AN EXTENSIBLE VISUAL CONSTRAINT LANGUAGE

By

Brian Broll

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

May 11, 2018

Nashville, Tennessee

Approved:

Ákos Lédeczi, Ph.D.

Gabor, Karsai, Ph.D.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Ákos Lédeczi, for all his support, guidance and invaluable feedback. I would also like to thank Róbert Kereskényi, Tamás Kecskés, László Jurácz and the rest of the WebGME team for their feedback and assistance. Lastly, I would like to thank my wife, Cassie, for her continued support, love, and patience for the many hours spent listening about WebGME.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . . . **ii**

LIST OF FIGURES . . . . . **iv**

I Introduction **1**

I.1 Programming Languages 1
I.2 Model-Integrated Computing 3
I.2.1 Background 3
I.2.2 WebGME 4
I.3 Visual Constraint Languages 4
I.4 Overview 7

II Abstract Syntax **8**

II.0.1 Core Concepts 8
II.0.2 Data Types and Coercion 10
II.0.3 Functions 13
II.0.3.1 Boolean Functions 13
II.0.3.2 Numerical Functions 15
II.0.3.3 Collection Functions 15
II.0.3.4 String Functions 16
II.0.3.5 Generic Functions 16
II.0.3.6 Constraint Functions 17
II.0.4 Control Flow 18
II.0.5 Commands 20

III Architecture **23**

IV Language Visualization **27**

IV.1 Inter-Block Design 27
IV.2 Individual Block Design 31

V Constraint Generation **33**

V.1 Asynchronous Code Generation 33
V.2 Additional Features 35
V.3 Algorithm 36

VI Examples and Discussion **42**

VI.1 Unique Name Constraint 42
VI.2 One Start Block Constraint 43
VI.3 Equal Incoming and Outgoing Connections Constraint 44

VII Conclusion & Future Work **46**

BIBLIOGRAPHY **47**

LIST OF FIGURES

Figure		Page
I.1	Dataflow Programming Example	1
I.2	Instruction Flow Programming Example	2
I.3	Visual OCL Example	5
I.4	Constraint Diagrams Example	6
II.1	Core Concepts	8
II.2	Block Type Examples	9
II.3	Base Data Types	11
II.4	**attribute of** Block	12
II.5	**filterByNodeType** Block	12
II.6	Logical Boolean Functions	13
II.7	Numerical Boolean Functions	14
II.8	Collections Boolean Functions	14
II.9	Numerical Functions	15
II.10	Collection Numerical Functions	15
II.11	Collection Functions	15
II.12	String Functions	16
II.13	Generic Functions	16
II.14	Node Functions	17
II.15	Node set Functions	17
II.16	Collection Functions	18
II.17	Constraint Generic Function	18
II.18	Control Flow	19
II.19	Custom Loops	20
II.20	Commands	21

II.21 Casting 21
II.22 Map Commands 22
II.23 Constraint Commands 22

III.1 Visual Constraint Language Components 23
III.2 Basic Block Relationships 24
III.3 Hierarchical Structure of Figure III.2 24

IV.1 Block Ordering Example 29

V.1 Placeholder Block Example 37
V.2 Block Attribute Example 39
V.3 Block Pointer Example 39

VI.1 Block Pointer Example 42
VI.2 One Start Block Example 43
VI.3 Equal Connections Constraint 44
VI.4 Metamodel modification 45

CHAPTER I

Introduction

I.1 Programming Languages

Decreasing the complexity of programming has been the goal of many programming languages since the early 1960's [12]. These efforts were often motivated by making programming more accessible to a larger number of people. Additionally, reducing the complexity of programming can be useful for the more broad goal of simply improving human performance when programming [25].

Various programming languages have taken a number of different approaches to simplifying the task of programming. These include simplifying the language, creating new programming models and allowing the user to create the programs using physical or graphical objects [12]. Visual programming languages (VPLs) belong to the last category as they provide the user with graphical objects to use to develop their program. There are two fundamentally different approaches to visual programming: dataflow and instruction flow programming.

Dataflow programming models a program as a directed graph in which the nodes represent instructions and edges represent the data "flowing" between these instructions [10]. Unlike most imperative programming languages, this programming paradigm focuses on the movement of the data through the program rather than the sequence of instructions being run. There are a number of visual programming languages using this paradigm. [14, 5, 23]

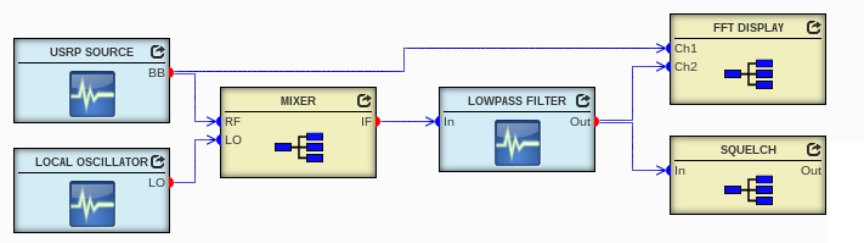


Figure I.1: Dataflow Programming Example

One example of a dataflow programming model can be found in the signal flow diagram in Figure I.1. In this example, the data is represented by the blue lines and operations are represented by the light blue and yellow boxes. As each edge in the graph is directed (from left to right), it is easy to see that the model represents the data as it visits a number of instructional nodes (including "MIXER" and "LOWPASS FILTER") where instructions are run which potentially modify the data.

Unlike dataflow programming, instruction flow programming models the sequence of instructions to be

executed. For users familiar with imperative programming, this is often more intuitive and familiar. While dataflow programming models the program with respect to the data, instruction flow programming structures the program with respect to the instructions being executed. Many textual languages fit into this model as, often times, each line of code represents an instruction to be run.

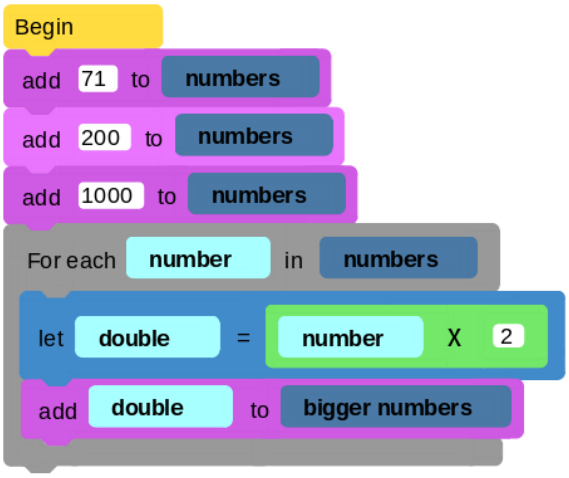


Figure I.2: Instruction Flow Programming Example

Figure I.2 provides an example of instruction flow programming. In this example, the program starts at the top block and executes the instructions of each connected block sequentially. Unlike the dataflow example, the model is not organized with respect to any data used at each step. That is, the parameters of any of the instructions could change in Figure I.2 without causing any significant changes to the model. Although some visual programming languages model data flow, this paper will focus predominantly on VPLs which model the instruction flow of the program.

The use of graphical objects to create programs provides additional benefits to improve development time. Using graphical objects removes the potential for creating subtle syntactic errors such as line terminators (such as semicolons) or misspellings in variable and method names. When reviewing textual source code, it is relatively easy for the user to overlook a misspelled word in a method name or variable name, but a misplaced block is certainly more apparent at a glance.

Graphical objects allow for more visual cues to provide intuition about the functionality and usage of the given block. Many languages will create links between objects with distinct shapes to imply the compatibility between two blocks [19, 17]. This allows any user to quickly understand how the blocks can be arranged (although he/she may not understand why any given blocks are incompatible). This also enables the user to better understand not only the inputs to a function but also understand the types of the inputs without reading any documentation.

Although visual programming languages provide a number of benefits, they also can introduce some challenges to programming. One such challenge is the Deutsch limit which states that the maximum number of objects that can be displayed on a page is approximately 50; this would significantly restrict the practicality of using a visual programming language for any nontrivial project. Another common challenge to visual programming is the limited extensibility of the languages [1]. However, this challenge has been addressed by some modern visual programming languages which allow the creation of custom blocks [17, 19, 2]. Visual programming languages also have the potential to be messy and difficult to read as code can be placed in a number of locations on the page and still produce the desired output.

Visual programming languages currently lack many development tools used with textual programming languages [4]. One such example is version control, virtually a necessity of any significant project. As changes in text can be easily captured using "diff" and applied using "patch", it is easy to see that version control is easily applied to any textual programming language. However, as visual programming languages are represented with a different underlying data structure, capturing and applying changes in a project is a nontrivial task. Another required feature is library generation; however, unlike version control, this feature has been added to some modern visual programming languages [19, 2].

I.2 Model-Integrated Computing

I.2.1 Background

Model-Integrated Computing (MIC) is the use of models to define a system which then can be interpreted to create the desired output with respect to the models representing the system and environment [20]. The use of model-integrated computing allows domain engineers to model their problem and then automate the solution of the problem for their given scenario. Model-Integrated Computing has been used in a variety of applications from development of embedded software to fault detection in space vehicles [8, 15, 20, 11].

Constraints are a set of rules used to validate a model in MIC. For example, a simple constraint in a model may simply verify that all nodes in a project have a unique name. Although this example was very simple, constraints can become rather complex as the complexity of the domain and model increase. Especially in large models, constraints can be invaluable in verifying that the project conforms to the set of rules as manually checking the rules can be impractical.

Currently, constraint rules can be written in a variety of ways. Some current constraint languages include Object Constraint Language (OCL) and Microsoft FORMULA [16, 9]. Although these languages provide a concise and unambiguous representation of the constraint, they may not be the most approachable for domain engineers. For example, suppose we have a model of a finite state machine and want to ensure that no state has a transition to itself. The OCL expression can be represented as follows

```
self.transTo->forAll(s | <> self)
```

where “self” and “forAll” are OCL keywords and “transTo” is a domain specific keyword for the finite state machine[1]. However, as these models and constraints may need to be created by domain engineers who may not have prior exposure to a constraint language, the unnatural syntax can provide additional complexity to writing constraints which can provide barriers to learning to create models and constraints as well as increase the likelihood for introducing errors in the model validation.

I.2.2 WebGME

WebGME is a modeling toolkit for Model-Integrated Computing supporting the definition of custom domain specific modeling languages (DSML) [18]. After defining a DSML, users can then create custom plugins to interpret models and create the desired output [7]. Unlike many other MIC tools, WebGME is designed for the creation of large, collaborative projects in a distributed environment.

One novel feature of WebGME is its distributed nature; however, this distributed nature has significant implications for creating and running constraints. As the client application is run in a web browser, the constraints are written in JavaScript to allow evaluation on the client application. Also, as the project is in a distributed environment in which the browser only loads the immediately required nodes from the server, not all nodes required for evaluating a given constraint may be accessible. Therefore, the client application may need to request nodes from the server during the evaluation of a constraint. To accomodate this, the constraint code often must make asynchronous calls to the server to request nodes.

The unique environment and the versatility of WebGME made it an ideal option for supporting a visual constraint language. As the constraints are currently specified using asynchronous JavaScript, simplification of the process of creating constraints would be beneficial. The distributed environment of WebGME provides a unique opportunity for creating a more intuitive and understandable constraint language as the visual language must not only provide an inuitive constraint syntax but also must be able to generate asynchronous JavaScript code to allow the constraint evaluation to be completed on the client application.

I.3 Visual Constraint Languages

As visual programming languages have been used to reduce the complexity of programming and constraint creation can be rather complex and unintuitive (especially to a domain engineer), it seems natural to simplify the constraint creation by creating a visual constraint language. This should allow us to consider the techniques used in simplifying general programming and apply them within the context of constraint languages.

[1] This example can be found in “Composing Domain-Specific Design Environments.” [16]

In recent years, there have been two similar approaches to visualizing constraints in object oriented models: Visual OCL and Constraint Diagrams [13, 21, 6].

Visual OCL is a logical, typed, object oriented language which provides an graphical representation of the Object Constraint Language to make OCL easier to use and integrate into diagrams [3]. As UML is a standard in modeling, Visual OCL is designed to adhere to the UML standard to minimize the requirements of learning a new language [22, 21].

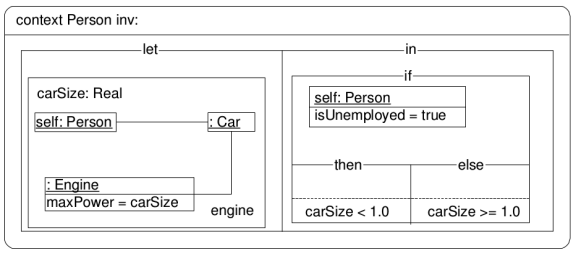


Figure I.3: Visual OCL Example

Figure I.3 provides an example constraint that validates the size of a person's car given his employment[2]. This constraint uses a "let-in" box to first define the "carSize" variable as the maximum power of the engine associated to "self" through the "car" and "engine" associations. Then, in the "in" portion of the "let-in" box, Figure I.3 uses an "if-then-else" block to represent that if the person is unemployed, then the "carSize" variable should be less than 1. Otherwise, the "carSize" variable should be at least 1.

Constraint Diagrams is another visual represention of constraints in object oriented models [13]. Like Visual OCL, Constraint Diagrams is also a logical, object oriented language. Constraint Diagrams uses a syntax similar to Euler diagrams and maintains a natural conversion to predicate logic. Constraints can also be expressed in a very compact manner using Constraint Diagrams. However, the Euler diagram style syntax can become cumbersome if an element is a member of more than three sets [13, 6].

[2]This example can be found in "Two Visualizations of OCL: A Comparison" [6]

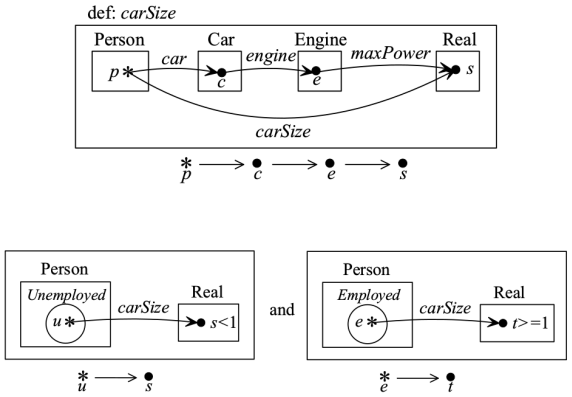


Figure I.4: Constraint Diagrams Example

Figure I.4[3] first defines "carSize" by creating the "carSize" link from the "Person" to the "Real" box at the end of the associations through "Car" and "Engine". Then, using this new association, Figure I.4 follows the "carSize" association from a person who is an element of the "Unemployed" set to a "Real" element and verifies that this element is less than 1. Using an analogous approach, it also verifies that any person in the "Employed" set has a "carSize" value of at least 1.

These visualizations of constraints share some significant differences from our visual constraint language developed within WebGME. Unlike both Visual OCL and Constraint Diagrams, the visual constraint language uses an imperative, instruction flow programming paradigm. This provides the user a significant amount of flexibility over the constraints and allows the user a more natural way to perform any necessary operations on data prior to evaluating any given value. That is, in a case where the value to be evaluated must be calculated through the traversal of a large model, an imperative language may be more natural. However, as the visual constraint language is imperative, this requires the user to not only consider the constraint to be enforced for the given model but also how to check if the constraint is violated. This can provide unnecessary complexity to creating the constraints and a less natural way to considering model validation.

Our visual constraint language is also created with the ability to be easily extended and customized for specialized domains. Rather than simply creating a robust, generic visual constraint language, this allows users the ability to create domain specific visual constraint languages for their respective domain by simply extending or restricting the generic language. As the language is defined using a metamodel within WebGME, the modification of the generic visual constraint language to a domain specific visual constraint language can be performed without requiring the user to learn new languages or tools. An example of an extension of the base language is provided in Chapter VI.

[3]This example can be found in "Two Visualizations of OCL: A Comparison" [6]

I.4 Overview

I will present a visual constraint language created for generating JavaScript constraint code for use in the distributed environment of WebGME. The visual constraint language not only supports asynchronous code generation from synchronous visual programming blocks but also provides scalability features not present in other modern visual programming languages such as version control and real-time collaboration. The visual constraint language also supports library functionality as well as easy extensibility using a visual interface. The extensibility not only allows the user to easily add new blocks to the language but also to easily modify the base language. That is, if desired, the user can simply import the structural patterns of the base language and, using the modeling interface of the WebGME, he/she can customize the language to even support an alternative paradigm.

In this paper, I will discuss the language semantics of the visual programming language, the architecture and the details of both the visualization of the language as well as the robust code generation. This paper is organized as follows: Chapter II provides the core concepts and syntax of the language. Chapter III describes an overview of the architecture of the visual constraint language. The design choices made for visualizing the language are explained in Chapter IV. Compilation and constraint generation are described in Chapter V. Finally, examples are provided in Chapter VI and future work is discussed in Chapter VII.

CHAPTER II

Abstract Syntax

II.0.1 Core Concepts

At the most fundamental level, the visual constraint language contains the following abstract concepts: **base**, **hat**, **command**, and **predicate** [1]. As the name suggests, the **base** concept is the most fundamental element of the language and simply represents a syntactic element of the language. The **hat** is an element that can only have subsequent elements and the **command** can have both predecessors and successors in a code block.

In the metamodel, relationships between blocks are represented with pointers and user specified values (such as text entered into a block) are represented as an attribute of the node in the metamodel. If the block accepts either a block or user specified values, the block will contain both a pointer and an attribute of the same name. This will be discussed further in Chapter III.

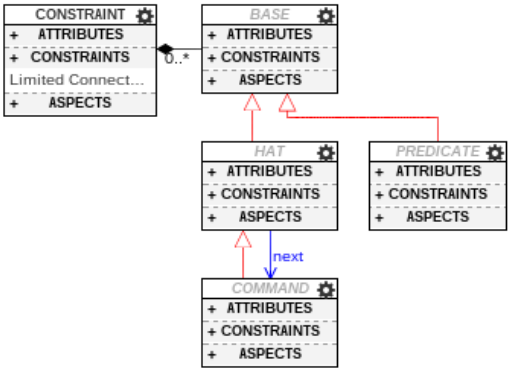


Figure II.1: Core Concepts

Figure II.1 shows the abstract syntax of these core concepts. The **constraint** block represents an element containing constraint code and the blue arrow represents a pointer, named "next", from the **hat** block definition to the **command** block definition. The "next" pointer represents the next block to be executed in the interpretation of the code.

As the **command** block inherits from the **hat** block, the **command** also can have a "next" pointer to other **command** blocks (as can the **hat** block). The **predicate** represents a code element that is dependent on either a **hat** or **command** block; that is, a **predicate** can be used only to supplement the meaning of another block. This is shown in Figure II.1 as the "next" pointer points only from a **hat** block to a **command** block and a **predicate** can be neither the source or destination of this pointer.

[1] This is similar to the abstract syntax of Snap! [19]

As shown in Figure II.1, all elements of the code blocks are descendents of **base**. This allows for the easy inclusion of these language blocks in future projects as well as easy extensibility of the language. That is, if the base language is exported to be used in another context, the visual programming blocks can be added to an object by simply specifying containment of the **base** in the project metamodel. An example of this is given in Figure II.1 as it defines every **constraint** type to contain **base** and, as a result, any visual programming block.

The **predicate** concept contains both the data types and *functions* of the language where a *function* is any block with a return value. The **predicate** also serves as the base type for our data types. These data types are first organized by plurality, then by any coercive relationships between the data types. This is discussed in more detail in Section II.0.2. Functions are children of their respective return data type. As children in the metamodel inherit the relationships from their parents and our data types are always "recipients" of relationships with other elements[2], creating the functions as children of their return data type allows a function to be used in place of its return type. The orange **children** block in Figure II.2 provides an example of a **predicate**.

Figure II.2: Block Type Examples

Examples of the block types can be found in Figure II.2. In this example, the **Begin** block is a type of **hat** block and can only have subsequent elements. This behavior is also implied by the shape of the **Begin** block as it can only connect to the top of other blocks. The **let** block provides an example of the **command** type as it can both follow blocks in the execution sequence and have other blocks connected below it (and, thus, occurring later in the code execution). The **children** block provides an example of a **predicate** block type. Unlike the **Begin** or **let** blocks, the **children** block shape does not contain any indents implying a connection to prior or subsequent blocks. As this shape suggests, the **children** block is not an independent code statement but is used to complement the meaning of the **let** block by providing the variable to be set. As **base** is the abstract base type of every block, all three blocks in Figure II.2 are descendents of the **base** type.

In our metamodel definition of the language, base types are generously used to promote extensibility and understandability of the language. For example, if one was to extend the base language (that is, the language not including the constraint data types and functions) to a custom domain or application, one could simply export the base language components as a library from within WebGME and then create children of the

[2]In this context, a "recipient" of a relationship is represented by an object being the destination of a pointer in the metamodel

command object in the metamodel. The user could also add custom data types by simply creating children of **predicate** (or a more specific data type) resulting in a language tailored to the desired application. That is, the generous use of base types creates a base language which can be easily extended for a variety of different applications.

In the following sections, the individual elements of the visual constraint language are discussed. These sections are organized as follows: **Data Types and Coercion**, **Functions**, **Control Flow**, and **Commands**.

II.0.2 Data Types and Coercion

The base language has 5 default data types: **boolean**, **string**, **number**, **map**, **collection**.

Adding this structure to the language provides multiple benefits to the visual language. As one major benefit of visual languages is the absence of syntax issues, these types allow us to further prevent semantic issues as it restricts the user from creating meaningless or unintelligible code (such as adding a string and a boolean). This also allows the user interface to be more intelligent when the user is connecting two blocks. That is, when the user is connecting boxes, the data types allow us to remove incompatible possible connections between two blocks and better predict the action intended by the user.

These specific data types were chosen as they provide intuitive, natural data types that can be easily understood without the user needing to learn the idiosyncrasies of specific types and their interactions. **Boolean**, **string**, **number**, and **collection** types are natural concepts to a person without a programming background. Although the **map** data type is less natural than the previous data types by itself, it should allow the user to write more natural and concise code.

The creation of the language as a model allows us to imply data coercion from the structure of the metamodel; data coercion can be represented with inheritance in the metamodel. As a prototypical child in the metamodel will inherit its parent's relationships, it follows that any child of a data type allows the child to be used in place of the given data type in any block connection. In the context of the visual constraint language, this results in an implicit casting of any child data type to the parent data type. Given this relationship between metamodel inheritance between data types and data coercion, we will now look at the specific implicit casting allowed by the metamodel of the base language in Figure II.3.

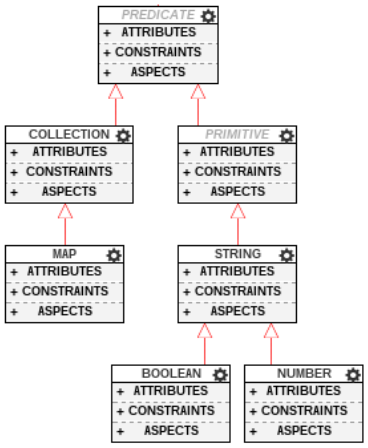


Figure II.3: Base Data Types

The two most important coercion relationships in the metamodel can be seen between the **collections** and **maps** data types and the **string**, **boolean**, and **number** data types. As the **map** data type inherits from the **collection** data type, it follows that the hashmap can be used in place of a **collection** data type. This implies that a **map** is treated as simply a specific type of **collection**. The implications of this with respect to code generation will be discussed in "Constraint Generation".

The coercive relationship between the **string**, **boolean**, and **number** data types can also be derived from the Figure II.3. As is convention in most modern programming languages, **boolean** and **number** data types can be implicity casted to a **string**. However, as an implicit casting from a **number** to a **boolean** can add unnecessary complexity to the language (as these implicit casting rules are not always intuitive), we do not allow any implicit casting between **number** and **boolean** data types.

In creating a visual constraint language, we extended the base language with two more data types: **node** and **node set**. These data types represent WebGME node elements that compose the projects and will need to be validated by our constraints. The creation of this **node** data type allows us to create custom functions for the retrieval of node attributes and project traversal. The **node set** data type represents a collection of nodes and inherits from **collection** in the metamodel. This allows any **node set** to be used in place of any **collection** data type such as in **For each** and **length of** blocks.

An example of a block specific to the **node** data type can be found in Figure II.4. In this example, the **attribute of** block retrieves the "name" of the given **node** block using the WebGME API for constraints. Just as this easy extensibility of the language allows for the addition of **node** data types with custom **node** commands, this flexibility of the language also allows for the easy creation of domain specific constraint

blocks.

Figure II.4: **attribute of** Block

The creation of the **node set** data type allows us to create functions specific to sets of nodes, such as **filterByNodeType**. As the name suggests, **filterByNodeType** is a function that allows the user to filter a set of nodes given a node type. That is, the function will return the set of nodes from the collection that are the given type. This abstraction certainly reduces the complexity of the constraint code and can be seen in Figure II.5.

Figure II.5: **filterByNodeType** Block

In Figure II.5, the **currentNode** block (of type **node**) is an argument to the **descendents of** block, a block which returns the containment subtree of the given WebGME project. This subtree is returned as a **node set** data type and is passed to the **filterByNodeType** block. The **filterByNodeType** block then returns a **node set** containing all **command** nodes in the subtree of **currentNode**. Finally, the **length of** block returns the number of nodes in the **node set**.

Figure II.5 provides a strong example of how the extension of the language can simplify the writing of the constraint code. Consider the **descendents of** block. As the project may be distributed over the network (that is, nodes may be stored on the browser, server and the database), node retrieval is asynchronous. The **descendents of** block entails asynchronously traversing the entire containment subtree from the **currentNode**. For a domain engineer, this is certainly an unnecessary complexity which may very easily introduce bugs in their constraint code. The **filterByNodeType** block then allows the user to consider the nodes in a very natural way as it allows the user to consider the desired blocks rather than thinking of the programmatic way to retrieve these blocks. When combined with the **length of** block, we have a rather natural way to express the code fragment:

"(the) length of (the) descendents of (the) currentNode of type Command"

Therefore, this extension of the base language abstracts the meaning of the constraint further from the evaluation of the constraint; this should better allow domain engineers to focus on the complex constraints and models that they are creating rather than the challenges of representing the given constraint in JavaScript, OCL or any other complex constraint language.

The ability of this language to be extended to support custom data types and custom functions for these data types allows this constraint language to be easily customized for different domains. The domain engineer can create not only a domain specific modeling language but also a domain specific constraint language which allows them to more intuitively describe constraints in their given domain.

II.0.3 Functions

Functions in this section are organized by their return type (as they are in the metamodel) with the additional constraint specific functions grouped in their own subsection. That is, a function with a **boolean** return value will be labeled a "boolean function". This section is organized as follows: The first two sections explain the boolean functions and numerical functions. Then Section II.0.3.3 discusses the functions with a **collection** block return type. Section II.0.3.4 provides the string functions supported by the language. Finally, Section II.0.3.5 and II.0.3.6 present the generic functions and constraint functions of the language, respectively.

II.0.3.1 Boolean Functions

The basic boolean functions can be divided into 3 groups: logical operators, numerical comparisons, and collection containment.

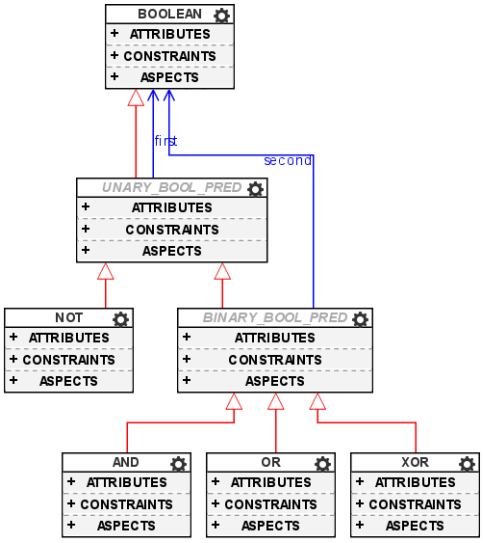


Figure II.6: Logical Boolean Functions

The base language contains **not**, **and**, **or**, and **xor** logical operators. As shown in Figure II.6, these logical operators are divided into two groups: unary boolean predicates and binary boolean predicates. These groups are created by the similarities in the input of these blocks. That is, the functions are already classified by their return type (as they are descendents of their return data type) and these further classifications are created with respect to the input arguments for the function blocks. In general, this simplifies the metamodel as it allows

for a reduction of the number of pointer connections. It also simplifies extending the language as it reduces the number of connections to make in the metamodel; that is, if the user can find an abstract descriptor block (such as **binary_bool_pred**) which is a descendent of the appropriate data type, then the user can simply add the block as a child of the given descriptor block without adding any extra pointers. This technique is utilized frequently in the function definitions.

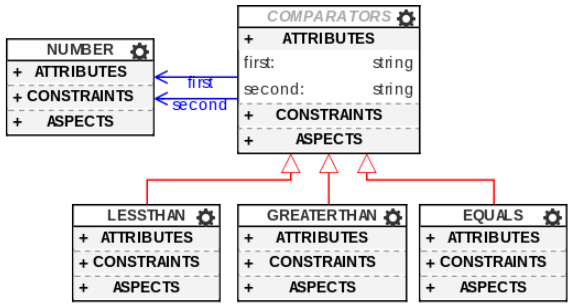


Figure II.7: Numerical Boolean Functions

Numerical boolean functions are also included in the base language. These blocks represent basic relationships between two numbers. Currently, the base language contains **less than**, **greater than**, and **equals** relations between numbers. As before, these blocks share a parent that characterizes their input, **comparator**.

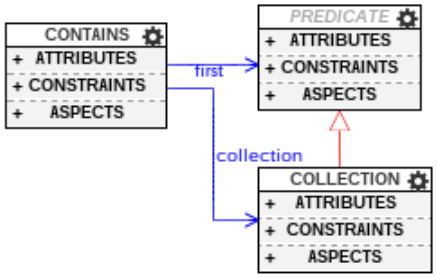


Figure II.8: Collections Boolean Functions

The last group of boolean functions in the base language is a very small group; it simply is a function which checks if a collection contains an item. As the **map** data type inherits from the **collection** data type, this method supports both checking either data type if it contains the given element. Of course, if the function is checking a **map**, it will check to see if the given element is contained as a value of the hashmap (as opposed to a key). The element that is checked is stored as the destination of a "first" pointer from the function. This element is a **predicate** type; this allows the element to be of any data type. That is, **predicate** serves as a generic data type in the metamodel as the data types contained in the collection is unknown.

II.0.3.2 Numerical Functions

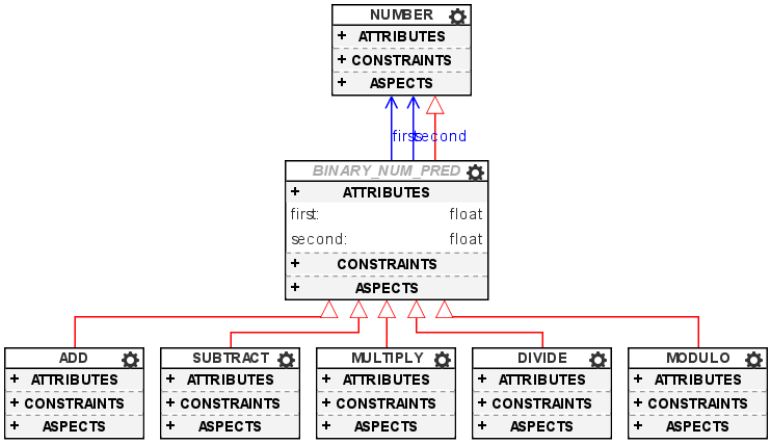


Figure II.9: Numerical Functions

Currently, the base language supports elementary math functions. As shown in Figure II.9, these include addition, subtraction, multiplication, division and the modulo operator. These are all children of a block representing numerical functions with two numeric inputs. This parent block is especially useful as adding additional math functions on two numbers can be done by simply adding an instance of **binary_prim_pred** to the metamodel and creating respective entries in the output language specification (for compilation) and an SVG (for visualization). These will be discussed in further detail in Chapter V and IV, respectively.

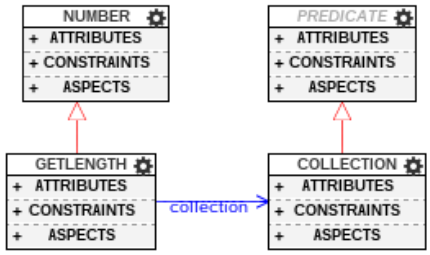


Figure II.10: Collection Numerical Functions

Another numerical function supported in the base language is the **getLength** function. This function will return the number of elements in a collection or map.

II.0.3.3 Collection Functions

Figure II.11: Collection Functions

The only function returning a **collection** type in the base language is **getKeysFromMap**. This function returns all the keys from a hash map and returns it as a **collection** type.

II.0.3.4 String Functions

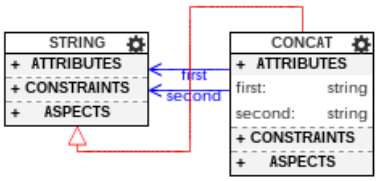


Figure II.12: String Functions

The **concat** block is currently the only function in the base language which returns a **string** type. As shown in the picture, the **concat** block accepts two **string** blocks as input, "first" and "second".

II.0.3.5 Generic Functions

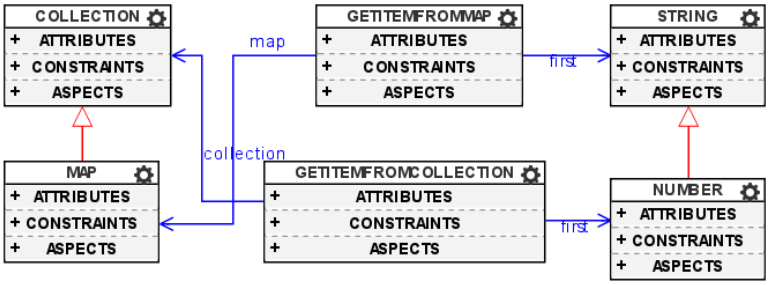


Figure II.13: Generic Functions

There are two generic functions in the base language. These are **getItemFromMap** and the **getItemFromCollection** blocks. As shown in the Figure II.13, **getItemFromMap** accepts a **map** and a **string** type as input and returns a generic. Similarly, the **getItemFromCollection** block accepts a **collection** and **number** block specifying the collection and the index. As these functions return generics, they will need to be casted to the appropriate data type before they can be used in a way specific to their given data type using a **let** block. This will be discussed further in the Section II.0.5.

II.0.3.6 Constraint Functions

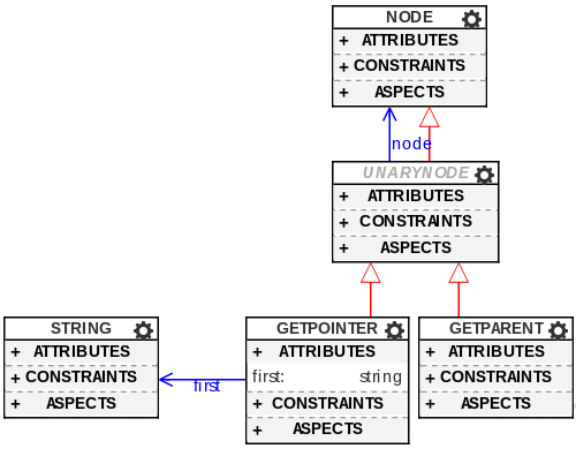


Figure II.14: Node Functions

Currently, there are two functions that return **node** blocks. These include **getPointer** which gets the target of a given node given an id ("first" as shown in Figure II.14) and **getParent** which returns the hierarchical parent of the node with respect to the containment tree of the project.

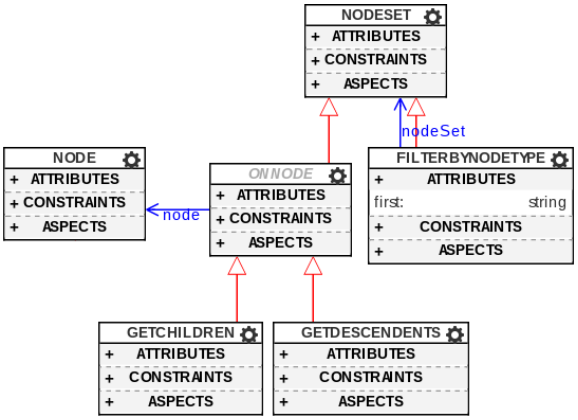


Figure II.15: Node set Functions

There are currently three functions which have **node set** return data types. These functions are **getChildren**, **getDescendents**, and **filterByNodeType**. The **getChildren** block returns the contained children of a given node allowing for traversal of the project. The **getDescendents** block retrieves all nodes in the containment subtree. The third function, **filterByNodeType**, allows the user to specify which nodes he/she is referring to within a given node set. As shown in Section II.0.2, these blocks can allow the user to simply refer to the nodes that he/she wants to use without worrying about the programmatic process to retrieve and filter the given nodes.

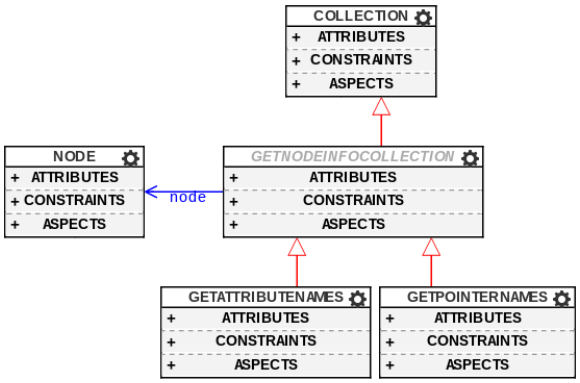


Figure II.16: Collection Functions

The constraint language also contains functions for retrieving information about a node in the form of a collection. These functions are **getAttributeNames** and **getPointerNames**. These functions, as given by their names, return a collection containing either the names of their pointers or a collection of their attribute names.

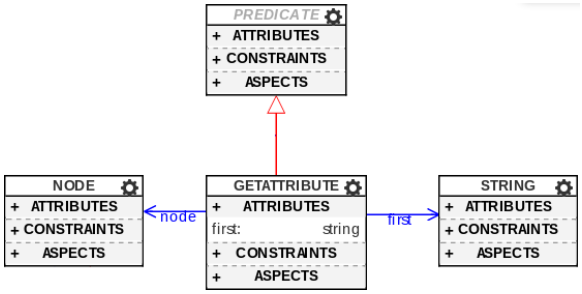


Figure II.17: Constraint Generic Function

As shown in Figure II.17, there is a function, **getAttribute**, which retrieves the value of a node. As neither the value (nor the type) of the node is known, this function returns a generic data type (**predicate**).

II.0.4 Control Flow

There are also a number of blocks for controlling the flow of the program. These have been divided into two parts: **Control Flow** and **Custom Loops**.

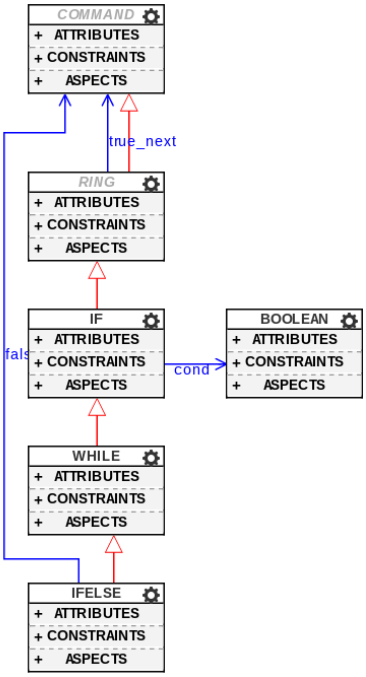


Figure II.18: Control Flow

As shown in Figure II.18, the base language contains the standard elements for control flow of the program: **if**, **ifElse**, and **while**. However, similar to Snap!, the visual constraint language contains a **ring** block [19]. The **ring** block is a generalization of "if" statements and loops as it contains command blocks and contains a "true_next" pointer to one of these commands. The **if** block then inherits from the **ring** type (allowing it to contain commands and have a "true_next" pointer to one of them) but also contains a **boolean** type which can be the destination of a "cond" pointer. This allows the **if** statement to contain a conditional to be evaluated; this conditional can then determine if the "true_next" pointer will be executed (with all subsequent blocks).

The **while** block has the same components as the **if** block. That is, they both contain a **boolean** condition and **command** blocks to be executed if their conditional is true. Therefore, the **while** block inherits from the **if** statement without adding any additional functionality; the differences between **if** and **while** are only in their concrete syntax and compilation.

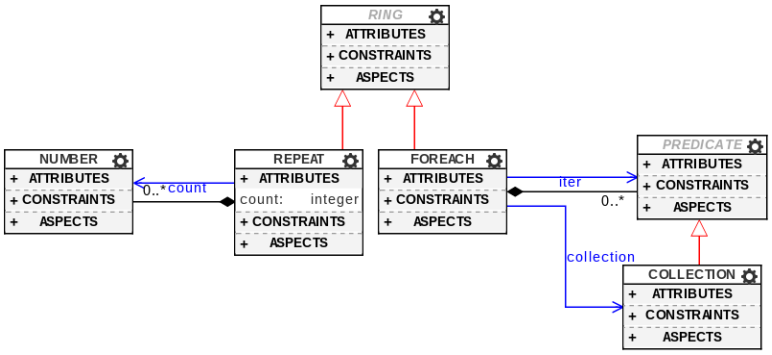


Figure II.19: Custom Loops

As shown in Figure II.19, the base language also contains two custom loops, **repeat** and **forEach**. These loops inherit from the **ring** block as they all contain a set of **command** blocks to be executed and a pointer, "true_next", to the first **command** block. As opposed to having a "cond" pointer like the **if** and **while** blocks, the **repeat** block contains a "count" pointer to a number type and a "count" attribute to specify the number of times to execute the contained code blocks. The **forEach** block has two pointers, "iter" and "collection", which specify the **collection** block to be iterated over and the iterator to be used.

Figure II.19 also shows the benefits of the abstraction of the **if** and loop block types into a **ring** block type. This allows not only for the easy creation of the **repeat** and **forEach** loops, but also allows the user to easily extend the language to contain other specialized loops such as a "do-while" or standard "for" loop.

II.0.5 Commands

The base language is rather lightweight and the commands included are predominantly for manipulating and managing the base data types. Creating a large number of blocks in the base package would add potentially inappropriate blocks for the language. Unlike textual languages, where the unknown features are virtually invisible to the end user, extra unused functionality in a visual programming language creates visual clutter that can hinder development through requiring excess searching by the user to find the desired block. As the language was designed to promote extensibility, any necessary functionality can be easily added as needed to guarantee the language does not contain excess blocks or visual clutter.

The commands have been organized into three parts. The **collection** commands and **let** command can be found in Figure II.20. Commands for the **map** data type are in Figure II.22 and constraint commands can be found in Figure II.23.

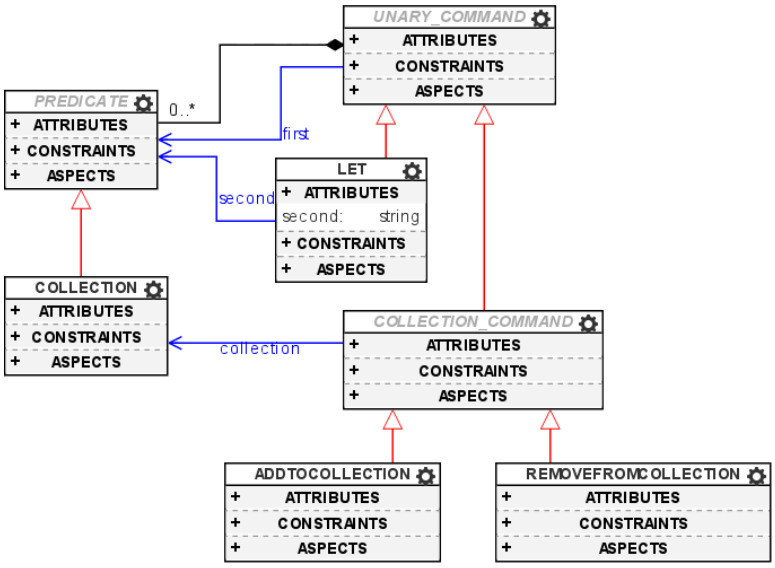


Figure II.20: Commands

The most fundamental command in Figure II.20 is the **let** block. This block allows the assignment of variables. As the **let** block accepts generic data types (as the recipient of "first" and "second" are both **predicate** blocks), this also allows for the casting of variables. That is, Figure II.17 shows that the block **getAttribute** gets an attribute from a **node** data type and returns a generic data type (as the type of the attribute is not known). This makes using the attribute directly often impossible as the type is unknown and the block will not have the required relationship in the metamodel. An example of this can be seen in Figure II.21.

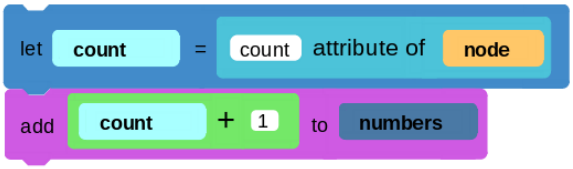


Figure II.21: Casting

In Figure II.21, the "count" attribute is retrieved from a node. As the attribute of a node could be virtually any data type of the base language, this block returns a generic. However, in this example, the "count" is likely a **number** block and, thus, should be able to be used as a **number** block. In order to do this, the generic is assigned to a variable of the type that it is expected to be. This allows for the generic variable to be casted to the given type and used accordingly. In Figure II.21, this is shown by the "count" block being then incremented by one as the **add** block only accepts **number** blocks as input. Unfortunately, if the cast is invalid, such as the "count" attribute being a **string** block type, this may result in a runtime error or undesired

behavior in the constraint evaluation.

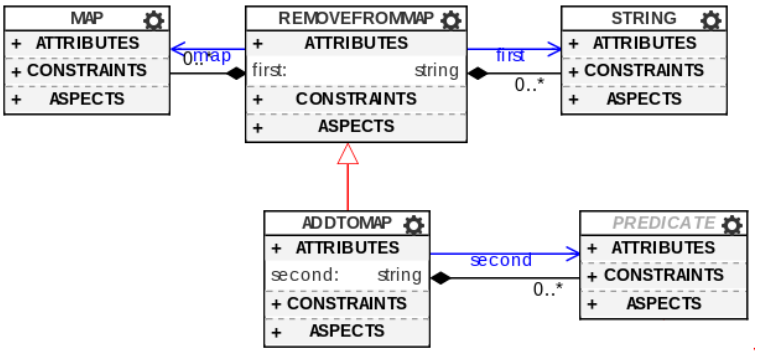


Figure II.22: Map Commands

In Figure II.22, there are two main **map** commands: **removeFromMap** and **addToMap**. The **removeFromMap** block contains a pointer to the key to be removed "first" and to the **map** block from which the given key will be removed. The **addToMap** block requires the same functionality as well as a pointer to the element to be added, "second". The **removeFromMap** block also contains an attribute called "first" which allows the user to specify the key to be used manually. The **addToMap** block inherits this text attribute as well as allows the user to specify the element to be added with a string attribute as well.

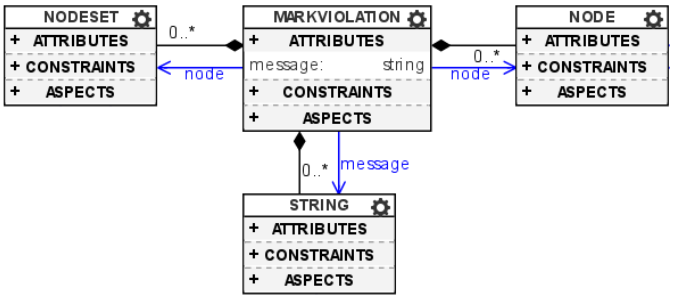


Figure II.23: Constraint Commands

The extension of the base language for constraints added only a single command: **markViolation**. This command records a violation of a constraint and allows the user to specify the message to the user as well as any node(s) causing the violation. As shown in Figure II.23, the message can be provided by either user input or given by a **string** block. This block also illustrates an example of how a single block can accept multiple types of input when the block cannot simply select a parent block. The **markViolation** block accepts either a **node** or **node set** data type as input to it's "node" field; this can be seen in Figure II.23 as the **markViolation** block has two pointers of the same name to the **node** and **node set** blocks.

CHAPTER III

Architecture

Our visual constraint language is developed within WebGME as it provides a number of advantages including a generic data model and a framework for custom data visualization. Also, using WebGME provides additional features such as version control and import/export functionality to the visual constraint language. These advantages stem from both the flexibility of the component-based nature of WebGME as well as the advanced functionality provided natively by WebGME.

The visual constraint language is composed of four main components: metamodel, model, visualizer and the compiler (implemented as a plugin to WebGME). The metamodel defines the syntax of the programming language as shown in Chapter II. The model represents constraints created using the visual programming language. The visualizer provides the concrete syntax for the data by visualizing the model in an intuitive way for the user. The visualizer also allows for editing the model with respect to the rules defined in the metamodel. Finally, the compiler (using the provided language specification for the constraints) will parse the model and generate the constraint code with respect to the provided specified language. These relationships are given in Figure III.1.

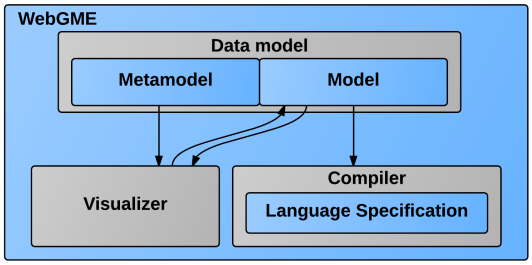


Figure III.1: Visual Constraint Language Components

We use visual blocks to represent the syntactic code elements of the visual constraint language. Blocks can have two different types of relationships between one another: either one precedes another or one block supplements the meaning of the other. If we consider the example given by Figure III.2, we can see that the **Begin** and **Let** blocks are visually connected, representing the relative ordering of the two blocks. If we consider the orange **children** block and the **Let** block, we can see that the **children** block provides meaning to what is being assigned by the **Let** block. It is apparent that the **children** block supplements the meaning of the **Let** block.

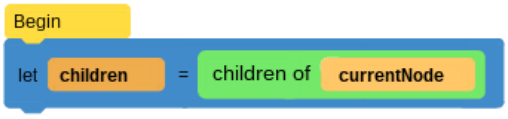


Figure III.2: Basic Block Relationships

Using the WebGME data model, we can utilize two concepts to represent the relationship between our language blocks: hierachical containment and pointers from one node to another. That is, when a block in the language precedes another, we will simply create a pointer from the former to the latter (eg, from **Begin** to **Let** in Figure III.2), creating a singly linked list of block ordering. When a block supplements another, like **children** and **Let** in Figure III.2, the supplementing block will be contained by the other in the WebGME data model and there will be a pointer from the given block to the supplementary block.

Pointers in the WebGME data model are named with respect to the relationship between the given blocks. As pointers designating a supplementary relationship between blocks result in the supplementing block becoming a child of the other (with respect to the containment tree), these pointers are called "children pointers". As the precedence relationship between blocks is the only relationship which does not imply one block supplementing the other, this pointer (called the "next" pointer) is considered a "sibling pointer".

This representation of our language in the data model can be easily illustrated. If we consider Figure III.2, our resulting model will have two nodes at the current level, the **Begin** and **Let** nodes. The **Let** node will then contain two children: **children** and **children of** nodes. The **Begin** node will also have a pointer to **Let** as it is connected into **Let.** Intuitively, this is understandable as each contained element can be viewed as **supplementing** the meaning of the parent object. Finally, the **children of** block contains the **currentNode** block. This hierarchical structure can be seen in Figure III.3.

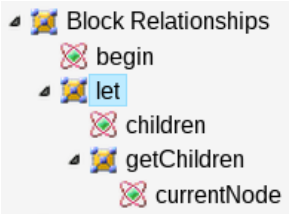


Figure III.3: Hierarchical Structure of Figure III.2

Using this representation provides not only an intuitive mapping from the language elements to the WebGME data model but also provides us with a structure that is intuitive to traverse and manipulate. That is, each node's child blocks can be viewed as supplementing the meaning of the given block. This can be seen in Figure III.2; although we can infer the general meaning from the design of the "Let" block, the details of

the block are contained in its children blocks ("children" and "children of"). Therefore, this representation of the language in the data model also provides an encapsulation of the meaning provided by certain syntactic elements as each node will contain all other nodes needed to complete the meaning of the given node. This allows operations on a given language element, such as hierarchical move or deletion, to also affect any other language elements relevant to the affected node. That is, deleting the "Let" node from Figure III.2 would automatically result in a deletion of "children" and "children of" (consequently, "currentNode").

Along with relationships between blocks, a block may need to also allow a user specified value in the block. In the model, these fields are by represented simply adding an attribute to the given block. However, if the block contains a value that can be specified with either another block or a user specified value, then the block will contain both an attribute and pointer with the same name.

Although the default WebGME data visualizer (using boxes and lines) is not appropriate for the visualization of a visual programming language, the component-based nature of WebGME allows for the creation of a custom visualizer to fit the needs of the visual programming language. This visualizer must be able to intuitively represent both the precedence and supplementary relationships as described above. The precedence relationship will be represented by visually connecting the subsequent block to its predecessor. As in the data model, the supplementary relationship will be represented with containment. Both relationships can be illustrated in Figure III.2.

In order to show these relationships, the visual blocks must contain basic information about how to be modified when containing or connecting to other blocks. Therefore, our blocks will contain information about where they can be connected to preceding or subsequent visual blocks and information about how to stretch appropriately to contain any supplementary blocks. In order to allow for programmatic manipulation of our blocks while still maintaining a modular and extensible data visualization paradigm, we use Scalable Vector Graphics (SVG) with custom data attributes. That is, each language block is represented with a custom SVG which informs the visualizer how the block should be manipulated given another block is contained within it or another block is connected to it. As any pointer will result in the recipient being connected to the base block, these relationships are called "connections" within the visualizer.

As WebGME allows for the creation of custom plugins to interpret models, the compiler can be implemented as a JavaScript plugin. This plugin will then have access to the model and will be able to traverse it as needed to generate the JavaScript constraint code. The WebGME plugin also contains an output language specification which provides the necessary information about the relationship between the blocks and the desired output language. Using this output language specification and the access to the WebGME model, the compiler can then generate the necessary asynchronous JavaScript constraint code.

Using WebGME not only provides a flexible, extensible platform on which to develop our language but

also provides additional features to the language. As WebGME projects are automatically version controlled, the visual constraint language built within WebGME is also version controlled. Also, WebGME supports exporting projects and parts of projects as libraries. This enables the visual programming language to be easily extended or restricted to create a customized visual programming language.

CHAPTER IV

Language Visualization

As previously mentioned, the component-based design of WebGME allows for the creation of a custom data visualizer for the visual constraint language. The visualizer has two main design choices: the design choices for visualizing relationships between blocks (**Inter-Block Design**) and the design choices made for visualizing a given block (**Individual Block Design**).

IV.1 Inter-Block Design

Using WebGME to design the language provides us with two useful predefined constructs that will function as the backbone to the inter-block design of our language: "containment" and "pointers". Pointers between nodes are represented by allowing the blocks to visually "connect". Naturally, containment in the data model is represented by resizing the parent block so it visually contains the child block.

As previously described, the pointer concept is used to represent relationships between blocks in WebGME. A pointer to the next block is only stored within the source WebGME object and, thus, the destination WebGME object in the project has no information about it's predecessor. After being loaded on the screen, the rendered blocks corresponding to WebGME objects contain references to both their successor and predecessor. Although this design choice added some complexity to the loading of WebGME objects, it allowed for a reduction in complexity for the metamodel creation of the visual constraint language and prevented storing redundant information in WebGME objects.

As described in Chapter III, pointers in the metamodel can be categorized as either "sibling" or "child" pointers based on the relationship between the given nodes in the project containment tree. These pointers are visualized as *connections* between the respective blocks where a *connection* is a physical linking of the blocks directly to one another. Like pointers, connections between blocks are also considered to be either a "sibling" or "child" connection. Sibling connections are links between nodes sharing a parent while children connections are links between a node and one of its children in the containment tree. Each connection is labeled with the pointer relationship it represents; this allows for a deterministic representation of the blocks relationships to other blocks. In our visual constraint language, we have one sibling connection type, "next", and multiple children connection types. That is, "next" will connect the given block to the next block to be executed while any children connections will connect the block to blocks that further complement the functionality of the given block. When two blocks are connected, I will refer to the block with the pointer as the *source block* and the block that is the recipient of the pointer in the metamodel as the *destination block*.

For example, an “if” statement will have a child pointer, “cond”, to the conditional statement to determine whether to execute the code in the block. It also has a “true_next” child connection which connects the “if” statement to the hierarchical child to execute if the conditional statement is true. The “next” statement will designate the subsequent block to be executed and will be visually connected to the “next” connection area of the “if” visual block. In this example, the “if” statement will be the source block of the “cond”, “true_next”, and “next” relationships with other blocks. Similarly, a block connected to the “cond” connection area will be a destination block of the “if” statement (with respect to the “cond” connection area) and the connected block will have a source block of the “if” statement.

When two blocks are connected, the destination block is moved to be visually connected to the source block. Consequently, the position of the destination block is dependent upon the location of the source block. As all rendered blocks on the screen have information about any blocks that could be connected to them[1], any destination block’s position depends solely on the location of the respective source block (a trivial calculation). I will call these destination blocks *positionally dependent* as the position of these blocks depend on the location of their source block. To avoid unnecessary and excessive WebGME object updates in the database, positionally dependent blocks do not update their positions in the data model; until they are disconnected from their respective source blocks, their position is determined solely from the position of their source block.

Every block can be the destination block in at most one relationship. As any block can have multiple connections to other blocks and can have at most one source block, we can create a dependency tree where a child node is a destination block and the parent is the source block for some connection between the two nodes. As all destination blocks are positionally dependent, this tree also represents a dependency of blocks in which the parent block’s position must be determined before the child block’s position can be determined. As there may be multiple blocks on the screen without any source blocks (or blocks connected into them), this will give us a forest of dependency trees where a block’s children are the union of the block’s child connections and sibling connections to other blocks.

As child connections represent both a pointer and containment between blocks, the destination block of a child connection is visually contained by the source block. This often requires that the source block is resized to fit the destination block inside. This resizing of the source block can affect the connection areas of the given block and, consequently, require a repositioning of any positionally dependent blocks. As each node in the dependency tree may need to be resized to contain its children, the resizing of a node in the tree may also force a recalculation of the sizes of all ancestors in the dependency tree as the ancestors may no longer

[1]There will only be one block connected into another as, if there were multiple blocks connected into a single block, the given block would have a position depending on the location of two other blocks. As it may not be possible to have the given block visually connected to the two arbitrary connection areas of the sources, this could result in impossible scenarios.

contain the given resized block. It follows that rendering visual blocks will require a careful ordering of the nodes based on the dependency trees of the model to prevent excessive recalculations.

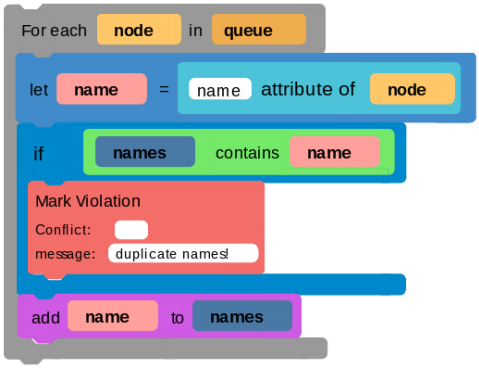
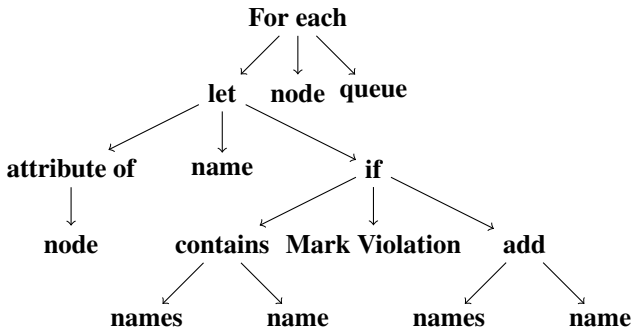


Figure IV.1: Block Ordering Example

For example, consider the blocks shown in Figure IV.1. In the corresponding dependency tree, the **node**, **queue** and **let** blocks depend on the **For each** block for their position. However, the size of the **For each** block depends on the size of the **let** block and it's siblings as the **For each** block needs to be large enough to encompass these blocks. As the **For each** block needs to encompass all contained blocks, the locations of the contained blocks also is relevant for determining the combined size of all the children. That is, the **For each** block cannot be sized until all it's contained blocks are both sized and positioned correctly.

As each block's sibling and child pointers are children of the given node in the dependency tree, creating the dependency tree for Figure IV.1 gives us the following graph:



This dependency tree illustrates a few unique characteristics of dependency trees created as described. One such feature is that the sequence of execution of the blocks corresponds to a path in this directed dependency tree. For example, the blocks **let**, **if**, **add** are performed sequentially[2] and can certainly be found in the tree. More importantly, a block, say b, contains all it's child dependents, say c_i, in it's subtree. As sequential blocks form a path in this directed tree, it follows that b's subtree also contains all subsequent blocks of c_i

[2] Although **Mark Violation** is contained in the **if** block and may be executed between **if** and **add**, it is connected to a child pointer of **if** ("true_next") and is considered to be a part of the **if** block.

with respect to the execution ordering. It follows b can be resized once it's entire subtree has been resized and b can be positioned (with all it's dependents) once it's parent in the dependency tree has been resized.

Formally, this ordering is performed by taking a root node of the dependency forest (a block which does not depend on any other block for its position) and adding it to a queue. The algorithm will then repeatedly add dependents to the front of the queue to find a node with no dependents on the queue. This will be given by the leading element on the queue that either has no dependents or has already been visited. Specifically, the algorithm for updating blocks is as follows:

Algorithm 1 The updating blocks algorithm.

updateBlocks[*updatedBlocks*]

for each updated blocks, b_i **do**
  Let t_i be the dependency tree containing b_i
  Let $n = b_i$
  while n has a parent in t_i **do**
    Let $n = \text{PARENT}(b_i)$
  end while
  if *roots* doesn't contain n **then**
    Add n to *roots*
  end if
end for
for each root, r_i **do**
  Add r_i to queue, q
  Let c, s be the child, sibling dependents of r_i, respectively
  while c is non-empty **and** r_i is not marked as visited **do**
    Mark r_i as visited
    Add s to the beginning of q
    Add c to the beginning of q
    Let r_i be the first element of q
    Let c, s be the child, sibling dependents of r_i, respectively
  end while
  Update the size of r_i given it's child dependents
  Update the position of the dependents of r_i
end for

This algorithm first finds the highest root of t_i as this will ensure that all potentially affected blocks will be updated. Then the algorithm uses the ordering given by a depth first search in which a node is added to the ordering once the DFS has backtracked from the given node. The algorithm then uses this ordering of the tree to update the size of the blocks then update the positions of the given block's dependents. That is, a block will update its size when it is r_i but will not be positioned until after its parent in the dependency tree is resized. As the whole tree may need to be updated and all non-root blocks have a parent in the tree, it follows that all positionally dependent blocks will be placed after their parent in the tree has been resized.

This organization of the connections allows us to store any features of a given block as subcomponents of

this block. This yields an organization that is not only more intuitive but also allows us to be more efficient with rendering nodes and generating code of a given block.

IV.2 Individual Block Design

Scalable vector graphics (SVG) are used for the visual representation of the blocks. This allows us to decouple the visual components from the language definition in the metamodel. This decoupling allows for easier modification and creation of new visual aspects of the language. The decoupling of the language definition and the visual aspects requires the creation of general rules for transforming a given SVG when another block is connected or nested within the given SVG component. Therefore, every dynamic SVG in the visual programming language must contain relevant information for the application of the appropriate stretching or shifting allowed by the given SVG.

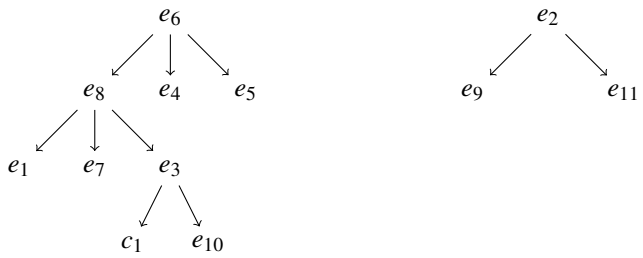
SVG transformations are composed of a series of stretches and shifts applied to the individual elements of the SVG. That is, the placement of a block inside of another results in the stretching and shifting of individual elements of the parent block to visually represent the containment of the new child element. As shown below, these transformations can be rather significant.

These shifts and stretches must be specified in the svg file. Stretches are specified with respect to the name of the connection that will cause the given element to stretch and the given axis to stretch. Shifts are specified with respect to the id of the SVG element which will cause shifts to the location of the given element and the given axis to shift. That is, the SVG element will designate an element which, when stretched or shifted, will cause a stretch or shift in the given SVG element.

Each block on the screen has current stretch values organized by the connection name that is being stretched. When another block is connected to a connection area, the parent block's stretch values for the given connection area (termed "stretch class") are updated and will be updated on the next screen refresh. Organizing the elements into stretch classes allows the SVG elements to be stretched to the maximum of the stretches of the stretch classes to which the SVG elements belong. As the SVG element's stretch classes' stretches are all recorded, the size of the SVG element is easily updated on the event of a significant decrease in the largest stretch class value. That is, if an element has three stretch classes, say "first", "second" and "third", with values 15, 17 and 20, respectively, the element will be stretched to 20. On the event that "third" is changed to 10, the SVG element can easily update itself to depend upon the stretch value of the new maximum, "second" (with a value of 17).

Unlike the stretching of the SVG elements, shifting is dependent upon other SVG elements rather than connection names. Given the shifting specifications in the SVG, our visualization component of WebGME will create a forest of shift trees for each of the elements of the SVG. Shifts are then applied by applying a

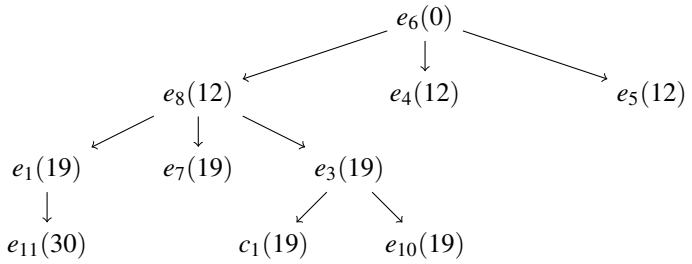
shift to the children of each stretched node and allowing the shifts to propogate down the tree. Connection areas designated in the SVG support the same functionality and can be added as leaves to the shift tree for the given SVG. For example, if we let e_i designate elements and c_i designate a connection area of a given SVG such that $e_{8,4,5}$ have a shift dependency on e_6, $e_{1,7,3}$ have a shift dependency on e_8 and c_1, e_{10} have a shift dependency on e_3, we will create the left shift tree in Figure IV.2. Letting e_2 be an unshifting element with dependents e_{11}, e_9 will give us the tree on the right.



Using this constructed forest of shift trees, we can then apply shifts by first shifting the children of stretched elements by the given amount. As previously described, these shifts will then propogate down the tree to update all dependent elements. Suppose, for our example, we have the following stretches in the SVG:

$$e_6 = 12$$
$$e_8 = 7$$
$$e_1 = 11$$

This will update the left shift tree as follows:



It is worth noting that the stretching elements and shifting elements have significantly different requirements; this is both with respect to what the elements depend upon and how their values are calculated given their dependents. The stretching elements depend upon the values of their given stretch classes (which are determined from size of any block connected to a given connection area). Given these stretch class values, the SVG element will stretch to the max stretch value dictated by the stretch classes. The shifting elements, however, are dependent on their descendents in their given shift tree; the shift value is then given by the summation of the stretch and shift values of the parent.

CHAPTER V

Constraint Generation

Constraint generation is done in 3 major steps. First, variables are hoisted to the top of the file and declared with respect to their given data type. Next, the blocks are traversed and the code is generated from the block code templates. Finally, any dependent functions required by the code templates are prepended to the generated code.

As discussed in Chapter III, a block may have both an attribute and pointer with the same name to represent a value that can be specified by either user specified text or a relationship with another block. During compilation, any values from block relationships take precedence over any user specified text (attributes in the data model). Constraint code for a given block is generated by first finding any values that would be replaced by the attributes of the given block (such as input of text fields). Then, for any pointer name required in the template, code is generated for the target block of that given pointer. This ordering allows the information about any connected block to override any attribute value that may be inputted to the given block. This approach effectively results in a creation of the code which generates the code of children and sibling connections before the node's code is generated. The constraint generation uses a robust, template based code generation. The use of templates allows our constraint generator to be easily extendable with the addition of new language concepts and with any future language or syntax changes. Like Scratch, the constraint code is generated only from code accessible from the main entrypoint; disconnected blocks are ignored [17].

The most novel aspect of the code generation for the visual constraint language lies in it's ability to generate asynchronous code. Using this asynchronous code generation, the compiler enables the user to create more manageable, synchronous code using the blocks and will compile the block source code into the necessary asynchronous JavaScript code as necessary. The constraint generator also uses intelligent variable name mapping to allow the user to ignore character restrictions or reserved words restrictions in naming variables.

V.1 Asynchronous Code Generation

As WebGME allows the loading of only the relevant parts of a project, the nodes being accessed may not be currently available to the client application. This requires the constraint code to be asynchronous as the client application may need to request the node from the server.

The constraint generator supports asynchronous code creation from the synchronous block input. Supporting asynchronous code function generation in the template requires support for moving subsequent code

into a callback while preserving and variable assignment or similar function that requires the return value of the asynchronous function. Consider the following example of loading a node in JavaScript and assigning it to the variable "myNode" (where "nodeId" is the id of the desired node).
Using naive template based code generation:

```
myNode = getNode(nodeId, function(node){
});
```

However, as *getNode* is an asynchronous function, the resulting node is actually the input to the callback function (rather than the return value of the function). Therefore, in our asynchronous code generation, we handle this by allowing the block's code to move it's parent's code (the *assignment* block in our example) inside of the given code's callback.

To prevent undesirable behavior in the case of nested asynchronous functions, we only allow this movement of the parent code inside of the callback to occur once. As the parent code snippet contains the placeholder for the subsequent commands to be executed, this single movement will account for movement of all following generated code. Using this code generator, the previous example correctly moves the parent code inside of the child code as shown below.

```
getNode(nodeId, function(node){
    myNode = node;
});
```

Nesting the parent code within the child block's code effectively allows the parent block to use the result of the asynchronous function. However, as subsequent code may also depend on the result of the the callback, the subsequent code is also executed within the scope of the callback. Just as every visual block contains a connection area to connect to subsequent code blocks, every block's code snippet contains a placeholder for the following block's code snippet. Maintaining the location of the following block's code snippet allows the subsequent code to be lifted into the asynchronous callback with the appropriate parent block's code snippet. Allowing the current insertion point of subsequent code to be held with a placeholder facilitates the generation of more complex asynchronous code.

This gives us a more robust template-based code generator as it allows our previously synchronous code to be converted to asynchronous JavaScript code. This mapping of synchronous to asynchronous code allows for a reduction in complexity of the constraint language required in our distributed modeling environment.

Supporting asynchronous functions also requires some modification to loops in the code blocks as they cannot necessarily be mapped to a synchronous "for" or "while" loop. This mapping could cause unexpected behavior as, if the loop contains asynchronous calls, the loop may enter subsequent iterations before the asynchronous call returns. In order to ensure the appropriate behavior, we map loops to recursive function calls where the recursive call is moved into the callback of the asynchronous function. As a loop with many iterations could result in a stack overflow, the recursive call is made asynchronously. Performing the recursive call asynchronously prevents the call stack from growing during subsequent iterations.

In JavaScript, this is implemented using the **setTimeout** function. JavaScript is implemented with an event queue which contains functions to be executed by the global object. The **setTimeout** function allows functions to be placed on this event queue. In the generated constraint code, loops are converted to recursive calls where the subsequent iterations of the loop (recursive function) placed on the event queue using **setTimeout**. This utilization of the event queue effective shrinks the call stack as desired and prevents any stack overflow errors as a result of any large loops in the constraint code.

V.2 Additional Features

Along with the support for asynchronous code generation, the visual constraint language code generation also contains a framework for testing of new constraint code blocks, an intelligent variable name mapping, basic name collision avoidance, lazy loading of nodes and a decoupled output language specification.

The visual constraint language also includes a framework for testing new constraint code blocks. The framework allows the user to create a test case and then simulates the asynchronous network communication used in WebGME to allow the user to run the compiled constraint code locally. To test the accuracy of the output, the user can write a JavaScript function testing the same constraint to be compared with the compiled code. As the testing is all performed locally, the user-provided JavaScript code is given direct access to the test case and the code can be written synchronously (ignoring the added complexity from the network communication). This allows the user-provided JavaScript to be written very simply while still performing as desired. This testing framework also allows for the programmatic creation of large test cases to also test the performance of the newly created code block in the context of the test constraint.

At compile time, variable names are mapped to a valid JavaScript variable name. Mapping the variable names to valid JavaScript variable names allows the user to create more readable variable names (such as "maximum nodes" as opposed to "maximumNodes" or "maximum_nodes"). Although subtle, this should promote the readability of code to non-programmers.

Also, the code generator allows the user to create boilerplate code for the template and specify variables from the boilerplate code as private. That is, variables can be specified as unavailable to the constraint.

During compilation, any variables created by the user matching a private variable from the boilerplate code will be renamed to avoid the name collision between the variable specified by the user and the variable from the boilerplate.

Another feature of the constraint code generation is the lazy loading of nodes in the constraints. When a node object is created in the constraint, it's value is simply the id of the node (as assigned by WebGME). That is, until the node is passed as input to a function, the node is simply the string id of the given node represented. However, when the node is passed as input to a function, the node is then retrieved from the local node cache of the constraint or requested from the browser cache (or, if it is not cached locally, requested from the remote WebGME server). This allows the nodes to only be loaded when they are used and can limit the number of nodes stored in the browser; this may be significant when running constraints on large projects. As the nodes are stored as simply string ids which the node object is retrieved only when needed, the user can also use a node block as an index to a hashmap.

To promote easy extension of compiler functionality, the output language specification is modular and decoupled from the functionality of the compiler. That is, the output language is defined using template snippets in a separate JSON file which can be easily modified or customized for extensions of the visual constraint language. This modularity also facilitates future extension of the compiler for general code generation rather than simply constraint generation.

V.3 Algorithm

The algorithm consists of two main functions: **createCode** and **generateBlockCode**. As each code snippet contains the placeholder for the subsequent code block's snippet, **generateBlockCode** only needs to be called for the first block.

Algorithm 2 The compilation algorithm.

createCode[language, blocks]

1: **initializeLanguage**(language)
2: *vars* be the blocks representing a variable
3: *start* be the starting block for the code
4: **for** each block, b_i **do**
5: **if isVariable**(b_i) **then**
6: Add b_i to *vars*
7: **else if isStartingBlock**(b_i) **then**
8: Set $start = b_i$
9: **end if**
10: **end for**
11: Let *code*=**declareVariables(language,** *vars***)**
12: Let $first$ be the block at the "next" pointer of *start*
13: Add **generateBlockCode(language,** $first$**)** to *code*
14: Let *code* = **mergeCodeSegments(language,** *code***)**
15: **return** *code*

Algorithm 2 starts by initializing the current language. This gives the compiler the code snippets for each block as well as some other relevant information (such as variable declaration snippets and boilerplate). Then the compiler finds all variables (and the starting block). Finally, the **declareVariables** method will declare all boilerplate variables and then the user created variables. This results in a hoisting of all user created variable blocks to the top of the generated code. Next **generateCodeBlock** will be called on the first of the code blocks and generate the majority of the code. Finally, **mergeCodeSegments** will place the code in the boilerplate provided in the language definition and add any utility functions used in the main code and defined in the language definition.

In Algorithm 3, the code for a single block and all successors is generated. First, the snippet for the given block is found from the language definition. Next, the snippet field values are recorded in a hashmap (*snippetTagContent*) from the values of the placeholders in the snippet, attributes and pointers of the code block. Finally, the snippet fields are populated given the current recorded values in *snippetTagContent*.

Placeholders are custom tags defined in the language definition and are optionally used in the code snippet to specify temporary variables. This allows for multiple of the same code block to use temporary variables that are not present in the code blocks without any collisions. For example, consider the code example given in Figure V.1.

Figure V.1: Placeholder Block Example

These code blocks will generate the following JavaScript code:

```
getNode(myNode, function(node){
    name = core.getAttribute(node, 'name');
});
```

In the generated code, it is easy to see that the "myNode" value is from the **node** block, "name" is the **string** block, and the "name" in the **getAttribute** function is provided as text input to the **attribute of** block in Figure V.1. However, as this function is asynchronous, the **let** block is "lifted" into the callback and the node represented by **myNode** is passed as an argument to the callback of **getNode**. As the nodes are lazily loaded, **myNode** is actually the id of the given WebGME node. Consequently, we need another variable in the snippet representing the actual value of the node object. This is done through the use of placeholders; as a placeholder can be specified in the language definition then a new variable will be used in the generated code

Algorithm 3 Generating the code for a single block.

generateBlockCode[language, block, *b*]

```
1:  Let snippet be the code snippet for b
2:  Let snippetTagContent be a string hashmap
3:  Let placeholders be all placeholders of the language
4:  Let attributes be the attributes of b
5:  Let pointers be the pointers of b
6:  for each p in placeholders do
7:      Let key = getPlaceholderValue(p)
8:      if snippet has field key then
9:          Let content = createVariableName(key)
10:         Add content to snippetTagContent at key
11:     end if
12: end for
13: for each a in attributes do
14:     if snippet has field a then
15:         Let content be attribute, a, of b
16:         if a is "name" then
17:             content = getVariableMapping(content)
18:         else
19:             content = getFormattedAttribute(content)
20:         end if
21:         Add content to snippetTagContent at a
22:     end if
23: end for
24: for each p in pointers do
25:     if snippet has field p then
26:         Let b_2 be the value of the pointer, p, of b
27:         if b_2 is not null then
28:             content = generateBlockCode(language, b_2)
29:         else if isOptional(language, p) then
30:             Set content to the empty string
31:         else
32:             Let content = getUndefinedValue(language)
33:         end if
34:         Add content to snippetTagContent at p
35:     end if
36: end for
37: for each key in snippetTagContent do
38:     Let content be the current value of key in snippetTagContent
39:     if hasAsyncTags(content) then
40:         Let snippet = swapSnippetAndContent(snippet, content)
41:     else
42:         Set field k of snippet to content
43:     end if
44: end for
45: return snippet
```

that does not collide with any existing variables in the generated JavaScript code.

After the placeholder values are recorded, the attributes of the given code block are recorded followed by the pointer values of the code block. As placing a code block in another block with text entered does not remove the attributes of the code block, this ordering of populating *snippetTagContent* allows the block values to overwrite the attribute values in the snippet. For example, suppose we have the code block given in Figure V.2 and place a block in the left field (as given in Figure V.3).

Figure V.2: Block Attribute Example

Figure V.3: Block Pointer Example

Certainly, if we remove the **count** block, we should still see the original number (1000888). It follows that the attributes clearly should not be cleared when a block is placed in the field but they should be overridden by the pointer value. This characteristic must be reflected in the generated code; storing the values for the fields given the attributes then overwriting them with the pointer values allows this overriding of the attribute value.

In Algorithm 3, there are also a number of functions used which require explanation. One such function is **createVariableName**. This function generates a variable name to be used in the source code that matches the format of the variables in the language (specified with a regular expression) and does not collide with any other variables from the boilerplate, placeholders, or already defined variables in the code blocks.

isOptional and **getUndefinedValue** are two more important functions used in code compilation. This allows the generation of blocks with missing attributes or pointers. That is, if a block's code snippet contains a field that cannot be populated by the given block (as it does not have the appropriate attribute or pointer), **isOptional** checks if the field is needed. If so, the field will be populated with the empty string. Otherwise, the field will be populated with the "undefined value" of the language (retrieved from the language definition with **getUndefinedValue**).

An easy example of the use of **isOptional** and **getUndefinedValue** can be found with the "next" pointer. As the user's code is not infinitely long (nor cyclical), there must be some code block that does not have a "next" pointer. This pointer is considered optional and, thus, is replaced with the empty string. However, if it was not optional, then it would be replaced with the undefined value of the language. In the example of the code generated from Figure V.1, the generated code would actually look as follows:

```
getNode(myNode, function(node){
    name = core.getAttribute(node, 'name');
    undefined
});
```

Given this code example, it may seem compelling to simply allow all fields to be optional as adding "undefined" to a snippet field may not seem to be beneficial. There are two natural alternatives to this approach: replace unknown values with the empty string or ignore unfilled field entries.

Suppose the compiler replaced all unknown fields with empty string. As demonstrated in the previous code snippet, this is sometimes a satisfactory solution. However, this approach can cause problems as some fields certainly may not be necessary or used but still must exist. This could be the case for functions with multiple arguments or with setting optional parameters for an object; a value may need to be present though leaving undefined may simply specify that the optional parameter is left unspecified. If the values are input to a function, then inserting an empty string may cause a syntax error in the generated code. As it is possible that the value is not needed by the function, this would cause an error in the compiled code which is not in the code blocks.

Suppose the compiler simply ignored fields without values specified by the current block. As each code block contains the subsequent code, a block leaving a field unfilled would allow the field to be filled by any of it's ancestor blocks (predecessors in the execution sequence). This could result in unexpected behavior by the compiler. As this option could cause nondeterministic for the code of a given block (as the generated code snippet could be modified by the given block's predecessors) and the functionality of the first option is included in the current design of the compiler, the current design of the compiler results in a more robust and reliable compiler supporting more flexible code generation.

Finally, the function that facilitates generating asynchronous code is **swapSnippetAndContent**. As described in Section V.2, this function will "lift" the current snippet into the callback of the block representing an asynchronous function. This function will recognize asynchronous code snippets and insert the parent block snippet code into the child block snippet. For example, consider the code generated from Figure V.1. When this code is being created, the snippet will be

```
name = {{second}};
{{next}}
```

and the content for the "second" pointer will be

```
getNode(myNode, function(node){
    {{_async_start_}} core.getAttribute(node, 'name'){{_async_end_}}
});
```

The **swapSnippetAndContent** function will recognize the asynchronous tags and place the content between the "start" and "end" tags in the "second" pointer of the snippet:

```
    name = core.getAttribute(node, 'name');
    {{next}}
```

Then the snippet will be inserted into the content for the given pointer in place of the asynchronous section to get the new snippet value:

```
getNode(myNode, function(node){
    name = core.getAttribute(node, 'name');
    {{next}}
});
```

CHAPTER VI

Examples and Discussion

In this section, I will provide three constraint examples using the visual constraint language: **Unique Name Constraint**, **One Start Block Constraint**, and **Equal Incoming and Outgoing Connections Constraint**. The first two constraints provide constraints which would be applied to a project container (the root node in the containment tree) and verify that all contained nodes conform to the constraint rule. The third constraint provides an example of an extension of the visual constraint language for a domain using a visualization paradigm with "connection" nodes[1]. The third example will also provide the small modifications to the metamodel performed to make the given extension.

VI.1 Unique Name Constraint

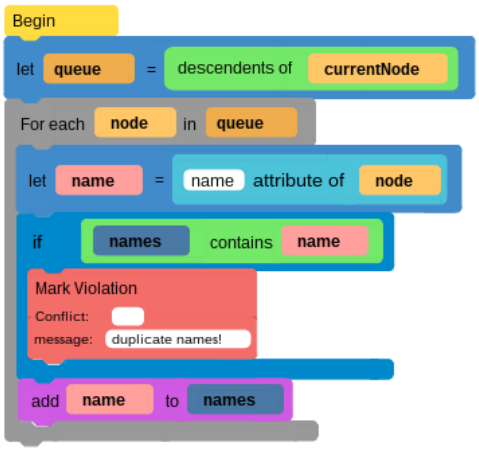


Figure VI.1: Block Pointer Example

Figure VI.1 shows an example of validating that all containment descendents of a node have a unique name. As with all constraints, this constraint starts with a "Begin" node which marks the entrypoint of the constraint. The constraint then retrieves all descendents of the current node and assigns them to the "queue" variable. The "queue" node set is then iterated through using the **forEach** loop using "node" as the iterator. For each node, the name attribute is retrieved from the node and assigned to the "name" string variable. If the name has already been visited (and added to the "names" collection variable), then the constraint marks the current node as violating this constraint. Otherwise, the code will simply add "name" to the list of seen names (recorded in the collection, "names") and continue.

[1] In this context, a connection node is a node which is visually represented as a connection between two other nodes in the model.

VI.2 One Start Block Constraint

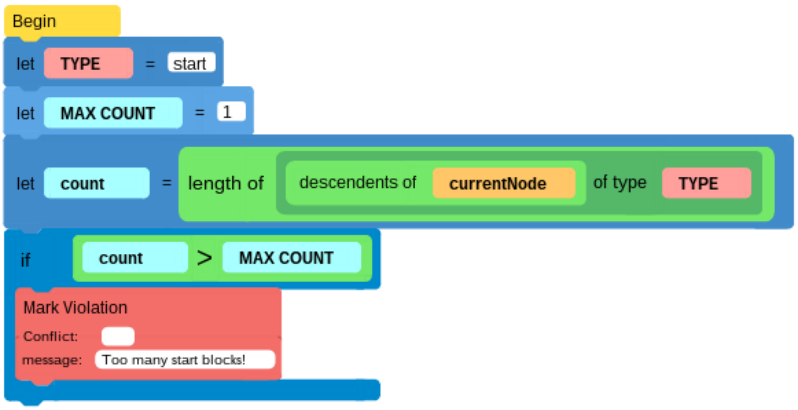


Figure VI.2: One Start Block Example

Figure VI.2 starts by setting the "TYPE" variable to "start" and the "MAX COUNT" variable to 1[2]. Next, the number of descendents of the current node of type "start" are retrieved and assigned to the variable "count". Finally, it simply checks if "count" is larger than "MAX COUNT" and marks a violation if appropriate.

This example illustrates the significant reduction of complexity provided by the visual constraint language. Specifically, the retrieval of the number of descendents of a given type is performed by simply performing three operations on "currentNode" in Figure VI.2 while these blocks encompass asynchronously loading all containment descendents, filtering the results by the specified node type and returning the number of remaining nodes. As this requires a large amount of asynchronicity, representing it in a synchronous manner provides the user with a significant reduction of code complexity.

The simplicity of this example is a result of more than simply providing useful functions for node retrieval within WebGME. Providing a library for writing JavaScript functions could provide the functionality of convenient blocks such as **getDescendents** in Figure VI.2. However, as the loading of nodes in WebGME is an asynchronous call, this would require the **getDescendents** method to also be an asynchronous call and would consequently affect how the method must be used. Using this visual programming language allows the user to not only encapsulate the convenient functionality within a custom block but also to hide the asynchronicity of the code to the end user.

[2] As the variable names are converted to valid variable names (if needed) within the compiler, the space within "MAX COUNT" is a valid variable name.

VI.3 Equal Incoming and Outgoing Connections Constraint

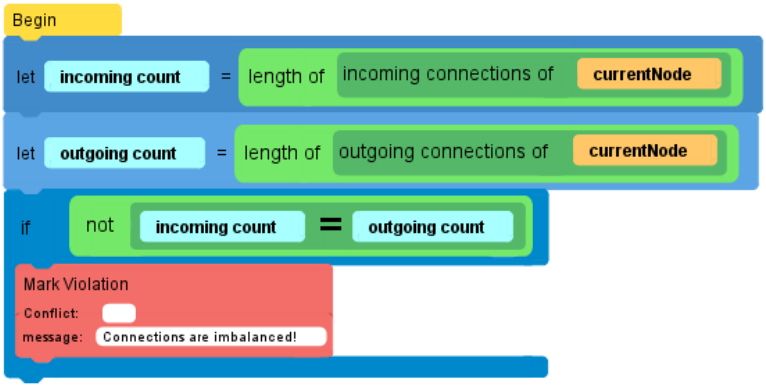


Figure VI.3: Equal Connections Constraint

Figure VI.3 presents a constraint which will verify that the given node contains an equal number of incoming and outgoing connections. The constraint first retrieves the incoming connections of the current node and stores the count in the "incoming count" variable. Similarly, the number of outgoing connections is stored in the "outgoing count" variable. Next, the two counts are compared and the node is marked as violating the constraint if the two values are nonequal.

Unlike the previous two examples, this example provides an extension to the standard visual constraint language for models which are using connections[3]. Unlike the **filterByNodeType** block, these blocks can be used to filter by the type recognized by the visualizer; rather than filtering the nodes by a metamodel type, they can be filtered by type recognized by the visualization paradigm.

This constraint represents one basic extension that can be made to the visual constraint language given knowledge about the model. In Figure VI.3, we created a new data type which **node** represents a type of node in the WebGME model, namely **connections**, and functions to retrieve this new data type. However, this abstraction can be used to create custom data types representing structures in the model. For example, consider we are modeling a simple workflow with data and tasks to perform on the given data. As the workflow should likely be acyclic, it may be useful to consider a cycle in the graph as a custom data type. When considering the cycles as a unit, creating a constraint verifying the graph is acyclic is trivial. This allows constraints to be created at a higher level of abstraction than simply considering the nodes in the graph (and discovering the cycles in the constraint).

[3]These are connections which are nodes in the model and usually represented with a line connecting two boxes. The connections described in the visualization of the visual programming language are simply the presence of a pointer from one node to another and not actual objects in the data model.

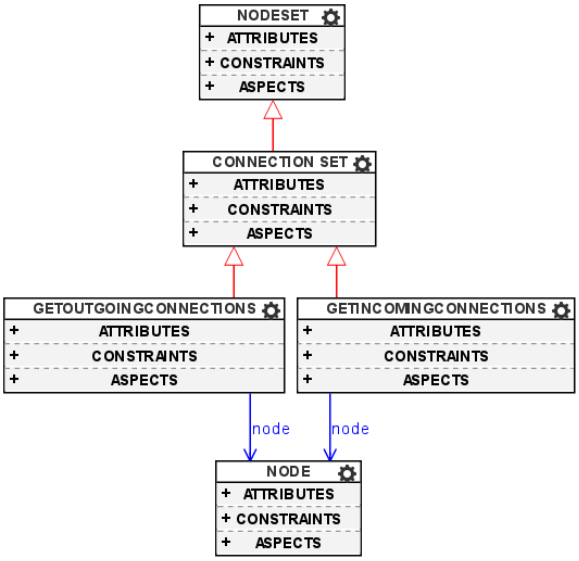


Figure VI.4: Metamodel modification

Figure VI.4 shows the three boxes added to the metamodel to create the customizations to the visual constraint language used in Figure VI.3. As the connections are a type of node recognized by the visualizer, the **connection set** block inherits from the **node set** block. **GetIncomingConnections** and **getOutgoingConnections** both return a block of type **connection set**. As functions in our visual constraint language inherit from their return data type, these blocks inherit from the **connection set** data type. As both functions also are performed on a given node, they both have a pointer to a block of type **node**.

Although this provides a simple example of the extensibility of the language, it provides an example of using new data types to refer to types of nodes which could be difficult to define otherwise. Given a domain specific application, this concept can be further utilized to create very precise data types and functions that are unique to the given domain. As the visual constraint language is easy to edit as well as extend, this would allow the user to then remove unwanted generic types to create a smaller, more precise language to define their constraints. As this new language is domain specific, this can also create constraints that are more natural and familiar to a domain engineer.

CHAPTER VII

Conclusion & Future Work

This language provides a number of benefits to constraint creation. As domain engineers are not necessarily software engineers, our visual constraint language provides an alternative to more complex constraint languages such as Object Constraint Language [24] or programming languages (such as JavaScript as in WebGME) [18]. In the context of WebGME, the visual programming language also reduces the additional complexity as a resulting from the distributed environment of WebGME. The semantics of the visual constraint language also promote a reduction of complexity as it contains a number of blocks that entail common functionality for WebGME project traversal and constraint generation.

The construction of the visual constraint language as a model allows the visual constraint language definition to not only be extended for custom domains but also provides the language definition in a familiar format for domain engineers. This will allow the domain engineers to not only extend the language to better match the needs of their domain but also enable them to remove and modify the base language to create custom fundamental syntactic structure as needed. That is, this allows subsets of the language to be forked and modified to fit the needs of the domain; this feature of the language allows for easy creation of domain specific constraint languages.

Creating the visual constraint language as a model within WebGME provided a number of additional advantages over other visual programming languages [2]. Building our visual constraint language within WebGME provided a version control system and a collaborative development environment. These enable our visual constraint language, as well as any future versions or variations to be useful for large scale projects.

However, despite these benefits, the language is not always ideal for constraint creation. Complex constraints can be cumbersome in creation and creation with language blocks is not as flexible as creating the constraint code by hand. Also, the language does not currently support first class functions that can be called from within a constraint definition as in some other visual programming languages [19, 17]. Unfortunately, this can result in cumbersome definition of complex constraints.

This constraint language not only provides a useful tool for constraint definition within the distributed nature of WebGME, but provides the basis for future research. First class functions could allow for the creation of more complex constraints without cumbersome creation. Also, modifying the base language to a logic programming paradigm could potentially create a more concise constraint representation for some domains.

[1] Andrew Begel. Logoblocks: A graphical programming language for interacting with the world. *Electrical Engineering and Computer Science Department, MIT, Boston, MA*, 1996.

[2] Blockly. https://developers.google.com/blockly/. Accessed: 2014-12-19.

[3] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. A visualization of ocl using collaborations. In *¡¡UML¿¿ 2001The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 257–271. Springer, 2001.

[4] Margaret M Burnett, Marla J Baker, Carisa Bohus, Paul Carlson, Sherry Yang, and Pieter Van Zee. Scaling up visual programming languages. *Computer*, 28(3):45–54, 1995.

[5] Chance Elliott, Vipin Vijayakumar, Wesley Zink, and Richard Hansen. National instruments labview: a programming environment for laboratory automation and measurement. *Journal of the Association for Laboratory Automation*, 12(1):17–24, 2007.

[6] Andrew Fish, John Howse, Gabriele Taentzer, and Jessica Winkelmann. Two visualizations of ocl: A comparison. 2005.

[7] Graham Hemingway, Himanshu Neema, Harmon Nine, Janos Sztipanovits, and Gabor Karsai. Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach. *Simulation*, page 0037549711401950, 2011.

[8] Larry Howard. Cape: A visual language for courseware authoring. In *Second Workshop on Domain-Specific Visual Languages*, 2002.

[9] Ethan K Jackson, Dirk Seifert, Markus Dahlweid, Thomas Santen, Nikolaj Bjørner, and Wolfram Schulte. Specifying and composing non-functional requirements in model-based development. In *Software Composition*, pages 72–89. Springer, 2009.

[10] Wesley M Johnston, JR Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.

[11] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[12] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, 37(2):83–137, 2005.

[13] Stuart Kent. Constraint diagrams: visualizing invariants in object-oriented models. In *ACM SIGPLAN Notices*, volume 32, pages 327–341. ACM, 1997.

[14] Jeffrey Kodosky, Jack MacCrisken, and Gary Rymar. Visual programming using structured data flow. In *Visual Languages, 1991., Proceedings. 1991 IEEE Workshop on*, pages 34–39. IEEE, 1991.

[15] Zsolt Lattmann, Adam Nagel, Jason Scott, Kevin Smyth, Joseph Porter, Sandeep Neema, Ted Bapty, Janos Sztipanovits, Johanna Ceisel, Dimitri Mavris, et al. Towards automated evaluation of vehicle dynamics in system-level designs. In *ASME 2012 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 1131–1141. American Society of Mechanical Engineers, 2012.

[16] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001.

[17] J. Maloney, M.Resnick, N. Rusk, B.Silverman, and E.Eastmond. The scratch programming language and environment. *Trans. Comput. Educ.*, 2010.

[18] Miklos Maróti, Tamas Kecskés, Robert Kereskényi, Brian Broll, Peter Völgyesi, Laszlo Jurácz, Tihamer Levendoszky, and Ákos Lédeczi. Next generation (meta)modeling: Web- and cloud-based collaborative tool infrastructure. In *Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS 2014, Valencia, Spain, September 28-October 3, 2014. Proceedings*, volume 8767 of *Lecture Notes in Computer Science*, Valencia, Spain, 2014. Springer.

[19] J. Mönig and B. Harvey. Bringing “no ceiling” to scratch: Can one language serve kids and computer scientists? *Constructionism 2010*, 2010.

[20] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *Computer*, 30(4):110–111, 1997.

[21] Visual OCL. http://www.user.tu-berlin.de/o.runge/tfs/projekte/vocl/. Accessed: 2014-12-19.

[22] Visual OCL: A visual notation of the object constraint language. http://www.user.tu-berlin.de/o.runge/tfs/projekte/vocl/gKTW02.pdf. Accessed: 2015-1-19.

[23] William W Wadge and Edward A Ashcroft. Lucid, the dataflow programming langugage. 1985.

[24] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[25] Kirsten N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, 8(1):109–142, 1997.