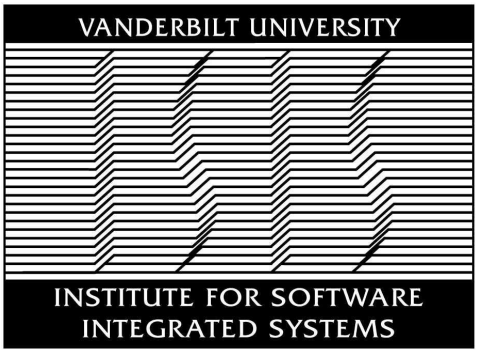


Institute for Software-Integrated Systems

Technical Report

TR#: **ISIS-18-101**

Title: **How to Build a Design Studio with WebGME**

Authors: **Patrik Meijer, Anastasia Mavridou**

Introduction

1 The Scope of this Tutorial **1**
- 1.1 Target Audience . . . 1
- 1.2 Prerequisites . . . 1

2 What is a Design Studio? **3**
- 2.1 Example Design Studios . . . 3

3 What is WebGME? **5**
- 3.1 Extensible . . . 5
- 3.2 WebGME-cli . . . 6

4 Other resources **7**

5 Dependencies **9**
- 5.1 Installing Node.js . . . 9
- 5.2 Installing MongoDB . . . 10
- 5.3 Git . . . 10
- 5.4 Redis . . . 10

6 The App Repository **11**
- 6.1 Installing webgme-cli . . . 11
- 6.2 Creating a Repository . . . 11
- 6.3 Installing the node_modules . . . 12
- 6.4 gmeConfig . . . 12
- 6.5 Starting the server . . . 12

7 Terminology **15**

8 Creating a Webgme Project **17**

9 What is Meta-modeling? **19**
- 9.1 Simple Containment Example . . . 19

10 Modeling in Webgme **21**
- 10.1 Data-model . . . 21
- 10.2 Restrictions Beyond the Meta-model . . . 22

11 Meta-modeling Concepts **23**

11.1 Containment . . . 23
11.2 Base Relationship/Inheritance . . . 23
11.3 Attributes . . . 24
11.4 Pointers . . . 24
11.5 Connections . . . 25
11.6 Sets . . . 25
11.7 Mixins . . . 25
11.8 Aspects . . . 26
11.9 Constraints . . . 26
11.10 Meta Properties . . . 26

12 Creating the Meta-model **29**
12.1 Which types do we need? . . . 29
12.2 Containment . . . 30
12.3 Sub-types of Components . . . 30
12.4 Connections and Ports . . . 32

13 Read-only Meta-model **33**

14 Model Interpreters/Transformations **35**
14.1 Plugins . . . 35
14.2 Add-ons . . . 35
14.3 Webhooks . . . 35
14.4 iCore . . . 35

15 Creating the Code Generator **37**
15.1 How to Implement the Interpreter? . . . 37
15.2 Generating a Plugin Template . . . 38
15.3 Registering the Plugin at Circuits . . . 38
15.4 Querying the Model . . . 39
15.5 Generating the Code . . . 40
15.6 Uploading the Generated File . . . 41

16 Integrating Analysis Tools **43**
16.1 Creating a process from the plugin . . . 43
16.2 Executor Framework . . . 43
16.3 WebGME Routers . . . 43

17 Result Presentation **45**
17.1 Generate an Artifact from the Plugin . . . 45
17.2 Store the Result in the Model . . . 45
17.3 Notifications while Analysis is Running . . . 45

18 Creating the Simulator Plugin **47**
18.1 Calling OMC from cmd/shell . . . 47
18.2 Generating the SimulateModelica Plugin . . . 48
18.3 Invoking the ModelicaCodeGenerator . . . 48
18.4 Simulating the Model . . . 48
18.5 Notes for Developers . . . 49

19 Deployment **51**
19.1 Systemd/Init-system . . . 51
19.2 Docker . . . 51

20 Authentication **53**

21 SSL/TLS **55**

22 Scaling Up Deployment **57**

23 Integrating a WebGME Design Studio with CPS-VO **59**
23.1 Step 1: Enabling authentication on your WebGME Server 59
23.2 Step 2: Linking the WebGME Server from CPS-VO 61

CHAPTER 1

The Scope of this Tutorial

This tutorial will guide you on how to build a Design Studio with webgme. If both of these terms are completely new to you, that's fine. It will also explain what these are.

The tutorial will go through all steps needed to host your own deployment, create your own meta-model, implement interpreters and customize the visualization. In order to give you an idea of how all these things play a roll in a Design Studio, these steps will be illustrated through an example of a simple electrical circuit domain. In short we will:

- Create a meta-model suitable for modeling electrical circuits
- Write an interpreter that generates Modelica code
- Extend the interpreter to simulate the generated Modelica code using OpenModelica
- Integrate the generated results with the model

1.1 Target Audience

This is for you, a researcher or an engineer, who might have seen an example of an application build up in webgme and are interested in building your application fitting your domain.

It also for you who just heard about webgme and are new to meta-modeling and would like to learn more about it.

1.2 Prerequisites

Although we try to explain the underlying technologies that webgme depend on, users are assumed to have a basic understanding of these. If not, it's recommended to use the internet and read up on the basics behind these technologies. In addition it's required that you have some experience in JavaScript. To customize the visualization basic knowledge of CSS and HTML5 is needed.

CHAPTER 2

What is a Design Studio?

A framework with a set of tools and services for simplifying one or many (typically engineering) tasks. Design studio components can be organized in the following three categories: 1) semantic integration, 2) service integration, and 3) tool integration.

Semantic integration Semantic integration components comprise the domain of the modeling language, i.e., its meta-model that explicitly specifies the building blocks of the language and their relations. This portion is covered in the **Modeling** section.

Service integration Service integration components include dedicated model editors, code editors, and GUI/Visualization components for modeling and simulating results. Additionally, service integration components include model transformation and code generation services, consistency and type checking mechanisms, model repositories, dedicated model interpreters, and version control services. Part of these are covered in the **Model Interpreters** section.

Tool integration Tool integration components consist in interfaces and integration services towards third party tools such as run-times and verification tools. This is covered in the **Integrating Analysis Tools** section.

The following figure shows the main steps of the design flow. Initially, models are designed using dedicated model editors. Optionally, design patterns stored in model repositories can be used to simplify the modeling process. Next, the checking loop starts (step 1), where the models are checked for conformance, optionally by using dedicated tools. If the required conformance conditions are not satisfied by the model, the checking mechanism must point back to the problematic nodes of the model in the model editor and inform the developer of the inconsistency causes to facilitate model refinement. Finally, when the conformance conditions are satisfied (step 2), the refined models can be analyzed (e.g. for safety properties) and/or executed (step 3) by using integrated, into the design studio, third party tools. The output of the tools is then collected and sent back to the model editors (step 4) for visualization of analysis or execution results.

2.1 Example Design Studios

DeepForge DeepForge is an open-source visual development environment for deep learning providing end-to-end support for creating deep learning models. This is achieved through providing the ability to design architectures, create training pipelines, and then execute these pipelines over a cluster. Using a notebook-esque api, users can

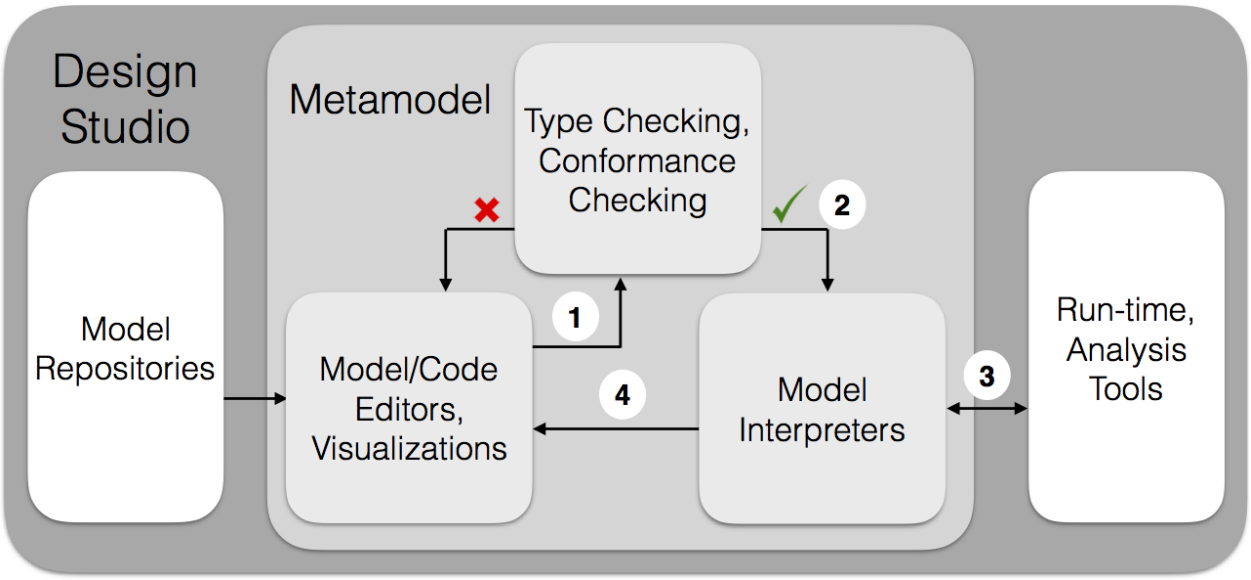


Fig. 1: Overview of a Design Studio

get real-time feedback about the status of any of their executions including compare them side-by-side in real-time.

DesignBIP DesignBIP is an all-in-one, web-based, collaborative, modeling and execution environment for building concurrent systems using the BIP (Behavior-Interaction-Priority) component framework. DesignBIP allows specifying BIP interaction and behavior models in a graphical way and generating the equivalent code. DesignBIP provides model repositories, design guidance services, code generators and integration with the BIP tool-set.

ROSMOD The Robot Operating System Model-driven development tool suite, (ROSMOD) an integrated development environment for rapid prototyping component-based software for the Robot Operating System (ROS) middleware. ROSMOD consists of: 1) The ROSMOD Server, which provides collaborative model-driven development, and 2) The ROSMOD Comm layer, which extends ROS to provide a more well defined component execution model with support for prioritization of component event triggers.

CHAPTER 3

What is WebGME?

WebGME is a web-based, collaborative meta-modeling environment with a centralized version controlled model storage. WebGME is a client-server based application, where both the client (browser) and server-side (NodeJS) use JavaScript. The clients carry quite a lot of the work-load and the role of the server is mainly to store and retrieve the raw model-data and propagate events between collaborating clients.

The model storage of webgme is influenced by git. All model objects and commits are immutable and all states in the evolution of a project are identified (and retrievable) via unique commit-hashes. Branches are light-weight mutable labels referencing one of these commits.

Clients do not load the entire model, instead they register and listens to events at territories (subtrees) within the model hierarchy. The major portion of the communication with the server is retrieving raw model-data. Therefore the storage model has been optimized to reuse as much of that data as possible; both over revisions (structural sharing of immutable data) and within the models (prototypal inheritance). In addition, when two versions of the data exist, the communication of changes is done using small patch objects (diffs).

This model allows for immediate small commits, which in turn minimizes the risk for accidental branching. Every change made using the GUI will create a new commit in the GIT-like storage. In the event of concurrent changes it's guaranteed that only one client will update the model state (branch hash). In such cases the other client has the option to attempt to merge in its changes.

3.1 Extensible

WebGME is made to be customized. Even though the generic GUI adapts itself based on the meta-model (the model describing the models), the visualization can be augmented and/or replaced on multiple levels; From small decorators displaying certain characteristics for nodes on the canvas, to completely swapping out the GUI and only use the Client API of webgme-engine as a library inside any front-end framework.

In addition to visualization webgme provides a framework for user-defined plugins - javascript code running either inside the browser or on the server (or both). These scripts (typically invoked by the end-users) are triggered on specific commits and contexts of the models and can work along side the users without any need for locking etc.

3.2 WebGME-cli

To ease the process of creating new components webgme provides a tool, webgme-cli, that generates boilerplate code and automatically updates the configuration so webgme can find and apply these.

In addition to this it also allows users to easily share and import components between repositories, on webgme.org a list of published extension components is updated every 15 min.

WebGME-cli will be used throughout the tutorial, however you don't need to install it right away. As you move forward detailed instructions will be provided.

CHAPTER 4

Other resources

WebGME-Seminar A youtube play-list with videos and screen capturing from a full-day webgme seminar in May 2015. The target audience for the seminar were people with some previous knowledge of meta-modeling.

Wiki pages This is the primary documentation hub for webgme. It is target towards people who already are using webgme and are trying to look-up more details about a certain area or extension point of webgme. Some of the pages there will be link to from this tutorial.

Source code documentation The API documentation generated from the jsdoc inside the webgme source code. This documentation is served from every webgme deployment. Once you're at the point of having your webgme-server running, you should be able to access the API docs for your currently running version at `<host>/docs/source/index.html`.

Sections of these will be referenced throughout the tutorial.

CHAPTER 5

Dependencies

One of the advantages of being a web-based modeling environment is that end-users don't need be concerned with installing any dependencies on their machine. All they need is a web browser! We aim to support all the major modern browsers. However we recommend using Chrome for a couple of reasons:

1. Manual testing is mostly done using chrome
2. Performance profiling is done against the V8 JavaScript Engine

As a developer of a webgme app you will however be required to host your own webgme server and for that you will need to install some dependencies in addition to having access to a browser.

- Node.js (version >= 6, CI tests are currently performed on versions 6.x, 8.x and LTS is recommended).
- MongoDB (version >= 3.0).
- Git (must be available in PATH).
- (Optional) Redis Note that this is only needed if you intend on running multiple webgme nodes.

5.1 Installing Node.js

When you have followed the instructions below make sure that the command below works and prints v8.9.4 or similar.

```
node --version
```

Windows Simply click on the link above and make sure to install the LTS! At the time of writing this that would be version v6.11.4. Alternatively install nvm (node version manager) which enables you to have multiple version of node installed.

Linux based operating systems (and macOS) On linux based systems it is recommended to install node using nvm (node version manager). It allows you to have multiple versions installed. The instructions below are borrowed from d2s' gist.

1. Open new Terminal window.
2. Run nvm installer with either curl or wget.

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.5/install.sh | bash
```

or

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.33.5/install.sh | bash
```

3. If everything went well, open new Terminal window/tab.
4. List what versions of Node are currently installed (probably none).

```
nvm ls
```

5. Install latest Node.js LTS release.

```
nvm install v8.9.4
```

5.2 Installing MongoDB

Webgme stores the models, project metadata and user info inside a mongo database. The Community edition works fine.

After you've followed the instructions and successfully installed mongodb. Either Launch a daemon (mongod) with the default options or pass the dbpath flag to store files at another location.

Windows

```
mongod --dbpath C:\webgmeData
```

Linux based/MacOS

```
mongod --dbpath ~/webgmeData
```

5.3 Git

For this tutorial you will need to have git installed. On linux/macOS this is typically already installed. Check by typing:

```
git --version
```

If not installed following the instruction at git's webpage.

5.4 Redis

This is optional and we won't be needing it for the tutorial.

CHAPTER 6

The App Repository

All of the code, configurations, images etc. for the web-app will be contained in a single repository. In order for webgme to pick up the correct configurations and locate the code there is an imposed structure on our repository. Fortunately, webgme-cli will automatically set this up for you.

6.1 Installing webgme-cli

webgme-cli is a tool for managing webgme apps. It provides a command line interface for creating, removing, installing from other webgme apps, (etc) for various webgme components.

Make sure you have node.js and npm (npm comes with node.js) installed.

```
npm --version
```

Now install webgme-cli as a global npm module (the flag `-g` ensures this)

```
npm install -g webgme-cli
```

If the installation was successful the following command should print a version number (the version of webgme-cli).

```
webgme --version
```

6.2 Creating a Repository

Pick a location on your file-system where you want your repository to be located (a new folder will be created).

```
webgme init electrical-circuits
```

Navigate into the newly created directory and you will see the content...

```
cd electrical-circuits
```

.gitignore Contains a list of patterns of files that will not be checked into the repository if using git as version control system.

app.js This is the javascript file that starts the webgme server. Note that it requires a mongo database server to be available at the specified mongo-uri from the gmeConfig. If you need to make some actions before start up - this is the place to put that code.

package.json The *package.json https://docs.npmjs.com/files/package.json* contain information for npm. It's main purpose is to store the dependencies of a module (this repo can be seen as a module and used by others). Notice that webgme is a peerDependency. The reason for this is that webgme-cli provides ways to share and reuse components between webgme repositories and in order to avoid multiple installations of webgme (when the versions do not match exactly) webgme is by default set as a peerDependency. If you're not intending ot share this repo you can move over the declaration under dependencies instead.

README.md It's always good to give a highlevel introduction about a repository together with some steps of how to run the app. This is the place to put it.

webgme-setup.json This is where webgme-cli stores meta-data about generated components of this repository. It should not be manually edited.

6.3 Installing the node_modules

In order to be able to launch the server you need to install all dependencies. From the root of the repository do:

```
npm install
```

In case you didn't move webgme to dependencies and it's still a peerDependency you need to explicitly install it...

```
npm install webgme
```

To check if the installation succeeded the following command should print a tree-like structure and include webgme at the root level.

```
npm list
```

6.4 gmeConfig

The configuration files in the `config` directory is where you set the configuration for your webgme app. By default the `config/config.default.js` and is the entry point of where you manually can overwrite the parameters. It loads `config/config.webgme.js` where webgme-cli adds parameters for the generated components (e.g. plugin). A full list of all possible parameters and how to switch between configuration files is documented here.

6.5 Starting the server

Make sure you have mongodb running. Look back at the dependencies section. As mentioned before the app.js file is the starting point for the webgme server. To invoke it with node.js simply do:

```
node app.js
```

There is a short-cut defined in package.json that lets you start the app by typing:

```
npm start
```

You should see some logging listing the port where webgme is available. By default this is 8888, so open up a browser and enter `localhost:8888` in the address bar. The webgme GUI should be loaded!

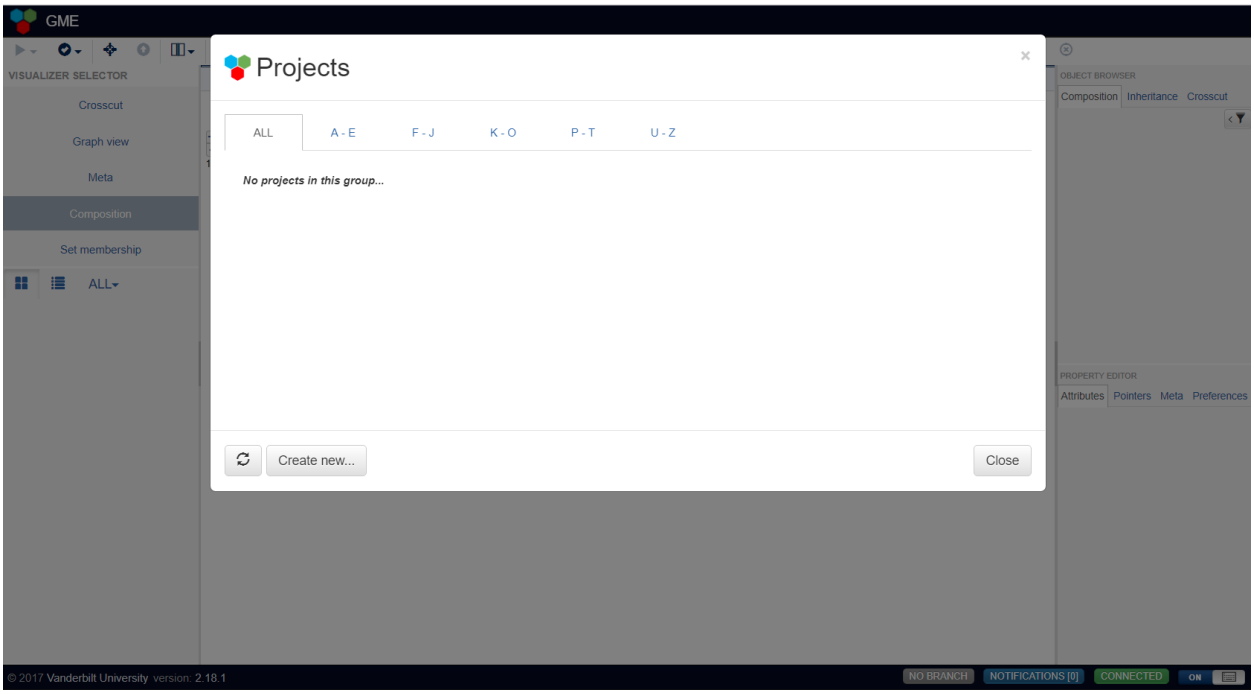


Fig. 1: The WebGME GUI

CHAPTER 7

Terminology

This is a list of webgme-terminology that will be used throughout the tutorial. It's recommended to at least skim through the list at this stage, but you can always return to it later.

Project Version (and access) controlled model repository containing (model)data-blobs, commits, branches and tags.

Commit Immutable object with a reference to a single state of a project and the ancestor commit(s) [of that state].

Branch Mutable label referencing a single commit. [Changes to branches are broadcasted to clients.]

Tag Similar to a branch, but it does not change. [It can be deleted and recreated but not updated.]

Node Atomic modeling element in the project tree.

Containment/Composition Single rooted tree among the nodes related as parent/children. Strong relationship.

Inheritance Single rooted tree among the nodes related as base/instance. Strong relationship that infers prototypal/prototypical inheritance.

Project Tree Combination of the containment and inheritance tree. [Where the nodes exist.]

FCO and ROOT node The FCO is the root of the inheritance tree and the ROOT is the root of the containment tree.

Meta-rules Definitions that governs the properties of, and relationships between the nodes.

Meta-node Node that is part of the MetaAspectSet (owned by the ROOT) and typically contains meta-rules.

Meta-type A node's first parent in the inheritance tree that is a meta-node is the meta-type of that node.

Relid Unique identifier of a child within a parent.

Path Unique position of a node in the containment hierarchy. [The '/'-concatenation of the parents' and its relids.]

GUID Immutable unique identifier of a node.

Library and namespace A project embedded within another project. Each library defines a unique namespace.

Core API API for querying and manipulating nodes that forms a state of a project.

Client API API on top of the Core API that also handles the evolution of a project.

Plugin Scripts invoked on a specific state of a project. [Typically use the Core API to interpret or transform models.]

Visualizer UI-component that visualizes the state and evolution of the project tree. [Typically uses the Client API to track the changes]

Decorator UI-component subordinate a visualizer that visualizes/decorates a single node. [The Model Editor and Part Browser defines their own APIs for controlling their decorators.]

CHAPTER 8

Creating a Webgme Project

Alright, looks like you have all the dependencies set up and your own webgme server running!

In order to get familiar with how modeling is done in webgme and what the different panels do, you need to create a project. If this is the first time you visit your deployment and there currently aren't any projects available, you should see a modal window titled `Projects`.

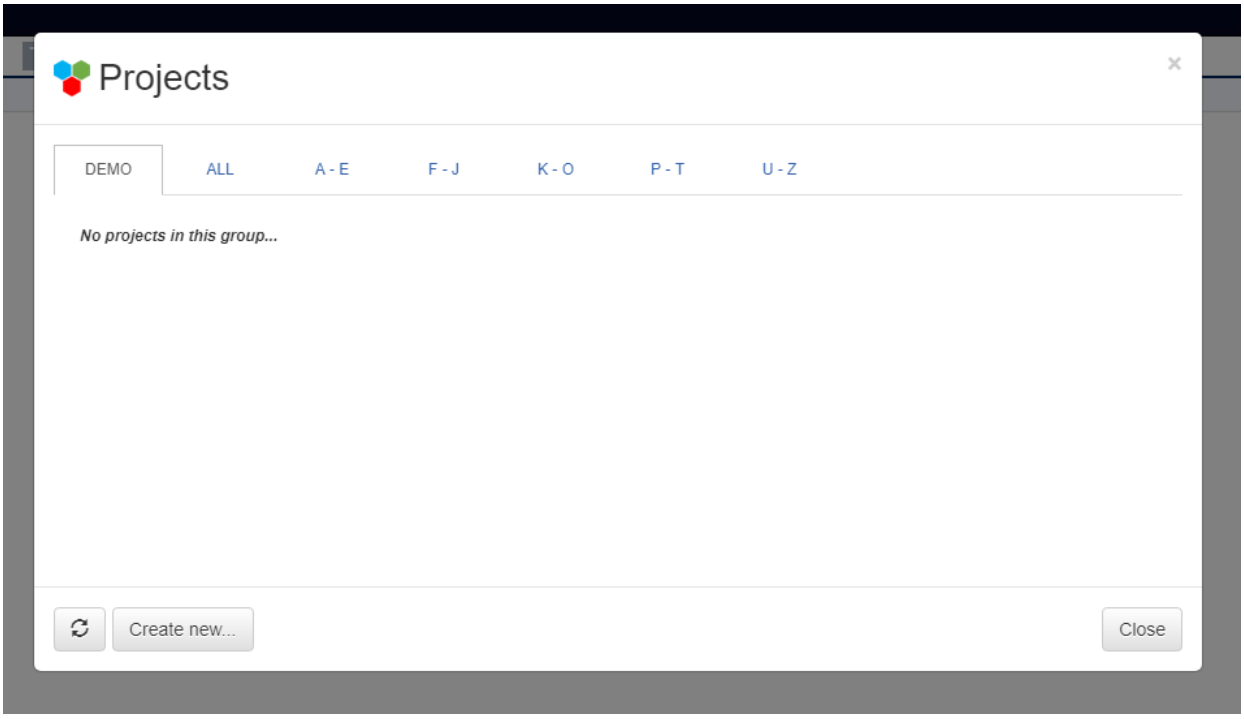


Fig. 1: Manage project dialog

In case you happened to have closed it or it doesn't show up for any other reason you can bring it up from the **Project Navigator**, by hovering the GME icon and clicking `Manage projects ...` (to save a click you can also click

`New project ...` right away.)

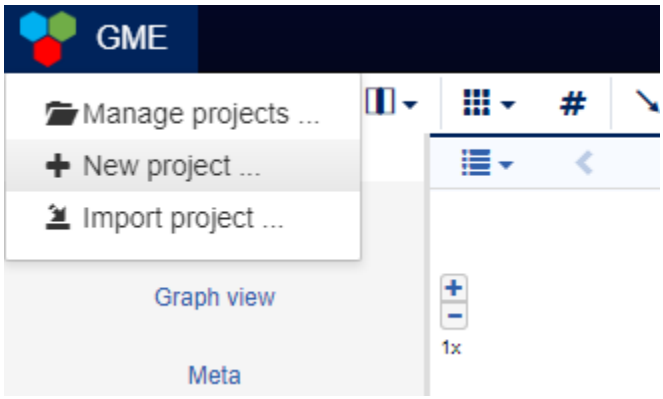


Fig. 2: Main menu in **Project Navigator**

Select a suiting name for you project, why not `ElectricalCircuits`, and proceed with the instructions. You should create a project from a seed and select the `EmptyProject` from one of the Template Files.

After this step you should have an "empty" project containing only the **FCO** and the **ROOT** node. Every webgme project contain these and they cannot be removed. (Looking back at the terminology page you'll see that these are the roots of the inheritance and containment trees respectively, and since both of these are **strong** relationships, the removal of any of these would remove all other nodes).

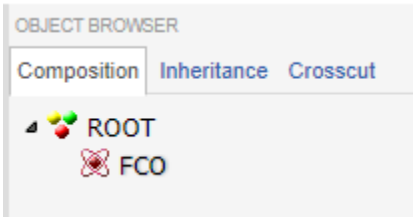


Fig. 3: The **Composition** tab in the **Object Browser** shows all the nodes in our project-tree.

Click here to open the video in a browser..

Note that authentication is currently not turned on your deployment and every user will be identified as the `guest`.

CHAPTER 9

What is Meta-modeling?

Alright, so you have created your first webgme project and probably want to make a bit more interesting than just containing the **FCO** and **ROOT** nodes...

In a typical modeling environment (including the Design Studio you will have built up at the end of this tutorial) there will be a syntax or schema for how models can be constructed. In many tools this syntax is implied and enforced by the source code itself. But webgme, and meta-modeling tools in general, work differently. The rules for how models can be composed (structural semantics) is captured in a model itself - the meta-model.

The following definition of meta-modeling is borrowed from Wikipedia[1].

```
A metamodel or surrogate model is a model of a model, and metamodeling
is the process of generating such metamodels.
```

So a meta-model is a model that governs how other models can be composed.

9.1 Simple Containment Example

The domain, or meta-model, we are targeting in this tutorial is a domain for building electrical circuits. Two obvious concepts that comes into mind are the notions of a `Circuit` and a `Component`.

- `Circuit` - A diagram where different types of electrical-components will be place and wired together.
- `Component` - An abstract base type for various electrical-components such as `Resistor`, `Ground`, `Inductor`, etc.

Conceptually, just the notion of these two meta-types constitutes a meta-model. So far we have declared that our models can have instances of `Circuit` and `Component`, however nothing has been said about how such instances can be related to one another...

A natural way of relating electrical-circuits and electrical-components is to say that circuits *can contain* components. Most meta-modeling environments have the notion of **containment** that defines exactly this. (It is typically visualized as an edge between the two types, where the end at the container is a black diamond.)

[1] This tutorial uses the hyphenated version of meta-model and not metamodel.

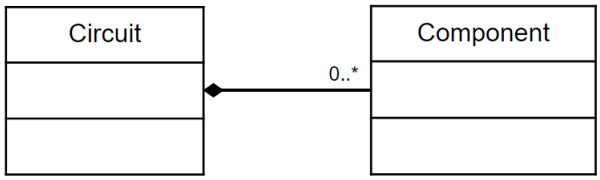


Fig. 1: Containment as depicted in UML

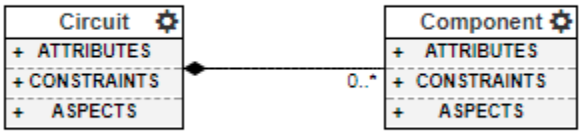


Fig. 2: Containment as depicted in WebGME

In the two graphical representations of **containment** the **cardinality** is also shown. For this example it says that there can be any number of `Components` contained inside of a `Circuit`.

Containment is also a strong relationship. This means that when the parent (the container) is removed from the model so are the children. This suites well for electrical-circuits - if we discard a circuit the components are discarded as well.

Model elements can be related in more ways than **containment**, in the next sections we will go through all the different meta concepts, meta rules, that webgme support. At this point you just need to have an idea of what meta-modeling is.

CHAPTER 10

Modeling in Webgme

OK, so in webgme you can make a (meta-)model that describes how models can be composed. That's all sounds pretty neat, but in order for that to work in practise some more concepts are needed.

- How do nodes (model elements) relate to each other in general, what are the restrictions?

Before going into the details of the data-model let's first have a look at the default GUI of webgme…

Click here to open the video in a browser..

10.1 Data-model

Leaving the adaptive UI on the side, it is completely possible to construct models using webgme's APIs that do not adhere to any specific meta-model. Below are the built-in concepts of the webgme data-model listed. Note that they are tightly coupled with concepts of the meta-models.

FCO "first class object", "object", "thing" every node is derived from this (except the **ROOT**)

Inheritance Single tree rooted at the **FCO**. Every node has a base, except the **FCO** and **ROOT**. Inheritance is a strong relationship: If the base is removed so are all derived nodes. Webgme uses prototypal inheritance.

Containment Single tree rooted at **ROOT**. Every node has a parent except the **ROOT**. Inheritance is a strong relationship: If the parent is removed so are all children nodes.

Pointer A one-to-one named association. The equivalent in UML is Directed Association.

Set A one-to-many named association. Similar to pointers but the owner can associate with more than one target. Also similar to containment by it is not a strong relationship. The equivalent in UML is Aggregation. In webgme sets can store data about their members.

Attribute/Registry Textual or numerical information stored at the nodes. Attributes adhere to the meta-model, whereas registries can hold any type data without meta-violations.

10.2 Restrictions Beyond the Meta-model

The data-model itself does set some restrictions on models in addition to the meta-model - these are mainly stemming from how prototypal inheritance is implemented in webgme. Consider a meta-type `A` that can contain instances of meta-type `A`.

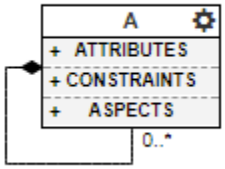


Fig. 1: Nodes of meta-type `A` can contain other nodes of meta-type `A`.

Consider two instances of `A`, `a` and `a'`. (Yes in webgme creating instances is not restricted to meta-types, and it's perfectly fine to create a new instance from any node.)

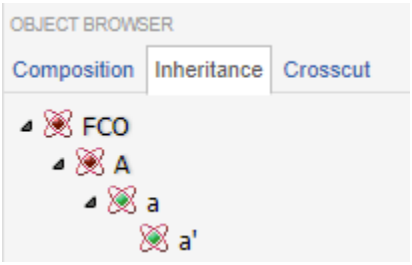


Fig. 2: `a'` is an instance of `a`, which in turn is an instance of `A`.

One immediate restriction (that should not be very surprising) is that node `a` cannot contain itself - a node can only exist in once place in the containment hierarchy.

In this scenario it is not possible to move `a'` into `a`, nor the other way around even though either case would not a violate the meta-rules.

Nodes cannot contain any of its instances Prototypal inheritance in webgme also includes structural inheritance - an instance inherits the nodes contained by its base. If `a'` were contained in `a` it would mean that `a'` would inherit an instance of itself as a child. This child would in turn inherit yet another instance of itself as a child, and so on until the people in the white lab enters...

Nodes cannot contain any of its bases This is due to how nodes are loaded in webgme. If you remember from the introduction, all nodes do not have to be loaded in webgme - instead subtrees are loaded on demand. Before a node is loaded all its parents and bases are loaded. In the case where `a'` were to contain `a` this look-up would result in a loop. To load `a` its parent `a'` would have to be loaded first and to load `a'`, its base `a` would have to be loaded..

In general the graph formed by the intersection of the nodes and union of the edges from the containment- and intersection-tree must form a tree (a graph with no loops). Luckily the webgme UI and the API ensures that this won't happen.

CHAPTER 11

Meta-modeling Concepts

The meta-meta-model in webgme describes the syntax for the meta-models. Below follows a short description and a video elaborating on each concept defined in the meta-meta-model. To follow this tutorial it is perfectly fine to skip **Sets**, **Mixins**, **Aspects** and **Constraints**.

This entire section can also be seen as a reference manual and it's recommended to return to these when creating the meta-model.

11.1 Containment

- Defines where in the containment hierarchy a node can be created
- One-to-many relationship
  - The cardinality can be restricted (e.g 0..* means any number of children)
- All nodes have a parent
  - Except the ROOT node which is the root of the Composition Tree
- Containment is a strong relationship: if a parent node is deleted so are all its children

Click here to open the video in a browser..

11.2 Base Relationship/Inheritance

- The base is a built-in pointer (one-to-one relationship)
- All nodes have base pointer defined

- All nodes (except the ROOT and FCO) have another node as their base
  - The FCO is the root of the Inheritance Tree (the ROOT lives outside!)
- We can modify the value of base in the Meta Editor
- The base/instance relationship is a strong relationship: if a base node is deleted so are all its instances
- A node's base is its prototype
  - Prototypal/Prototypical inheritance: asking for a property of a node traverses the base-chain till a value is reached

Click here to open the video in a browser..

11.3 Attributes

- Textual or numerical information
- Types: string, boolean, integer, float and asset
  - Assets can store a SHA-256 hash acting as an identifier for an artifact stored outside the model database
- name is by default defined on the FCO
  - Is used to identify meta-nodes
- Depending on the type, attributes can have different properties:
  - Default value - the value stored at the node defining the attribute
  - Read-only - the value can only be edited at meta-nodes
  - Enumeration, min, max, regex, multiline, etc.

Click here to open the video in a browser..

11.4 Pointers

- Named one-to-one relation between two nodes, the owner and the target
- Target of a pointer is either another node or NULL
- base is a built-in pointer (inheritance)

Click here to open the video in a browser..

11.5 Connections

- Not an actual concept of meta-meta-model
- Using reserved named pointers, src and dst, we can create connections
- Connections are a visualization technique on the canvas for nodes with pointers src and dst defined
- We can assign attributes, children etc. to a connection and we can create any number of connections from/to a node
- In the Property Editor we can see that it's just two pointers
- If any of the src or dst pointers are NULL, the connection appears as a regular box annotated with << Connection >>

Click here to open the video in a browser..

11.6 Sets

- One-to-many relationship between a set owner and members
- The cardinality can be restricted (e.g 0..* means any number of members)
- Has special visualizer: Set membership
  - Similar to Composition - but members are visualized/edited
- Meta-nodes are the members of the built-in set, MetaAspectSet, owned by the ROOT

Click here to open the video in a browser..

11.7 Mixins

- Meta-nodes can be used by other meta-nodes as mixins
- Meta-definitions are inherited from mixins
  - No actual data (e.g. attr values) of the mixin node is inherited
- A meta-node can have more than one mixin (but only one base)
  - In case of colliding definitions, the base node has precedence
  - Collisions among mixins resolved based on GUID

Click here to open the video in a browser..

11.8 Aspects

- Defines filtered views of your composition/containment (by selecting a set of valid children types)
- Defined aspects show up as tabs in the Composition view

Click here to open the video in a browser..

11.9 Constraints

- Functions defining constraints that cannot be captured by other meta-rules
- Such custom constraints are evaluated at the server and by default turned off
  - To enable `config.core.enableCustomConstraints = true;`
- Constraints will be evaluated for every node that is of the meta-type where

Click here to open the video in a browser..

11.10 Meta Properties

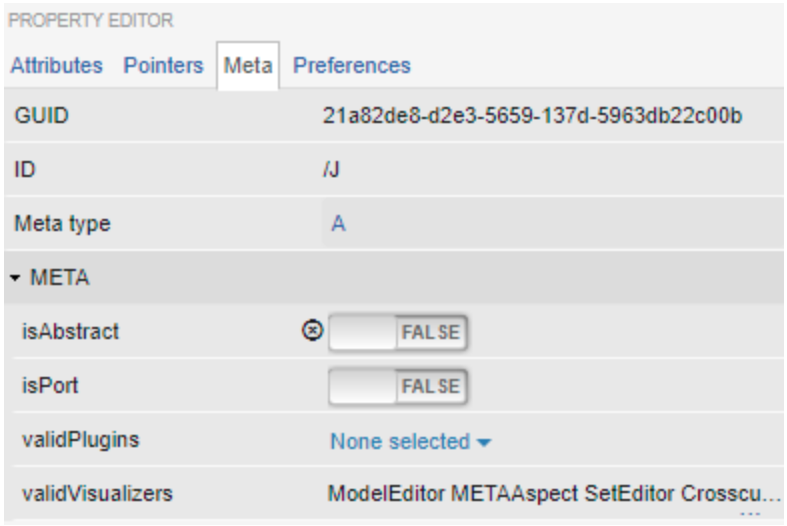
Meta properties are properties that are typically set on meta-nodes with main purpose to guide/constrain the end-user while modeling. In contrast to the other concepts they are not defined using the Meta Editor, instead they are set in META tab in the Property Editor.

isAbstract An abstract node cannot be instantiated or copied

isPort A port node will visually be elevated to the border of its parent.

validPlugins A list of plugins that can accepts the node as active-node.

validVisualizers Which visualizers should be listed for the node and which one should be opened when navigating to it.


PROPERTY EDITOR
Attributes
Pointers
Meta
Preferences
GUID
21a82de8-d2e3-5659-137d-5963db22c00b
ID
/J
Meta type
A
META
isAbstract
FALSE
isPort
FALSE
validPlugins
None selected
validVisualizers
ModelEditor METAAspect SetEditor Crosscu...

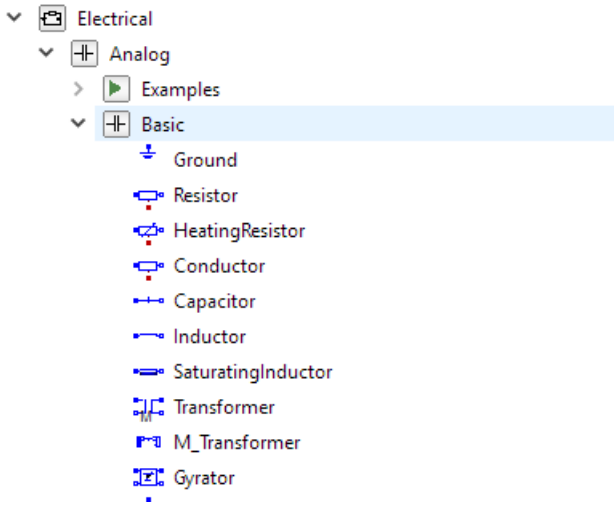
CHAPTER 12

Creating the Meta-model

At this point you should have a basic understanding of the concepts of meta-modeling in webgme and be somewhat familiar with the GUI of webgme. In this section we will build up a meta-model for the electrical circuits domain.

12.1 Which types do we need?

The first thing to layout when constructing a meta-model is what types our domain should include. Which these are does not only depend on the domain, but also the analysis tools being targeted. Since our target is **Modelica** and more specifically the Analog Electrical library as part of the **Modelica Standard Library**, we should take that into consideration.



This library contains a range of electrical components with a various number of pins. Looking at one of the examples we see that the pins are connected via electrical connections.

With or without Modelica a very natural breakdown of this domain is include the types: `Component`, `Pin`, and `Connection`. Additionally, since our goal is to build electrical circuits, we also need a `Circuit` type. Later

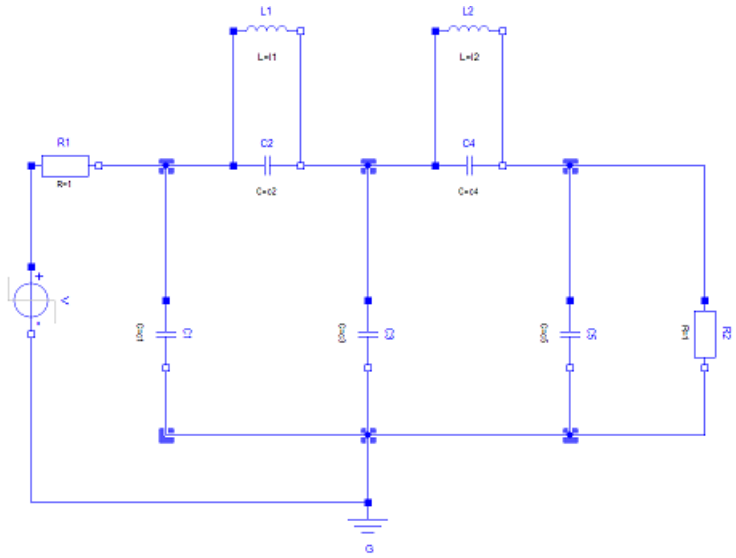


Fig. 1: A Cauer Low Pass Analog Circuit in Modelica

we will add sub-types of components corresponding to components such as `Resistor`, `Ground`, `Capacitor`, `Inductor`, etc.

The video below shows how you can add these types as meta-nodes starting from your empty project.

Click here to open the video in a browser..

12.2 Containment

Now let's model where these concepts can be added in the containment-hierarchy.

A `Circuit` should be able to contain `Components` wired together by `Connections`. The way connections are constructed in webgme requires us to add a containment rule for the `Connection` w.r.t. the `Circuit`. Next section illustrates how we can make the `Connection` in to an actual connection (an edge on the drawing canvas).

Inside the `Component` the `Pins` determine where the `Connections` connect the `Components` together.

The video below shows how to add these containment rules to our meta-model using the Meta Editor.

Click here to open the video in a browser..

12.3 Sub-types of Components

So far our meta-model only contains a generic `Component` for representing electrical components, but we need a way to represent specific electrical components such as `Resistor`, `Ground`, etc.

There are multiple ways we can achieve this by extending the meta-model. One apporach is to add a meta-type for each type of electrical component and add the related Modelica parameters as attributes to these respectively.

Another approach is to create the different types of electrical components outside of the meta-model and treat the Modelica parameters as separate child nodes of the `Components`. This approach allows for creation of new types without modifying the meta-model itself, but also makes the modeling a bit more cumbersome using the default visualization. (Visualizing and modifying the parameters of a component could no longer be done from a single node.)

For the sake of simplicity we will take the first approach and limit our domain to the five `Components` below (we will also leave out the heat transfer portion in the domain).

The associated Modelica parameters can be extracted from the **Modelica Standard Library** using a Modelica tool, such as OpenModelica. For each component we need to indicate its unique path or identifier within the **MSL**, this will be captured by the read-only attribute `ModelicaURI`. In order to map directly to Modelica we name the ports and the other parameters the same way they're named in **MSL**.

Resistor

Attributes

- `ModelicaURI` - `Modelica.Electrical.Analog.Basic.Resistor`
- `R` - The resistance of the resistor in Ohm. A float greater or equal to `0` with a default value of `1`.

Ports Two `Pins`, `p` and `n`.

Ground Modelica requires each electrical system (`Circuit` in our case) to contain a ground component in order to make the system solvable.

Attributes

- `ModelicaURI` - `Modelica.Electrical.Analog.Basic.Ground`

Ports One `Pin`, named *p*.

Inductor

Attributes

- `ModelicaURI` - `Modelica.Electrical.Analog.Basic.Inductor`
- `L` - The inductance of the inductor in Henry. A float greater or equal to `0` with a default value of `1`.

Ports Two `Pins`, `p` and `n`.

Capacitor

Attributes

- `ModelicaURI` - `Modelica.Electrical.Analog.Basic.Capacitor`
- `C` - The capacitance of the capacitor in Farad. A float greater or equal to `0` with a default value of `1`.

Ports Two `Pins`, `p` and `n`.

StepVoltage

Attributes

- `ModelicaURI` - `Modelica.Electrical.Analog.Sources.StepVoltage`
- `V` - The voltage of the source in Volt. A float with a default value of `1`.
- `startTime` - Time offset (when the voltage goes from 0 to `V`) in seconds. A float with a default value of `1`.

Ports Two `Pins` `p` and `n`.

With the approach taken the `Component` meta-type itself will not have any interpretation w.r.t. our domain and will only act as an *abstract* type that cannot be instantiated.

In the video an additional *abstract* base type, `TwoPinComponent`, that defines two `Pins`, `p` and `n` is also added. In general this approach is not only more elegant and convenient, but also more efficient since the raw data for the two pins can be shared and requires less data to be loaded from the server.

Click here to open the video in a browser..

12.4 Connections and Ports

In order to create connections between `Components` or rather between the `Pins` of the `Components` we need to make our `Connection` into a connection like object. In webgme's meta-model there is no first order concept of a connection, instead such can be constructed using reserved named pointers; `src` and `dst`. The target of each will be the source and destination of the `Connection` respectively. For more details on connections revisit the videos in the **Meta-modeling Concepts** section.

Just like with connections, there is no first order concept of a port in webgme either. Connection sources and destinations are only constrained by the valid pointer ends defined in meta-model and can crosscut over the containment hierarchy. To make modeling more comprehensible, it is often useful to be able to visually propagate port like children up to boundary of the parent node. The way this is solved in webgme is through the meta-property (implemented as a registry) `isPort`. Note that the usage of this property only takes effect if the decorator (the UI component responsible for drawing the box on the canvas) implements logic using this property - this is the case for both the default Model- and SVG-Decorators.

Click here to open the video in a browser..

At this point we have a complete meta-model and a small example `Circuit`.

CHAPTER 13

Read-only Meta-model

When designing a web app in webgme it may be preferred to provide the end-users with a meta-model that they cannot edit. Webgme has the concept of a library, which is a separate project that can be added to and linked from another project.

For detailed information see the documentation of libraries here.

CHAPTER 14

Model Interpreters/Transformations

14.1 Plugins

Plugins are custom extension points to a webgme-deployment that are intended to be used for e.g. querying, interpreting and building models. The framework and API are designed to enable both server- and browser-side execution of the same code. At the point where a plugin is executed it will have access to the context it was invoked and various webgme APIs, e.g. `Core`, `Project`, `BlobClient` etc.

Plugins are the typical starting point when adding interpretation to models and in this tutorial we will focus on how to create and write plugins.

14.2 Add-ons

Add-ons are extensions that run on the server and are triggered by changes made in the model. Registered add-ons are started when there is user-activity in a certain branch of a project and are kept running as long as (configurable) changes are being made. Add-ons have access to the same APIs as plugins (except the project API).

14.3 Webhooks

Webhooks are similar to add-ons but more loosely coupled. They can be triggered by different events on the webgme storage and the implementation requires a server accepting the post-requests sent out at the events. For more detailed documentation see the webgme-wiki pages.

14.4 iCore

iCore is a visualizer for webgme that enables editing and execution of code using the same APIs as a plugin directly in the webgme GUI. This is a good way to get familiar with the different APIs without the need to host your own

deployment. For larger projects/deployments and more advanced interpreters it's not recommended to rely on the iCore and it should be treated as an educational feature.

To get a quick start to this tutorial it is possible to complete the majority of the steps by creating the meta-model and using the iCore for implementing the code generator on webgme.org. Since the iCore does not have access to the server, it is however not possible to execute any simulation tool with this approach.

CHAPTER 15

Creating the Code Generator

So far we have constructed a meta-model for our electrical-circuit domain. It enabled us to create models that resembles circuits. Up to this point though there is no actual meaning to the circuits. As humans we can infer certain properties from these circuits, but the models themselves don't impose any particular interpretation. The meta-model infers the structural semantics, but no behavioural semantics.

There are many potential interpretations and interpreters of our circuit models. In this tutorial we will focus on generating Modelica code that can be used to simulate the dynamic behavior of the circuit in question.

15.1 How to Implement the Interpreter?

In webgme the typical extension point for writing interpreters are plugins. The plugin framework and API are designed to enable both server- and browser-side execution of the same code. At the point where a plugin is executed it will have access to the context it was invoked and through various webgme APIs; `Core`, `Project`, `BlobClient` etc.

In this tutorial we will create two plugins;

- *ModelicaCodeGenerator* - Responsible for traversing model hierarchy and extract the data needed for generating Modelica code corresponding to the circuit being interpreted.
- *SimulateModelica* - Responsible for invoking a Modelica tool that can simulate the dynamical behavior of the circuits. This plugin will also be responsible for communicating back the results to the users.

There a couple of reasons why this is a favourable division. To generate the Modelica code there is no restrictions on where the plugin is executed. The server does not have to have any 3rd party dependencies installed (or have any connected workers with available resources) and the plugin can even run in the browser. For some deployments restricting the set of features to only generate the Modelica code might be favorable. When it comes to writing tests it is typically also easier to divide functionality into separate implementations.

We will start with the *ModelicaCodeGenerator* here and continue with the *SimulateModelica* in the **Integrating Analysis Tools** section.

15.2 Generating a Plugin Template

To get a quick start we use the webgme-cli tool in order to create a new plugin. Navigate to the root of the repository created at the beginning of this tutorial and invoke the command below.

```
webgme new plugin ModelicaCodeGenerator
```

This should generate a range of new files..

src/plugins/ModelicaCodeGenerator/metadata.json This json-structure contains information about the plugin and is used by the GUI and plugin-framework. Details about what goes in here is explained in the wikipages.

src/plugins/ModelicaCodeGenerator/ModelicaCodeGenerator.js This is the code of the plugin itself. The very first lines show the dependencies needed for this code to run and is using requirejs hence the syntax `define(['path'], function (Module){ ... return ModelicaCodeGenerator; });`. The last return statement is the module that this file defines when required by another module (the plugin framework must be able to load our plugin).

test/plugins/ModelicaCodeGenerator/ModelicaCodeGenerator.spec.js This is the outline of a mocha test suite for the plugin and shows how to build up a test context and invoke a plugin from a unit-test.

You might also have noticed that the `config/config.webgme.js` was modified... In order for the webgme plugin framework to find our plugin the path to it is added to the configuration file. Note that both `config.default.js` and `config.test.js` load and reuse the added configuration parameters from this file.

The video below shows how to generate the new plugin and modify it so we have a map of all the nodes in the subtree of the `activeNode`. The `activeNode` is the invocation point of a plugin and in the next sub-section we will register our plugin so it's invokable at `Circuits`. (With the node map it is possible to retrieve nodes without any asynchronous function call - this makes the writing, and especially demonstration of the code easier. The asynchronous API functions in webgme do use promises which makes this a bit easier to deal with.)

Click here to open the video in a browser..

15.3 Registering the Plugin at Circuits

The generated plugin is available from the browser and the server, however in order to present it to the user on the GUI, we must register it at the appropriate nodes. In our case we want the *ModelicaCodeGenerator* to be invoked from nodes of meta-type `Circuit` so we edit the value at the meta-node and the registered value will propagate down the inheritance chain to all `Circuits`.

This video shows how we register the plugin and how we can enable the gme-logger for the *ModelicaCodeGenerator* in the browser. (Note that after updating the localStorage the page must be refreshed. The page must also be refreshed each time we update the plugin code on the server.)

Click here to open the video in a browser..

15.4 Querying the Model

At this point we have the context setup up for our plugin. The activeNode for the plugin will be a `Circuit` and all nodes in the sub-tree are pre-loaded in a map where keys are the path (a unique id) to the nodes and values are the node objects.

To extract data from the model we will be using the Core-API, and it's highly recommended to read the section on how to use the API at the link.

Before we start extracting the necessary data from the model we need to pin down what we need from our models in order to, in this case, generate Modelica code. The figure below shows the mapping from the `Circuit` to Modelica code and the related Core-API calls. (For simplicity we will leave out the Modelica parameters and use the default values from **MSL**.)

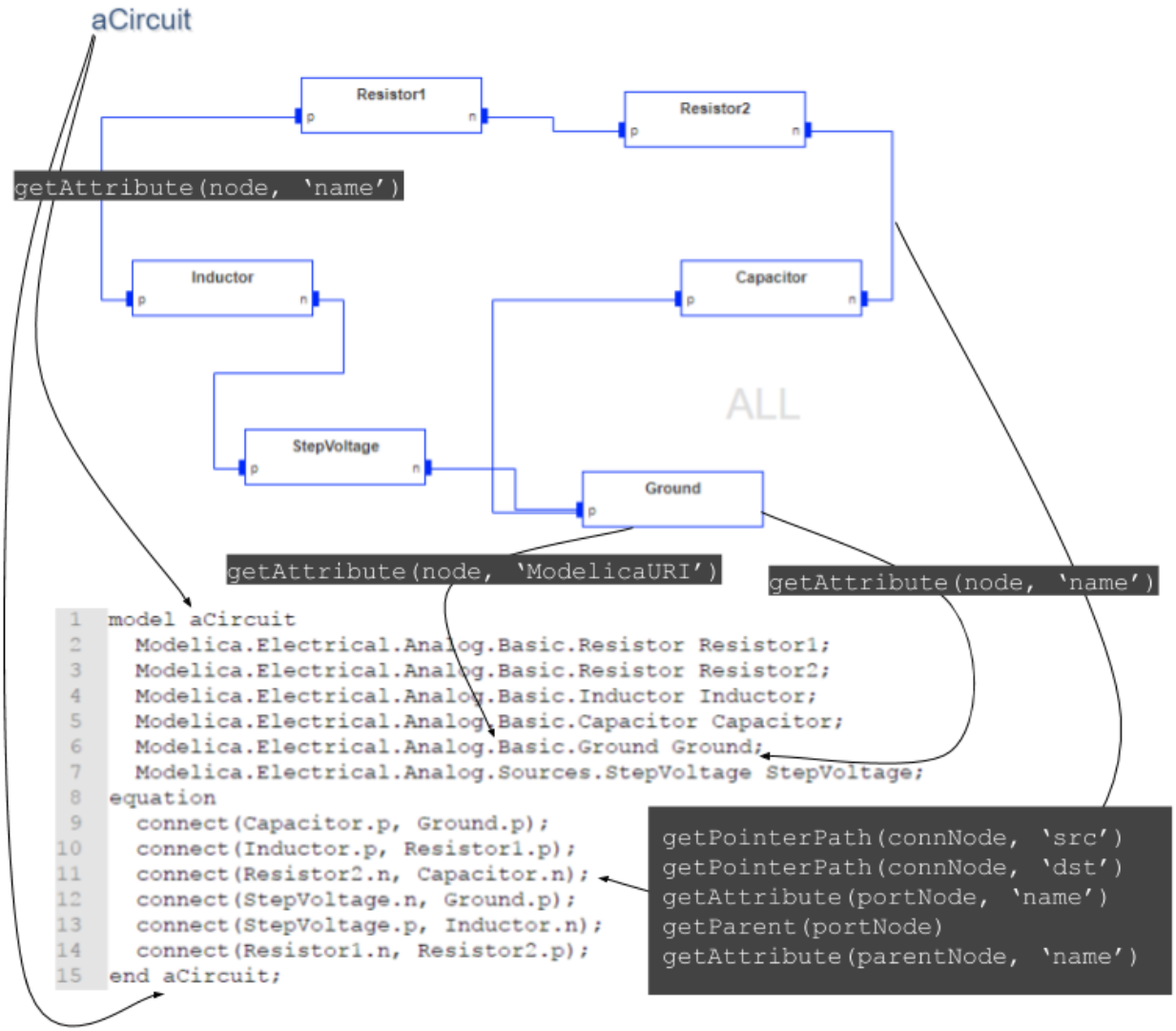


Fig. 1: Mapping from a Circuit in our webgme model to Modelica code

We will break up the task into two parts; 1) Extracting the data and 2) Generating the modelica code. The high-level outline of the first part is shown below in the code block where all the extracted data will be stored in serializable JavaScript object.

```
ExtractName(activeNode)
for all (Node child in activeNode) do
    if (child.metaType is Component) then
        ExtractNameAndModelicaURI(child)
    else if (child.metaType is Connection) then
        [srcNode, dstNode] = GetConnectedNodes(child)
        srcParent = GetParentNode(srcNode)
        dstParent = GetParentNode(dstNode)
        ExtractNames([srcNode, dstNode, srcParent, dstParent])
    end if
end for
```

The first video shows how to iterate over the children of the `Circuit` and check their meta-type. Important concepts here are; `self.META` property of the plugin which is a mapping from the name of a meta-node to the actual core-node, and the helper-method `isMetaTypeOf` which checks if the first node is of type of the second node.

Click here to open the video in a browser..

Next we need to implement the extraction of the data we need in order to generate the Modelica code. For this we will implement two helper functions that populates the `modelJson.components` and `modelJson.connections` array respectively. At the end we will serialize the data using the standard built-in `JSON.stringify` function.

Click here to open the video in a browser..

15.5 Generating the Code

In the previous section we extracted the data needed to generate the Modelica code in an easily accessible format...

```
{
  "name": "aCircuit",
  "components": [
    {
      "URI": "Modelica.Electrical.Analog.Basic.Resistor",
      "name": "Resistor2",
      "parameters": {}
    },
    ...
  ],
  "connections": [
    {
      "src": "Resistor2.n",
      "dst": "Capacitor.n"
    },
    ...
  ]
}
```

To generate the actual code we can use a templating engine such as ejs (an example of how to use this in webgme can be found here). There are also other more fitting templating engines available in JavaScript. For simplicity here, we will make a simple string concatenation to generate the Modelica code.

Click here to open the video in a browser..

15.6 Uploading the Generated File

After generating the modelica file we would like to make it available for the user to download. Webgme provides a storage for files through the blob-storage.

In the video below we will show how to upload the generated file onto the storage and how to provide a download link to the invoker of the plugin.

Click here to open the video in a browser..

CHAPTER 16

Integrating Analysis Tools

Whereas it is feasible for an interpreter to perform an analysis on its own - in a lot of applications the analysis is a performed by a separate program (could be a third-party tool). Typically it is the plugin that handles the triggering and monitoring of the analysis tool. The plugin also handles the result feedback to the user.

Depending on the type of analysis, the expected load on a deployment and requirements on result visualization, webgme offers a couple of different built in options for handling the analysis.

16.1 Creating a process from the plugin

The most straight-forward approach is to invoke the analysis tool by creating a child-process from the plugin. The major draw-back of this is that the analysis tool would be running directly on the same host as the webgme server.

One way to come around this is to replace the default server-worker-manager in webgme with the docker-work-manager.

In this tutorial we will simply assume that OpenModelica is installed on the host machine and invoke the compiler/simulator from the plugin. (The code we written here can without modification be used with the docker-worker-manager approach so we're not locking ourselves into a corner.)

16.2 Executor Framework

An other approach is to use the webgme executor framework, where external workers attach themselves to the webgme server and handles jobs posted by, e.g. plugins.

16.3 WebGME Routers

The approaches above both assume that the invocation point for the analysis job is a process call. This may not be the case for all analysis tools. In these cases the invocation call can either be directly implemented in the JavaScript code or the spawned process can handle the communication with the analysis service.

Alternatively a custom webgme router can be created to proxy requests and handle results. From such router, practically any tools or services can be accessed and integrated with the user interface.

CHAPTER 17

Result Presentation

There are multiple ways results from an analysis can be presented to the user. Here we list some approaches that can be used when running the analysis from a plugin.

17.1 Generate an Artifact from the Plugin

The most straight-forward way to present the results is to return a link to an artifact at the end of the execution.

17.2 Store the Result in the Model

The end result can also be stored back in the model and viewed at any time. In the case of long running analyses the invoker might no longer have their browser open to retrieve the results. A benefit of storing the result in the model is that the results will be version controlled and since they can be attached to the correct context - the evolution of the model and results can be traced.

For the Modelica simulation we will save the time traces from the dynamic simulation at the circuit and provide a download link to the results. With this approach a visualizer for presenting the results can be implemented, which would enable users to view the results embedded in the webgme GUI.

17.3 Notifications while Analysis is Running

The plugin framework in webgme supports sending notifications back to the invoker. These could be simple progress statuses, but could also contain partial results. The GUI displays messages like these in a console like notification widget, but the Client API allows any UI widget to listen to these and present the results in any manner.

CHAPTER 18

Creating the Simulator Plugin

This section will show how we can integrate the OpenModelica compiler (OMC) with the webgme. The example shown here is of course quite specific to Modelica, still the main takeaway is the pattern which can be reused for many analysis tools in general.

18.1 Calling OMC from cmd/shell

Download and install the open-source Modelica modeling and simulation environment OpenModelica. After the installation make sure the following command is working from an arbitrary directory.

Windows `> %OPENMODELICAHOME%\bin\omc.exe --help`

Linux/Mac `$ omc --help`

OpenModelica supports scripting of model-editing and simulation via mos scripts. From the `--help` output we see that these can be invoked by the command:

```
omc Script.mos           will run the commands from Script.mos.
```

The following mos code will load and simulate a circuit model and store the results in `aCircuit_res.csv` (details about the functions used is documented here.)

```
// Load Modelica Standard Library (MSL).
loadModel(Modelica); getErrorString();
// Load the generated circuit model.
loadFile("aCircuit.mo"); getErrorString();
// Simulate the model and generate the output as an csv file.
simulate(aCircuit, startTime=0.0, stopTime=1.0, outputFormat= "csv");␣
↪getErrorString();
```

Save the code snippet above in `simulate.mos` and place the generated `aCircuit.mo` file in the same directory. Providing you used the same name for your circuit confirm that the following command works and indeed generates the result file.

Windows `> %OPENMODELICAHOME%\bin\omc.exe simulate.mos`

Linux/Mac `$ omc simulate.mos`

Alright so we have a programmatic way of simulating our circuits. Now let's implement this code in a plugin!

18.2 Generating the SimulateModelica Plugin

Just like when we generated the *ModelicaCodeGenerator* plugin we again use the webgme-cli tool.

```
webgme new plugin SimulateModelica
```

This plugin will be responsible for the following tasks:

1. Invoking the ModelicaCodeGenerator to retrieve the modelica code
2. Generating a mos script that simulates the circuit from the modelica code
3. Calling OMC to execute the simulation
4. Reading in the results and storing them in the model

Since this plugin will execute commands on the server we need to enable execution of server side plugins in the gmeConfig.

In the plugin's `metadata.json` we will disable browser execution of the specific plugin, add a configuration parameter and register that *ModelicaCodeGenerator* is a dependency of the plugin. Additionally we register that the plugin requires write access to the project, that way users without write access won't be able to execute the plugin. For detailed info about the `metadata.json` documentation is available here.

Click here to open the video in a browser..

18.3 Invoking the ModelicaCodeGenerator

Plugin can be invoked from other plugins and the invoker will receive the results generated from the invoked plugin. The video below shows how to do this.

Click here to open the video in a browser..

18.4 Simulating the Model

At this point we have access to the model-content and a way to invoke OpenModelica from command line. We will create a unique directory on the server where the `.mo` and `.mos` files will be written out. After that we will execute the command using nodejs's child_process module. (From the same link documentation about the other built-in modules of nodejs can be found.)

The first video shows how to generate the files and the second one shows how to simulate and store the result in the model.

Click here to open the first video in a browser.. .. raw:: html

** **
<iframe width="560" height="315" src="https://www.youtube.com/embed/q9AS35VhAYg?rel=0" frameborder="0" allowfullscreen></iframe>

Click here to open the second video in a browser.. .. raw:: html

** **
<iframe width="560" height="315" src="https://www.youtube.com/embed/Ol8YqcNnSNs?rel=0" frameborder="0" allowfullscreen></iframe>

18.5 Notes for Developers

When developing plugins it is typically faster to execute the plugin directly from command line and much easier to debug server side code than running and restarting the server. The webgme bin script for running plugins is available and documented at `npm run plugin` (the script itself is located at `./node_modules/webgme-engine/src/bin/run_plugin.js`.

This tutorial has not touched on how to write tests for the plugins. Webgme provides a range of helper methods to build up the model context for a plugin, see the generated test files for some examples.

CHAPTER 19

Deployment

Once your application has reach a certain level of complexity and stability you probably want to host a centralized deployment where multiple users can connect and work together. In the end it is up to you how you solve it. This section contains common approaches and things to keep in mind when deploying webgme.

19.1 Systemd/Init-system

Detailed instructions on how to run webgme as a systemd service can be found here.

19.2 Docker

Another approach is to create a docker image of the webgme-app and run the webgme-app inside a container. Details information on how to create a webgme docker image and configure it to persist appropriate files outside of the container is documented here.

The docker image for the webgme-app hosted at webgme.org is published and publicly available at hub.docker.com.

CHAPTER 20

Authentication

By default authentication and authorization is turned off. This works fine for development and is really no restriction when the server is running on the user's machine. However once an actual deployment is setup up and users are starting to collaborate on projects - it's time to turn the authentication on.

The video below shows how to enable authentication and how the user schema works.

Click here to open the video in a browser..

For detailed instructions of how to turn on authentication see the instructions here.

CHAPTER 21

SSL/TLS

The WebGME server itself does not provide a secure connection. Instead we recommend using a reverse-proxy in front of the WebGME server that encrypts the data. Nginx is a commonly used one and an example configuration for a webgme-app can be viewed here here.

To obtain a certificate you can either create a self signed one using OpenSSL. If you have domain name Let's Encrypt offers free certificates.

CHAPTER 22

Scaling Up Deployment

Horizontal scaling of webgme servers is possible, but requires a some additional dependencies and specific configurations.

CHAPTER 23

Integrating a WebGME Design Studio with CPS-VO

This section is intended for you who have developed a design studio with WebGME and want to make it publicly accessible for users on https://cps-vo.org.

An integration between a WebGME design studio within CPS-VO simply means that a link to the design studio is available from a group on CPS-VO. Additionally the user's identity at CPS-VO is forwarded to the WebGME server. By disabling the option for users to register on your WebGME deployment, you can ensure that only people with user-accounts on CPS-VO (and access to your particular group) are able to logon to your deployment.

In order for this to work properly this documentation will guide you through the necessary steps on how to host such a WebGME deployment (most of these steps apply regardless of whether or not the deployment is integrated with CPS-VO.)

The steps are as follows:

1. Enabling authentication on your WebGME server
2. Allowing CPS-VO to authenticate users in WebGME.

In addition to these steps you need to setup a secure deployment. Notes on that can be found in the previous section.

23.1 Step 1: Enabling authentication on your WebGME Server

Before turning on authentication you must generate a set of RSA Keys. These are used to encrypt (the private key) and decrypt (the public key) the tokens containing the users identity. Sharing the private key with CPS-VO allows it to forward the users identity to WebGME. (One way of doing this is to set the token query when navigating to your WebGME site, e.g. `https://mywebgme.org?token=<tokenString>`.)

1. Create a new directory, e.g. `mkdir token_keys`, next to your checked out repository. (It's important that these keys are outside the cwd of the running WebGME server.)
2. Using openssl (available for windows here) a private key is generated with: `openssl genrsa -out token_keys/private_key 1024`
3. The public key is generated from the private key with: `openssl rsa -in token_keys/private_key -pubout > token_keys/public_key`

With the keys generated the next step is to configure WebGME to enable authentication/authorization. Since you probably want to have a different configuration for deploying the server then when you're developing your application it's advised to create a new configuration that appends to your default configuration.

Create a new file in your repository at `./config/config.deployment.js`, this section describes how to make sure it is being used.

Read through the section here and add the content to your new file.

```
// you can remove this line once you know this file is picked up
console.log('### using webgme config from config.deployment.js ###');

// require your default configuration
var config = require('./config.default');

// enable authentication
config.authentication.enable = true;

// by default non-authenticated users are authenticated as 'guest' this can be
↪disabled
config.authentication.allowGuests = false;

// this allows users created via the cps-vo route to create new projects
// which may or may not suite your deployment (but most likely you want this enabled)
config.authentication.inferredUsersCanCreate = true;

// disable user registration
config.authentication.allowUserRegistration = false;

// this assumes the keys are placed outside the webgme-app folder,
// as mentioned in the key generation section ../../token_keys directory
config.authentication.jwt.privateKey = __dirname + '/../../token_keys/private_key';
config.authentication.jwt.publicKey = __dirname + '/../../token_keys/public_key';

// finally make sure to export the augmented config
module.exports = config;
```

The full list of all available configuration parameters regarding authentication is available here.

It's always a good idea to have a site-admin account for your deployment which allows you to create, delete and manage access level of users, etc. from the WebGME profile page interface.

Since WebGME version 2.25.0 you specify an admin account to be created (or ensured to exist) at server startup. The following addition to the `config.deployment.js` above will create a user `admin` with password `password`.

```
// The password defined here will be visible so make sure to change it once the
↪server has
// been run at least once.
config.authentication.adminAccount = 'admin:password';
```

You should now be able to login to the profile page at `<host>/profile/login`, once logged in make sure to change the password from the WebGME profile interface.

Alternatively, or if you should forget the admin password, you can change the password from using the following command (make sure to set the correct environment variable for the configuration by setting `NODE_ENV=deployment`):

```
node node_modules/webgme-engine/src/bin/usermanager.js passwd admin newPassWord
```

For more details about authentication and authorization in WebGME these tutorials have a dedicated section.

23.2 Step 2: Linking the WebGME Server from CPS-VO

Share the private key with your contact at cps-vo.org and provide a url to your WebGME interface.