Extensible Visual Constraint Language

Brian Broll & Akos Ledeczi,
Vanderbilt University
Background
Constraints

- Well-formedness rules for a DSML
- Typically written in OCL or Microsoft Formula

WebGME

- Next generation of the GME
- Runs in a web browser
- Component based
- Provides:
  - Scalability
  - Real-time collaboration
  - Version control
  - Custom data visualizations
Related Work
Other Visual Constraint Approaches

Visual OCL

- Logical, typed, object oriented
- Adheres to UML for simplicity

Constraint Diagrams

- Logical, typed, object oriented
- Similar to Euler diagrams
- More compact than Visual OCL
Extensible Visual Constraint Language

- Inspired by Scratch from MIT
  - Explicitly focuses on making programming more accessible
  - Imperative
- Implemented as a DSML
  - Supports domain specific customization
- Supports evaluation in distributed environment
Architecture
Components

- Composed of 4 main components
  - Metamodel
    - Defines language syntax
  - Model
    - Contains constraint instances
  - Visualizer
    - Provides the Scratch-like representation
  - Compiler
    - Transpiles the visual blocks to asynchronous Javascript
    - Implemented as a WebGME plugin
Language Syntax
Core Concepts

- Represents the basic structure of the elements
- Blue lines represent pointers in the metamodel
  - The visualizer will interpret these as connected or contained blocks
Data Types

- Data types are instances of a *Predicate* block
- Inheritance allows the child block to be implicitly casted to the base block type
Functions

- Functions inherit from their return type
  - This allows them to be used as their respective return type
- Pointers represent the input for the given function

![Diagram of function inheritance and pointers](image-url)
Control Flow

- Inheritance represents structural similarities
- The “true_next” pointer references the block contained in the parent which is executed if the conditional (target block of the “cond” pointer) is true
- Similarly, the “false_next” pointer references the block to be executed if the conditional is false
Constraint Generation
Code Generation

- Asynchronous support
  - Hoisting the necessary code into the callback
  - Deterministic evaluation of loops (without causing a stack overflow)

- Additional Features
  - Framework for testing new constraint code blocks
  - Variable hoisting
  - Promotes scalable constraint code
    - Lazy loading of WebGME nodes
Callback Hoisting

- In synchronous code, the generated code for a *predicate* block is the return value of the block
- Example:

```javascript
let nodes = children of currentNode;
// next block's code
```
Callback Hoisting

- In asynchronous code, the generated code for a *predicate* block will include the creation of the context in which the return value is defined.
- Example:

```javascript
getChildren(currentNode, function(children) {
    {{= RETURN_VALUE.START }}children{{= RETURN_VALUE.END}}
});
```
Callback Hoisting

- In asynchronous code, the generated code for the parent block is hoisted into the correct context (placed around the return value)
- Generated code for command blocks stores the location of the subsequent code
- Example:

```javascript
let nodes = children of currentNode

getChildren(currentNode, function(children) {
    nodes = children;
    // next block's code
});
```
Deterministic Loops

- Loops are converted to recursive functions
- To eliminate stack overflow, recursive calls are placed in a “setTimeout” call
  - This flattens the call stack
  - Each subsequent call is placed on the event queue at the end of the execution of the prior iteration

```javascript
var asyncLoop = function() {
  if (myConditional) {
    // Executed if "myConditional" is true
    setTimeout(asyncLoop, 0);
  } else {
    // Executed once "myConditional" is false
    // Subsequent blocks' code is hoisted here
  }
};
```

```javascript
while (myConditional) {
  // Executed while
  // "myConditional" is true
}
// Executed once
// "myConditional" is false
```
Example
Unique Name Constraint

```javascript
function (core, currentNode, callback) {
    'use strict';
    var names = [];
    var name = null;
    var node = null;
    var queue = [];

    getNode(currentNode, function(arg0_7) {
        getDescendants(arg0_7, function(arg1_6) {
            queue = arg1_6;
            var fn_4 = function() {
                var arg1 = Object.keys(queue);
                var arg2 = arg1[0];
                while (arg2.valueOf(arg2) <= arg1.length) {
                    arg2 = arg1.pop();
                }
                if (arg2.valueOf(arg2)) {
                    arg2[2] = true;
                    node = queue[2];
                    getNode(node, function(arg0_4) {
                        name = core.getAttribute(arg0_4, "name");
                        if (names.indexOf(name) !== -1) {
                            violationInfo = {
                                hasViolations: true,
                                message: "Duplicate names!",
                                nodes: null
                            };
                        } else if (getDimension(names) === getDimension(name)) {
                            names = names.concat([name]);
                        } else {
                            names.push(name);
                            setTimeout(fn_1, 0);
                        }
                    })
                }
            }
            callback(err, violationInfo);
        })
    });
    var arg0_2 = ();
    fn_1();
});
```
Questions?