Extensible Visual Constraint Language

Brian Broll
Vanderbilt University
brian.broll@vanderbilt.edu

Ákos Lédeczi
Vanderbilt University
akos.ledeczi@vanderbilt.edu

Abstract

The paper presents a visual, imperative language for the specification of constraints corresponding to domain-specific modeling languages (DSML) in the new WebGME modeling environment. The language is based on the visual notation introduced by Scratch. The novel feature of the approach is that the constraint language is just another DSML defined through UML class diagram-based metamodels. As such, it is easily extensible via metamodel inheritance. The visual constraint programs are automatically translated into asynchronous JavaScript code that utilizes the native WebGME APIs for evaluation.

Keywords DSML, constraints, visual programming

1. Introduction

Model-Integrated Computing (MIC) is one approach that advocates the definition of domain specific modeling languages (DSML) via metamodels [11]. The DSML, in turn, enables domain engineers to model their system and use various analysis, simulation and code generation tools to solve their problems.

Constraints are used to specify well-formedness rules for a DSML. They allow the user to enforce rules that are difficult or impossible to capture by the metamodel. For example, a constraint may state that all models in a project have a unique name. Although this example is very simple, constraints can become rather complex as the complexity of the DSML and the models increase.

Typically, constraints are written in a declarative manner. Some current constraint languages include the Object Constraint Language (OCL) [12] and Microsoft FORMULA [5]. Although these languages provide a concise and unambiguous representation of constraints, they tend to be unnatural for domain engineers, increasing both development time and the likelihood of errors. On the other hand, using a traditional imperative programming language for constraint specification would not necessarily overcome these problems. Such constraints would need to utilize the general API of the given modeling environment with its associated—typically steep—learning curve.

Recently, many visual programming languages emerged with the explicit goal of making programming more accessible to novices, typically children. Of particular importance is Scratch [10] because it has gathered millions of users and many other environments adopted its intuitive visual notation. This paper presents our experience in creating a visual language based on Scratch for constraint specification for DSMLs.

1.1 WebGME

The Generic Modeling Environment (GME) [7] is a well-know tool supporting MIC. Recently the newest generation MIC toolsuite called WebGME has been introduced [8]. Designed with a strong focus on scalability and extensibility, WebGME provides many features improving the viability of modeling in large, real-world contexts including its web browser-based user interface, version controlled cloud-based backend and collaborative, real-time editing support similar to Google Docs.

Probably the single most important distinguishing feature of GME has been its support for prototypical inheritance. Any model in GME can be used as a prototype for the creation of a derived model. Any subsequent changes in the prototype (or base model) automatically propagate to the derived model. In turn, the derived model can be used as a prototype for further specialization. This way an entire inheritance tree can be constructed. This feature promotes model reuse and can be used, for example, to model product lines. WebGME takes prototypical inheritance one step further and uses it to fuse the metamodel and the domain models. The metamodel and domain models reside in the same inheritance tree and any change to the former immediately propagate to the latter. This makes DSML evolution seamless and fits nicely with the inherently iterative nature of DSML design.

As the WebGME client needs to run in the browser, it is implemented in JavaScript. WebGME supports the creation of custom plugins that, when executed by the user, can access the model through a set of JavaScript APIs and perform meaningful operations including modifying the models and/or generating output such as code. Similarly, constraints can be written directly in JavaScript to allow them to be evaluated on the client side. Note that the models are edited in a distributed environment in which the browser only loads the immediately required data from the server; hence, not all model elements required for evaluating a given constraint may be immediately accessible. Therefore, the client application may need to request additional data from the server during the evaluation of a constraint. To accommodate this, the constraint code often must contain asynchronous network requests to retrieve the desired data. Writing asynchronous JavaScript code is beyond the expertise of most domain engineers who would otherwise be inclined to use WebGME for their needs.

Therefore, the paper presents a visual alternative to both JavaScript code and declarative constraint languages such as OCL and FORMULA. This visual constraint language is designed for generating 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 due to its WebGME-based implementation, it 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 extensibility using a visual interface. The extensibility not only allows the user to 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 WebGME, she can customize the language to support an alternative paradigm.

2. Related Work

In recent years, there have been approaches to visualizing constraints in object oriented models such as Visual OCL and Constraint Diagrams [6][1][4]. Visual OCL is a logical, typed, object oriented language which provides a graphical representation of the Object Constraint Language to make OCL easier to use and integrate into diagrams [3]. Visual OCL is designed to adhere to the UML standard to minimize the requirements of learning a new language [2][1].

Constraint Diagrams is another visual representation of constraints in object oriented models [6]. 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 [6][4].

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 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 is typically a better fit. 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 increased complexity to creating the constraints and a less natural way to considering model validation.

Our visual constraint language is created with the ability to be easily extended and customized for specialized domains. Rather than simply using the 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.

3. Visual Constraint Language

3.1 Architecture

Our visual constraint language is developed within WebGME, hence, 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 in WebGME). The metamodel defines the syntax of the programming language. The model represents constraints created using the visual 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. The dataflow among these relationships is illustrated in Figure 1.

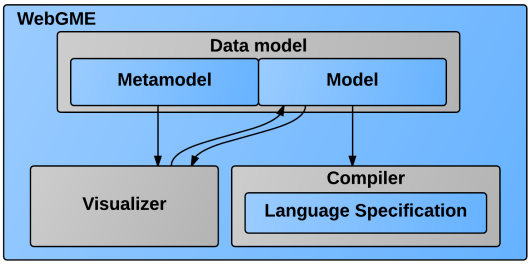

Figure 1: Visual Constraint Language Components

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 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.

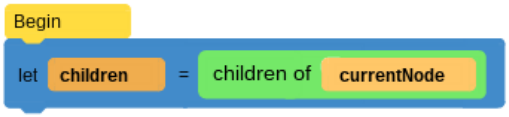

Figure 2: Basic Block Relationships

Using the WebGME data model, we utilize two concepts to represent the relationship between our language blocks: hierarchical 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 (e.g., from **Begin** to **Let** in Figure 2), creating a singly linked list of block ordering. When a block supplements another, like **children** and **Let** in Figure 2, the supplementing block will both be contained by the other in the WebGME data model and be the target of a pointer from the given block. Consider the containment tree visualization of Figure 2 below.

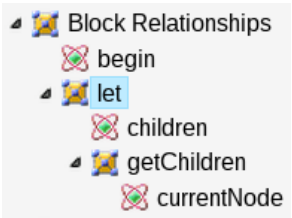

Figure 3: Hierarchical Structure of the Constraint in Figure 2

Along with relationships between blocks, a block may need to also include a user specified value in the block. In the WebGME data 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 a 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 2.

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 WebGME data model and will be able to traverse the model 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.

3.2 Language Syntax

The syntax of the visual language is based on Snap! (a Scratch derivative), the educational visual programming language developed at Berkeley[9]. Like Snap!, the most fundamental level of the visual constraint language contains the following abstract concepts: **base**, **hat**, **command**, and **predicate**. 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. User specified values (such as text entered into a block) are captured as an attribute of the node. 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.

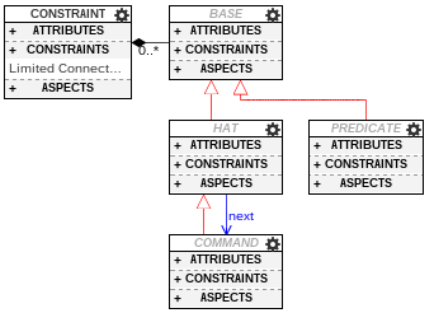

Figure 4: Core Concepts

Figure 4 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 4 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.

3.2.1 Data Types and Coercion

Using these core concepts, we extend the metamodel to add data types and functions to the programming language. The base language has 5 default data types: **boolean**, **string**, **number**, **map**, **collection**.

These specific data types were chosen as they provide intuitive, natural data types; **boolean**, **string**, **number**, and **collection** types are certainly natural concepts to a person without a programming background. The **map** data type, although it may be less natural, 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 of 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 5.

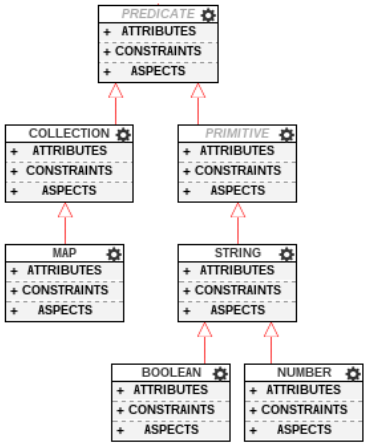

Figure 5: Base Data Types

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**.

3.2.2 Functions

In the metamodel, functions inherit their return type. This allows a function with a given return type to be used in place of any blocks of this data type (similarly to coercive data types).

Figure 6 provides an example of a function definition in the metamodel, the **concat** block. As the **concat** block returns a **string** type, it inherits from the **string** data block in the metamodel. As it contains 2 pointers to the **string** block, we know that it accepts 2 string arguments. Also, as the **concat** block has attributes with the

same name as the pointers, the **concat** block supports user entered text as arguments.

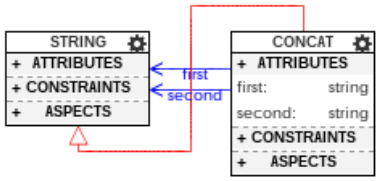

Figure 6: String Functions

3.2.3 Additional Concepts

Along with the given data types and a number of functions, the programming language utilizes a number of additional concepts to facilitate the creation of constraints. This includes standard elements of imperative programming languages such as if and if-else statements, while loops, repeat loops and for-each loops as well as concepts specific to constraint generation.

The constraint concepts include a number of different types of concepts from model traversal to model querying. For facilitating model traversal, the language contains concepts for loading the hierarchical children or parent of a given node as well as loading entire subtrees of the containment tree. Concepts for querying the names and values of a node's attributes and pointers are also provided. Additionally, concepts for filtering collections of nodes by type and marking constraint violations in the model are also supported by the language.

3.3 Constraint Generation

As the generated constraints are to be executed in the distributed context of WebGME, the constraint generator must support creating asynchronous code from the synchronous visual programming blocks. This generation of asynchronous code from the synchronous visual programming blocks is perhaps the most novel aspect of the constraint generation. Using this asynchronous code generation, the compiler enables the user to create more manageable, synchronous code using the blocks and hides the complexity introduced by the asynchronous nature of network programming.

Supporting asynchronous code function generation in the template requires support for moving subsequent code into a callback while preserving any 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 its 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, all 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.

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.

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.

3.4 Example Constraints

3.4.1 Unique Name Constraint

Figure 7 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 entry-point 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.

Given the visual constraint defined in Figure 7, the following Javascript code is generated [1].

```
function (core, currentNode, callback){
  "use strict";

  var names = [];
  var name = null;
  var node = null;
  var queue = [];

  getNode(currentNode, function(arg0_7){
    getDescendents(arg0_7, function(arg1_6){
      queue = arg1_6;
      var fn_1 = function(){
        var arg1 = Object.keys(queue);
        var arg2 = arg1[0];
        while(arg0_2[arg2] && arg1.length){
          arg2 = arg1.pop();
        }
        if (!arg0_2[arg2]){
          arg0_2[arg2] = true;
          node = queue[arg2];
          getNode(node, function(arg0_4){
            name = core.getAttribute(arg0_4,
                "name");
            if (names.indexOf(name) !== -1){
              violationInfo = {
                hasViolation: true,
                message: "duplicate names!",
                nodes: null
              };
            }
            if(getDimension(names) ===
                getDimension(name)){
              names = names.concat(name);
            } else {
              names.push(name);
            }
            setTimeout(fn_1, 0);
          });
        } else {
          callback(err, violationInfo);
        }
      };
      var arg0_2 = {};
      fn_1();
    });
  });
}
```

3.4.2 Equal Incoming/Outgoing Connections

Figure 8 presents a constraint which will verify that the given node has 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.

[1] In the generated code, there are some convenience functions defined prior to the variables, such as "getDimension". For brevity, these convenience functions have been omitted from this code snippet.

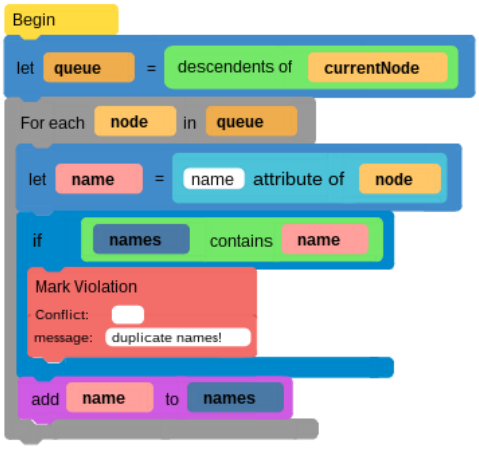

Figure 7: Block Pointer Example

This constraint demonstrates one basic extension that can be made to the visual constraint language given knowledge about the model. In Figure 8, 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).

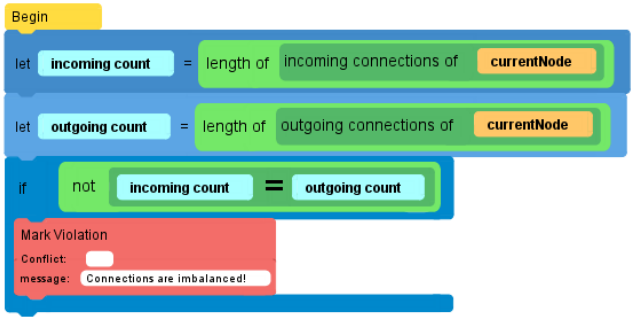

Figure 8: Equal Connections Constraint

Figure 9 shows the three boxes added to the metamodel to create the given extension of the visual constraint language. 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**.

Along with the modifications to the metamodel, these concepts will need to be added to the target language specification provided to the compiler. In this language specification, the names of the new visual blocks need to be provided with the corresponding code snippet. The code snippet for this example is provided below.

```
getIncomingConnections:
    'getConnectionsWith(\'src\', {{ node }},
    function('+placeholders.ARG(0)+') {\n'+
 '{{' + async.START + '}}' + placeholders.ARG(0)
     + '{{' + async.END + '}}'+
 '\n});',

getOutgoingConnections:
    'getConnectionsWith(\'dst\', {{ node }},
    function('+placeholders.ARG(0)+') {\n'+
 '{{' + async.START + '}}' + placeholders.ARG(0)
     + '{{' + async.END + '}}'+
 '\n});',
```

In this code snippet, "async.START" and "async.END" mark the beginning and end of the desired content in the asynchronous function and allow the compiler to hoist the earlier code (i.e., **length of** block) into the callback of the "getConnectionsWith" function. The argument placeholders mark the insertion of an anonymous argument to be created by the function callback. The number specifies that the given arguments are the same value; argument placeholders with different numbers will be given unique names to prevent name collisions. As with "getDimension" in the code generated in the previous example, "getConnectionsWith" is a convenience method for retrieving all connection nodes with the given source or destination.

Although this provides a simple example of the extensibility of the language, it also illustrates 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.

4. Conclusion

The paper introduced a constraint language for using visual programming blocks to represent model constraints to be evaluated in a distributed environment. The visual language is based on a proven notation that works well for novice programmers. Building our visual constraint language as a WebGME DSML provided a number of advantages over other visual programming languages including version control support 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.

Despite these benefits, the language is not always ideal for constraint creation. Complex constraints can be cumbersome to specify and working with visual language blocks is not always as flexible as writing textual constraint code. Also, the language does not currently support first class functions that can be called from within a constraint definition as in some other languages [9] [10]. Unfortunately, this can make the definition of complex constraints quite cumbersome.

The presented constraint language not only provides a useful tool for constraint definition within WebGME, but it also constitutes the basis for future research. First class functions would ease the effort required for the creation of more complex constraints. Also, modifying the base language to a logic programming

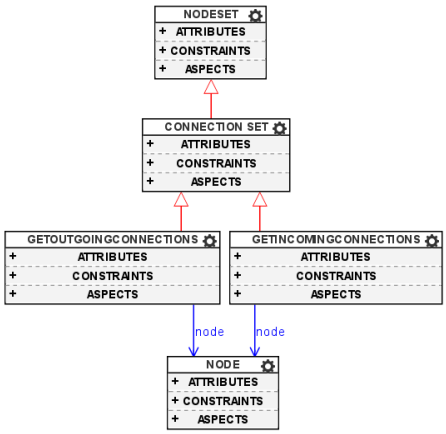

Figure 9: Metamodel modification

paradigm could potentially create a more concise constraint representation for some domains.

References

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

[2] 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.

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

[4] A. Fish, J. Howse, G. Taentzer, and J. Winkelmann. Two visualizations of ocl: A comparison. 2005.

[5] E. K. Jackson, D. Seifert, M. Dahlweid, T. Santen, N. Bjørner, and W. Schulte. Specifying and composing non-functional requirements in model-based development. In *Software Composition*, pages 72–89. Springer, 2009.

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

[7] Á. Lédeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001.

[8] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Levendoszky, and Á. Lédeczi. Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure. *Proceedings of MPM*, page 41, 2014.

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

[10] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52 (11):60–67, 2009.

[11] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4):110–111, 1997.

[12] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-37940-6.