on the relationships defined by the predicates inside and disjoint:

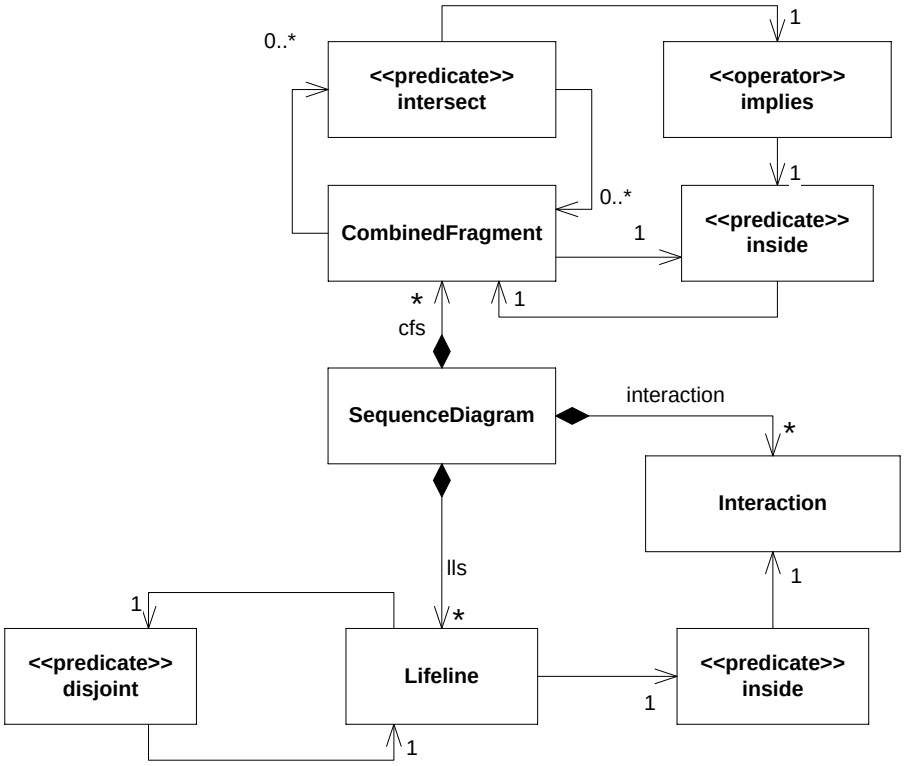

Figure 2-6 MOF-type representation of a subset of a GDL instance (GDSQ) for SequenceDiagram Symbol in Figure 2-3

```
context SequenceDiagram
    inv:
        self.cfs -> forAll(cf1, cf2 : CombinedFragment | cf1 <> cf2 and cf1 ->
intersects(cf2) implies cf1 -> inside(cf2) or cf2 -> inside(cf1))
        self.lls -> forAll(lfl1, lfl2 : Lifeline | lf1 <> lf2 and lf1 -> disjoint(lf2));
        self.lls -> forAll(lfl : Lifeline | lfl -> inside(self.interaction))
```

The diagram of the model GDSQ presents a translation of GIS extended OCL expressions into relationships and stereotyped nodes. This so that we may give a class-diagram representation that shows the model-representation of the constraints defined on the elements, in our case constraints defined on the SequenceDiagram. Although not a formal model representation of OCL-expressions in any way, the above model *does* represent the relationships we require between elements. We do this by deducing what the OCL expressions, defined on the SequenceDiagram, mean *structurally* with respect to the relationships between nodes and the multiplicities on the relationships. Importantly, what

the model *does not* represent are the parts of the OCL expressions dealing with non-equality of objects (<>). CombinedFragment, Lifeline, Interaction, SequenceDiagram are all Symbols.

The GDL Definition (GDSQ) in Figure 2-6 can be viewed as a subset of a model which in total represents the graphical definition of a SequenceDiagram, as it describes nothing about the elements in the syntax, only their relationships to each other. The nodes disjoint and inside (stereotyped to <<predicate>> for readability*)* can be viewed by the reader as a model representation of the OCL predicates used in the SequenceDiagram context and that we have *extracted* from the SequenceDiagram element and visualized. The associations between the predicates define the direction of the predicate evaluation (e.g. Lifeline -> inside(Interaction)) and the multiplicities define that they must always exist in a valid instance of this model.

Based on GDSQ we may define the relationships between graphical elements in instances of this meta-model. Figure 2-7 depicts a model A that conforms to its GDL Definition meta-model (GDSQ) and that defines the spatial predicates currently true between elements in the diagram. We use bi-directional relationships to denote the existence of 2 uni-directional relationships of the same type that relate the same elements for conciseness.

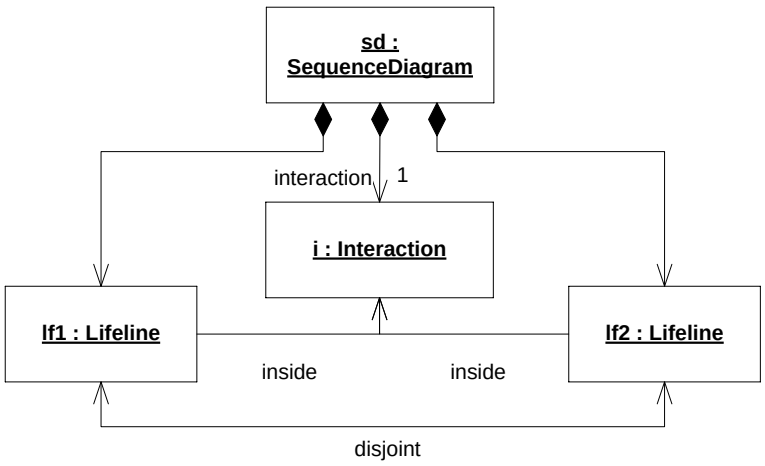

Figure 2-7 Model A conforming to GDSQ

Figure 2-8 is a graphical representation that is a true representation of the model A in Figure 2-7.

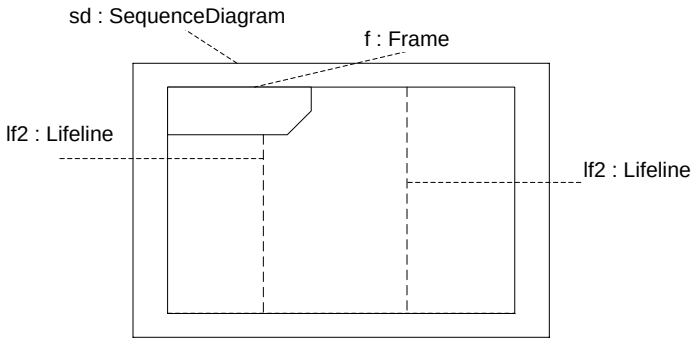

Figure 2-8 Graphical representation of Model A

The GDSQ places constraints on which relationships must be present at all times in a valid GDSQ instance. Also by negating an expression with a predicate like disjoint (not lf1 : Lifeline -> disjoint(lf2 : Lifeline2)) we get what relationships must *never* be present at all times in a valid GDSQ instance like A. We define that relationships *missing* from the model means that its constraint has evaluated to false in the model; e.g. if the statement

```
lfl : Lifeline | lfl -> inside(self.interaction)
```

evaluates to false we remove the relationship between the lfl and the Interaction. Importantly this renders the model *inconsistent* with respect to its meta-model GDSQ (a violation of the constraints defined on the meta-model), of which the model A' in Figure 2-9 is an example of.

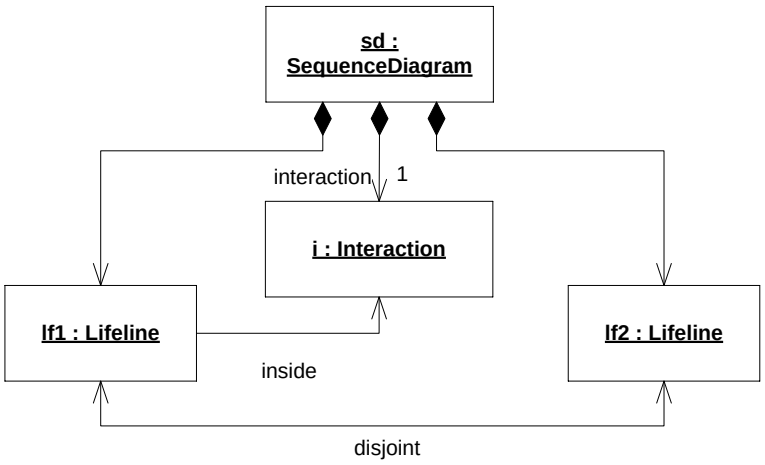

Figure 2-9 A inconsistent model A' not conforming to GSDQ

Another example is the model R in Figure 2-10 conforming to GDSQ but where we only show a subset of the model, focusing on the Lifelines and CombinedFragments.

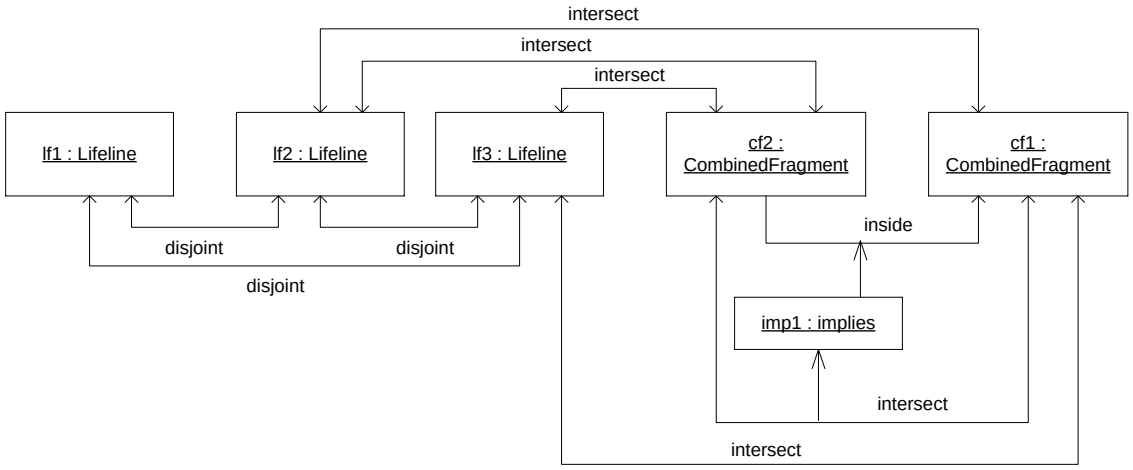

Figure 2-10 Model R conforming to GDSQ

Figure 2-10 shows an instance of a model where the implies *operator* binds the two relationships intersect and inside together using a association class-like notation. We have chosen this notation for elements stereotyped as <<operator>> since association classes only exist while the association exists (the directed arrows show which association was responsible for creating it; the source-association). This coincides nicely with what the invariant for CombinedFragments states: that if 2 CombinedFragments intersect one must be inside the other. We further define that for inside relationships between CombinedFragments it is the responsibility of the association class to manage the information about the existence of the relationship. The multiplicities on the relationships in GSDQ between intersects and implies [1..1], implies and inside [1..1] show that the relationships are *strict*. One may not exist without the other. Since we use bi-directional relationships to denote the existence of two "equal" relationships between elements (different in direction only) we say that the iff viewing them as uni-directional they would *both* have a relationship to the same implies instance. Figure 2-11 is a graphical representation of model R.

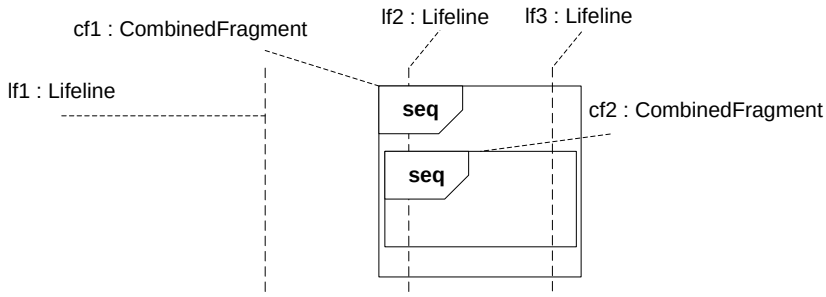

Figure 2-11 Graphical representation of Model R

Figure 2-12 and Figure 2-13 show an *inconsistent* model G with respect to its meta-model GDSQ. Looking at the graphical representation we see that we have intersecting CombinedFragments where none is *inside* the other. This creates a *dangling* implies association class (similar to the well-known dangling-else problem in programming language development and compiler theory [15], a problem regarding ambiguity in concrete syntax) that has no reference to its required inside relationship, which is missing from the model.

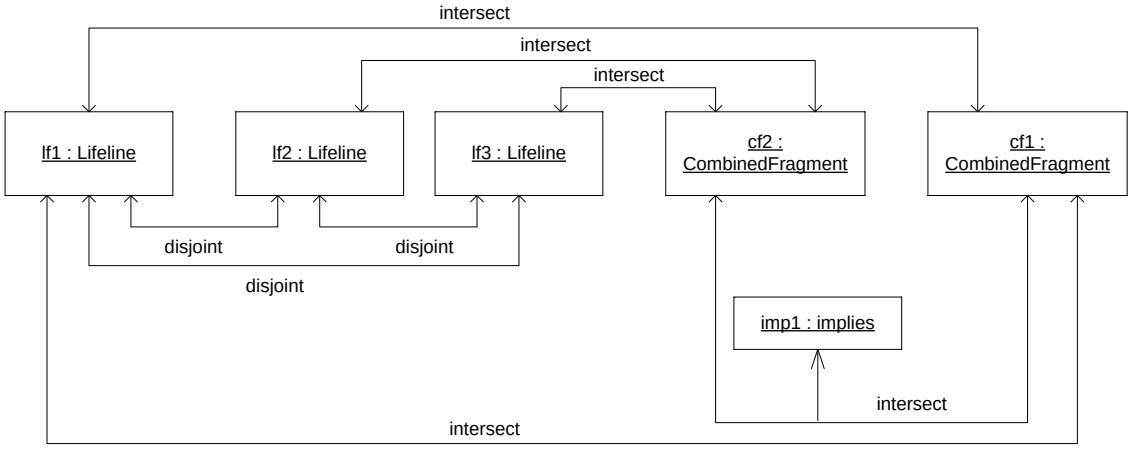

Figure 2-12 Inconsistent model G: dangling implies with missing inside-relationship relationship

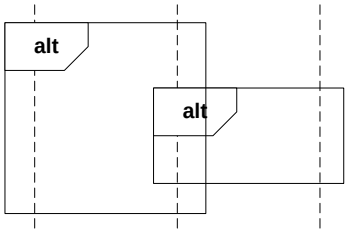

Figure 2-13 Graphical Representation of model G

3 BACKGROUND: EDITORS FOR GRAPHICAL LANGUAGES

" If thought corrupts language, language can also corrupt thought" - George Orwell

3.1 SYNTAX-DIRECTED EDITORS

Programs are often written and manipulated with text editors. A text editor, as the name implies, manipulates textual entities that are organized into a basic hierarchy that consists of characters and lines. The editor provides simple functions such as insertion and deletion of characters and lines. However, a program is not merely text - they are only *represented* textually. A program is a collection of syntactically and semantically meaningful objects such as identifiers, procedures, loops and data types. We therefore often build editing tools that employ knowledge about the programming language constructs, and allow users of the editing tools to create and manipulate programs in the terms of these language constructs [24].

We call these structured, language-sensitive or syntax-directed editors. Structured editors employ operations from the programming languages compiler to give users information about errors in the program while it is written rather than compiled. This is often solved via running a portion of a typical compiler process while the user is interacting with the editor, like Eclipse's Abstract Syntax Tree (AST)[13] . While editing a user may create a program that is *inconsistent,* in the sense that it would not compile if we tried to compile it. The editor would notice this and proceed with notifying the user of the error. One of the first such editors was the Cornell Program Synthesizer [25].

Similarly we have editors for *graphical languages*. Most of which are *completely syntax directed* or *completely structured,* such as [5, 26, 27]. By this we mean that *inconsistent* states, which are common in textual editors and result in the editor notifying the user of the error, are not permissible *at all*. Such editors place syntactic correctness as an *absolute* requirement at all times.

We can construct an example of this difference:

Figure 3-1 Graphical language: Dangling Association. Textual language: If-statement with dangling Else

A completely syntax-directed editor for a graphical language would not permit the above graphical situation, as it is not syntactically correct. If a user attempted to accomplish the above *inconsistent edit* the editor would deny the edit. We may easily claim that most modern and popular editors for a textual language *would* permit the above textual situation, even though it is syntactically incorrect, since inconsistent interim states are permitted in most such editors, as they are not *completely* syntax strict. If a user created the above situation a syntax oriented editor would inform the user of the error, but not deny the mere existence of it.

This example highlights the major difference between textual and graphical editors for formal languages. In editors for graphical languages the user is vastly more *constrained* when it comes to possible edits, and we therefore need to closely examine the implications of this and how we deal with inconsistency creating edits.

3.1.1 WORKING WITH COMPLETELY SYNTAX-DIRECTED TEXTUAL EDITORS

The implications of completely syntax-directed editors on textual languages has been researched for many years. Since the programs in such editors much remain syntactically correct after each editing operation, a large number of edits that are otherwise very simple become awkward and frustrating [24]. An example of this is the following well-known *if-to-while* transformation problem. The following program is supposed to calculate the factorial:

```
read(n):
a := 1;
fact := 1;
IF a <= n THEN
BEGIN
  a := a + 1;
  fact := fact * a;
END
```

Figure 3-2 Syntactically correct textual program

The above program does *not* calculate the factorial ***n!*** although it is perfectly syntactically correct in our pseudo-language. The error is that instead of using a *while-loop* an *if-statement* is used. But fixing this in a completely syntax-directed editor is not simple. If we were to try to replace the IF with a WHILE we would create a syntactically *incorrect* program since the syntax for *while-statements* is while |condition| do, not while |condition| then. The solution in this case is to use "trickery" to accomplish the desired program; for instance by creating a new *while-statement* and copying the conditions and statements of the *if-statement* into it, and then deleting the old *if-statement*. This is of course more awkward than in a regular text-editor in which we simply type in while in place of if and proceed with typing in do in place of then.

3.1.2 WORKING WITH COMPLETELY SYNTAX-DIRECTED GRAPHICAL LANGUAGE EDITORS

This method of "getting-around" constraints put upon the user by the editor is exists also in modern graphical editors. Figure 3-3 is a screenshot of a state-of-the-art editor for UML Class Diagrams [4]:

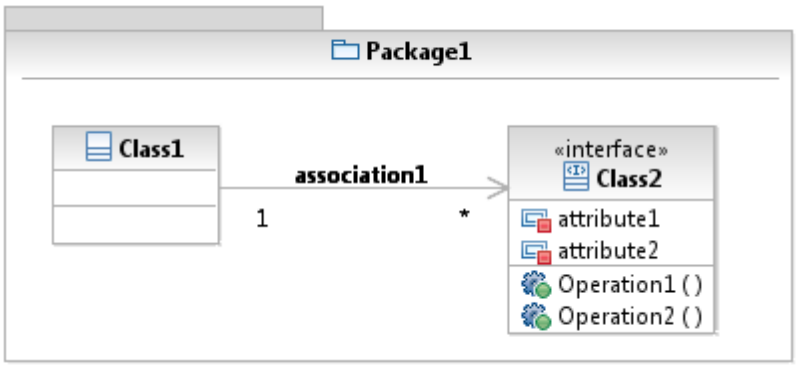

Figure 3-3 Interface erroneously created by user instead of Class. Neatly depicted inside a Package

In this case we presume that the user did not intend to create an *Interface*-element or wishes to alter the element to a *Class*-element. To accomplish this without losing the content of Class2 (attributes and operations) the user must do the following:

1. Create a new *Class*-element : Class3
2. Drag-and-drop the attributes and operations from the old Class2 to the new Class3.
3. Move the association-end from Class2 to Class3
4. Delete Class2.
5. Rename Class3 to Class2

There are several problems with this approach:

1. It is time-consuming compared to just deleting the <<interface>> stereotype (which is not possible in the editor).
2. It entails expanding the Package1 graphical element to be able to fit in the new Class3.
3. It also entails that we need to scale Package1 back again once we have accomplished the goal.
4. Fourthly and importantly, it also means that we cannot guarantee that elements in *other diagrams* that *referenced* the old Class2 have updated their reference to the new Class2 (which was temporarily Class3).

A more elegant solution would be for the editor to accomplish the above without the user needing to use "trickery", while at the same time ensuring that all references to the *old*

Class2 are updated to the *new* Class2. To do this the editor would need to expose the <<interface>> element for selection and deletion, and have some routine that atomically does the steps above and updates references. We call this type of routine an *Editing Behavior*.

3.1.3 BENEFITS OF COMPLETELY SYNTAX-DIRECTED EDITORS

Although completely syntax-directed editors can be awkward from a usability stand-point, as we have seen in the previous chapter, there are several benefits to such editors.

One benefit is that it allows us to implement the well-known Model-View-Controller pattern (MVC-pattern) in editors. The *Model-View separation* principle states that the model (domain) objects should have no *direct knowledge* about the view (commonly interface objects) [28]. In our case we may view the model as an instance of a DSL Meta-Model, and the view as an instance of the DSL Graphical Definition (a model) with some direct relationship to an on-screen rendered figure (what is commonly referred to *as* the view, but we will think of the view as instances of the Graphical Definition).

A relaxation of this principle is the O*bserver pattern*, famously described by the "Gang of Four" [29], which allows entities to *observe* and see events. In the context of an MVC-pattern we say that model elements are permitted to send messages to view elements, but then only by using an *interface*. This way the model is able to communicate notifications, for instance *updates,* about events that have taken place to the view without actually having any knowledge about the view element other than that it implements the *interface*.

To summarize, the benefits of complete syntax-direction with respect to editor development are as follows;

1. The MVC pattern allows us to have multiple representations (Views) on *the same model element*
    a. This is particularly important for multi-diagram editors, in which the same model element may be represented multiple times in several different diagrams.

2. The Observer pattern allows us to keep model and view *synchronized* at all times; if a model-element is updated via some other mechanism than the controller itself, the controller is notified and it may update the view, or vice versa if the view updates.
    a. This fits well with completely syntax-directed editors in that we are never allowed to have inconsistent or illegal models. We may then continuously, and in the "background", synchronize and update either view or model whenever one or the other has been updated.
3. The MVC pattern also allows us to create a direct link between Controllers and a DSL Mapping Model. Controllers may be viewed in a sense as *executing instances* of a *Mapping Model* in the DSL development process[7]. Said using meta-modeling terms we may say that Controllers may be seen as *instances of* elements in the mapping model between the abstract and concrete syntax. This since controllers and maps fulfill some of the same tasks; to provide the link between model and view and keeping them synchronized at all times.

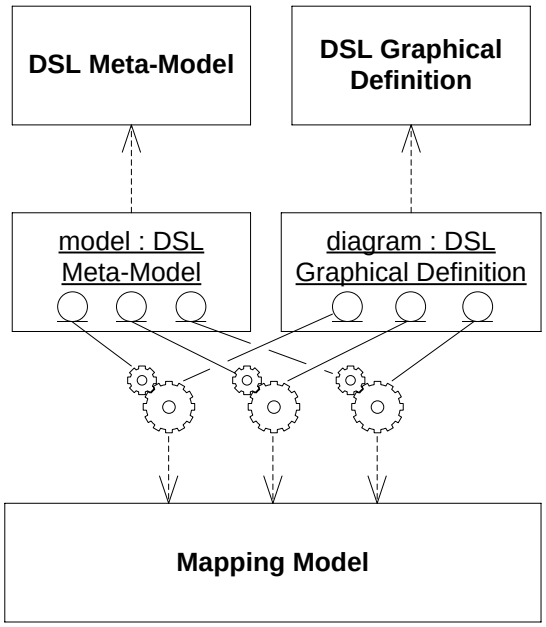

Figure 3-4 Controllers as instances of the Mapping Model elements

3.2 MODELING EDITORS FOR GRAPHICAL LANGUAGES

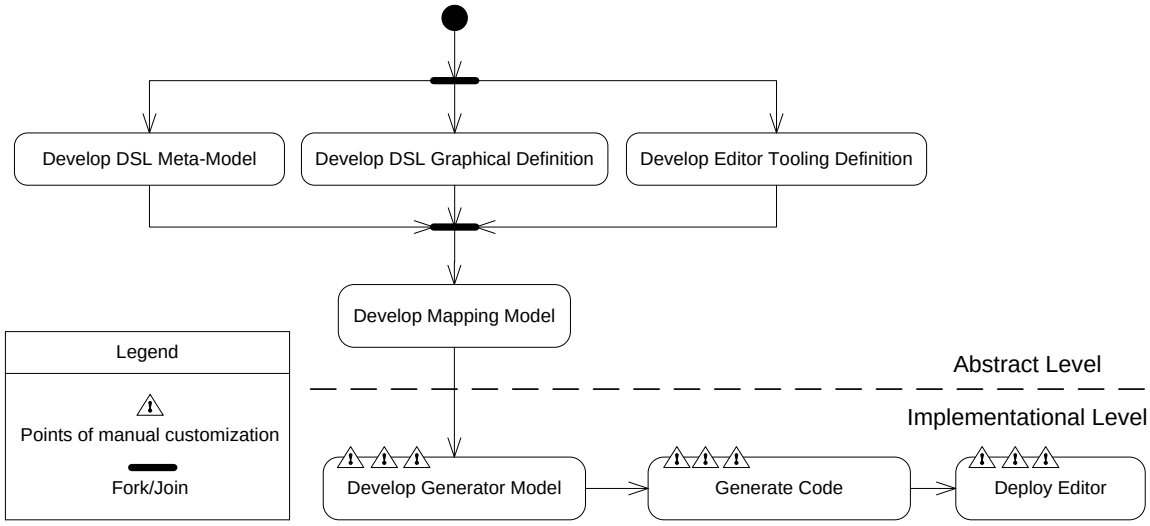

Figure 3-5 The workflow when modeling an editor using state-of-the-art tools

Modern state-of-the-art frameworks for the development of editors for graphical languages (graphical-editor-modeling-frameworks) like GMF, employ a model-driven development approach (MDD). Graphical languages consist of both an abstract syntax and a concrete graphical syntax. These are separated into two different models when creating an editor with a model-driven development approach. In GMF these are respectively called the *Domain Model* (instances of the EMF *Ecore* meta-model [18]) and *Graphical Definition* (instances of the GMF *notation* meta-model). GMF also employs a *Tooling Definition* (instances of GMF *tooling* meta-model); as editors typically include a palette and other tools to create, modify and delete content in the diagram and model. The Tooling Definition specifies these elements on an abstract level; defining what buttons should be in which menus and so on; basically defining a simple model of the user-interface.

Henceforth and throughout this thesis we will refer to what GMF calls a *Domain Model* as a *DSL Meta-Model* (DSL-MM), so as not to confuse the very distinct terms Meta-Model (defining the DSL) and Model (*instances of* the DSL Meta-Model). We will also rename what GMF calls a *Graphical Definition* to *DSL Graphical Definition* (DSL-GD) to more closely bind the Graphical Definition to the *DSL* it was created for. The *Tooling Definition* will rename to *Editor Tooling Definition* (ETD), again to more closely bind the model to it intention; to define the tooling for an *editor*.

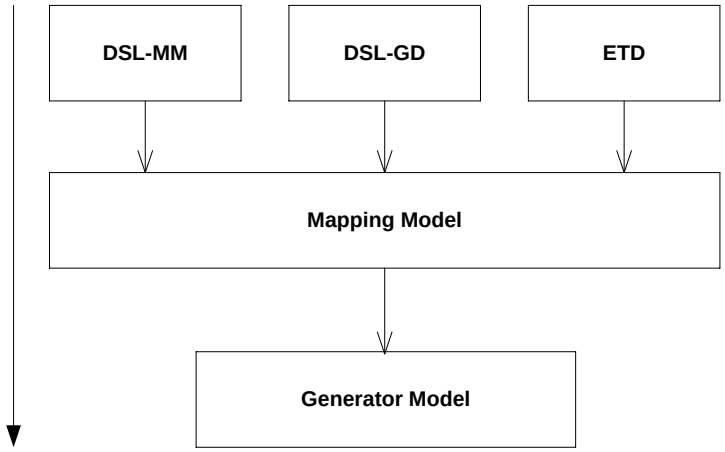

Figure 3-6 Models involved when modeling an editor using state-of-the-art tools (GMF)

Once these three models (DSL-MM, DSL-GD, ETD) have been created we create a *mapping model* (MAP) (GMF *Mapping Model*) that binds the elements of the 3 models together into a coherent map; tying the underlying DSL meta-model together with its DSL graphical definition and the editor tools needed to create and manipulate it.

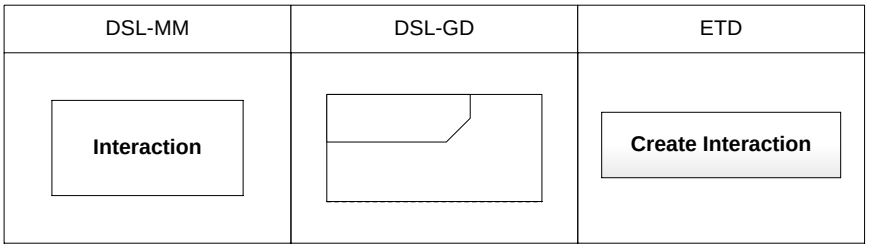

Figure 3-7 Creating a MAP-element

A typical mapping in a GMF mapping model would be similar to the figure above. We map the *Interaction* (DSL-MM element) to its graphical representation (DSL-GD) and to a button with a label "Create Interaction" (ETD-element), which defines a button in the editor responsible for creating an *Interaction*. Once a mapping model has been created a model-to-model transformation (M2M) generates a *Generator Model* from which executable code may be generated. Different from the Mapping Model the Generator Model necessarily contains all the information needed to automatically generate an editor for the DSL and therefore contains information about the technicalities of the intended implementation platform.

We differentiate between two levels in the workflow in Figure 3-5; the *abstract* and *implementational*. On the abstract level reasoning about the editor under development takes place in an abstract manner; a Toolsmith (Editor Developer) focuses on modeling the editor using the concepts from the DSL Meta-Model, DSL Graphical Definition and the Editor Tooling Definition without needing to use modeling concepts that refer to the concrete implementational platform. On the Implementational level we begin work on these technicalities; what runtimes and APIs to use, we define necessary identifiers used in the generation, define the file extensions needed, copyright information, and perhaps import into the models some *action language* for the *advanced behaviors* the editor is required to exhibit. Within GMF the Generator Model is the most likely to be extended and manipulated to provide *customizations* [8].

3.2.1 CUSTOMIZATIONS

Customizations are alterations of the default generated Generator Model. We may have specific requirements for how an editor is to *behave* with respect to the DSL that a Toolsmith needs to implement. But which is not possible to define in any of the previous models, and that is not possible to inherit from the graphical-editor-modeling-framework being used. The amount of customization needed for a language depends on how little the language *deviates* from the type of languages the creators of the framework have anticipated it being used for. Typically, very simple languages just containing nodes and lines in its DSL-GD and few elements in its DSL-MM may require no customizations of the generator model at all, while others more complex (dare we say more domain-*specific*?) require heavy customization. In Figure 3-5 we have used ⚠ -icons to depict the imagined amount of customization required during editor development for a DSL. We may say that;

- The number of ⚠ required in the workflow is directly correlated with how much the language in question *aligns* with the languages the graphical-editor-modeling-framework developers had in mind when developing the framework.

GMFs Generator Model supplies some basic methods for accomplishing customizations via *Custom Behavior* elements. A *Custom Behavior* element simply allows for a class-name of an EditPolicy to be entered. EditPolicies are coded elements in Java, a concept stemming

from GEF and provide the main and undisputed mechanism for adding behavior to a diagram element [8].

More often than not, customizations do not only take place in the Generator Model but also in the code that has been generated. This "generated-code customization" process is a artifact when using a model driven development approach, perhaps due to a lack of expressivity in the meta-models used in the development. This is not always desirable, as we can quickly create modifications in the code that are not in-sync with the models; leading to code customizations being overwritten in an iterative editor development process, or becoming inconsistent with the code generated.

We may also customize by affecting and customizing the *generation* itself via *templates* [8]. This is somewhat more in-line with a model-driven development approach, but as with the process of direct code manipulation, we need to create such templates with the concrete implementation in mind, instead of reasoning about the customization with a more abstract and DSL-"near" approach.

4 EDITING BEHAVIOR

"The quality of our thoughts is bordered on all sides by our facility with language"
- J. Michael Straczynski

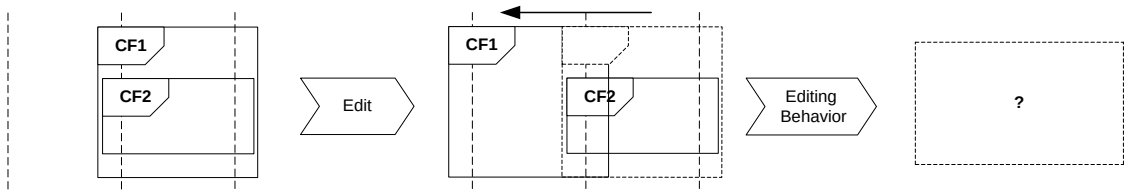

Figure 4-1 Edit resulting in inconsistency, Editing Behavior resulting in ?

When we do not have strict editors that only allow users to perform a basic set of edits (e.g. buttons for all edits, grayed out if currently illegal), but allows users to attempt a wide range of edits, which may or may not be consistency preserving (e.g. move, scale, place with a mouse), then we must examine the implications of the edit. We need to find out what to do if the edit violates some constraint in the DSL. To do this we need to examine the procedures in the editor that deals with such edits. There are multiple possible solutions to the problem of a constraint violation as a result from an edit (inconsistency creating edit):

1. To deny the edit and revert to a previous consistent diagram state (at least visually as the diagram may not have been altered at all). Notify the user of which constraint was broken. This is the MOF-default action for constraint violations [22].
2. For the editor to permit the edit. Depending on how the model repository reacts and how strictly it enforces constraints, will either create an inconsistent model or will result in the repository denying the edit, and not the editor.

3. To solve the violation by automatically initiating an e*diting behavior* that modifies the original edit (what some have called "compensating actions" w.r.t. to inconsistency management in general [30]) and failing that do 1.
4. To *not* automatically initiate a behavior, but find all the possible editing behaviors capable of solving the violation/inconsistency, and present these to the user for user-selection. If no behaviors are found or the user does not select a solution, *then* do 1.

The first solution is partially implemented in most state-of-the-art editors; if a user attempts to do something deemed illegal the editor *undo's* the interaction, and in some cases notifies the user about why. As is the case in IBM's Rational Software Modeler that runs a model validator in the background checking pre-defined and user-settable constraints [26]. The most common solution is however to merely deny the interaction without feedback, as is the case in popular tools such as Eclipse's UML2Tools, Papyrus UML and in the perhaps less popular tool; Limyr's SeDi [5, 6, 27]. IBM's Rational Software Modeler also denies most illegal edits without explanation, although a structure for feedback does at least exist, as we have mentioned.

The second "solution" may exist in editors that have been manually developed, heavily customized or loosely constrained. Illegal situations not anticipated by the developers may not be tested for and therefore not detected. This is also perhaps a direct consequence resulting from the immaturity of the field of formally defining the constraints on meta-models, especially on the concrete graphical syntax. E.g. the DSL does not define enough constraints to be able to guarantee that it is consistent with respect to the intention of the DSL developers. The worst case scenario in this "solution" is the corruption of the model(s), due to a controller trying and failing to synchronize an inconsistent diagram with the model.

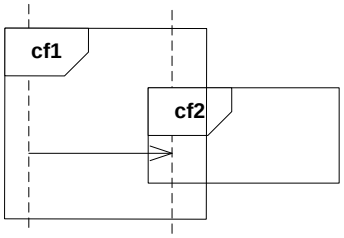

Figure 4-2 Who owns the message head?

Figure 4-2 is an example of such a situation where the *worst* may occur: Which box is the arrow-head contained within? If we automatically deduce parentage of model elements based on the diagram every time we receive a notification from the model of updates (which is common for controllers in MVC-type editors to receive) we may find ourselves stuck in an infinite loop: Checking first `cf1`, setting it as parent of the arrow-head. Receiving an update from the model. Skipping `cf1` as it already has all its children, not resulting in model-alterations. Then checking `cf2`, setting *it* as a parent. Receiving an update from the model. Setting `cf1` as parent etc.

The third solution is the standard way of dealing with constraint violations (although not necessarily violations against model constraints, but also constraints given in the code to specify how an editor reacts). This is often exhibited by the way arrows in box-arrow-type diagrams *route* themselves around other elements in order to maintain visibility, and not become overlapped/hidden by other elements. This behavior could be a reaction to a constraint defined in a Graphical Definition, or just constraints defined in the editor code). These types of *behaviors* are often included in editor frameworks such as in GMF and GEF [1, 2], as they are behaviors that the Toolsmiths request, or take for granted exist in the framework. The Toolsmith therefore merely inherits the behavior from the framework, requiring little or no developer effort to implement (as with predefined *EditPolicies* in GEF and GMF mentioned earlier). Behaviors that fall outside of what the framework developers envisioned being needed, are defined manually via *customizations* of a generated editor.

The fourth solution is what we will try to accomplish in this thesis. This type of solutions is akin to how syntax-oriented editors that are *not* completely strict w.r.t. the syntax react to an inconsistency, presenting errors and possible fixes based on a background parsing strategy. This while at the same retaining the MVC-pattern used in state-of-the-art editor

development, that in theory ensures that model and diagram are in constant consistency. An important requirement of such a solution is that it does not in any way *lock* the editor during its search for solutions, but runs in parallel with the editor, as do background parsers in syntax directed textual language editors.

4.1.1 WHY GIVE THE USERS A CHOICE OF BEHAVIOR?

"We can think of the "scaffolding" here as providing a knowledge framework upon which a learner can learn while gaining expertise, as a way to help the user climb the learning curve. This suggests organizing content to build on the learner's accumulated knowledge." [31]

An important aspect of the success of any DSL is the tool-support surrounding it. Consequently the success of the tools will also determine the success of the DSL. An important aspect of tools are their usability. Several industry leading tools, such as [26] acknowledge this and provide users with small *helpers* or "scaffolding" from which a user may learn about the DSL while using it. Examples of this are depicted in Figure 4-3 and Figure 4-4. Other examples are how many editors support drag-and-drop of elements, initiating an editing behavior that automates an otherwise complicated process. For instance IBM Rational Software Modeler [26] supports dragging the Class1 in Figure 4-3 out of its Package1 and into another, automating the operations needed to change parentage, maintain certain associations etc., instead of the user manually creating a duplicate Class and reproducing it in the other package.

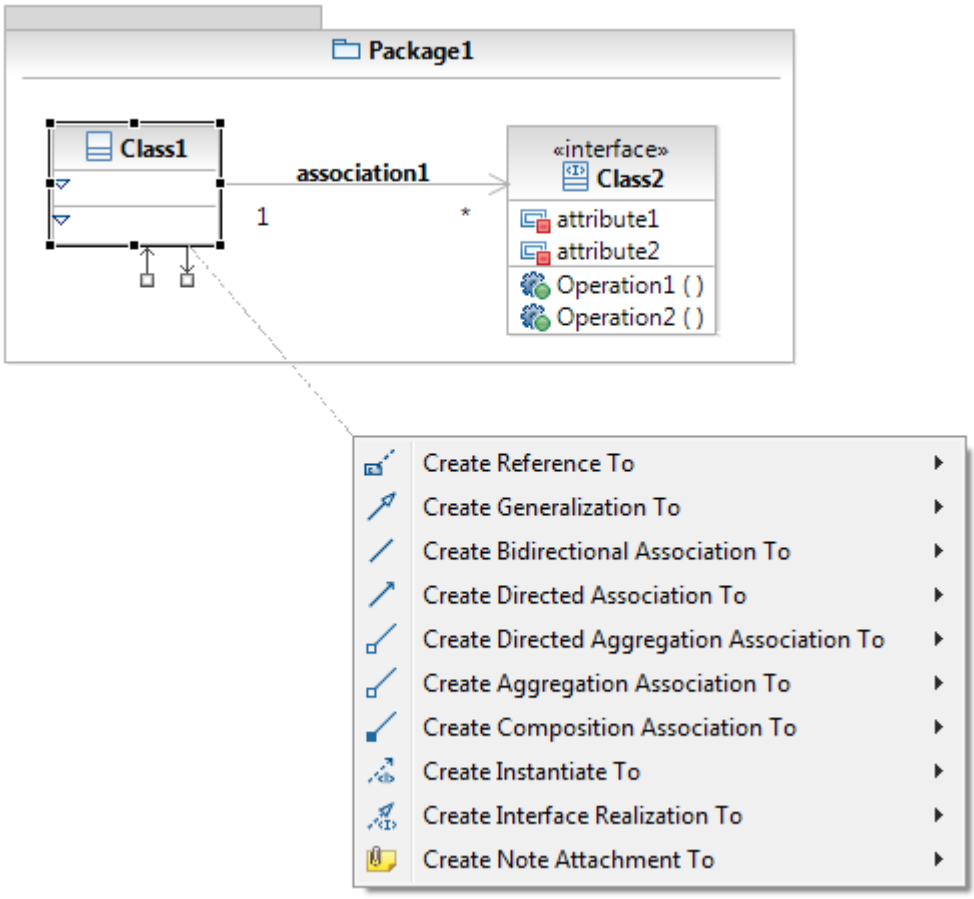

Figure 4-3 Helper: Associating to "nothing" leads to a context-menu with list of possible Creates

Figure 4-4 Helper: Mouse-over on blank space leads to "bubble" of possible Creates

In these editor interface screenshots we see how an editor *helps* the user learn about the DSL that is being used, by presenting the user with all the different DSL elements (known and unknown to the user) that the user may create as a result of attempting to create an element on a blank space in the diagram.

It is scaffolding such as these that we will attempt to lay the foundation for with the Behavioral Framework, including but not limited to element creation. During editor usage we may often find ourselves attempting to create what the editor deems as *inconsistent* with

respect to the DSL. This is of course reasonable as we usually want consistent models. However, the current method of denying user-interactions without any given reason, or merely by stating to the user that a constraint has been violated is not enough. What is needed is a form of error-reporting mechanism *with* solution finding capabilities that does not necessarily automatically implement the *inconsistency solving* solution, but gives the users the possible solutions for user selection. This we believe will improve editor usability and provide the users with a greater chance of "learning (the DSL) by doing".

4.1.2 DEFINING EDITING BEHAVIOR

4.1.2.1 Edits as model transformations

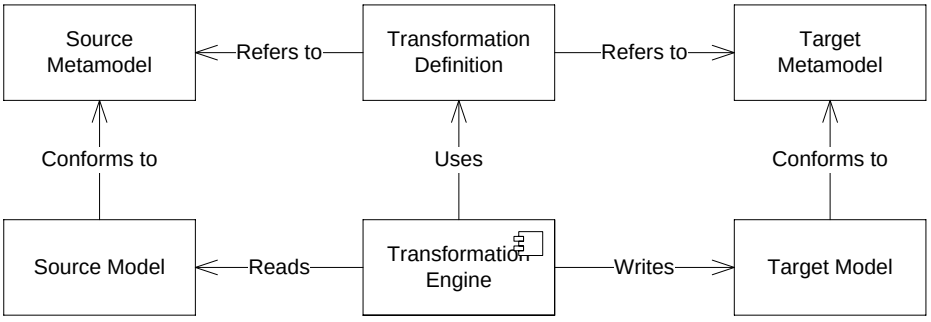

Figure 4-5 Basic concepts of model transformation

Many have researched the topic of combining graph transformation with field of model transformation. Of the many model transformation techniques, graph transformation-based techniques are present in many current model transformation implementations [32]. Some have also used graph transformations and rules to actually define graphical languages and editors for them, by viewing the transformations as *edits* on graphs representing the language [33, 34]. Others have also examined how to translate OCL Constraints, which lack a model representation, into graphs for efficient evaluation and visualization of them [35].

There exists a considerable interest for establishing standards that deal with model transformations in Model-Driven-Architecture (MDA) [32], and there exists several languages with different design choices that provide the definition of model transformations. Among which are the ATLAS Transformation Language (part of Eclipse's model-to-model (M2M) project) (ATL) [36] and OMG's standardized specification QVT [37]. The full extent of the interest in model transformations are far beyond the scope of this thesis, but importantly there exist a general consensus that it is possible to use graph patterns as rules for transformations and match those patterns to patterns in a source model for the transformation. This is called the graph-transformation-based approach to model transformation. According to Czarnecki and Heckel, patterns can be represented using both the *structure* of a model (strings, terms and graphs) or the abstract or concrete syntax of the corresponding source or target model language, the syntax may be either textual or graphical [32].

Graph transformation rules have a LHS (left-hand-side) and a RHS (right-hand-side). LHS patterns are matched in a model being transformed and replaced by the RHS *in place*. LHS usually also have NACs (negative application conditions) and PACs (positive application conditions). Heckel in [38] gives the following three steps to performing a graph transformation given a rule *t:*

1. *Find* an occurrence of the LHS of t in a given graph R.
2. *Delete* from G all vertices and edges matched by LHS but not in RHS.
3. *Paste* to the result a *copy* of the RHS, yielding the new graph G.

An important aspect of graph rule patterns w.r.t. DSL transformation is that it is theoretically possible to render them in the concrete *graphical* syntax of their respective source or target language [32]. The ability to incorporate the source language's *own* syntax into rules for its transformation would greatly simplify the creation of such rules.

However, those approaches that we have found that use graph transformations and rules to define editing behavior [39, 40] differ from what we are after in this thesis; they explore graph transformation rules that *never* create inconsistencies; the only edits available to the editor are rules that operate on initially consistent models and lead to consistent models. In this way they may guarantee the consistency of both the LHS and the RHS of any transformation. We on the other hand wish to use graph transformations to represent edit executions, and that transform from a consistent target model w.r.t. to its DSL *into* a source model which might be inconsistent w.r.t to the DSL. And use information from this transformation process to deduce the applicability of predefined editing behaviors.

4.1.2.2 Edits as transactions

Other approaches to defining how edits are handled (editing behaviors) are related to transaction processing; how to handle an edit as a transaction (like in EMF's TransactionalEditingDomain [3]). We may define a transaction as "a collection of operations on the physical and abstract application state" [41]. ACID is a set of properties on the transaction that must hold for a valid transaction on a state (we may view this state as the models in the DSL repository):

- **Atomicity:** A transaction's changes to the current state are atomic; all changes happen at once.
- **Cosistency:** A transaction is a *consistent* transformation of the state. It does not violate *any* consistency constraints on the state.
- **Isolation:** Even if transactions may execute concurrently, each transaction T views itself as either executing before or after another T.
- **Durability:** Once a transaction is completed successfully (committed) its changes to the state are persisted.

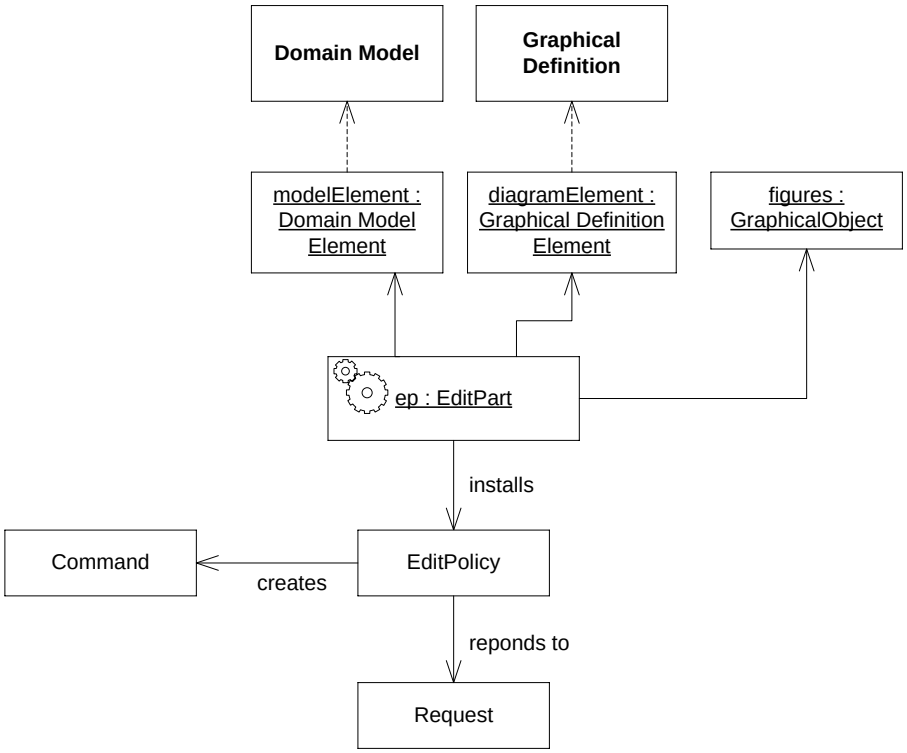

Figure 4-6 EditPolicy and Transactions: Commands

GMF uses EMF's TransactionalEditDomain along with entities called EditParts and EditPolicies (from GEF) to manage editing behavior. EditParts define the controller entity between model-elements(s) and view (on-screen-rendered elements), while EditPolcies are *pluggable contributions* to the overall editing behavior of an EditPart [8]. EditParts delegate the handling of edits to the EditPolicy classes, which are installed upon the EditParts during instantiation. EditPolicies respond to Requests originating from, among other elements, Tools and return Commands (and stacks of Commands) for execution within a transactional editing domain. EditPolicies may collect contributions from other EditParts by delegating

and/or forwarding Requests or new Requests. It is within these EditPolicies that most of the DSL specific editing behavior is usually performed, and is also among those concepts designated for manual customization by GMF. There are many benefits to this structure that we will not examine in this thesis, but we may in the context of this thesis think of Requests as close to what we have called previously called *edits* and Commands as the execution of said *edits*.

As we have mentioned before we would like to explore the use of graph transformation as a way of defining editing behaviors, instead of using the method employed by EditPolicies of programmatically defining them. We want to explore the possibility of using pattern matching on "inconsistent" models against patterns in rules, which when matched, define *when* and for *what* inconsistencies an editing behavior is applicable. This as a result of needing to allow users to perform inconsistency creating edits when the initial state of the models are consistent. If we are to use the concepts from graph transformation to express editing behaviors, and graph transformation rules to express *when* we can use editing behaviors we need a way to define a form of *inconsistent* model, capable of representing the result of an inconsistency creating edit.

4.1.3 LIVING WITH INCONSISTENCIES FROM EDITS AND EDITING BEHAVIOR

Goedicke et. al. [42] argue for the need to be able to live with inconsistencies during the lifetime of systems, and that tool-support is needed to tolerate inconsistencies and help developers use them to drive the development process forward. They put forth several activities needed to manage inconsistencies: *inconsistency detection, inconsistency classification,* and *inconsistency handling*. We will adapt these terms to our problem domain, in which we in need to *live* with inconsistencies resulting from edits, at least temporarily.

Inconsistency detection: We need to define what an inconsistency is in our terms and how it is *detected;* As we have mentioned before, we say that constraints in the DSL define what relationships must exist between elements (consistency conditions). Detection of an inconsistency is therefore done, in our view, by a MOF-like model repository; one that can check constraints defined in the DSL.

Inconsistency classification: Goedicke et. al. argue to classify inconsistencies as either minor (suitable for automatic solutions) or major (which may represent severe design errors). We view all inconsistencies as major, and therefore never actually attempt to fix them automatically (although we may easily envision such a solution for inconsistencies for which only *one* solution exists).

Inconsistency handling: We *handle* inconsistencies in line with point 4 in chapter 4, of the possible solutions to inconsistency creating edits.

For us to use graph/model transformation rules as representations for edits and editing behaviors that *might* create inconsistent models, we will need a method of *living* with the inconsistencies, at least until we are able to find editing behavior(s) that solve the inconsistency. A regular repository would not permit us to create inconsistent models, for good reason. However if we are to be able to use patterns representing inconsistent situations as rules for editing behaviors, we need to be able to represent the inconsistencies in a model capable of being pattern matched against.

4.1.4 THE NEED FOR A META-MODEL CAPABLE OF DEFINING AN INCONSISTENT DSL-INSTANCE CONSISTENTLY

What we need is a special meta-model capable of representing the inconsistencies not permitted in another meta-model. For instance, for a graphical definition, constrained according to the intended graphical syntax of the DSL, we want a variant of the graphical definition that is not *constrained,* so that we actually may create an "inconsistent" model consistently.

Hausmann et. al. in [43] employ graph transformations to express model transformations and use them to denote the consistency conditions between models. Model transformations describe the applications of techniques on a source model to produce a target model. The transformation may either be monolithic or done in steps. Transformations can be regarded as *functions of models*:

$$t : Model \rightarrow Model'$$

Models represent (views on) information. If the same information is represented in multiple models, we may say that they *overlap*. A typical example of overlapping models

could be a class diagram and sequence diagram. Although expressing different views they can do so on the same information; a sequence diagram showing the interactions of a class, which is also defined in a class diagram. Another example of overlapping models is how the UML meta-model represents the meta-view of a UML Model. The overlap relationship can be represented as:

$$overlaps : Model^n$$

When overlapping models are *consistent* we mean that they represent information in a *non-contradictory* way. Typically consistency between models is determined via *consistency conditions*. (e.g. the <<instance of>> consistency condition between a UML Class Meta-Model element and a Class in a model claiming conformance to UML). We regard *consistency conditions* as a relation over models.

$$cc : Model^n$$

We may use the above relationships to define what a well-formed model with respect to its meta-model is. Hausmann et. al. define it as the following:

$$\forall\, A, B \in Model : cc(A, B) \Rightarrow overlaps(A, B)$$

when they only view *static* consistency (the structure of models) and not *dynamic* consistency (the behavior of models), and only look at *binary* consistency relations. If a consistency condition holds between two models A and B, then the models A and B also are overlapping (e.g. if a Model:Class <<instance of>> UML:Class then Model and UML overlap). So following the above *cc* is a subset of *overlaps:* $cc \subseteq overlaps$

Hausmann et. al. also arrived at multiple interesting, and for this thesis relevant, questions when combining the concepts of consistency and transformation.

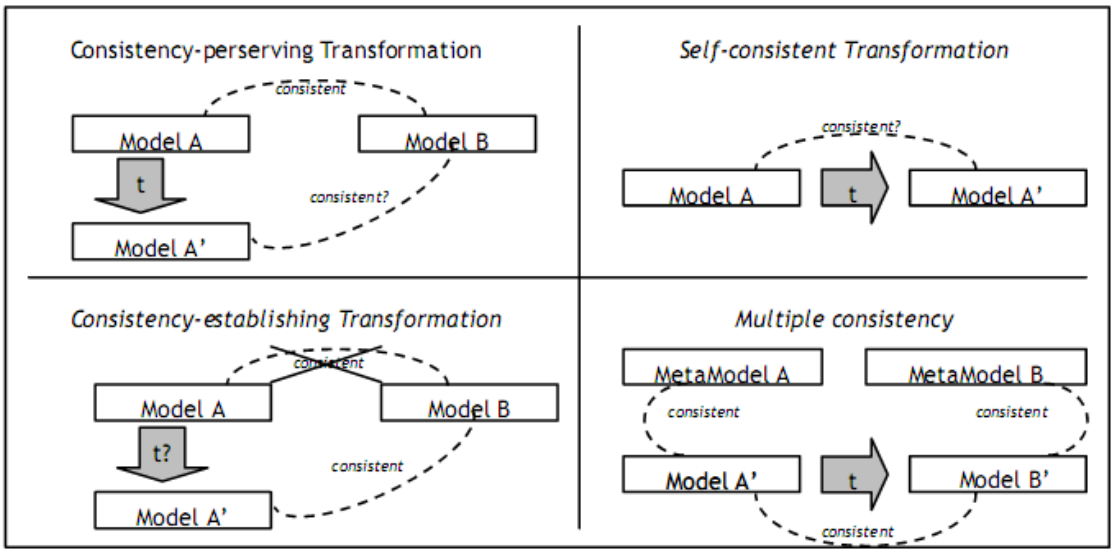

Figure 4-7 Scenarios with interrelation of consistency and transformations (fig. courtesy of [43])

1. Is the transformation consistency-preserving; does it alter a model (A) in such a way (A') that its consistency with another model (B) is preserved? E.g. if a UML Model A is consistent with the UML Meta-Model B, does a transformation *t* result in a model that is consistent with B? Formally we say that a transformation *t* is *consistency-preserving* iff

$$\forall A, B \in Model : cc(A, B) \Rightarrow cc(t(A), B)$$

2. Is a transformation *consistency-enforcing*; does it alter a model (A) in such a way that (A') is consistent with a model (B) when the original (A) was not? A transformation not proven to be consistency-preserving can result in models that are inconsistent. Some inconsistent situations can be solved by providing transformations that *re-establish* the consistency. Formally we say that a transformation *t* is *consistent-enforcing* iff

$$\forall A, B \in Model : \big(overlaps(A, B) \wedge \Rightarrow \neg cc(A, B)\big) \Rightarrow cc(t(A), B))$$

3. Is the transformation itself *consistent*, does it produce a target model that is consistent with the source model? This question is usually related to concerns regarding the preservation of semantic properties. Formally a transformation is a *consistent transformation* iff

$$\forall A \in Model : cc\big(A, t(A)\big)$$

E.g. is the code generated from the model (t(A)) consistent with the model (A)?

4. A combination of the above scenarios is a common case. The target model (B') must be consistent with its meta-model (B) and be semantically related to its source model (A'). For a transformation the consistency conditions would state the semantic relation, or that a target model is a true representation of the transformation defined given the source.

$$\forall A, A', B, B' \in Model : (cc_A(A, A') \wedge t(A') = B') \Rightarrow (cc_t(A', B') \wedge cc_B(B, B'))$$

E.g. A Model (A') is <<instance of>> the UML Meta-Model (A), the transformation of the Model (A') results in the Program (B'). This should imply that the Model and Program are consistent ($cc_t(A', B')$) and that the Program (B') is <<instance of>> Java (B).

We may use the above definitions to represent what we require in this thesis of the special "relaxed" DSL that we need to represent inconsistencies. Once we have a model that can represent inconsistencies we may *match* model transformation rules to the inconsistencies and attempt to *solve* them. Formally for an initial consistent model R instance of meta-model DSL_{GD}, a meta-model I which is consistent with DSL_{GD} but with none of its constraints**,** and a model G instance of I we say that:

$$R, DSL_{GD}, I, G \in Model : \left(cc_{DSL_{GD}}(DSL_{GD}, R) \wedge t(R) = G\right) \Rightarrow (cc_I(I, G))$$

meaning that for a model R, that is consistent with its meta-model DSL_{GD}, we need a transformation t on R resulting in the model G. G is consistent with the relaxed meta-model of DSL_{GD} called I. Importantly, since t is an inconsistency creating transformation w.r.t. R and its DSL_{GD}, then G is not consistent with R as G is not consistent with DSL_{GD} due to the introduction of an inconsistency by *t*.

Below we show a subset of a "relaxed" meta-model I based on the meta-model GDSQ (Figure 2-6) so that we can represent the model in Figure 2-9 consistently.

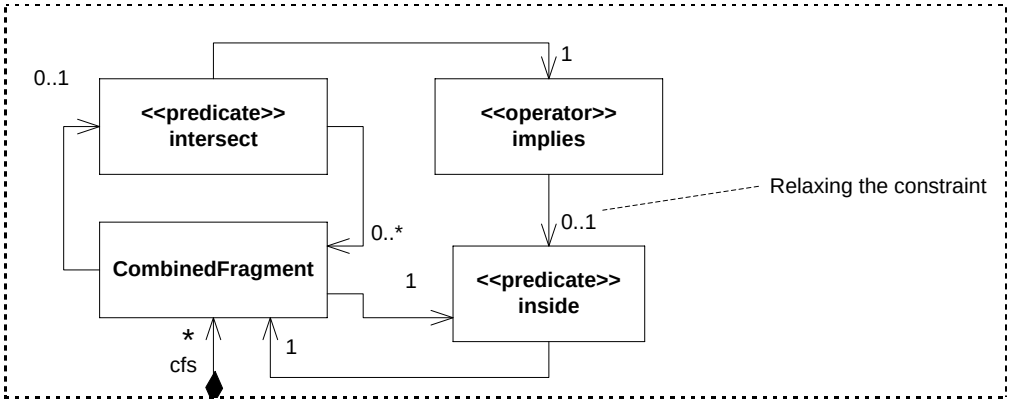

Figure 4-8 Meta-model I : Relaxing the implies constraint in GDSQ

Here we have relaxed one of the constraints on the syntax of GDSQ, giving us model I. We redefine the multiplicity of the relationship between the operator *implies* and the predicate *inside* to 0..*. This allows us to have 0 or 1 relationships between them, meaning that we now permit intersection without a CombinedFragment being inside another. In contrast GDSQ defines that the relationship must be 1; that if 2 CombinedFragments intersect one must be within another.

5 BEHAVIORAL FRAMEWORK

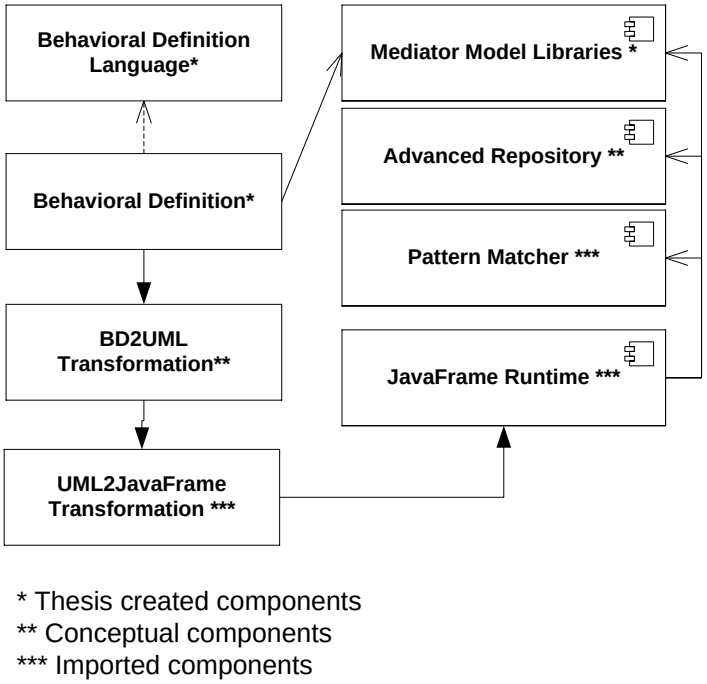

Figure 5-1 Framework Components

The framework we have created consists of components defined in this thesis, of components that exist conceptually, and of components that we have imported. We will here give a brief description of each.

1. **Behavioral Definition Language (BDL)**: a declarative language for the definition of editing behavior in editors for graphical languages, in a manner that allows for the use of rule matching to determine *when* the behaviors are applicable on a DSL-inconsistent model to make it consistent.
2. **Behavioral Definition (BD)**: instances of BDL that define the editing behavior for editors of a given DSL.
3. **BD2UML Transformation**: A conceptual component. BDL aligns with concepts from UML and is therefore transformable into UML.
4. **UML2JavaFrame**: An imported component. We have used a prototype, created directly in UML, for our experiments into editing behavior.

5. **JavaFrame Runtime [44]**: JavaFrame supports the execution semantics of a Behavioral Definition.
6. **Pattern matcher**: A conceptual component. We require the existence of a component able to match our patterns to a model.
7. **Advanced Repository**: A conceptual component. We query the repository for consistency when given an edit on one or several elements, and expect it to return either an OK-type message or NOK-type message. If it responds with NOK we also require that it is capable of representing the inconsistency in a model. We also require that it is capable of producing snapshots of this model pre edit and post edit so that we may incorporate knowledge of previous model states when determining applicable editing behaviors. These models should be returned to the executing behavioral definition.
8. **Mediator Model Libraries:** A simple model library of BDL-mediators that act as APIs between a Behavioral Definitions and GEF editor has been created as part of the prototype.

5.1 JAVAFRAME

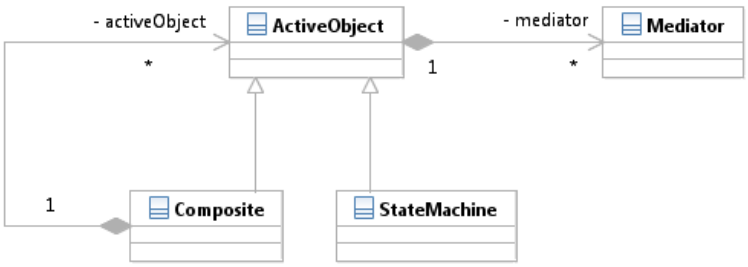

Figure 5-2 JavaFrame Concepts

We have for the prototype created UML models that are compatible with JavaFrame [44]. JavaFrame is a framework for implementing a subset of UML in Java. Consisting of a Java API (Application Programming Interface) for representing UML models in Java and programming guidelines. The API contains classes for UML model elements such as Statemachine, Composite and Port (in JavaFrame terminology called Mediator). A JavaFrame system is a Composite which contains ActiveObjects. ActiveObjects may

themselves either be Composites or Statemachines. ActiveObjects communicate by sending asynchronous messages through Mediators.

5.2 OUR META-MODELING ARCHITECTURE

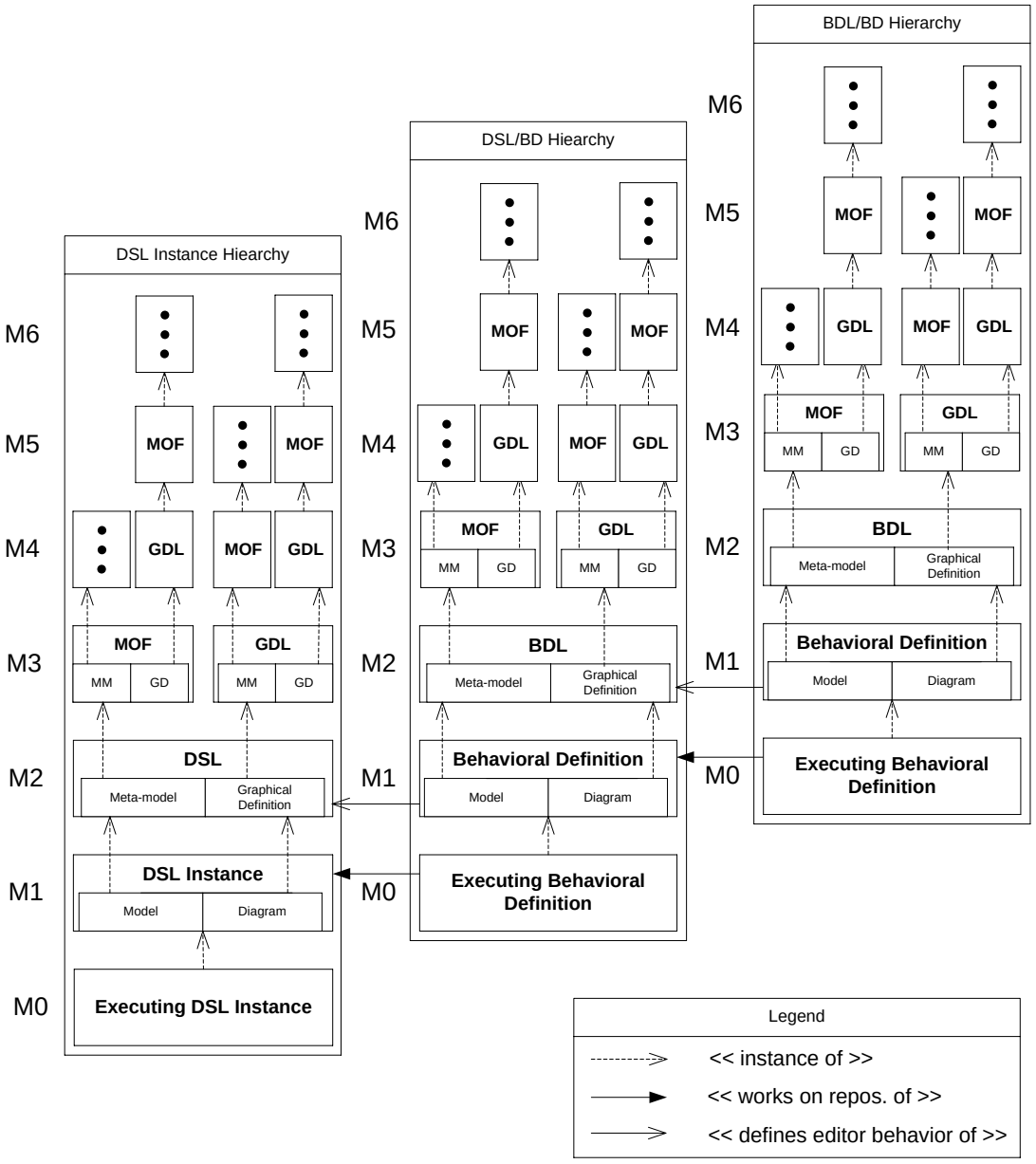

Figure 5-3 Meta-modeling architecture

5.2.1 CONCEPTS OF THE META-MODELING ARCHITECTURE

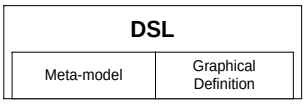

Figure 5-4 The DSL composite

Throughout the figure we make several visual statements when structuring the architecture, one of which is the boxing together of mapped models. This allows us to view DSLs in our meta-modeling architecture not only as the DSL meta-models, but as the *composition* of both Meta-Model (MM) and Graphical Definition (GD); the first containing the definition of the abstract syntax, the second of the concrete graphical syntax. Mapping models needed for binding the two models together are in our view implicit in the boundary between models in the composite. We will denote this composite of models defining a DSL as just **DSL** for the remainder of this thesis.

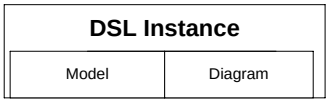

Figure 5-5 The DSL Instance composite

Viewing the DSL as a composite also allows us to view the DSL Instance as a composite, or more precisely as a *composite model* consisting of Model, Diagram and some synchronizing model (instance of the implicit mapping-model, also implicit in the boundary between the two models), that are instances of their respective meta-models.

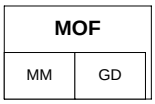

Figure 5-6 Simplified MOF composite

In our meta-modeling figure we assume that *every* language consists of both meta-model and graphical definition, as every language is a DSL. MOF and GDL included.

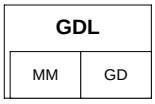

Figure 5-7 Simplified GDL composite

GDL is a term we use for **any** language capable of defining a DSL's concrete graphical syntax, and we assume that it is possible to express such a language using MOF. Instances of GDL are DSL Graphical Definitions (GD).

Figure 5-8 Element for Meta-Model-Only

When referring exclusively to a meta-model we may use the meta-model-only elements above; so that we may show how a DSL composite's GD is <<instance of>> GDL and GDL <<instance of >> MOF in a concise way.

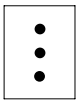

Figure 5-9 Ellipsis for "model instance of model" relations

We use an Ellipsis to show the infinite MOF <<instance of>> MOF relationship, since MOF can be defined by MOF. This also gives us a theoretical infinite number of meta-modeling levels.

Another important visual statement in the figure is the *upwards shifting of meta-modeling level* when defining Behavioral Definitions for DSLs, and the << *defines editor behavior of* >> and << *works on repos. of* >> relations that span hierarchies. More on this aspect in 5.2.3 Domain-Specific Language / Behavioral Definition Hierarchy (DSL/BD).

The hierarchies are named after the intended *goal* of the meta-modeling process.

5.2.2 DOMAIN-SPECIFIC-LANGUAGE INSTANCE HIERARCHY (DSL INSTANCE)

The first hierarchy we call the Domain-Specific-Language Instance Hierarchy. It represents the meta-modeling hierarchy involved when modeling Domain Specific Languages (DSLs)

capable of creating a DSL Instance models. The *goal* in this hierarchy is to create and represent domain specific information. From bottom to top we have:

- At the **M0** level we have the executing DSL Instances, typically runtime objects.
- At the **M1** level we have the models that are instances of the DSL itself, both the model (instance of the meta-model) and the diagram (instance of the graphical definition).
- At the **M2** level we have the meta-model of the DSL, represented as a model which is an instance of Meta-Object Facility (MOF) meta-model. And the Graphical Definition of the DSL, represented as a model which is an instance of the Graphical Definition Language (GDL) meta-model.
- At the **M3** level we have MOF and GDL. Both having internally within their composites meta-model and graphical definition.
- At the **M4** level and onwards repeated <<instance of >> relationships. We define GDL to be <<instance of>> the MOF meta-model. On this level we only represent meta-models and not graphical definitions.

5.2.3 DOMAIN-SPECIFIC LANGUAGE / BEHAVIORAL DEFINITION HIERARCHY (DSL/BD)

The second hierarchy we call the Domain-Specific-Language / Behavioral Definition Hierarchy, within which lies the focus of this thesis; the modeling of editor behavior for graphical DSLs. The *goal* in this hierarchy is to produce a Behavioral Definition for a DSL.

- At the M0 level we have the executing Behavioral Definition for the DSL. It *works on* a repository which contains the instances of the DSL in question (model and diagram).
- At the M1 level we have the Behavioral Definition composite for the DSL. It *defines the editor behavior* for an editor of the DSL meta-model and graphical definition. Since BDL is a graphical language we have both a model which is an instance of the BDL Meta-Model, and a diagram which is an instance of the BDL Graphical Definition. Element names in a BD should reflect the element names from the DSL to ease the process of reasoning about Editing Behavior. We therefore in the <<defines editor behavior of>> relationship say that the relationship also defines a

automatic initial generation of the Behavioral Definition using concepts and names from the DSL.

- At the M2 level we have the Behavioral Definition Language composite which contains both Meta-Model and Graphical Definition.

The rest of the hierarchy is straight forward and is the same as in the DSL Instance Hierarchy.

5.2.4 BEHAVIORAL DEFINITION LANGUAGE / BEHAVIORAL DEFINITION HIERARCHY (BDL/BD)

This hierarchy is included in the figure to show the benefits of the structuring that we have chosen. It shows how we may create a Behavioral Definition for the Behavioral Definition Language itself. Using BDL upon itself has not been attempted, but is deemed plausible.

5.2.5 BDL'S RELATIONSHIP THE DSL

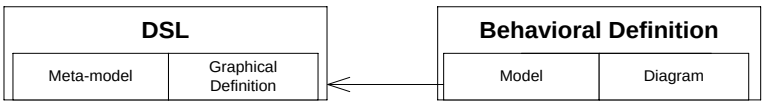

Figure 5-10 BD Behavioral Definition relationship to DSL

GEF and GMF employ a MVC-pattern, as mentioned. BD defines an integral part of what is usually defined in Controllers; editing behavior. A Behavioral Definition needs to employ and use knowledge about the DSL; what makes constraints in them invalid, and what makes them true, what possible behaviors can be attempted on which elements, and on which attributes. All this with the goal of making *consistent* models as a result of the behavior. This requires an intimate understanding of the DSL in question. We will in this thesis leverage a toolsmith's understanding of a DSL[1], but leave open the possibility for

[1] The writer of this thesis has extensive experience with Sequence Diagrams

automatically generating at least a part of the BD to help toolsmiths on their way. With this relationship we also show that we use the DSL's own graphical syntax within a Behavioral Definition (as patterns for transformation rules), although with certain modifications as we wish to be able to represent both *invalid* and valid graphical syntax.

5.2.6 BDL AND THE RELATIONSHIP WITH THE DSL INSTANCE

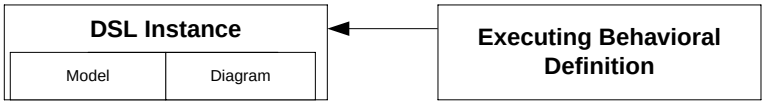

Figure 5-11 Executing Behavioral Definition relationship to DSL Instance

A vital component is the conceptual repository defined previously in the thesis. We require a repository capable of *creating* intentionally inconsistent models w.r.t. to the DSL, meaning that it conforms to a constraint-wise relaxed meta-model of the DSL. This is an important requirement to be able to use graph transformations in the way we have envisioned; where we match the "inconsistent" model against patterns representing inconsistencies on the left-hand-sides of rules. We also require the repository to be able to create snapshots of the model. We use these snapshots to reason about *how things were* in editing behaviors *action blocks* using a History-concept; as we will see later (for editing behavior 1, EB1) we sometimes need to determine the *difference* between the same attribute in two different snapshots to be able to extrapolate attribute values in the actions blocks. Additionally we require the repository to be able to provide transformations that conform to a given edit. Transforming an edit (consisting of a reference to one or several model elements, and one or more attribute values) into a transformation. We imagine this may be accomplished by mapping Edits to transformations using model element types and attribute types.

5.3 BEHAVIORAL DEFINITION LANGUAGE

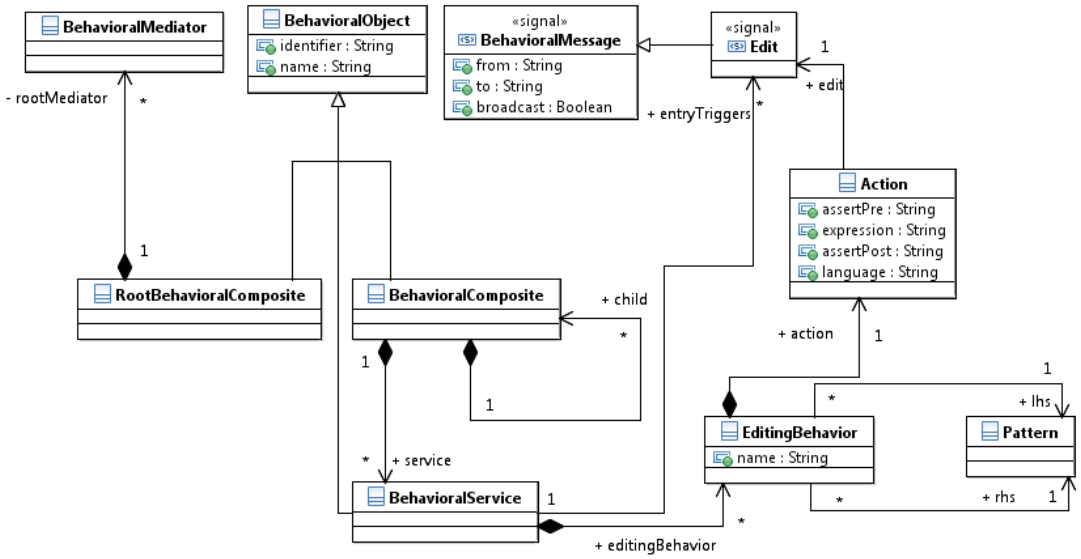

Figure 5-12 Behavioral Definition Language Meta-Model

The Behavioral Definition Language allows for the declarative description of the editing behavior for an editor of a graphical language. The metamodel focuses on defining the main structures needed for reasoning about editor behavior in an abstract way that is conceptually close to elements defined in the DSL using its own syntax in Patterns used to find applicable editing behaviors.

5.4 MODELING A BEHAVIORAL DEFINITION

We will here present an example Behavioral Definition, using BDL's (not formally defined in this thesis) graphical syntax, as we feel this is the best way to explain the concepts concisely.

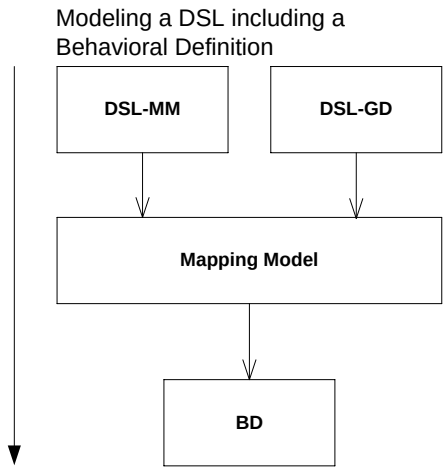

Figure 5-13 Behavioral Definition creation after mapping

The Behavioral Definition requires knowledge of which elements are mapped to each other to be able to give a coherent definition of how elements are to behave when interacted with by the user, and must therefore be created after the mapping has been accomplished. We will omit the tooling element from this point on as it, although an integral part of state-of-the-art graphical editor modeling, of little explanatory value once we have included a BehavioralDefinition.

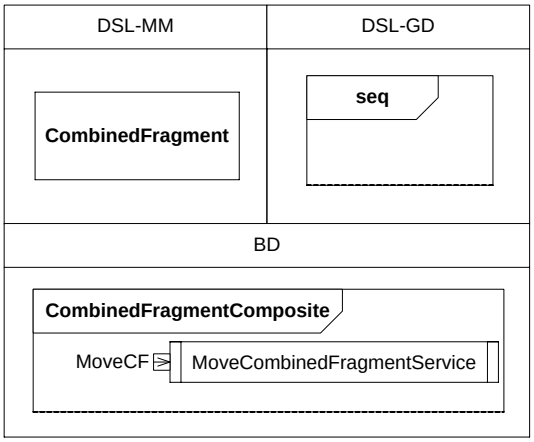

Figure 5-14 Example: mapping a BehavioralComposite to its meta-model and graphical definition element

Given a mapping for a CombinedFragment element from the DSL (UML) we may create a CombinedFragmentComposite that references this mapping element for knowledge of the DSL elements. This allows us to access both meta-model and graphical definition. In particular we use a variant of the DSLs graphical definition to render patterns in a BD graphical syntax. We also define a service called MoveCombinedFragmentService, with an

entry trigger Edit called MoveCF. This service will be instantiated every time a MoveCF-type edit is received from the editor.

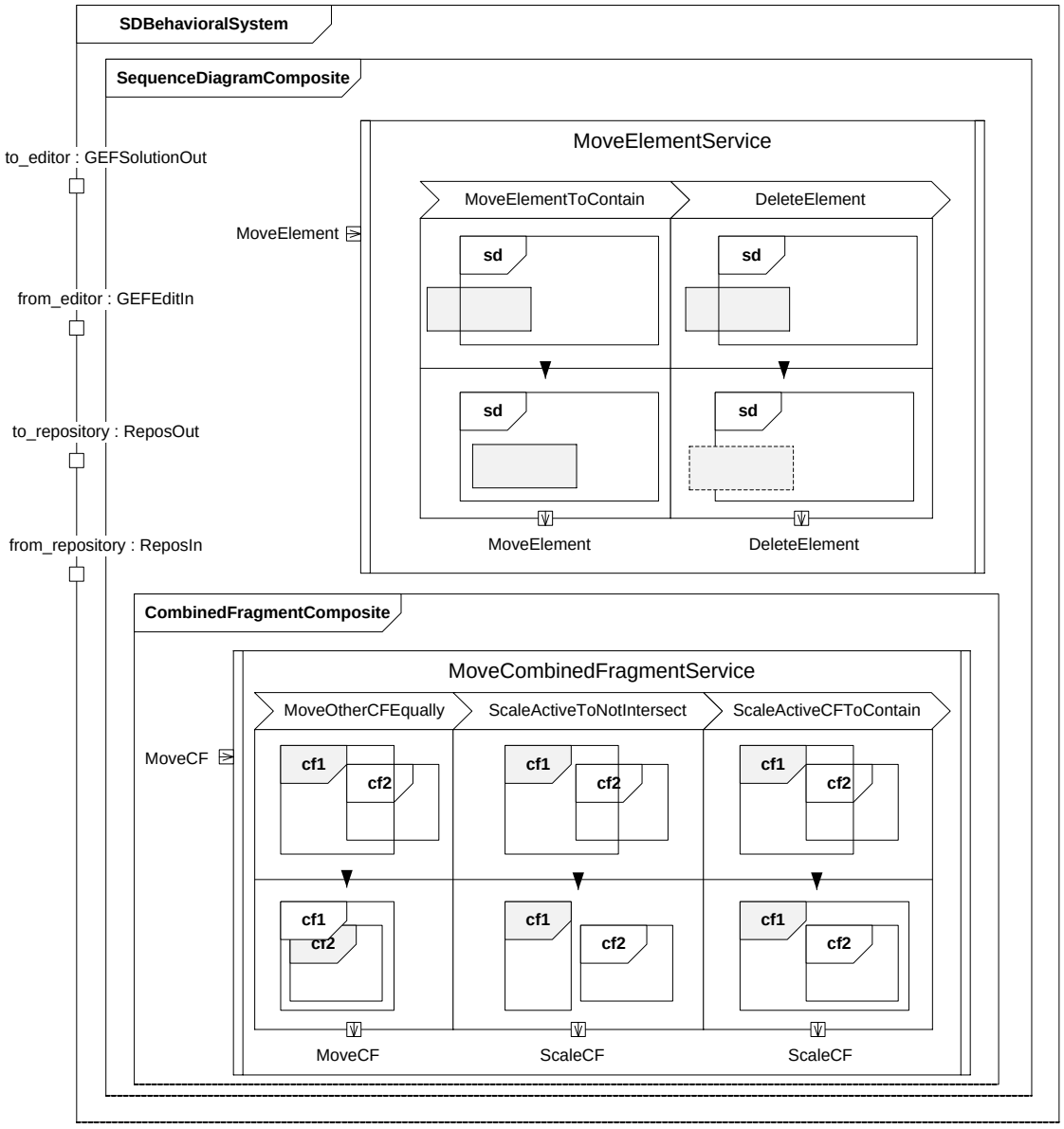

Figure 5-15 Example: Simple Behavioral Definition for a Sequence Diagram Editor with all composite levels visible

The above figure shows a BehavioralDefinition which includes the root-composite defining the mediators required to communicate with the context; here a GEF editor and an generic Repository. Internally in the services we show the different editing behaviors that services may provide, names of behaviors are shown above the patterns. The use of

"box-arrows" that "slide" into the next is not without consideration; although not explored in this thesis we may imagine the search for solutions to be possibly quite time consuming. For a toolsmith the ordering of editing behaviors from left to right can be used to prioritize the order in which the behaviors are pattern matched. Allowing the toolsmith to put the most likely behaviors first.

In this view the behaviors use the DSLs graphical syntax (relaxed, so that we may show actual *illegal* syntax) to render the left-hand-side and right-hand-side patterns. The black arrows in between denote the direction of the transformation rule. At the bottom of every behavior we show what kind of Edit it produces and sends back to the Editor for user-selection. This allows toolsmiths to quickly see whether or not they have defined services capable of supplying editing behaviors problems created by Edits stemming from other editing behaviors (which we have not in the figure above; we lack services capable of handling inconsistencies from ScaleCF and DeleteElement).

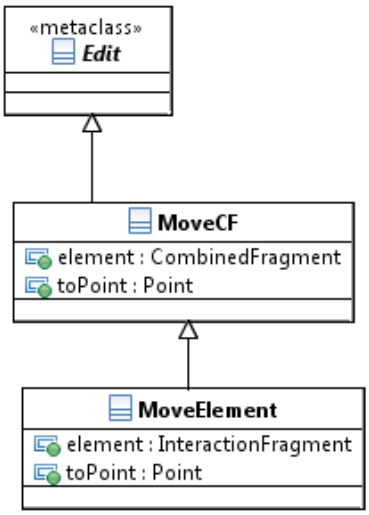

Figure 5-16 Example: Edit hierarchy

Another aspect of Figure 5-15 worth mentioning, but that is not explored in this thesis, are the Services in the DiagramComposite. Figure 5-16 shows that MoveCF is an extension of MoveElement, meaning that we may trigger both services when receiving a MoveCF. In this way we may have fundamental services in the DiagramComposite (e.g. MoveCF may also mean that it has moved outside of the Diagram; move it in again so it is contained). while having specialized services in the internal composites.

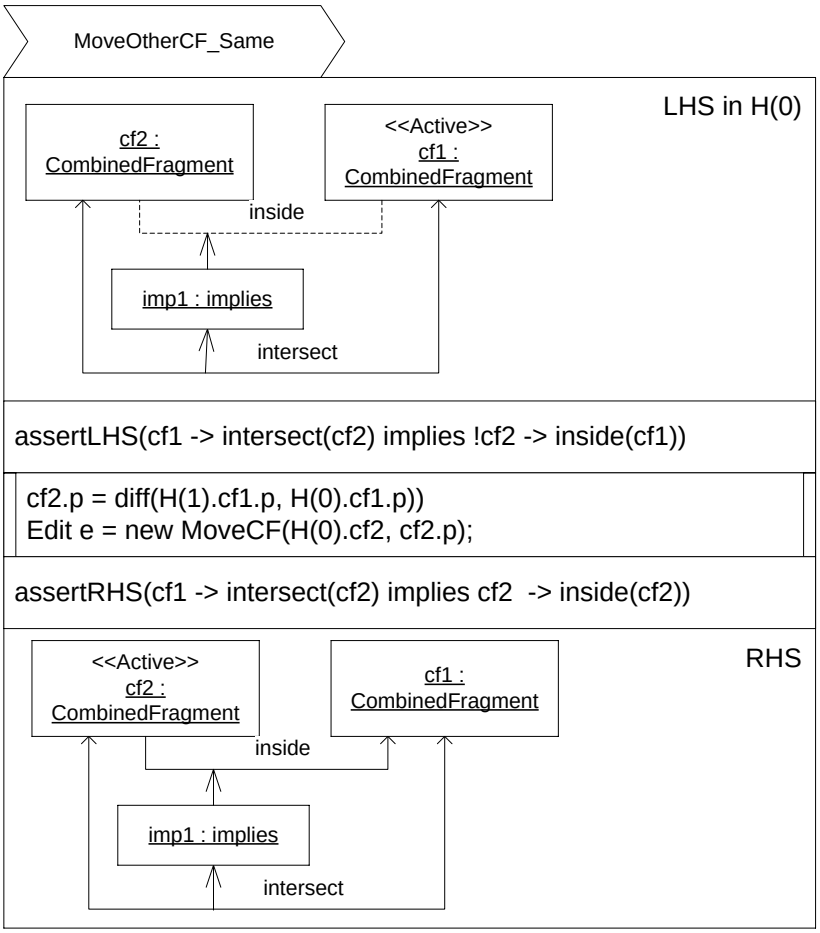

Figure 5-17 Example: Editing Behavior MoveOtherCF_Same

When viewing *just* an editing behavior we expand its complexity. We denote *negative application conditions* with dotted lines in LHS. This editing behavior (EB1 later in the thesis) is capable of solving a problem with intersecting CombinedFragments, where one is not inside the other, by *moving* the other element (cf2) the same amount as (cf1) was moved by the Edit that initiated the service (MoveCF).

The Action-block of the editing behavior, in between the patterns, contain assertions and expressions. Importantly the assertions of the LHS and RHS check all the predicates (the OCL statements defined on GDSQ) "just in case" the structural model is not consistent with the spatial attributes defined within the Symbols (e.g. the relationship intersect exists structurally, but evaluating a intersects predicate using the spatial data shows that they don't). LHS sides always refer to the last snapshot, while RHS always refer to the locally scoped snapshot. Modifications done on elements for calculation in the editing behavior are always done on a new snapshot scoped within the Editing Behavior. Asserting the

validity of the structure in RHS by checking attributes, rather than just structurally matching guarantees that the editing behavior has done what it has intended (e.g. cf2 might not have been inside cf1 before, then moving it the same amount would be an invalid behavior). This is in line with how one may define guards on patterns in ATL [36].

The expression in between the assertions define the calculation needed to find out how far to move cf2. We use the snapshot in the history to deduce how far it has moved as a result of the edit that triggered this service. The second line in the expression defines the new MoveCF edit that will be sent back to the editor. It contains the last snapshot's CombinedFragment cf2, and a copy of an object of type Point which was created in the locally scoped snapshot during calculation in the expression.

RHS patterns are actually not used to manipulate a model at all. In regular a graph transformation setting we would insert instances of the types in the RHS into the model. We however only use it to represent the consistency creating abilities of the *Action,* or more precisely the Edit that is returned to the Editor. It is the Editor that is responsible for executing the Edit, resulting in the DSL model changing. The RHS does however fill a vital purpose for the visualization of the editing behavior, especially when we render it using the DSLs graphical syntax as we have seen in Figure 5-15.

5.5 BDL DESCRIPTION

5.5.1 BEHAVIORALMEDIATOR

A **BehavioralMediator** defines the *interaction boundary* between an executing behavioral definition and its context. We may liken a mediator to a application-programming-interface (API). However as input to a mediator we require messages of type **BehavioralMessage**. E.g. a user interacts with an editor, the editor creates an Edit::BehavioralMessage, and puts this message on a queue in a mediator. The **Executing BehavioralDefinition** continuously checks and retrieves messages from the queue. Similarly we require of an Editor and a Repository that they also check the mediators for new messages and, importantly, that they do this in a separate execution *thread* so as not to lock the editor or repository during editing behavior searches.

5.5.2 BEHAVIORALMESSAGE

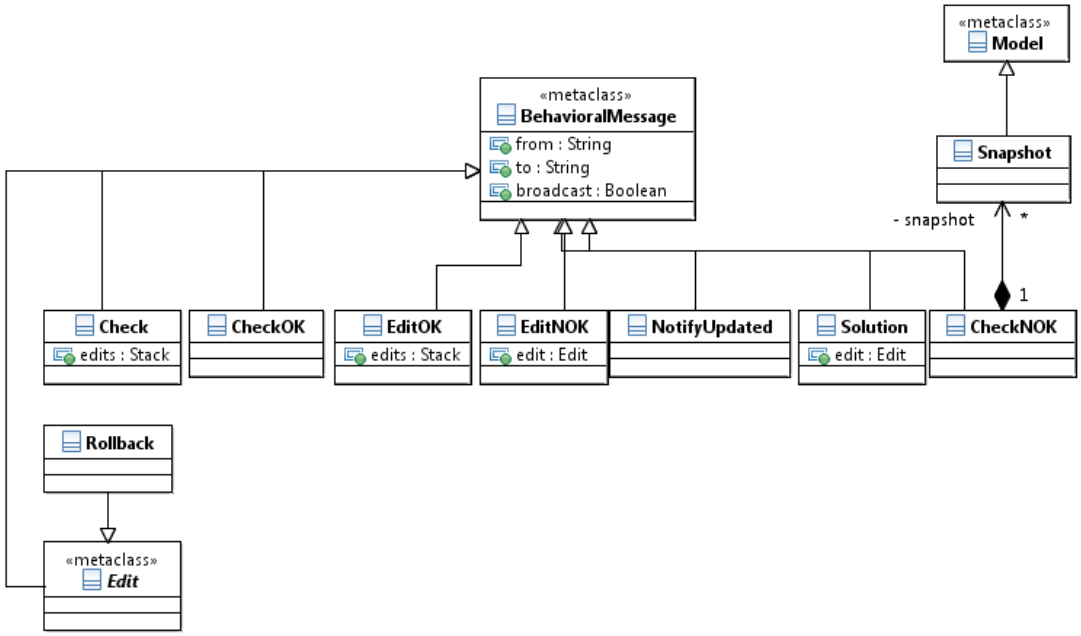

Figure 5-18 BehavioralMessages

BehavioralMessage which contains the necessary information to route and/or broadcast messages between the context and a executing behavioral definition, and internally between components in an executing behavioral definition. The instances of BehaviorlMessage, not marked as <<metaclass>> defined in the diagram above must be

handled by a BehavoralDefinition *editor and repository integrator* in accordance with BD execution semantics. In particular Rollback is the only BDL pre-defined Edit, used to allow Editors to send Rollback Edits as solutions to inconsistencies stemming from a previous Edit (which resets the executing BD to a non-searching state).

We refer to chapter 5.6 on *Execution Semantics* for details of the messages that an executing behavioral definition expects to receive and send with the context, and in which order.

5.5.3 EDIT

Edit is ***abstract*** and needs to be extended by a BD developer. **Edits** are BDL representations of attempted *edits* on the model via an Editor, versus an actually *executed* edit on the model. This distinction is important as BDL views **Edits** as "not yet executed", meaning that we liken them to *queries* stemming from the Editor about whether or not a particular **Edit** will result in a consistent model. We refer to "Figure 5-16 Example: Edit hierarchy" for a concrete example of **Edit** extension.

5.5.4 BEHAVIORALOBJECT

The main abstraction in BDL. Allows us to extend the **JavaFrame ActiveObject** to be able to inherit a runtime. It provides attributes to name services and composites. It additionally contains an identifier required if we need to route messages between elements.

5.5.5 BEHAVIORALCOMPOSITE

A **BehavioralComposite** is a composite structure that encapsulates the editing behavior of a particular element in the DSL. It may contain multiple services, in addition to multiple children **BehavioralComposites**. The **RootBehavioralComposite** is a special type of **BehavioralComposite**. It contains the mediators(ports) needed to communicate with the context, such as an Editor and a Repository.

A BehavioralComposite is a *static* structure, different from BehavioralServices that are dynamic. Composites act as *wrappers* for BehavioralServices and are responsible for creating instances of services when *Edits* are received.

5.5.6 BEHAVIORALSERVICES

BDL employs service-orientation to group editing behaviors together so that we may have multiple services running at once, each capable of searching for editing behaviors in parallel. **BehavioralServices** are created when messages of type *Edit* are received. BehavioralComposites are responsible for actually creating instances of BehavioralServices.

5.5.7 EDITINGBEHAVIOR, PATTERN AND ACTION

An EditingBehavior is a named element and is a BDL equivalent of a graph transformation rule.

Pattern: It has associations to both LHS and RHS patterns. We may generalize the pattern-element in the BDL meta-model to a pattern in some other meta-model (like ATL or QVT), so that we may in the future import the pattern matcher component in the framework. We can also allow for the BDL user definition of the super-type of Pattern if the user has a preference for a concrete pattern matching component.

Action: Actions have assertions and expressions. The assertions help us determine the editing behaviors applicability to the current problem in addition the LHS pattern. Expressions contain any calculations or logical statements needed to create a new Edit and populate its attributes with values.

5.6 BEHAVIORAL DEFINITION: EXECUTION SEMANTICS

In this section we will discuss the execution semantics of a Behavioral Definition. We require several more entities that those that are defined in a Behavioral Definition to be able to search for editing behaviors. We refer to the sequence diagrams in this chapter for these entities.

As we have mentioned earlier, there exists considerable efforts within the research community to formally define model and graph transformations. To create a formal search and pattern matching strategy is beyond the scope of this thesis. What we therefore propose is a simple rule-finding strategy that lacks formal backing, but that *can* find isolated solutions to problems when we do not consider its implications, but leave this up to the user for selection. We do this by assuming the existence of two major components: a

repository capable of producing inconsistent models, snapshots and converting Edits in to transformations, and a pattern matcher capable of matching our LHS patterns to the inconsistent models. The search strategy is dictated by the structure of BDL (composites and services) and our choice of communication paradigm. The concrete pattern matching strategy we leave to an external entity as this also is beyond the scope of this thesis.

5.6.1.1 Our solution finding strategy

The strategy we have chosen is a simple one; any BehavioralService that can accept the Edit, defined via its entryTriggers, is initiated no matter where in the composite-hierarchy they exist. We do this by *broadcasting* the Edit throughout the BehavioralComposite hiearchy. This via a special Coordinator entity in every composite, capable of creating services and routing messages. Another execution specific entity, called Archive, receives the Edit and queries the Repository via a BD-defined BehavioralMediator for repository communication, for any constraint violations pertaining to the a current stack of Edits.

The Archive maintains a stack of Edits so long as the Repository responds with CheckNOK on the stack. Once all Edits on the stack result in a CheckOK from the repository (meaning that the stack of Edits, evaluated atomically in the Repository, do not violate any constraints in the DSL). A CheckNOK consists of a set of models capable of representing the *inconsistency* in a consistent model w.r.t. to constraint-wise relaxed DSL model.

If a CheckOK is received from the Repository we return the current stack of Edits to the Editor for it to execution. In the intermediate time between returning the stack to the Editor, and receiving a NotifyUpdate from the Repository, we *lock* the Archive. This to ensure that we may not try to handle any other Edits in the time being.

If a CheckNOK is received from the Repository we send an EditNOK to the Editor containing the current stack of edits deemed illegal. We assume the editor itself realizes that a *roll-back* is one of the solutions and adds it to the solution view. The CheckNOK contains two model snapshots: one of the repository model with the Edit stack not applied (called H(1)), and one of the repository model with the stack applied (and inconsistent w.r.t. the DSL) called H(0). We define that snapshots are objects local to the Archive, and that any service may reference to these objects but not modify. When services modify snapshots they do so on ones that are locally scoped.

The CheckNOK is broadcasted to all services currently *in session* (those created by the previous Edit). The services search for EditingBehaviors matching the current inconsistencies in the H(0) models, use information from the snapshots to create new Edits, and return this Edit within a Solution message to the Editor. This searching process continues until the user has selected an Edit, upon which we terminate all current searches, and start the entire process all over. Adding the selected Edit to the stack, on top of the previous Edit. *Unless* the Edit is a Rollback::Edit. Then we remove all Edits from the stack and terminate all searches, finally responding to the Editor with EditOK and a empty stack.

5.6.2 GENERIC SOLUTION FINDING INTERACTION

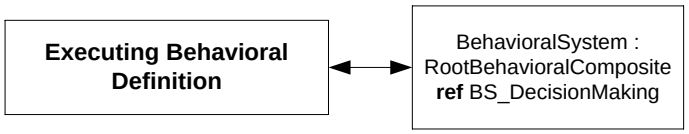

Figure 5-19 Executing Behavioral Definition ↔ BehavioralSystem

The following sequence diagrams and statemachine diagram depict how a generic Behavioral Definition would flow during execution. We view the Executing Behavioral Definition element in our meta-modeling architecture figure (Figure 5-3) as equal to the Lifeline called BehavioralSystem in the context sequence diagram.

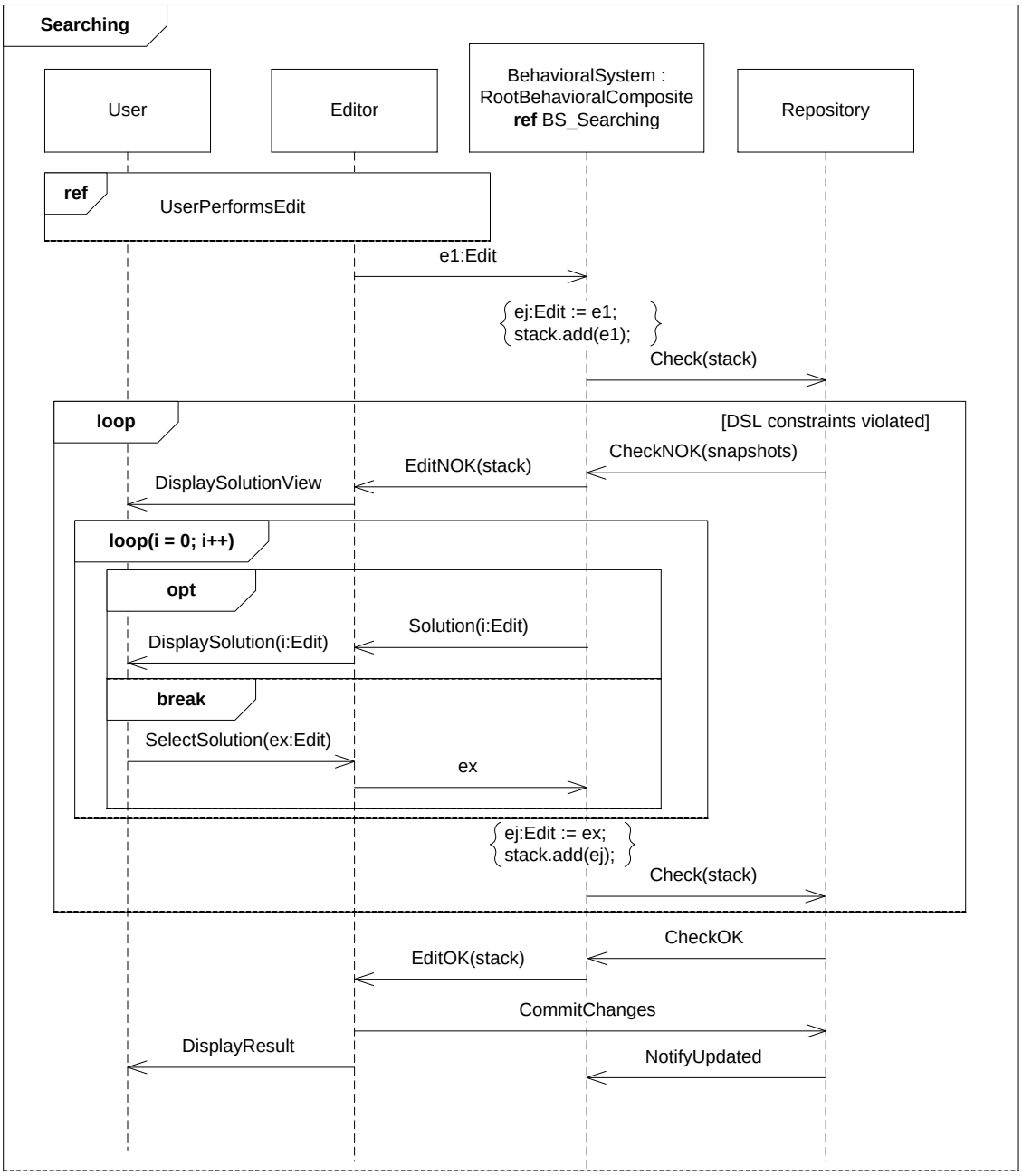

Figure 5-20 Searching : Context

This interaction specifies the generic interactions between a User, Editor, Executing Behavioral Definition, and Repository. We use the abstract message Edit in place of BD-defined Edits.

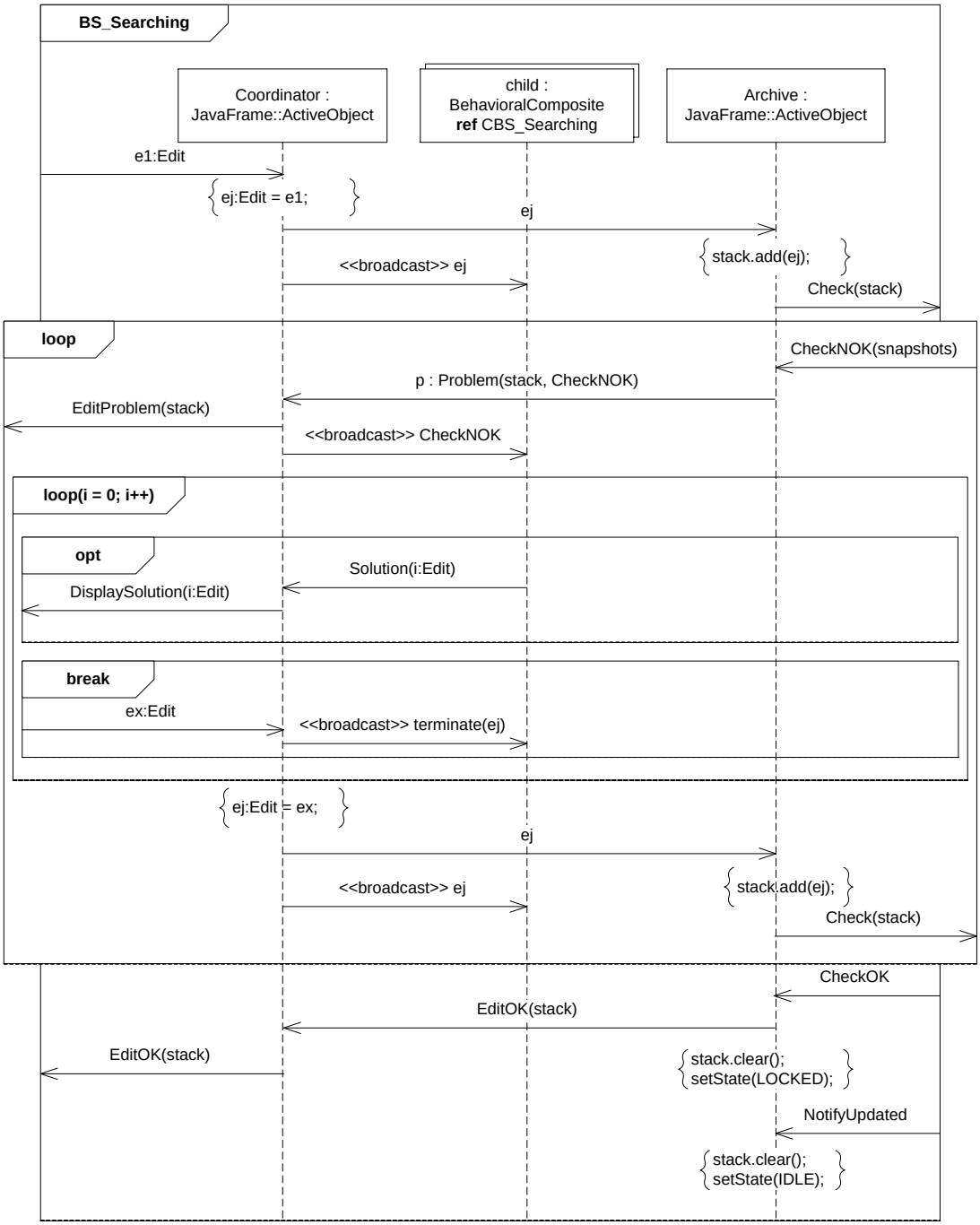

Figure 5-21 Searching: RootBehavioralComposite

This sequence diagram depicts the internal interactions of a generic RootBehavioralComposite. Coordinator and Archive are of type JavaFrame::ActiveObject, so that we may define them using UML in a future prototype. The special case where the solution selected by a user is a Rollback (in place of ex : Edit) would result in the Coordinator *not* broadcasting it, while the Archive would clear its stack, add the Rollback

and reply with an EditOK(stack). Clearing the stack afterwards. Internally in the system we only use one internally defined BehavioralMessage; Problem. Problems acts as wrapper messages to be able to send both the current stack and the snapshots to the Coordinator. The stack is used to tell the Edit exactly *what* stack has not been permitted, while the CheckNOK is extracted and broadcasted to all children of this composite.

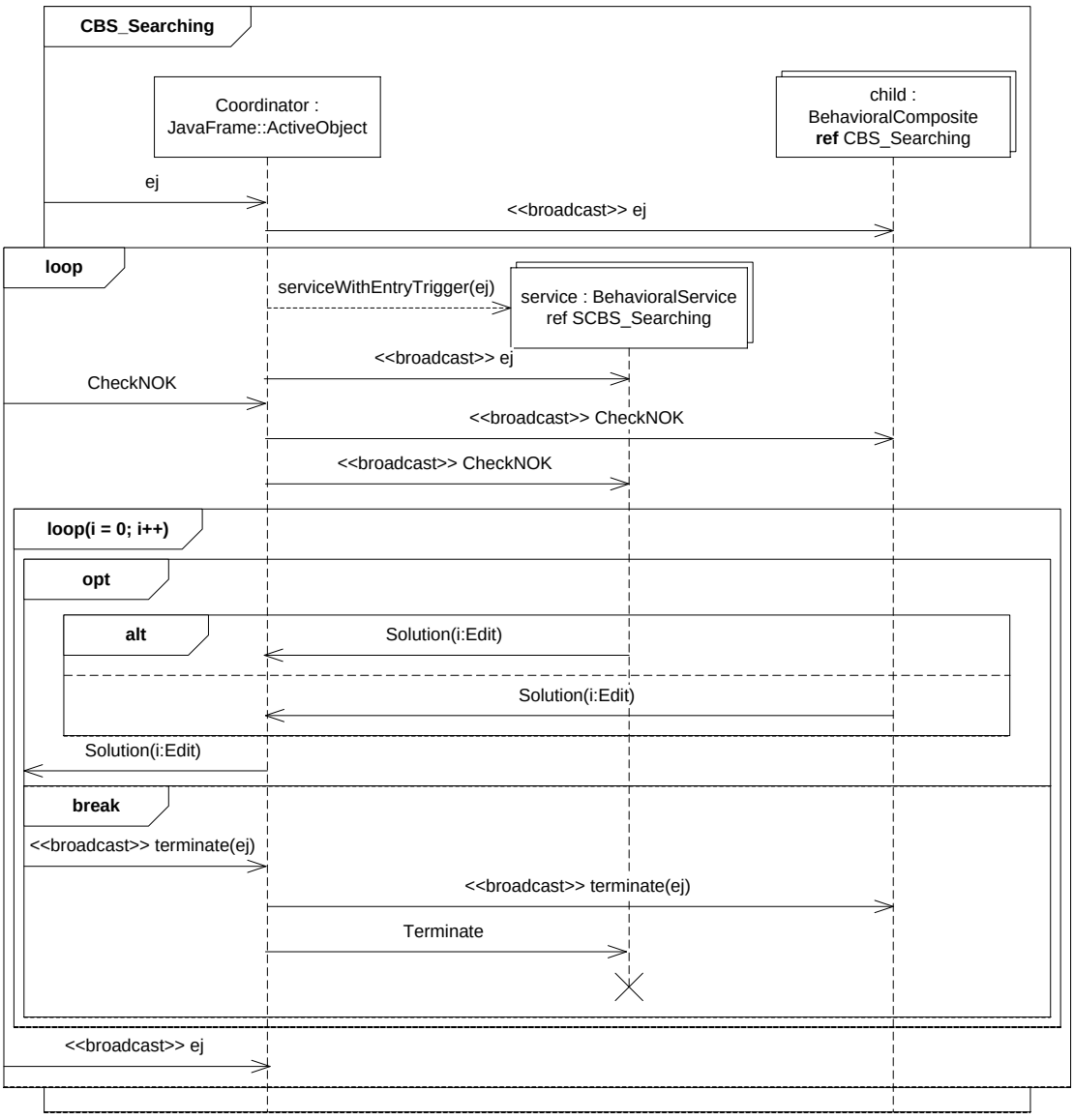

Figure 5-22 Decision Making with Behavioral Definitions : BehavioralComposite

This sequence diagram depicts the internal workings of a generic BehavioralComposite. At some first level in the hierarchy we would have a composite for a *diagram* element, along

with services for common Edits. The Lifeline *child* decomposes into *CBS_Searcing* to show that we repeat this interaction throughout the composite hierarchy.

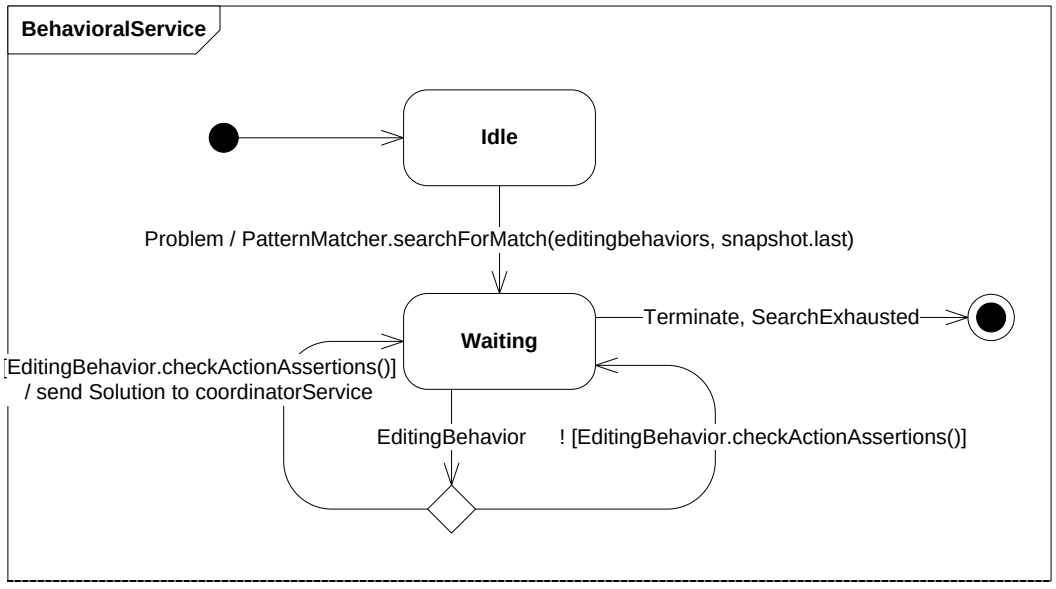

Figure 5-23 Internal workings of a BehavioralService

The above statemachine shows the internal workings of a generic BehavioralService, and how it finds solutions to problems with the Edits. First the service is initialized as the result of an Edit. Then it waits for the CheckNOK-message, containing the model snapshots needed to match editing behaviors against. It then takes its internally defined list of EditingBehavior instances and gives the list, along with the last snapshot to some *matcher* capable of matching a EditingBehavior instance's left-hand-side Pattern with the last (inconsistency representing) snapshot. We assume that this matcher returns a single match at a time by creating and sending EditingBehavior message. We do not depict the Pattern Matcher as its own Lifeline is the previous sequence diagrams, as we imagine the pattern matcher may either by a separate entity, or an internal entity in an executing behavioral definition.

We then run the assertions defined on the Action (similar to an ATL Action block [36]). If the assertions pass we extract the Edit instance created by the Action, and insert it into a Solution-message and send it to the Coordinator. If the user selects this Edit the Editor merely sends the back Edit to the BehavioralSystem in the same manner as when sending a

regular user-created Edit. And we may repeat the entire process once again, in case new constraint violations have been introduced.

5.7 INTEGRATING A BD INTO AN EDITOR AND REPOSITORY

Once a BehavioralDefinition for a DSL has been created we can integrate it into an existing Editor. We imagine this may be done by importing BD *interaction boundary elements* into a model similar to a GMF *Generator Model* capable of using BD-elements, and converting user-initiated edits on the diagram into BDL Edit-type. Interaction boundary elements between Editor and a executing BD consist of: instances of BehavioralMediators. Instances of elements extending Edit, Solution, EditOK and EditNOK. Interaction boundary elements between a Repository and a executing BD consist of: CheckOK, CheckNOK, Check, and NotifyUpdated.

6 EXAMPLE: PROBLEM 1

In this chapter we will formally analyze the possible editing behaviors that an editor may initiate in reaction to a given Edit (E1) that would result in an inconsistent DSL Instance (model R) if given to the repository and being allowed to commit; we call this problem Problem 1. We will see how we may view Edits and Editing Behaviors as transformation rules, and how we may use a Behavioral Definition to handle edits (both user-initiated and edits stemming from the Behavioral Definition itself) that result in inconsistencies, by searching for transformation rules (solutions) that match inconsistency *patterns* in snapshots of the model capable of representing the inconsistencies. We will show how we then may return these solutions back to the editor for presentation to the user. One a solution selected, we treat it in the same way as the user-initiated edit E1. By delegating the responsibility of transformation rule selection to the user, we remove several complexities involving deduction of user intent, and the possibility of cascading and never-ending transformation rule cycles due to rules competing and/or creating more inconsistencies. In the following we will employ a variant of the UML specification for Interactions [4], simplified for the sake of readability. We call it Simple-UML.

6.1 FROM CONSISTENT TO INCONSISTENT

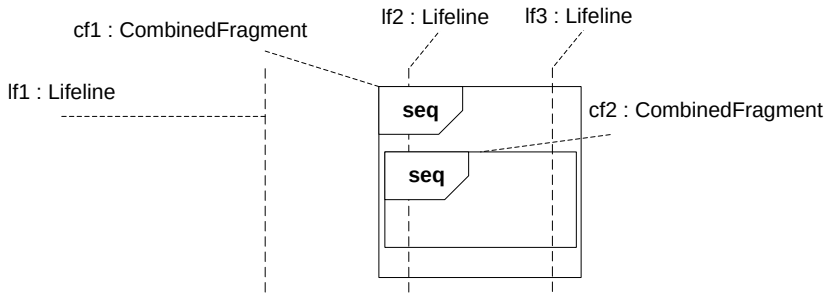

Figure 6-1 Graphical representation of Model R

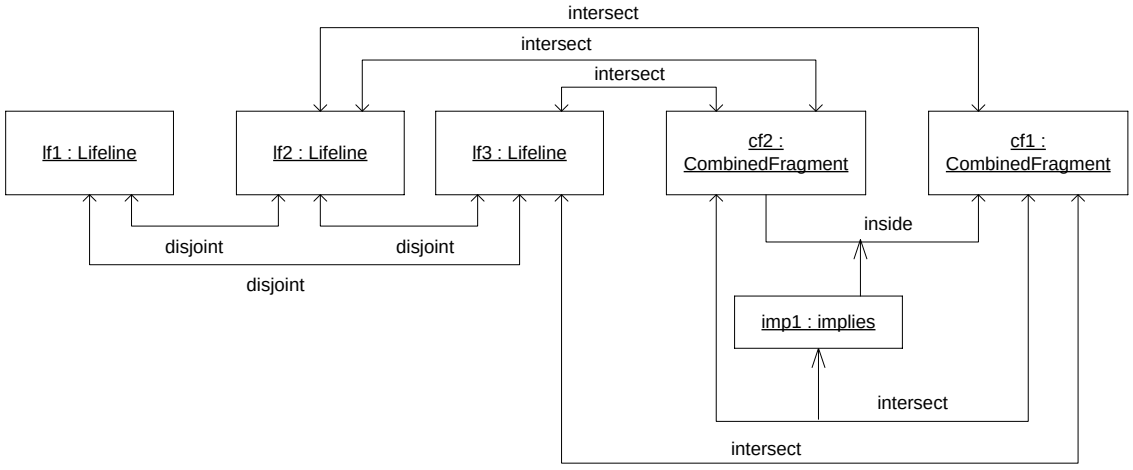

Figure 6-2 Model R conforming to GDSQ

In this chapter we will make heavy use of "Figure 2-11 Graphical representation of Model R" as the initial consistent diagram and its model representation (R) for our analysis of Problem 1. Repeated again here for readability.

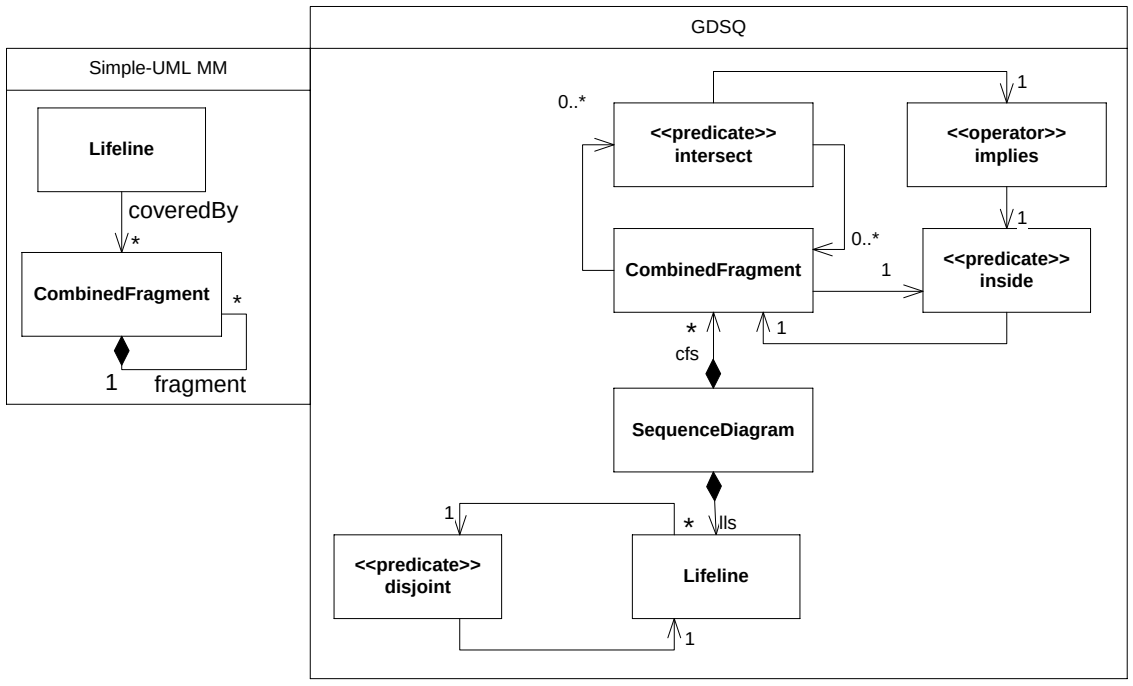

Figure 6-3 Abstract and Concrete Syntax Definitions, mapped

Using the format of a mapping-element given in Figure 3-7, but excluding the tooling element we define the in Figure 6-3 a map of meta-model and graphical definition for Simple-UML; the rules and constraints in the meta-model are given by a MOF-type diagram, while the constraints in the graphical definition is given as a sub-set of GDSQ, excluding the frame/interaction. We will still call the model GDSQ in the following. The constraints placed on the graphical syntax are the same those given in 2.2.1.3, only excluding the constraint dealing with containment of Lifelines within the Interaction. Also for the sake of readability we will only focus on the elements which are instances of GDSQ-element, and not include elements from the meta-model in our problem analysis.

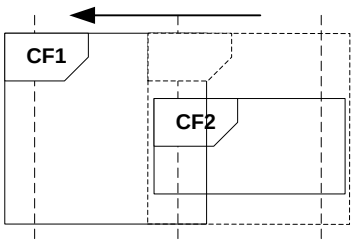

Figure 6-4 Problem 1: State during user-interaction which is illegal to commit to model: intersecting CombinedFragments.

From the consistent diagram the user attempts to accomplish the situation given in Figure 6-4. The user uses a movement-tool to move the outer CombinedFragment (cf1) translated by the Editor in an Edit (E1) which contains the element under manipulation and the values of the desired attribute modifications.

It now intersects a new Lifeline, does not intersect a previously covered Lifeline, and is intersecting with the previous inner CombinedFragment (CF2) which is no longer inside CF1. For the Editor E1 poses multiple questions that needs answering.

1. Who do we need to ask to find out if the Edit is consistency preserving (meaning that it results in a consistent model)?
2. What do we do if we find that the state is illegal (meaning an execution of the Edit would result in an inconsistent model?

To answer the first we need to examine the constraints and rules defined in the syntaxes of the DSL (abstract and concrete, meta-model and graphical definition). There exists no constraint violation from this new state in the meta-model, but there does exist a constraint violation in the graphical definition. This constraint states that the graphical elements of CombinedFragments are not allowed to intersect in a valid Diagram if a CombinedFragment graphical element is not inside another. The violation of the constraint in the graphical definition gives us the following *inconsistent* model G (which is the same as model G in 2.2.1.3):

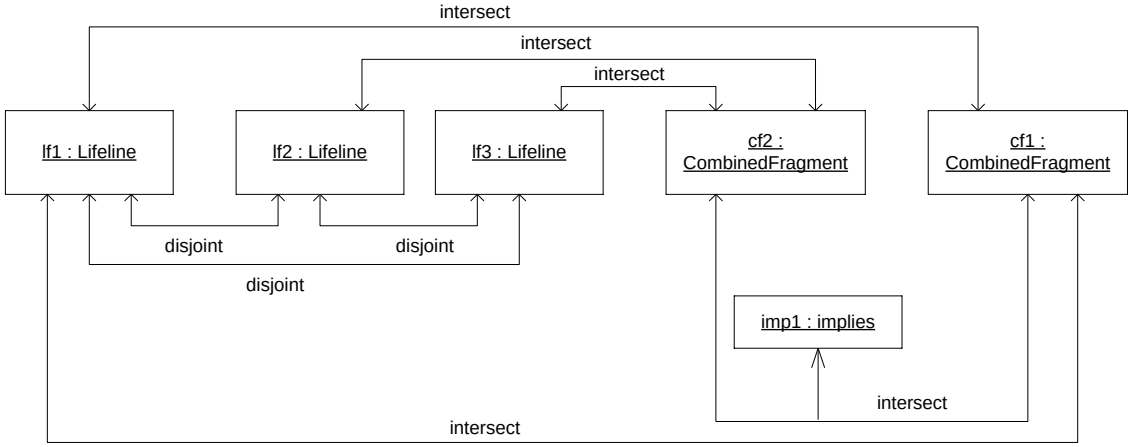

Figure 6-5 Inconsistent model G: dangling implies with missing inside relation

The *inconsistency* is in the model (Figure 6-5) the lack of a relationship between the implies-node (imp1) to the relationship *inside* (since GDSQ defines a 1-multiplicity on the relationship between an Implies instance and an *inside*-instance). We may also deduce this from evaluating the OCL constraints using the values in E1.

6.1.1 PROBLEM SOLVING WITH EDITING BEHAVIORS

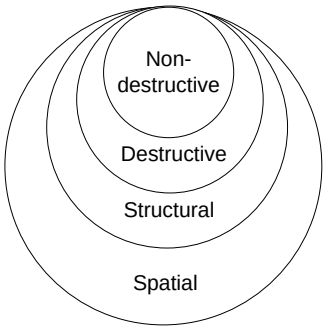

Figure 6-6 Solution space when reasoning about graphical definitions that define spatial attributes

There are several possible solutions to the situation given in Figure 6-4. We call solutions instances of Editing Behaviors; meaning that we may *find* an Editing Behavior capable of responding to the problem of inconsistency, but are in fact not solutions until they have been actually executed upon the model, resulting in an consistent model. We have an infinite number of possible solutions as we work with a model for graphical representation that includes the concept of 2-dimensional spaces. Our conceptual language GDL uses a combination of E-GDL and GIS extended OCL (both languages that use spatial attributes). A solution in this domain needs to include the positions and dimensions of the graphical elements. Since any x and y-coordinate in theory may be between 0 and approaching ∞ the solution-space itself approaches ∞ in size. The solutions below therefore only represent a small sub-set of the spatial solution space. However, if we examine the solution space using exclusively the structure of the models like that given in GDSQ we find that we may drastically reduce it. We may also further reduce the solution-space by defining a requirement on the possible Editing Behaviors:

1. **Requirement**: Editing Behaviors should not be destructive for modifications or creations. They should not delete any symbols from either of the models unless the Edit in question is of a deletion-type.

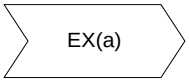

Figure 6-7 An user-initiated Edit on a symbol a

The above graphical notation denotes a user-initiated edit on a symbol *a*. In our case an edit E1 on a symbol cf1, depicted below. We may write this formally using a graph transformation notation as:

$$R\ \{Pre\} \xrightarrow{E(a)} \{Post\}\ G$$

where R is the initial consistent diagram, the function $E(a)$ is as defined above, and G is the resulting diagram, either consistent or inconsistent depending on $E(a)$. We also show pre- and post-conditions for the edit $E(a)$ with $\{Pre\}$ and $\{Post\}$, respectively. This allows us to show what must be true in the diagram before the edit, and what is true in the diagram after the edit. We state that these truth of these conditions are representations of the relationships in the model, and we define them using the GIS-extended OCL syntax. When we express that relationships *do not* exist in the model in pre and post conditions we will use the ! character as a logical not. When we are expressing bi-directional relationships in the notation we only define one uni-directional relationship for conciseness.

The exact contents of an edit E can be either; a structural modification (e.g. a deletion or creation of an symbol) or an attribute modification (e.g. modifying the spatial position of an symbol) or both.

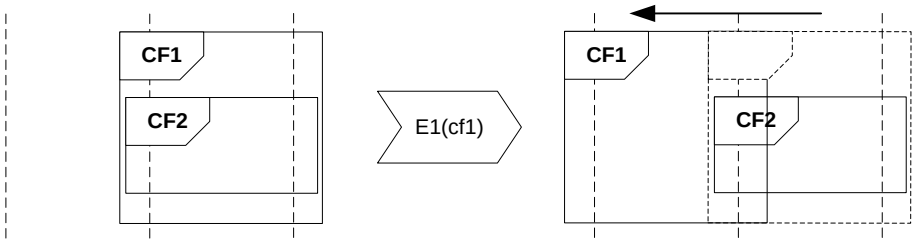

Figure 6-8 Graphical representation of E1

For the edit E1 was may then say that:

$$R\ \{cf1 \rightarrow intersect(cf2), cf2 \rightarrow inside(cf1)\}$$
$$\xrightarrow{E1(cf1)}$$
$$\{cf1 \rightarrow intersect(cf2), !\,cf2 \rightarrow inside(cf1), !\,cf1 \rightarrow inside(cf2)\}\ G$$

Here we exclude some relationships and refer to the models R and G for all the relationships. The interesting one, however, is the disappearance of the relationship with name *inside* (the two negations of the expression that use the predicate *inside* in the post-conditions of E1) creating the *inconsistent* model G with respect to its meta-model GDSQ.

By reducing the solution space and employing our knowledge about the DSL (GDSQ) *and* the information in the current diagram we find the following solutions:

1. Move CF2 the same amount as CF1, moving CF2 **inside** CF1.
2. Positive scale CF1 along its *y*-axis so that CF2 is **inside**.
3. Negative scale CF1 along its *y*-axis so that CF1 no longer **intersects** with CF2.
4. Roll-back to G, undo E1.

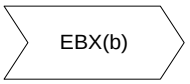

Figure 6-9 An editor-initiated edit performed on a symbol b

This graphical notation denotes an edit resulting from an EditingBehavior on a symbol b. We treat in the same manner as a user-initiated edit and may therefore use the same graphical and textual notation to show how it would affect the inconsistent model G. We define chain of edits as:

$$R\ \{Pre\} \xrightarrow{E(a)} \{Post\}\ G\ \{Pre\} \xrightarrow{EB(b)} \{Post\} G'$$

where G' is the resulting model of the editing behavior $EB(b)$. This model may be either *consistent* or *inconsistent* as we do not attempt to reason about *all* the possible implications of an editing behavior, only its defined effects in its post-condition.

6.1.1.1 Defining the Editing Behaviors

We excluded pre and post-conditions of E1 to shorten the statements and refer to them now as: p^1 and p^2. We will now define the 4 solutions in the previous chapter, named respectively EB1 through 4.

- EB1:

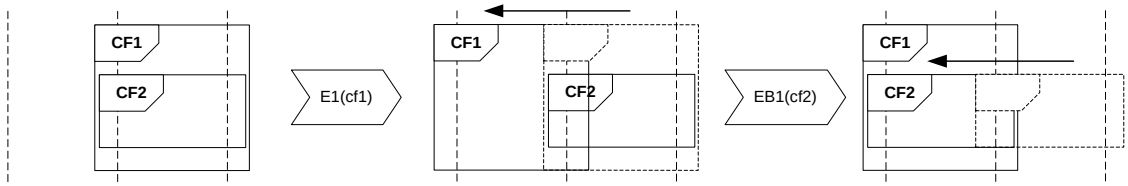

Figure 6-10 EB1 : graphical representaiton

$$R\ p^1 \xrightarrow{E1(CF1)} p^2\ G\ \{cf1 \rightarrow intersect(cf2), !\,cf2 \rightarrow inside(cf1), !\,cf1 \rightarrow inside(cf2)\} \xrightarrow{EB1(\,CF2)} \{cf1 \rightarrow intersects(cf2), cf2 \rightarrow inside(cf1), !\,cf1 \rightarrow inside(cf2)\}\ G'$$

- EB2:

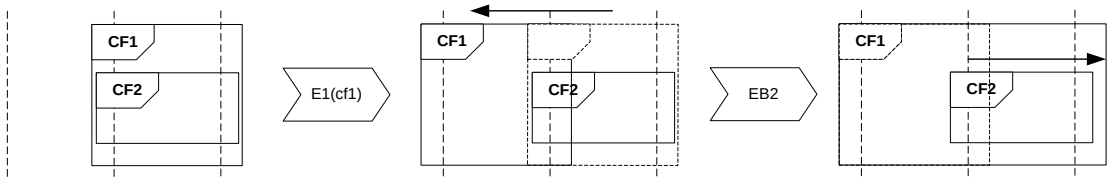

Figure 6-11 EB2 : graphical representation

$$R\ p^1 \xrightarrow{E1(CF1)} p^2\ G\ \{cf1 \rightarrow intersect(cf2), !\,cf2 \rightarrow inside(cf1), !\,cf1 \rightarrow inside(cf2)\} \xrightarrow{EB2(\,CF1)} \{cf1 \rightarrow intersects(cf2), cf2 \rightarrow inside(cf1), !\,cf1 \rightarrow inside(cf2)\}\ G'$$

- EB3:

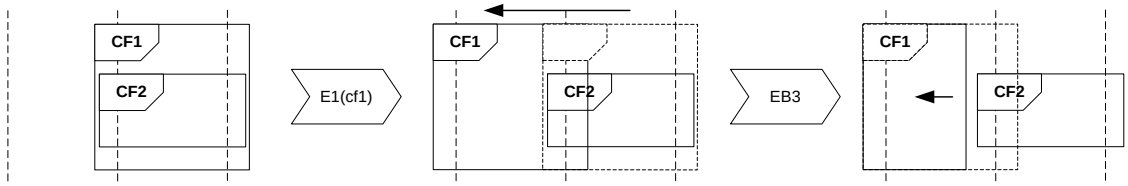

Figure 6-12 EB3: graphical representation

In this solution we do not need to represent the no longer existing relationships of type *inside* as the entire implication is removed by removing the *intersects* relationship.

$$R\ p^1 \xrightarrow{E1(CF1)} p^2\ G\ \{cf1 \rightarrow intersect(cf2), !\,cf2 \rightarrow inside(cf1), !\,cf1 \rightarrow inside(cf2)\} \xrightarrow{EB3(\,CF1)} \{!\,cf1 \rightarrow intersects(cf2)\}\ G'$$

- EB4:

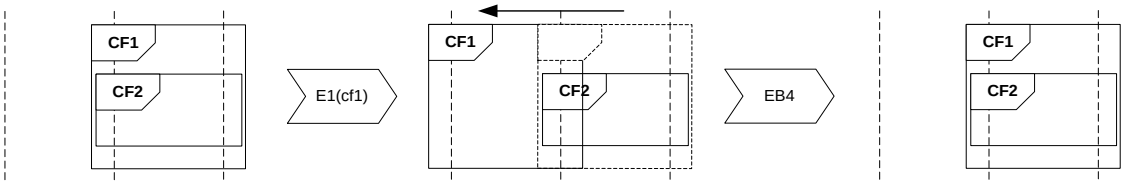

Figure 6-13 EB4: graphical representation

In this solution we roll-back to the initial model R. EB4 therefore restores E1's pre-conditions p^1.

$$R\ p^1 \xrightarrow{E1(CF1)} p^2\ G\ \{cf1 \rightarrow intersect(cf2), !\,cf2 \rightarrow inside(cf1), !\,cf1 \rightarrow inside(cf2)\} \xrightarrow{EB4(\,CF1)} \{p^1\}\ G'$$

Of the behaviors above it is reasonable to assume that EB1 is the most likely to be implemented by toolsmiths; it is common that symbols spatially inside another are moved with the same Δx and Δy as its containing symbol, as is the case in [5, 26]. However we may also envision a situation in which this should not be implemented; if the symbol was *not* inside another symbol in the initial model R, but are intersecting in the model G. Then moving CF2 with the same deltas as CF1 would not be very reasonable.

We may also imagine a situation where we do not want to move CF2; for instance by a *locked* attribute or if the user anticipates that moving it will result in further inconsistencies in the model. Then the behavior to execute would be EB2 or EB3, where we scale or shrink the Active-element and don't alter CF2. We may say that EB3 is perhaps the least plausible intention of the user if we employ our domain knowledge of UML and sequence diagrams, but nonetheless results in a consistent diagram state, as does the revert behavior EB4.

6.1.1.2 Possible Destructive Behaviors

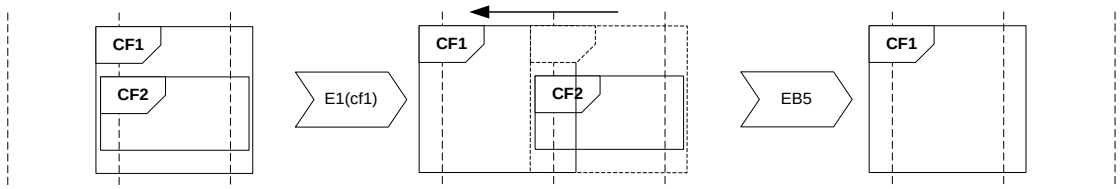

Figure 6-14 EB5 : Destructive behavior

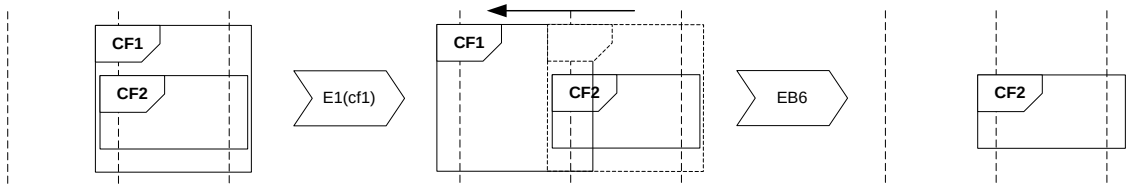

Figure 6-15 EB6 : Destructive behavior

The above 2 behaviors EB5 and EB6 we deem, although possible solutions, too destructive for implementation as they both delete one of the elements in response to an edit that was not a deletion-type edit.

6.1.2 EDITING BEHAVIORS AS MODEL TRANSFORMATIONS

The previous chapter helped us visualize and formalize the process of finding editing behaviors for the given problem 1 resulting from the edit E1, and that edits and editing behaviors resemble graph-transformation rules; We have left-hand-sides (LHS) of graphs and pre-conditions (graph-patterns), right-hand-sides (RHS) of graphs and post-conditions (graph patterns), with the EB's and E's between as graph transformations. We will now show how we can use concepts from graph and model transformations to find solutions to *inconsistent* models resulting from edits.

6.1.2.1 How Behavioral Definitions manage Inconistencies

A BD's response to inconsistencies is non-monolithic and step-wise, allowing the user to determine the most appropriate transformation at every step. BDL does not differentiate between an inconsistency stemming from an Edit performed by a user, or an inconsistency

created by a matched transformation rule that a BD has presented to the editor as a solution. This allows us to view Edits and Edits from Editing Behaviors in the same way and with the same formalism, as we have done in 6.1.1.1. Both Edits and Editing Behaviors may create inconsistent models, in which case we again try to match the new inconsistencies with transformation rules.

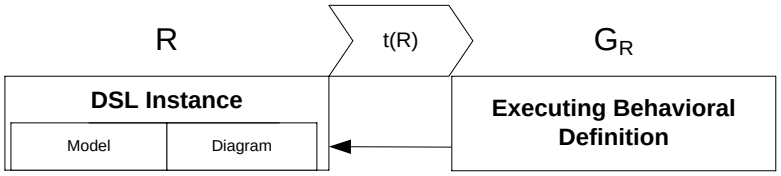

Figure 6-16 Building G

The Executing Behavioral Definition queries the repository of the DSL instance which finds that no constraints are currently violated. Here a transformation *t* is an initial transformation rule that transforms the model R (Figure 2-10 Model R conforming to GDSQ) into some model G_R that is an instance of a meta-model we call ***I***, capable of representing inconsistencies. As *t* is a non-modifying transformation (just transforming from a model conforming to GDSQ to a model conforming to I), and I is the same model as GDSQ with relaxed constraints then R and G_R are consistent. Formally:

$$R, GDSQ, I, G_R \in Model : \left(cc_{GDSQ}(GDSQ, R) \wedge t(R) = G_R\right)$$
$$\Rightarrow (cc_I(I, G_R) \wedge cc_t(R, G_R) \wedge cc(G_R, GDSQ)$$

where *t* is a non-structurally and attribute modifying transformation. We may say that the transformation is endogeneous.

$$R = G_R$$

Next the Executing Behavioral Definition receives an Edit ***E1***. Any service capable of handling the Edit is initiated. The Executing Behavioral Definition queries the repository asking for a model which is a transformation of ***R*** equal to what is defined in E1. We call the transformation ***tE1***. It structurally transforms and modifies the relevant attributes according to the edit E1 and produces the model ***G***.

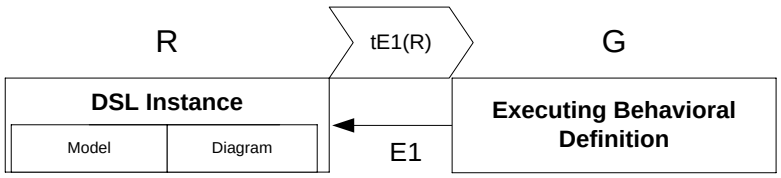

Figure 6-17 Building G

Here G does not conform to GDSQ, as it is a model that breaks a consistency in GDSQ. It does however conform to the relaxed meta-model I. Formally:

$$R, GDSQ, I, G \in Model : \left(cc_{GDSQ}(GDSQ, R) \wedge tE1(R) = G\right)$$
$$\Rightarrow (cc_I(I, G) \wedge \neg \mathrm{cc}_{\mathrm{GDSQ}}(\mathrm{GDSQ}, \mathrm{G}))$$

The following figure depicts the model G (same as in Figure 2-12) represented again here for readability.

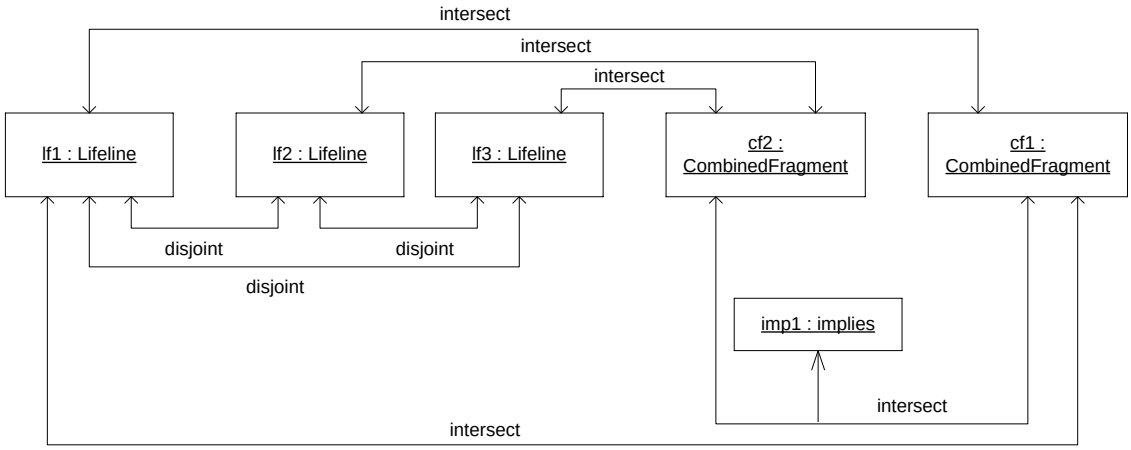

Figure 6-18 The resulting G model from the transformation tE1(R).

We see here the necessity of transforming into a separate model **G** that conforms to a special meta-model ***I*** capable of representing inconsistencies in GDSQ. GDSQ denies the existence of the above model; 2 CombinedFragments may not intersect without one being inside the other.

Our task (with a Behavioral Definition) is therefore to find a set of transformations *capable* of creating a model R' which *is* consistent with GDSQ while at the same time trying to

maintain the attribute modifications (for E1) and structural modifications (for some other Edit, e.g. a deletion) of the transformation tE1 (edit E1). Formally we may say:

$$GDSQ, R, R', G, G', I \in Model, \forall\, tE \in Editor, \exists \{tEB^1, \dots, tEB^{n-1}\} \in BehavioralDefinition : \left(cc_{GDSQ}(GDSQ, R) \wedge c_I(I, G) \wedge tE(R) = G \wedge tEB^1(G) = G^1 \wedge \dots \wedge tEB^{n-1}(G^{n-1}) = G^n = R' \Rightarrow cc_{GDSQ}(GDSQ, R')\right)$$

Meaning that for any edit E in the Editor there exists a ordered set consisting of editing behavior transformations $\{tEB^1, \dots, tEB^{n-1}\}$ existing in a Behavioral Definition, capable of rendering the target-model G^n of the transformation tEB^{n-1} a consistent model w.r.t. GDSQ. This definition is however somewhat problematic.

6.1.2.2 Cascading transformation rules and coordination

We cannot guarantee that we will not end up with an infinite long set of transformations based on editing behaviors. Never capable of producing a model that conforms to GDSQ. If we have multiple conditions we may also have rules that, while fulfilling one or more conditions, make others inconsistent. Thus if we attempt to automatically infer what transformations to execute we may find ourselves in non-terminating transformation process. One possible solution is to have more generalized rules that solves multiple inconsistencies at once, ideally solving any problems in the model by a single transformation. Although it might be theoretically possible to great such a "super"-consistency-creating-transformation, it would certainly be a cumbersome process and would affect both modularization and changeability [43]. We however do not need to take this into consideration as we never try to infer the "correct" behavior to initiate, but merely give the user a list of possible solutions that *might* result in consistency, but might also result in inconsistencies. This is similar to how syntax direct textual editors work; they may give suggestions of how to fix a syntactic error in the code, but never guarantee that any suggestion will not introduce more errors.

6.1.3 STRUCTURAL PATTERN MATCHING AGAINST RULES

A common matching strategy [35] for rule application in graph transformations is looking for a match $m : LHS \rightarrow G$ of the left-hand side into a host graph. A match is then a total mapping, i.e. were each object of LHS is embedded in the graph G. If a variable occurs several times in the rule's LHS they must be matched with the same value. There may be multiple matches of the rule's LHS into the host graph, or there may be no matches at all. In the last case the rule is not applicable. A rule is applicable if all its *negative application conditions* (NAC) and other *positive application conditions* (PAC) are met [32].

The second step entails taking the matching pattern found for the rule's LHS and take it out of the host graph and replace it with the appropriate matching pattern for the rule's RHS. Since the match is a total mapping, any object o of the rule's LHS has a "proper image" object $m(0)$ in G. If o has an image $r(0)$ in the rule's RHS, its corresponding object $m(0)$ in the graph G is *persevered* during the transformation. Otherwise it is *removed*. Objects in RHS that are not in the image of an object in LHS are *created* during the transformation. Objects of the graph G that are not covered by the match are not affected by the rule application at all.

The second step is in our approach irrelevant. We do not use the RHS for any actual transformations, and only for rule visualization. We will explain this in more detail in the next sub-chapter.

We will denote NACs with dotted lines, meaning that the element does not exists. We also use dotted lines to show the removal of elements by the rule in RHS.

6.1.4 "HEDGING OUR BETS": STEREOTYPES IN PATTERNS & ASSERTING ATTRIBUTE MODIFICATIONS IN ACTIONS

To find a transformation that results in a model with only valid relationships based solely on structural pattern matching and running a structural transformation is one thing, defining exactly what the transformation does to the *attributes* within the model is another ; the relationships in a graphical definition instance are not possible to create at will just by structural transformation, but depend on (as in our Problem1) the *spatial* data in our Symbols.

Our strategy for combating this complexity is relatively straightforward: any transformation rule also has some *action* that modifies the relevant spatial attributes where the symbols on which to modify are defined by matching the LHS pattern to the model. We ensure that the modifications fit with the transformation with respect to the RHS by asserting the truth of the predicates of the structural relationships.

Another important aspect is that we require of all transformations that they mark Symbols that have been modified with the stereotype <<*Active*>>. This so that we can direct our Editing Behaviors onto *non-active* elements so as to differentiate between behaviors that solve inconsistencies resulting from edits by altering the *edited* elements, and solutions that solve inconsistencies by altering *non-edited* elements. This allows us to define rules that can only be pattern matched against *edited* elements instead of pattern matching against the entire model which can be a time-consuming process. We ignore the possibility in this thesis of inconsistencies being created by between two non-active elements while the active element and all its relations are consistent. The default editing behavior is a *roll-back* that would fix such a inconsistency.

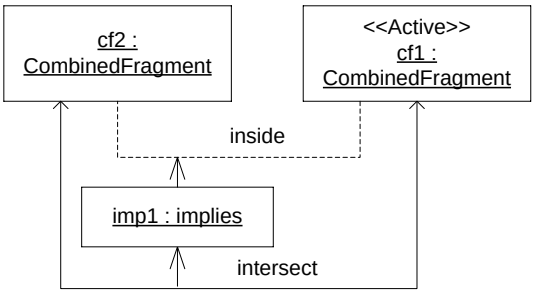

Figure 6-19 LHS : EB1-4

For our Editing Behaviors 1-4 we have LHS pattern Figure 6-19. Here the elements do not refer to actual symbols in the model, but are named and *typed* elements that we require instances of to exist in the model in this exact pattern. In the above figure we see that it is impossible to infer the *direction* of the uni-directional inside relationship (that was or never existed). This is why we will use assertions in combination with model snapshots later.

We have 6 possible RHS patterns that structurally restore consistency to a target-model with respect to the meta-model GDSQ given the source-model G.

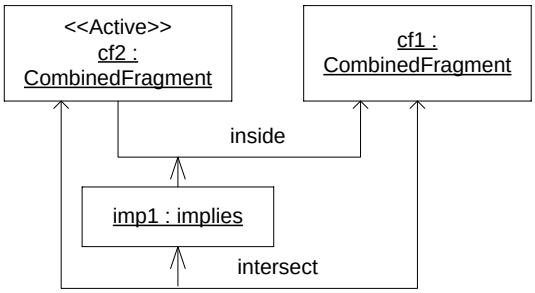

Figure 6-20 RHS 1: Connect implies to a inside relationship from cf2 to cf1 by manipulating cf2

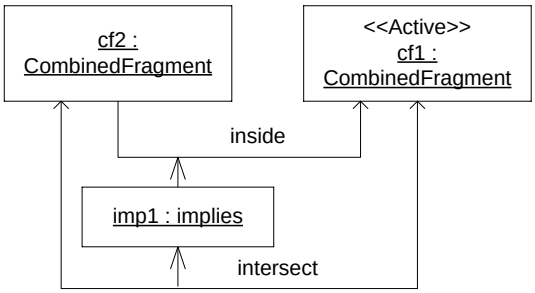

Figure 6-21 RHS 2: Connect implies to a inside relationship from cf2 to cf1 by manipulating cf1

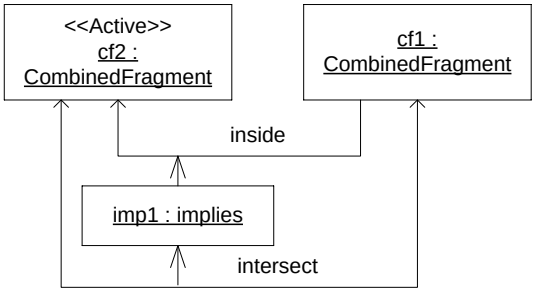

Figure 6-22 RHS 3 : Connect implies to a inside relationship from cf1 to cf2 by manipulating cf2

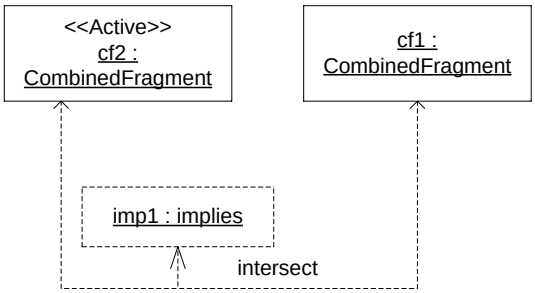

Figure 6-23 RHS 5 : Delete the intersect relationship (and therefore implies) between cf1 and cf2 by manipulating cf1

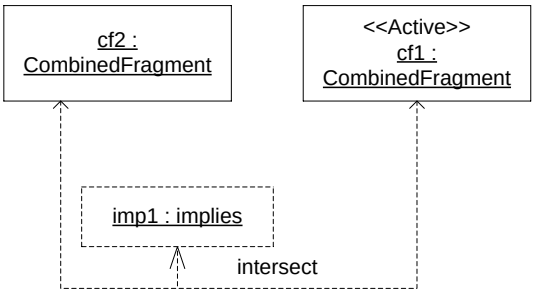

Figure 6-24 RHS 6 : Delete the intersect relationship (and therefore implies) between cf1 and cf2 by manipulation cf1

Of the 6 possible RHS patterns we only will define Editing Behaviors for those that render a non-active (not stereotyped with <<Active>>) CombinedFragments on the LHS, *inside* an LHS *active* CombinedFragment; i.e. putting the non-active element inside the active.

6.1.5 DEFINING EDITING BEHAVIORS WITH TRANSFORMATION RULES AND ACTIONS

The Editing Behavior EB1 seeks to restore the inside relationship by manipulation CF2. This transformation has as NAC that the relationship *inside* does not exist in the model (dotted-line). It has as PAC (*positive applications conditions)* that 2 CombinedFragments exist in the model and are connected by *intersect* with a *dangling-implication*.

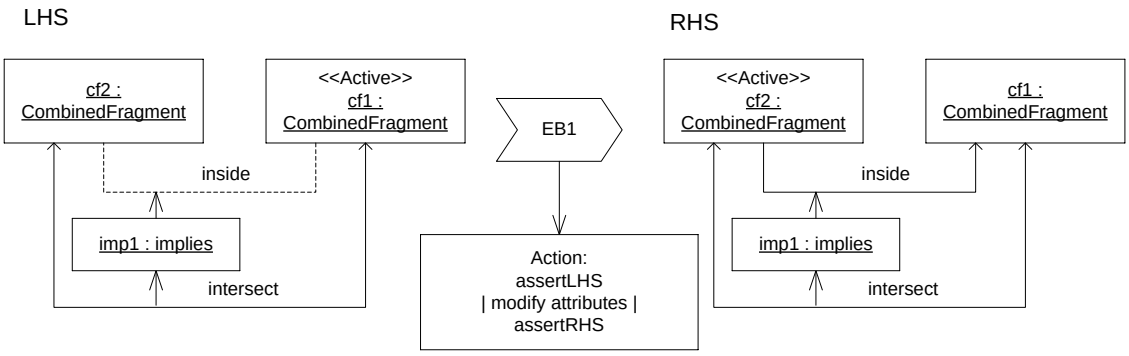

Figure 6-25 EB1 as a model transformation with Action on attributes

We further restrict the applicability of EB1 by defining that it may only work on the element that is not *Active* (making it Active after the transformation). This ensures that the Action will not attempt to modify CF1, which is not the intention of EB1. This also in part solves the problem of *irreflexive versus reflexive* relationships, and determining what element is

supposed to be inside another. The Active stereotype allows us to express that it is the target-model *Active* element that we intend to put inside another, and not the source. If this is the actually intention of the user's initial edit E1 is of little importance since we do not automatically apply the action.

Another benefit of this structure is that we answer a question posed in 6.1.1.1: what if CF2 was not *inside* CF1 in the model pre edit E1? If CF2 was not a child of CF1 in the initial consistent diagram we would not be able to deduce this from the relationships in the current model using patterns. We can, however, deduce this via assertions in the Action by referencing **snapshots** of the previous model (**R**). If the Action attempts to move CF2 with the same Δx,y as CF1, and CF2 was not inside previously and assertion of the RHS relationships using actual attributes and predicates and not only structural relationships will fail, letting us rule out EB1 as a possible solution.

```
Action:
assertLHS(cf1 -> intersect(cf2) implies !cf2 -> inside(cf1))

cf2.p = diff(H(1).cf1.p, H(0).cf1.p))
Edit e = new MoveCF(H(0).cf2, cf2.p);

assertRHS(cf1 -> intersect(cf2) implies cf2  -> inside(cf2))
```

Figure 6-26 EB1 Action

The set of snapshots is **H** (H for history). The most current snapshot is always referred to as **H(0)**. **H(1)** refers to the snapshot before H(0). In our case the model *pre* E1 is H(1) relative to EB1 during evaluation. The model *post* E1 is H(0) relative to EB1 during evaluation. Any attribute modifications always take place on a new snapshot local to the EditingBehavior, not needing a explicit reference to in the action. To shorten the statements we say that: a LHS assertion always refers to the H(0) snapshot, while a RHS always refers to the new local snapshot.We use a hybrid of GIS-extended OCL and Java notation to define the assertions and actions. The set of snapshots in H are also useful in that we may deduce the Δx,y by merely running a *diff* operation that returns a new point based on the two distinct positions of CF1 in H(0) and H(1). We use a *Point p* instead of our Symbol attributes lx and ly for conciseness.

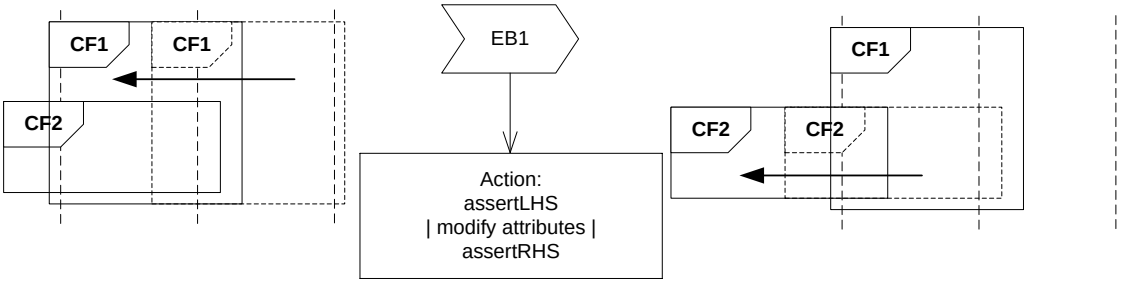

Figure 6-27 Assertion is ok for LHS, fails for RHS

The figure above shows the a valid LHS, but where CF2 was pre edit not inside CF1. We may say that CF1 has in fact move *onto* CF2. Since EB1 LHS matches the current model (intersecting CombinedFragments) we may try to use EB1 to move CF2 the same amount. This does not result in the attributes being modified in such a way so that the assertion of RHS passes (intersecting and an active CF2 inside CF1) and EB1 is marked as not applicable. We leave it up to the reader to imagine if EB2 is applicable in this situation.

6.1.6 DEFINING THE MOVECOMBINEDFRAGMENTSERVICE

First we define a MoveCombinedFragmentService which will contain the solutions. We also define what Edit(s) will trigger it and make it participate in any search for solutions to problems.

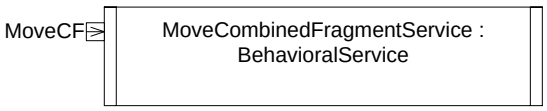

Figure 6-28 Basic MoveCombinedFragmentService

Then we define the solutions to possible problems that a MoveCF edit may create. Internally MoveOtherCF_Same also refers to a concrete action language for creating a message (Edit e) without actually sending it, just storing it within the Action's scope (we will use Java). It is up to the service to actually extract this created Edit and send it to the editor.

Exactly how *smart* an EditingBehavior is depends in addition to the patterns it is capable of matching, on the complexity of the expressions given in the Action. For EB1 we merely run a *diff*-operation on two points from the history of snapshots; finding out how much the Active CombinedFragment has moved and moving the non-active the same amount.

EB1 : MoveOtherCF_Same is defined previously in this thesis: "Figure 5-17 Example: Editing Behavior MoveOtherCF_Same".

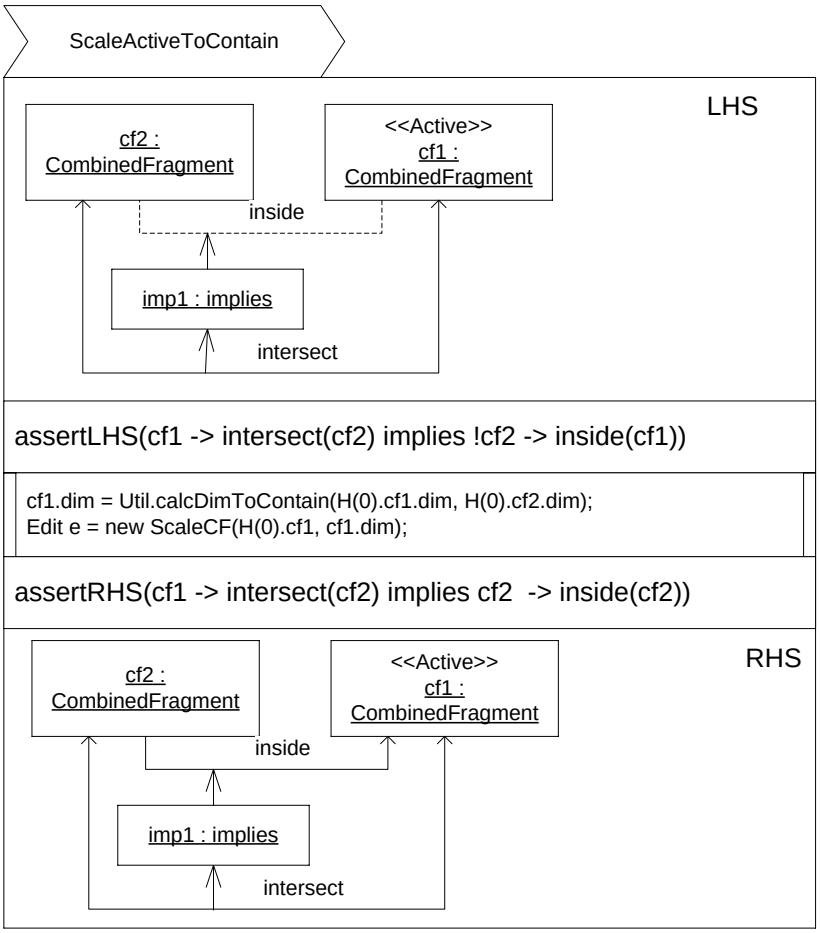

Figure 6-29 Solution 2 (EB2): ScaleActiveCFToContain

EB2 : ScaleActiveToContain Here the editing behavior acts a bit smarter: We use a static Util class capable of calculating the needed dimension for one dimension to contain another. In our case expanding cf1 enough so that cf2 is inside.

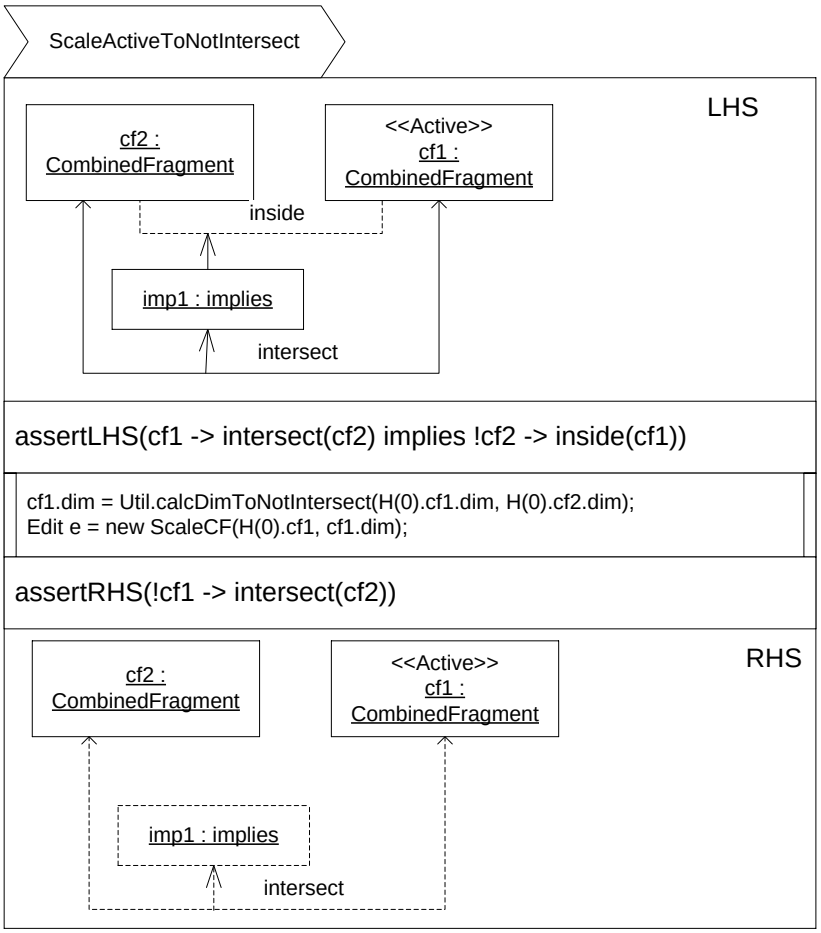

Figure 6-30 Solution 3 (EB3) : Shrink Active to Not Intersect

EB3 : ScaleActiveToNotIntersect has a different RHS; it removes the intersects relationship between cf1 and cf2, and therefore also the implies element. As inside was a NAC in the LHS and did not exist, is does not exist at all in the RHS.

Now we insert the Solutions into the MoveCombinedFragmentService : BehavioralService, which itself should be within a CombinedFragmentComposite : BehavioralComposite.

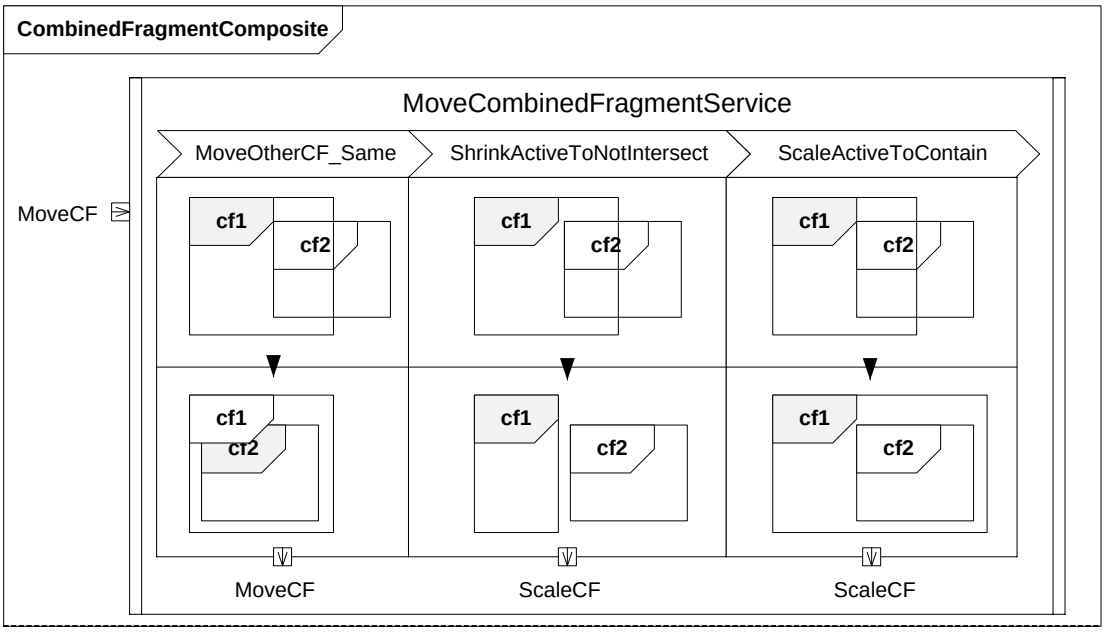

Figure 6-31 CombinedFragmentComposite with MoveCFService

In this view have hidden the details of the Solutions, and based on the LHS and RHS-sides drawn visual representations of the patterns the solutions match using elements from the DSL Graphical Definition itself. Highlighted in gray are the <<Active>> elements.

7 BEHAVIORAL SYSTEM PROTOTYPE

The prototype confirms two claims in this thesis; that it is possible to define mediators to communicate with a GEF-based editor at least in part asynchronously, and that it is possible to generate messages and mediators capable of communicating notifications stemming from an implementation of the Observable pattern in EMF-repositories to the prototype. The prototype focuses on the segment of our framework that deals with editor integration, and does not focus on the validity of using graph transformations and rules to find possible solutions to inconsistency creating edits. However the prototype was essential for experimenting with ideas of how to actually create an editor that contains more precise and formally defined editing behaviors, than state-of-the-art-editor frameworks such as those that use a programmatic approach. We did by attempting to use UML to describe editing behaviors. These experiments with the prototype are what eventually led us to the findings presented in this thesis, and to many concrete paths of examination for future prototypes within the same field.

7.1 INTERACTION BOUNDARY

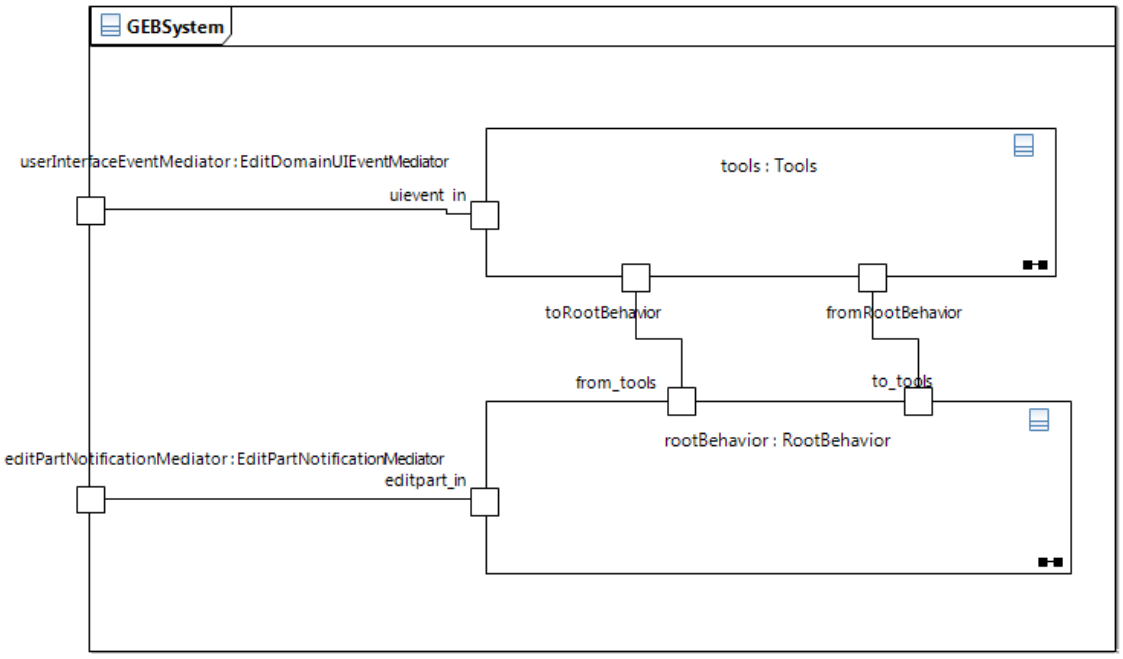

Figure 7-1 Prototype interaction boundary, user-interface event and editpart notifications

We have tested different approaches of how to integrate a behavioral system into a GEF-based editor; *where* to intercept what BDL calls Edits. The development of the prototype stopped at a point where we intercepted user-interface events and forwarded them (on the EditDomainUIEventMediator) to representations of the user-interface tools in the prototype. From there the idea was to translate and deduce the user-interface events into Edits and to send them to composites and services defined using statemachines and JavaFrame for examination. We also experimented with having references to all the information required to reason about editing behavior *within* the prototype. This is why we have the EditPartNotificationMediator connected to the RootBehavior, so that we may initialize Composites with the information contained within EditParts. EditParts are, as described previously in the thesis, the *controller* entities in GEF-based editors, giving us access to *figures* and both models (DSL meta-model model, and DSL graphical definition model).

Although beneficial for the process of examining where it would be best to place the *interaction boundary* between the Editor and the prototype, this approach proved ultimately to not be the most fruitful. As we have seen previously in this thesis, the interaction boundary is tied to Edits directly from the Editor, and not to user-interface events.

For future work with the prototype we imagine the interaction boundary between an Executing Behavioral Definition and a GEF-based Editor to exist by intercepting Requests in EditPolicies. Replacing incrementally those EditPolicies that the BD is capable of subsuming. In this way we may intercept Requests, translate them into BDL Edits, present Solutions to the user, wait until one has been selected by the user, and then ultimately return this stack to the Editor for execution upon the models if no more consistencies have been introduced (as per Behavioral Definition Execution Semantics). Translating the stack of Edits into a EMF TransactionalEditingDomain compatible stack of Commands to be executed atomically in the repository. An added benefit of placing the interaction boundary within EditPolicies is that we would then be able to quite simply insert the class responsible for interception in a GMF Generator Model using CustomBehavior elements (as mentioned previously in this thesis).

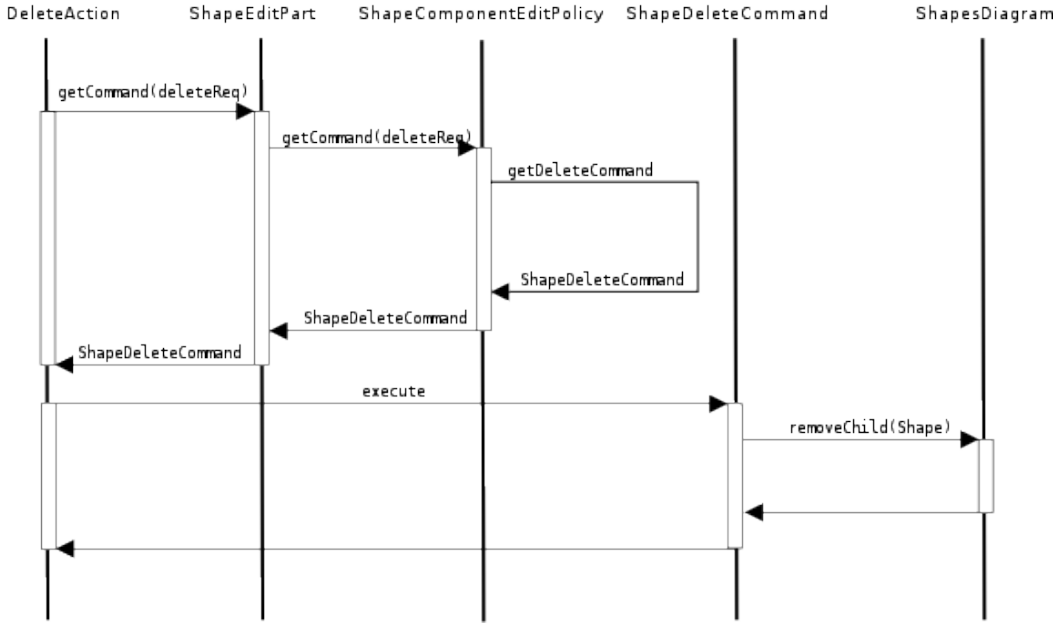

Figure 7-2 Example of Request-Command Interaction (from [45])

However there are problems with this approach, and they are related to how GEF handles sending Requests. Sending Requests in GEF to EditParts is done sequentially with a method call, and not with a signal or some other form of non-blocking operation. It effectively locks the editor until a Command has been received in return, as we see in Figure 7-2. A Delete "tool" calls a getCommand(Request deleteReq) method on an EditPart, and waits until a ShapeDeleteCommand is returned. This is one of the reasons why we

needed to intercept user-interface events in the prototype, as the sending of such events is non-blocking. We imagine that we may be able to solve this in part by automatically denying all requests.

An additional complexity stems from the fact that Requests in GEF are sent for every mouse-event received by a "tool", meaning that a prototype will become flooded with Requests (translated to Edits) if we forward all of them. To this problem we propose some counter strategies; to not send Edits to the prototype unless for instance the mouse has been idle for a set amount of time. This in combination with a hot-key that allows users to explicitly state that they want solutions to be presented. This hot-key would be absolutely necessary also to allow users to *select* a solution using the mouse without generating *new* GEF Requests, which would cancel the search process and cause all the solutions to disappear.

7.1.1 MODEL-LIBRARIES OF MEDIATORS AND MESSAGES AS API'S

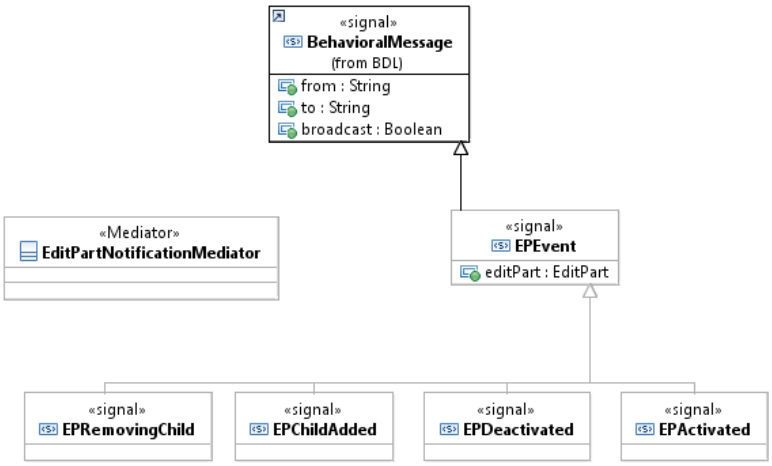

Figure 7-3 Model-library for the communication of EditPart Notifications

The prototype confirms an aspect of our framework regarding communication with the required context, the Editor, by using JavaFrame mediators and BDL messages (conforming to UML signal). We defined the required messages needed for communication, created classes stereotyped to be JavaFrame mediators and generated code functioning as the API between prototype and editor.

Although not a part of the definition of our interaction boundary in this thesis, Figure 7-3 shows such an example. The goal here is to intercept Notifications stemming from an implementation of the Observable pattern and send them as BehavioralMessages to a sub-system (our prototype) of an Editor, asynchronously and via JavaFrame Mediators.

EPEvent extends a BDL BehavioralMessage. Several other signals extend EPEvent for the different types of notifications that are sent by EditParts (incorporating the Observable pattern). The EditPartNotificationMediator acts as the mediator between the prototype and the EditPart-segment of a GEF-editor. We then transformed this model into its JavaFrame equivalent. The next step in the process was to inject calls to an instance of this mediator from within the GEF-editor. To do this we create objects of the EditPartNotificationMediator, set its scope to the rootmost object in the Editor (in our case an object BehavioralEditor *extending* a real sequence diagram editor, SeDiEditor), and then initialize our Behavioral prototype (called SequencedGEB) from within BehavioralEditor using a constructor that passed the initialized mediator to the SequencedGEB.

The next step in the process, once we have an initialized mediator connecting the sub-system (prototype) to the Editor, is to initialize a Listener which *subscribes* to notifications capable of sending BehavioralMessages to the mediator. This Listener may either reference the Editor for access to the mediator, or have a reference passed to it during construction. We implemented GEF EditPartListener interface on our Listener (BehaivoralEditPartListener), installed it on EditParts and sent messages using the mediator for every notification received; e.g. when an EditPart calls the listener's method childAdded(EditPart child, int index) we send a message EPChildAdded(child) on the mediator to the prototype.

Although the method given above pertains to EditParts and EditPartListener, we give this as an example of how to inject a Behavioral Definition *into* Editors that rely upon the Observable pattern to communicate important events. For instance, as EMF-repositories use the same principle as GEF w.r.t. observable pattern we could easily define the BD execution semantics required message NotifyUpdated.

7.2 BEHAVIORAL DEFINITION EXECUTION SEMANTICS

Our prototype does not conform completely to the framework given in this thesis, or to our definition of BD execution semantics. It *did* however allow us to experiment and find what we require of the framework and a future prototype.

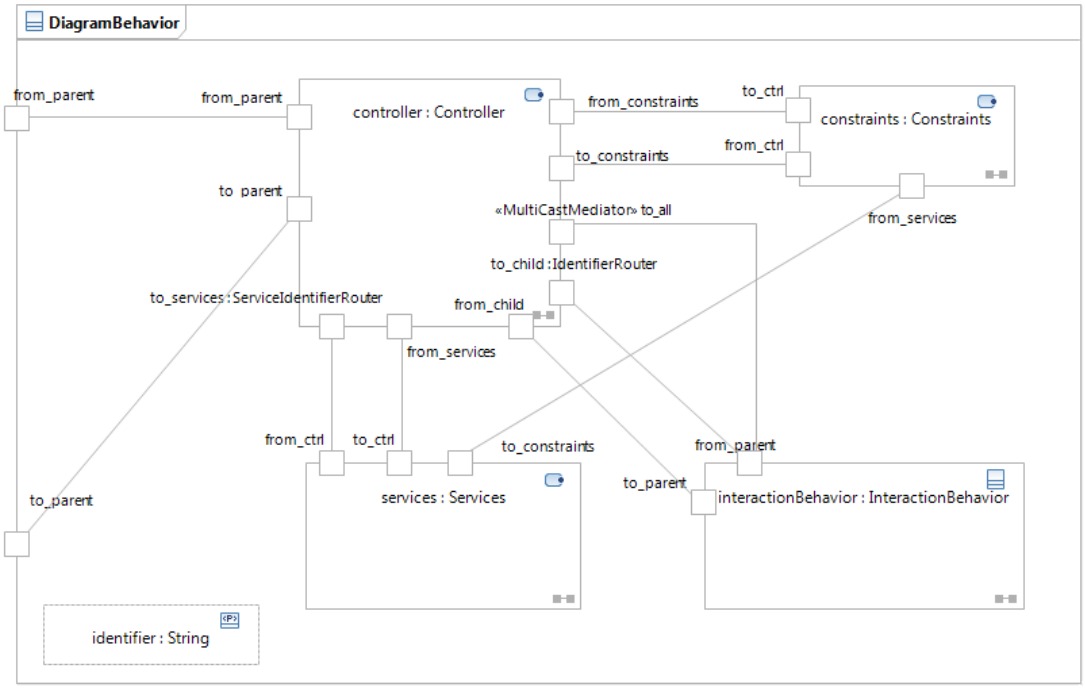

Figure 7-4 DiagramBehavior Composite

The prototype development stalled at a point in which we wanted to have a Composite *per* GraphicalEditPart in the Editor. GraphicalEditParts are controller entities for graphical elements in the editor, such as a CombinedFragment, representing actual elements in the model. The Composites were initialized with information from the EditPart, stored internally within the Composite in a part called we call Constraints. The idea was to query this part when Edits were performed to check constraints defined on the abstract and concrete graphical syntax of the element in question, using a purely programmatic approach.

Figure 7-4 shows an example of such a composite for a Diagram. We have a Controller entity responsible for routing messages, creating services and children (InteractionBehaviors in the figure). Service entities representing parts capable of reasoning about incoming messages pertaining to *editing behavior*. And a Constraint part providing an "archive" of

information and the current state of the editor element, usable by services when reasoning about what kind of editing behavior to perform.

The approach of having parts for constraints for *each* composite within the prototype proved ultimately not fruitful. It introduced several problems with mutual-exclusion on data, and in itself generating problems with respect to the *interaction boundary*. Since we stored objects received from the Editor, and manipulated them in the Constraint parts, we also ended up with added complexities regarding initializing and synchronizing the prototype with the Editor.

This is why we in our framework introduce the concept of a special type of Repository capable of providing snapshots of inconsistent models. Instead of using actual references to the model (as we have done via EditPart-storage in Constraint parts), checking constraints and manipulating values, we leave this up to the Repository and the Editor. Letting an executing Behavioral Definition (i.e. a future prototype) focus solely on finding Editing Behaviors using the strategy put forth in this thesis.

Services in the prototype were created with the thought of them being capable of reacting to a BehavioralMessage and respond with an editing behavior. This by using information stored in their local Constraint-part and by communicating with other services in other Composites. In fact we thought of Composites being akin to *Agents* in an hierarchical agent-oriented structure, were the hierarchy was given by the nesting of composites, and all agents having local descriptions of the *world* using the Constraint-part. The idea was to define editing behavior using advanced communication between agents; composites and services communicating with each other to try to find solutions to a problematic edit performed by the user. This strategy is more or less similar to how EditPolicies in GEF try to resolve problematic Requests, instead of denying them. EditPolicies in GEF may delegate Requests to *other* EditPolicies in order to create a set of Commands capable of solving the problematic Request. Letting other EditPolicies *reason* about local issues and constraints, and returning a Command if it succeeds to the EditPolicy that initiated it. However, trying to create a similar solution using agents, UML and a separate process for finding editing behaviors, proved unfruitful.

This is why in our framework we do not think of Composites as *agents* and do not try to reflect the structure of the Editor in our Behavioral Definitions. Rather we define a Composite in BDL as a *static* element.

Instead of having one composite per model element in the Editor (e.g. have a Composite for each CombinedFragment-EditPart in the Editor) we have in our framework *one* composite per element in the DSL (e.g. for a mapping-element for CombinedFragments).

Our concept called services has however remained more or less the same; its responsibility is to *find* editing behaviors. However instead of having services that communicate with other services trying to find a solution that makes a diagram *globally* consistent, we have in our framework only services that can find solutions guaranteeing only the consistency presented by the RHS pattern of an editing behavior. And then repeating the entire process if the solution selected by the user results in additional inconsistencies.

7.3 FUTURE WORK WITH PROTOTYPES

We assume the existence of several components such as our conceptual graphical definition language GDL, the special repository and a pattern matcher.

We therefore recommend that future work with the framework should be done sequentially and with smaller experiments in the following order:

(1) Create a DSL using state of the art methods for its development, defining especially the graphical syntax formally and within models. This so that all constraints capable of affecting editing behavior are defined in models. This DSL would be the foundation upon which the next experiments may be built.

(2) Create a repository capable of generating models of the DSL that are inconsistent with respect to it, when constraint breaking alterations are made on the models, instead of denying them. This would require the definition of a separate DSL with little or no constraints defined. We also imagine the introduction of special structural elements for inconsistency representation. So to even further relax the syntax by allowing orphaned elements etc. This experiment would lay the foundation of the next experiment: matching

inconsistent models to patterns of inconsistencies and creating rules capable of fixing them.

(3) Use BDL concepts and import an existing transformation language like ATL to define transformation rules that transform "patterns of inconsistency" into patterns of consistency w.r.t. the DSL. Not reasoning about global model consistency, but just patterns would greatly simplify this experiment. Of course we may encounter the pitfalls of such an approach, like the creation of additional inconsistencies and never-ending transformations if we try to apply the rules automatically, but this could be remedied by the next experiment.

(4) Create an editor that is capable of presenting rules from the above experiment that match a current inconsistency as a result of a current user initiated edit, to a user for user selection. This would give the final responsibility for the applicability of the rules to the users themselves, letting the users handle the responsibility of global consistency of the models. On this prototype we imagined multiple additional experiments may be undertaken, which we will speak of in the next chapter.

7.4 TOOLS: CHALLENGES AND PROBLEMS

A reason for the prototype's immaturity are, in addition to the overall *scale* of the prototype (in hindsight) and other reasons, the tools we used to create it. The application Rational Software Modeler from IBM [26] is based on Eclipse, and is a quite extensive tool for UML Modeling. It also supports the integration of model transformations, which suited us well as we wished to transform from UML to Java (JavaFrame). However, we experienced extensive problems with the tool once the prototype had reached a certain size; random and frequent crashes when modifying diagrams and model. Consequently at lot of time was spent trying to find out if we had a problem in the model, or if it was a problem with the tool. We also spent quite some time investigating whether or not it could be a problem relating to plug-ins that we incorporate into RSM (IFI-UML-Total tool-package [46]), used for creating sequence diagrams and transforming to JavaFrame. This was however not the case as RSM still continued to crash even without these plug-ins, once a model became of some undetermined size or complexity.

After a great deal of examination we believe the problems are related to a bug or several, regarding OpaqueExpressions and the Properties view in the tool. OpaqueExpressions may be used in UML models to refer to actual action language expressions within UML elements, using both a body attribute for the expressions and a language attribute to denote the language used. We use OpaqueExpressions heavily when creating JavaFrame compatible UML models, and it was during our work with these that most of the problems arose.

Another challenge stems from working with tools that are themselves in a somewhat prototype state, like the transformation engine creating JavaFrame consistent code. Although JavaFrame is a very stable platform, its UML2JavaFrame transformation is not. We suspect this to have mostly to do the *drift* that occurs between two components intended to be consistent, that necessarily happens when one side is updated more often that the other. Therefore a lot of time was invested into updating routines in the transformation-engine to make it more compatible with the UML-meta-model implementation in RSM.

The prototype development stalled at a point in which we were in the process of changing expressions of Java-statements into a formal UML-model. We were never able to complete this process and the prototype is therefore at the moment of writing quite non-functional, except for some basic functionality for initialization upon Editor initialization, and for intercepting and changing the state of Tool-statemachines from user-interface-events. We will in Appendix A give the diagrams of the prototype that we are successfully able to open.

8 CONCLUSION AND FURTHER WORK

This thesis lacks some of the empirical backing needed to conclude whether or not the approaches laid forth are valid, although extensive examples have been given in an effort to remedy the lack of a functional prototype. We will in the conclusion summarize our findings and in the section on Further Work give concrete suggestions for future examinations within the field of editing behavior for graphical language editors.

It is possible to view editing behaviors as a special form of transformation; as transformations on "consistent models of inconsistency". Or more precisely; to use transformation rules that have left-hand-side patterns depicting inconsistent models to find rules that lead to right-hand-side patterns capable of reintroducing consistency. The obvious problem with this approach is of course that the transformation itself may lead to inconsistencies as we do not try to infer its wider-reaching consequences on the model, other than what is defined in its right-hand-side. We solve this complexity by allowing the *user* to decide which transformations to use, allowing an editor to leverage the user's knowledge with respect to the current model and diagram to solve the inconsistencies. This is in line with how syntax-oriented editors of textual programming languages solve the problem of global consistency when presenting syntax and semantic corrections to programmers; they do not consider it a problem as it is ultimately up to the user to create a valid program and *not* the editor's.

Tied to the above is also the conclusion that we require a meta-model capable of defining the inconsistencies of another meta-model's instances. Our approach for our small example was to merely create another meta-model which was exactly the same as its source, only with a 0..1 multiplicity instead of 1. We have shown how such a "model of inconsistency" may be used in conjunction with transformation rules with "inconsistencies" in their left-hand-side patterns, to give actions capable of reintroducing consistency.

With BDL we see the benefits of how a specially tailored language for the definition of editing behavior may help editing-behavior-developers reason about the consequences of inconsistency creating *edits*. By incorporating a DSLs own graphical syntax into both

"inconsistent" patterns in the LHS and in consistent pattern in the RHS, BDL allows for a intuitive, declarative and elegant description of editing behaviors for different situations.

8.1 FURTHER WORK

We believe our approach to the definition and the implementation of editing behavior in editors for graphical languages deserves further investigation. In combination with the findings presented in this thesis, and with our findings of working with the prototype, we recommend 3 concrete steps for further work; (1) experiments examining the consistent representation of inconsistencies, (2) experiments regarding the determination of applicability of transformation rules using such an inconsistency representing model , (3) experiments with how to present the findings of such rule-matching to a user and leverage their knowledge of both DSL and current diagrams and models to the task of inconsistency management, simplifying the complexity needed when trying to find valid editing behaviors.

Several more questions arise if the 3 suggestions are proven to be valid approaches; some pertaining to the editor and editing behavior, some pertaining to DSLs and editing behavior. For the relationship between editors and editing behavior: Can we deduce any automatic behavior from the user-interaction instead of letting users choose every time an inconsistency is found? Would it be possible to create editors in this manner that purposefully *change* the way they auto-correct by *learning* from previous user selections?

For the relationship between DSL and editing behavior: What can we say about a specific graphical DSL if edits constantly result in inconsistencies?

Additionally, there already exists an approach similar to the one we have put forth in this thesis, but it did not become apparent until the very end. DiaGen [33] is a rapid prototyping tool for creating editors that supports both syntax-directed and freehand-editing. It uses an internal *hypergraph* to represent the current diagram state. This hypergraph may also be used for error-correction and editing behavior deduction. One of the main problems with DiaGen, and the hypergraph representation is, according to the authors themselves, that hypergraphs quickly become very large for even small diagrams. Although similar to our findings in this thesis, it has not, as far as we can tell, been aligned

with meta-modeling concepts, but is more closely aligned with classic compiler theory. A closer examination of DiaGen would be a an interesting approach for further work.

9 REFERENCES

1. Eclipse. *Graphical Modeling Framework (GMF)*. Available from: http://www.eclipse.org/modeling/gmf/.

2. Eclipse. *Graphical Editing Framework (GEF)*. Available from: http://www.eclipse.org/gef/.

3. Eclipse. *Eclipse Modeling Framework Project (EMF)*. Available from: http://www.eclipse.org/modeling/emf/.

4. OMG. *UML 2.2, OMG Document: formal/2009-02-04*. Available from: http://www.omg.org/spec/UML/2.2/.

5. CEA. *Papyrus UML*. Available from: http://www.papyrusuml.org.

6. Limyr, A., *Graphical Editor for UML 2.0 Sequence Diagrams*, in *Institute of Informatics*. 2005, University of Oslo: Oslo.

7. IBM. *Diagram Definition Version 0.1 - Initial Submission*. 2009; Available from: http://www.omg.org/.

8. Gronback, R.C., *Eclipse modeling project : a domain-specific language toolkit*. The Eclipse series. 2009, Upper Saddle River, N.J: Addison-Wesley. XXV, 706 s.

9. Sammet, J. *The early history of COBOL*. 1978: ACM.

10. Schmidt, D., *Guest editor's introduction: Model-driven engineering*. Computer, 2006: p. 25-31.

11. Haugen, Ø., *Hierarkibegreper i Programmering og Systembeskrivelse*, in *Institute of Informatics*. 1980, University of Oslo: Oslo.

12. Burns, Deville, and Meeker, *Power, Conflict, and Exchange in Social Life*, in *Working paper no. 88a*. 1977, Institute of Sociology, University of Oslo.

13. Thomas Kuhn, E.M.G. and I.O.L. Olivier Thomann (2006) *Eclipse - Abstract Syntax Tree*.

14. Di Ruscio, D., et al., *Extending AMMA for supporting dynamic semantics specifications of DSLs*. 2006.

15. Louden, K., *Compiler construction*. 1997: PWS Publ.

16. Espe, T.H., *A meta-language for UML concrete graphical syntax*. 2004, University of Oslo. p. 104.

17. OMG. *Diagram Definition RFP, OMG Document : ad/07-09-02*. Available from: http://www.omg.org/cgi-bin/doc?ad/2007-9-2.

18. Eclipse. *EMF- Ecore*. Available from: http://www.eclipse.org/modeling/emf/?project=emf.

19. Pinet, F., M. Kang, and F. Vigier, *Spatial Constraint Modelling with a GIS Extension of UML and OCL: Application to Agricultural Information Systems*. Lecture Notes In Computer Science, 2005. **3511**: p. 160-178.

20. OMG. *OCL 2.0, OMG Document: formal/2006-05-01*. Available from: http://www.omg.org/spec/OCL/2.0/.

21. Tor, B., *Geographic information systems: an introduction*. 2001: John Wiley and Sons, New York.

22. OMG. *Meta Object Facility (MOF), OMG Document: formal/2006-01-01*. Available from: http://www.omg.org/.

23. Mauw, S., *The formalization of message sequence charts*. Computer Networks and ISDN Systems, 1996. **28**(12): p. 1643-1657.

24. Arefi, F., C.E. Hughes, and D.A. Workman, *Automatically generating visual syntax-directed editors*. Commun. ACM, 1990. **33**(3): p. 349-360.

25. Teitelbaum, T. and T. Reps, *The Cornell program synthesizer: a syntax-directed programming environment*. Commun. ACM, 1981. **24**(9): p. 563-573.

26. IBM. *IBM Rational Software Modeler 7.5*. Available from: http://www.ibm.com/developerworks/rational/products/rsm/.

27. Eclipse. *UML2Tools*. Available from: http://www.eclipse.org/modeling/mdt/?project=uml2tools.

28. Larman, C., *Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process*. 2001: Prentice Hall PTR Upper Saddle River, NJ, USA.

29. Gamma, E., *Design patterns : elements of reusable object-oriented software*. Addison-Wesley professional computing series. 1995, Reading, Mass.: Addison-Wesley. xv, 395 p.

30. Verheecke, B. and R. Van Der Straeten. *Specifying and implementing the operational use of constraints in object-oriented applications*. in *ACM International Conference*. 2002. Sydney, Australia: Australian Computer Society, Inc.

31. VanderMeer, D. and K. Dutta, *Applying Learner-Centered Design Principles to UML Sequence Diagrams*. Journal of Database Management, 2009. **20**(1).

32. Czarnecki, K. and S. Helsen, *Feature-based survey of model transformation approaches*. IBM Systems Journal, 2006. **45**(3): p. 621-645.

33. Minas, M., *Concepts and realization of a diagram editor generator based on hypergraph transformation.* Science of Computer Programming, 2002. **44**(2): p. 157-180.

34. Bardohl, R., et al., *Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation.* Lecture Notes In Computer Science, 2004. **2984**: p. 214-228.

35. Bottoni, P., et al., *Consistency checking and visualization of OCL constraints.* Lecture notes in computer science, 2000: p. 294-308.

36. Eclipse. *ATLAS Transformation Language (ATL).* Available from: http://www.eclipse.org/m2m/atl/.

37. OMG. *MOF QVT, OMG Document : formal/08-04-03.* Available from: http://www.omg.org/spec/QVT/1.0/.

38. Heckel, R., *Graph Transformation in a Nutshell.* Electronic Notes in Theoretical Computer Science, 2006. **148**(1): p. 187-198.

39. Bardohl, R. and A. GENGED. *A Generic Graphical Editor for Visual Languages based on Algebraic Graph Grammars*. 1998.

40. Costagliola, G. and V. Deufemia. *Visual language editors based on LR parsing techniques*: Citeseer.

41. Gray, J. and A. Reuter, *Transaction processing: concepts and techniques*. 1993: Morgan Kaufmann Pub.

42. Goedicke, M., T. Meyer, and G. Taentzer. *Viewpoint-oriented software development by distributed graph transformation: Towards a basis for living with inconsistencies*. 1999: IEEE Computer Society.

43. Hausmann, J., R. Heckel, and S. Sauer. *Extended model relations with graphical consistency conditions*. 2002: Citeseer.

44. Haugen, Ø. and B. Møller-Pedersen. *Javaframe : Framework for Java enabled modeling*. in *Ericsson Conference on Software Engineering*. 2000.

45. Majewski, B. *A Shape Diagram Editor.* 2004; Available from: http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html.

46. IFI, U. *IFI UML Total.* Available from: http://www.uio.no/studier/emner/matnat/ifi/INF5150/index-eng.xml.

APPENDIX A

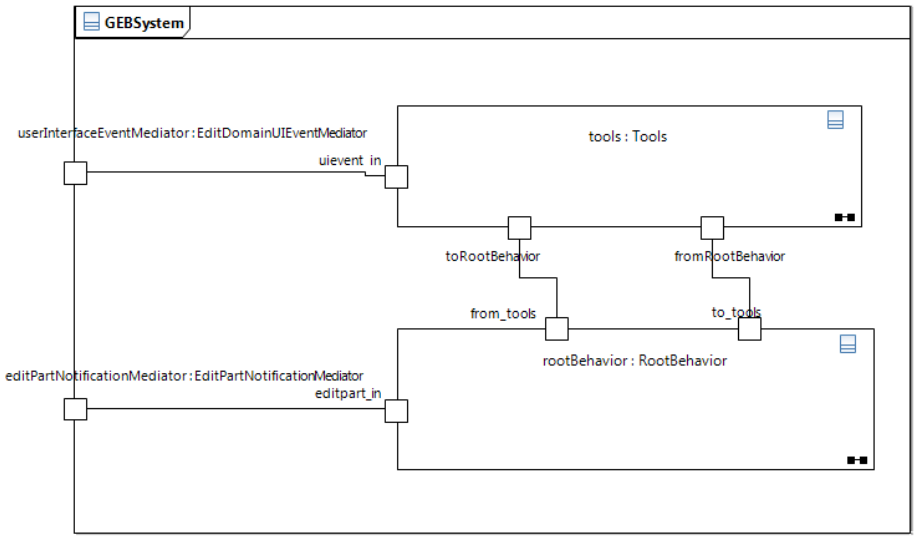

Figure A-9-1 Root composite GEBSystem

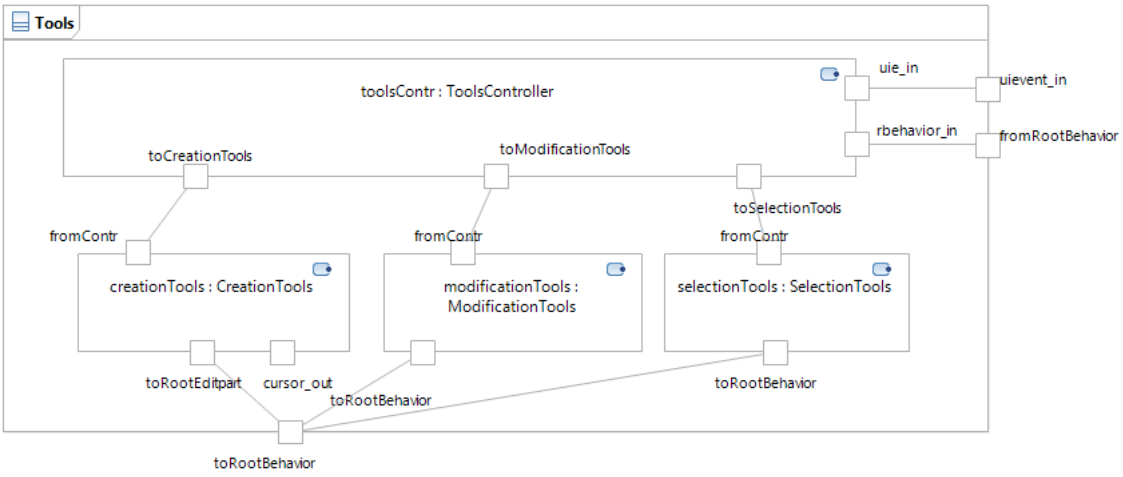

Figure A-9-2 Tools composite

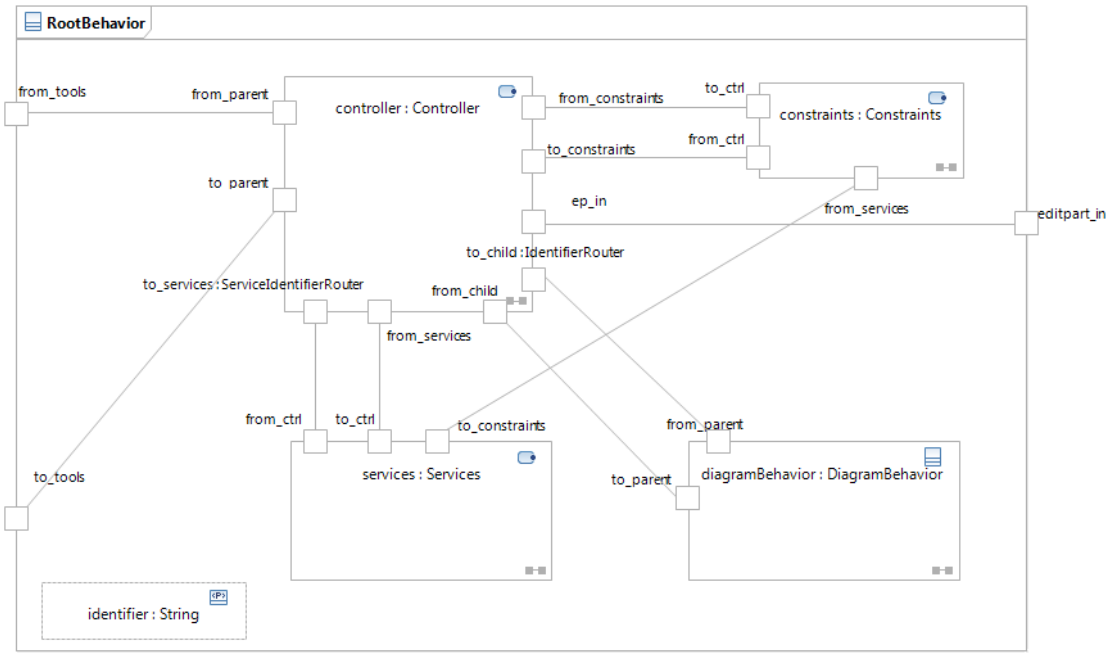

Figure A-9-3 RootBehavior Composite

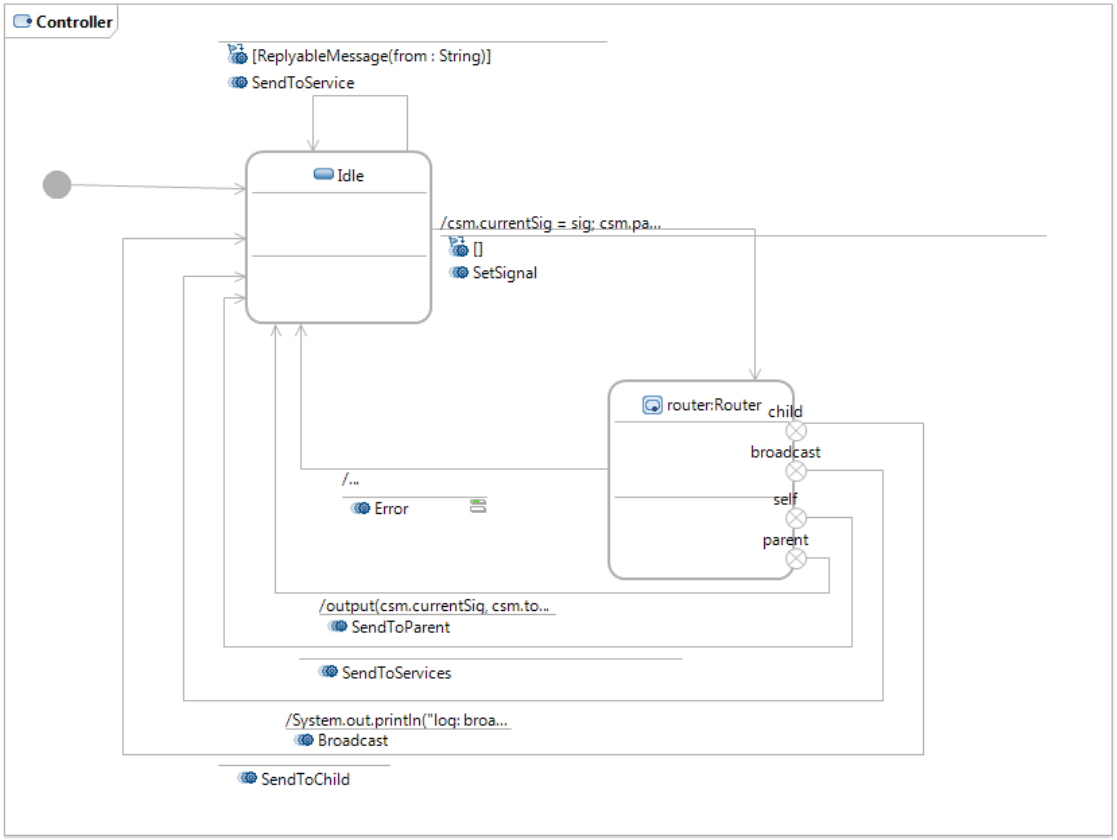

Figure A-9-4 Controller statemachine

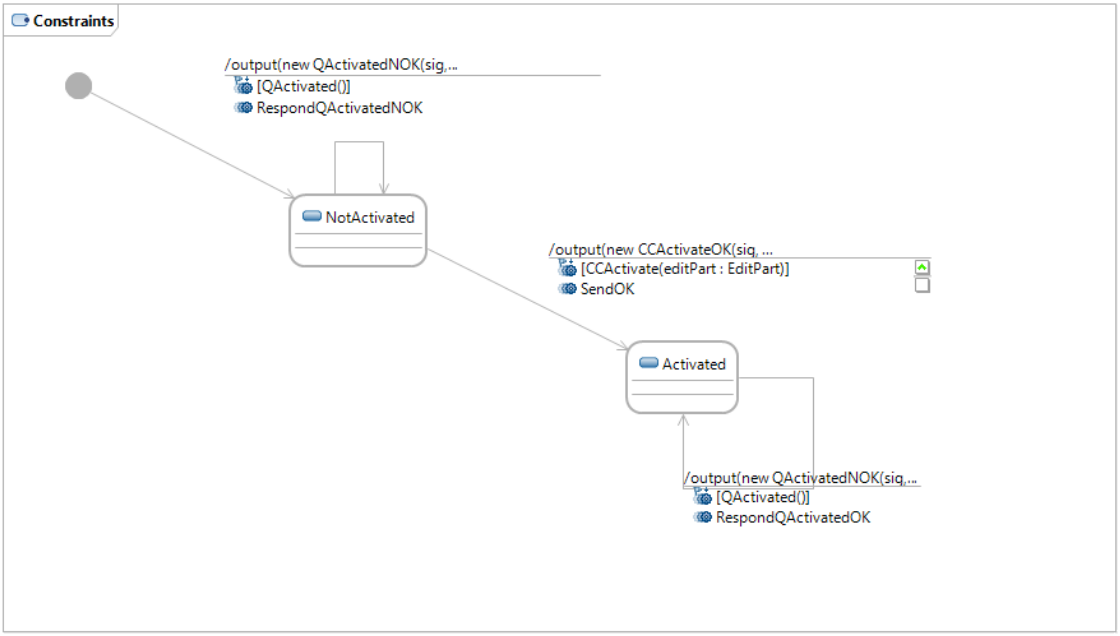

Figure A-9-5 Constraints statemachine

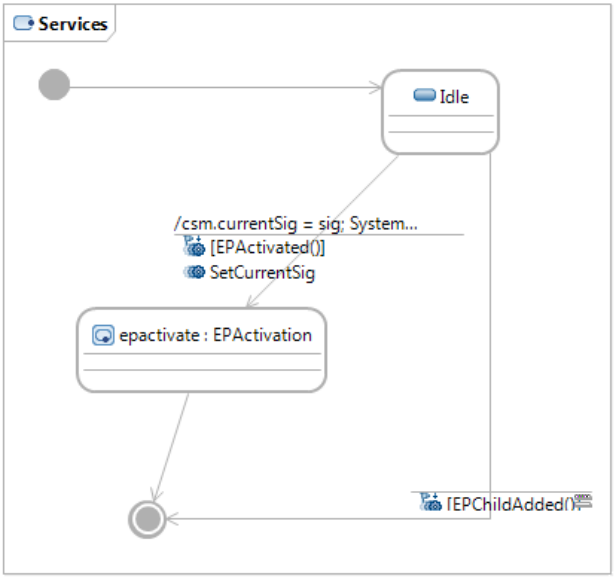

Figure A-9-6 RootBehavior Services statemachine - activation only

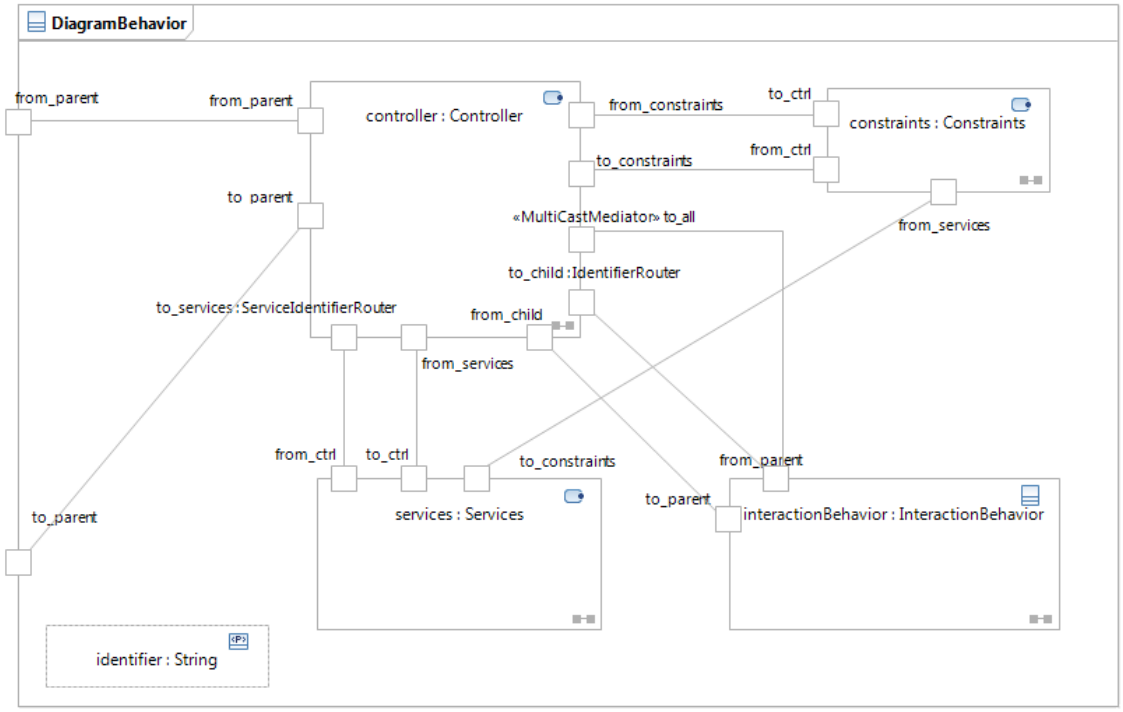

Figure A-9-7 DiagramBehavior Composite

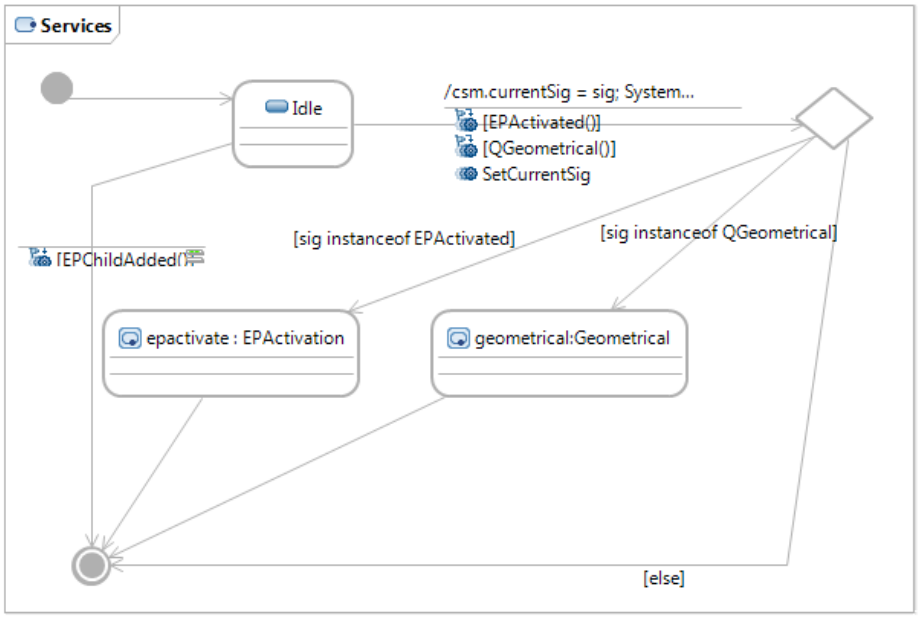

Figure A-9-8 DiagramBehavior Composite, with services for geometrical queries

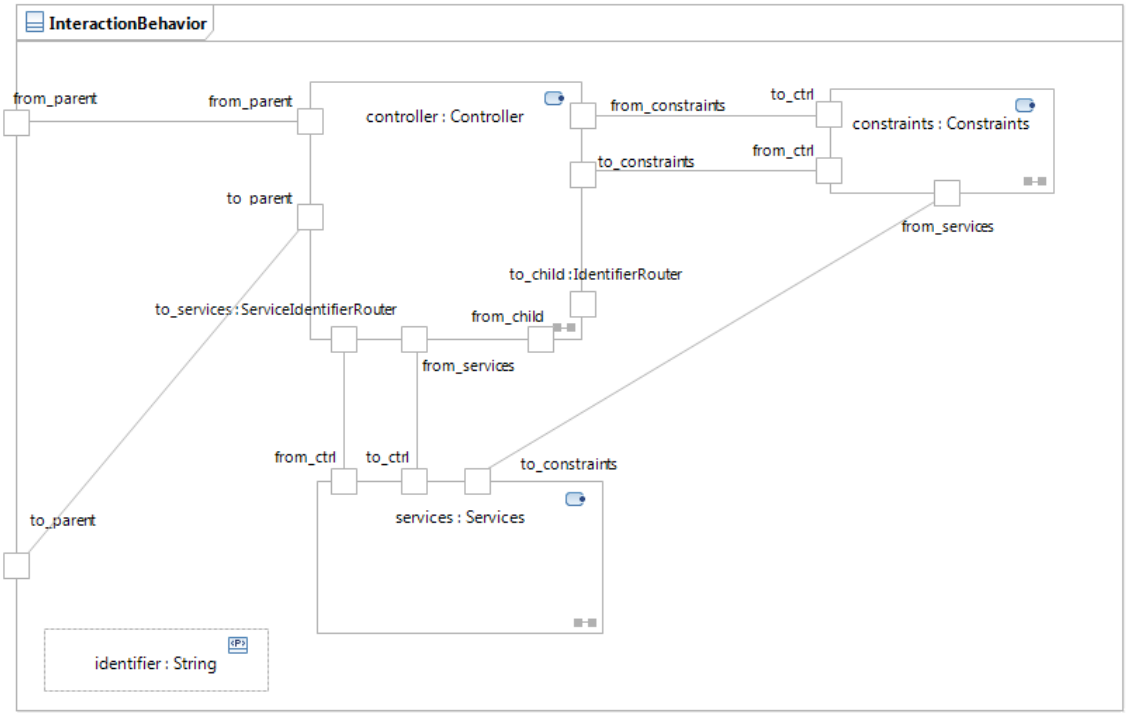

Figure A-9-9 InteractionBehavior Composite ; awaiting child composites like CombinedFragmentBehavior

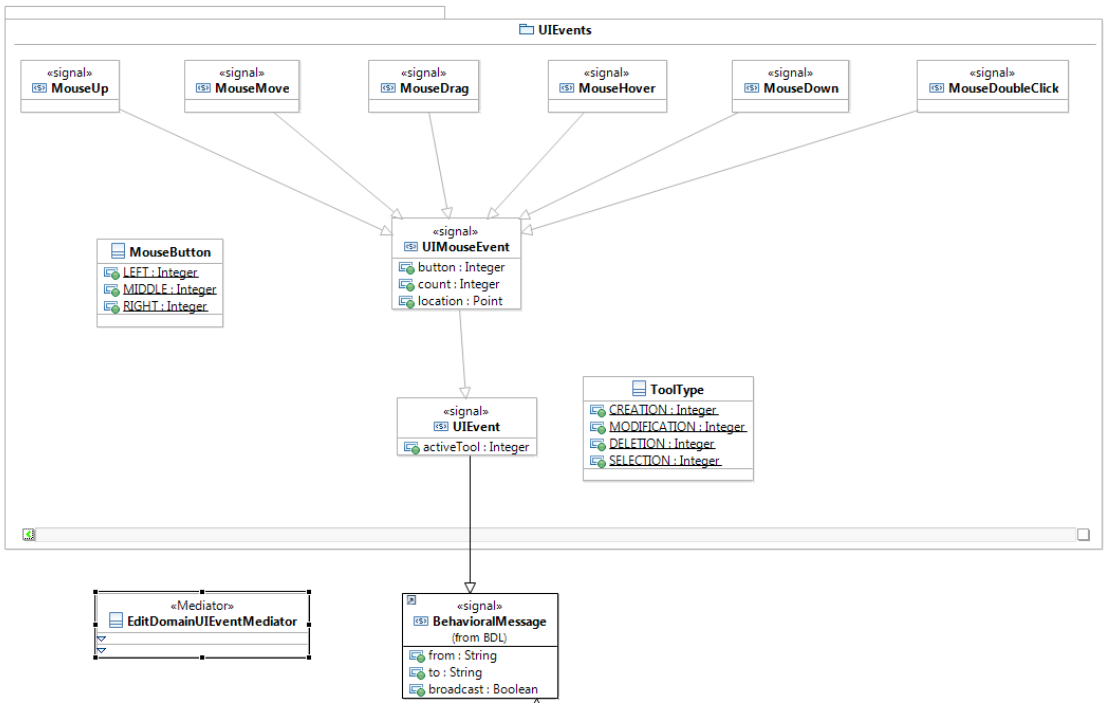

Figure A-9-10 UIEvent signals and EditDomainUIEventMediator in <<modelLibrary>> GEFMediators