A Playground for the Modelica Language

Michael M. Tiller[1]

[1]modelica.university, michael.tiller@gmail.com

Abstract

This paper introduces a Modelica playground which allows users to experiment with the Modelica language without having to install any specific Modelica tools. This web-based application also contains content and lessons that provide users with a guided tour of the language and the opportunity for advanced users to create domain specific content built on top of this same infrastructure. This paper will explain the various open source technologies employed in creating this application and discuss potential future work to further enhance the experience for the user as well as the reach for Modelica itself.

Keywords: Modelica, education, interactive, animation, playground, web

1 Introduction

1.1 Playgrounds

To help "onboard" users, many programming languages include a web-based environment that allows users to see working fragments of code in that language. What makes such an environment a *playground* is that it allows these code fragments to be edited and compiled as well. This enables users to explore the language and understand at least the basics of different syntactic constructs without having to install any of the normal tooling associated with the language.

These playgrounds are not only useful tools for users to "try out" a language before committing to installing all the tooling, they are also very useful as educational tools. Such playgrounds often include examples of specific features of the languages. In a sense, they are used to help "sell" users on the design of the language or help explain difficult concepts by giving the users running examples (created by language experts) to help users understand the particularly idiomatic ways of accomplishing various tasks in that language.

The reality is that Modelica lags behind many other language ecosystems. This is, in part, due to a lack of resources. Modelica is, after all, something of a niche language. Nevertheless, this application was developed in part because applications like VPython (Bruce Sherwood 2022) are being used in a classroom context to teach students about math and physics through the use of 3D visualization. But using VPython is quite tedious compared to Modelica because users must implement all the numerical methods themselves. By creating the Modelica Playground, we hope to provide a better platform for students.

1.2 Goals

This project was developed with several goals in mind:

- **Freely Available**: As with all other content at `https://modelica.university`, this content is made freely available. The goal here is to support, to the greatest extent possible, those interested in learning the Modelica language. There are many tools out there with greater commercial resources than those in the Modelica community which is why it is important that the unique value and capabilities inherent in Modelica are demonstrated by material that is as accessible as possible.
- **Collaboration**: When using the Modelica Playground, users create models (and post-processing reports). This can involve a significant amount of effort. As such, it should be possible for users to easily save, share and publish their work.
- **Visualization**: For most programming language playgrounds, it is sufficient to simply capture output from the running program and display that. But Modelica is a modeling language and the "output" of Modelica code is (generally) time-varying simulation results. So in order for the user to fully comprehend what their code "means" in a mathematical sense, it is essential that visualization tools are available to bring those simulation results to life. Although there are many ways to visualize data in a web browser, we don't want the user to be required to become a frontend web developer with full knowledge of Javascript, HTML and CSS in order to craft their visualizations. For this reason, a no/low-code approach was taken requiring minimal amounts of imperative code to be written.
- **Extensible**: This application is about more than just teaching people Modelica. It is to provide a means to communicate ideas via Modelica code. This platform has been designed explicitly to allow ordinary users to create content that can be organized into "lessons" such that these lessons can be shared among users *without the need to edit the source code of the playground application itself.*
- **Privacy**: Cookie consent popups have become ubiquitous since the rollout of GDPR. While it is use-

ful to track the popularity of the tools hosted at `modelica.university`, there is no useful purpose in tracking users as individuals. As such, there is no cookie consent popup because there are no cookies being dropped in the user's browser. It simply isn't necessary to track users in order to accomplish this application's goals. Sorry Google.

2 Modelica Editor

The first aspect of the Modelica Playground that we shall discuss is the Modelica editor. The Modelica editor, shown in Figure 1, is built on top of Monaco (Microsoft 2022a), the code editor that powers Visual Studio Code (Microsoft 2022d), a widely used open source integrated development environment. The Monaco platform provides a playground of its own at: `https://microsoft.github.io/monaco-editor/playground.html`.

```
1   parameter Real e=0.8 "Coefficient of restitution";
2
3   constant Real eps=1e-3 "Small height";
4   Boolean done "Flag when to turn off gravity";
5   Real h "Height";
6   Real v "Velocity";
7
8  initial equation
9   h = 7.0 "Initial height";
10  v = 0.0 "Initial velocity";
11  done = false;
12 equation
13  v = der(h);
14  der(v) = if done then 0 else -9.81;
15  when {h<0,h<-eps} then
16    done = h<-eps;
17    reinit(v, -e*(if h<-eps then 0 else pre(v)));
18  end when;
19
```

Figure 1. Modelica code editor in Modelica Playground

Currently, the Modelica editor provides syntax highlighting as well as "error decoration" (both syntax errors and compilation errors). The implementation details of these features will be discussed shortly. Monaco itself is quite a powerful platform and hopefully other features that it provides will be incorporated in the future.

Before talking about syntax in more detail, it is important to point out one way that the Modelica Playground **deviates from most Modelica tools**. For most Modelica tools, the user compiles a `model`. The understanding is that this `model` will be instantiated implicitly by the compiler and that the `model` is, in some sense, the fundamental compilation unit.

As shown in Figure 1, the Modelica Playground doesn't take this approach because no root level restricted class is required. The reason for this is that by allowing the user to start with simple variables and equations, no previous knowledge about restricted classes or object oriented programming features are required. As a result, the typical code fragments found in the Modelica Playground read more like a program in an interpreted programming language like Python or Javascript. The goal in making this change is to lower the barrier of entry for new users and provide them with an initial context that is more familiar and intuitive.

2.1 Syntax Highlighting

Syntax highlighting is an essential requirement for any kind programming language renderer whether it simply be rendering source code on a page or implementation of a text editor. The Monaco system provides something called Monarch (Microsoft 2022b) for implementing syntax highlighting as a series of simple rules. The goal, with Monarch, is to avoid the need to implement a complete language parser and instead reduce the process down to one that can be accomplished with a collection of regular expressions. This can certainly be done with Modelica, but that isn't how syntax highlighting is implemented in the Modelica Playground.

Ultimately, syntax highlighting is simply about identifying the semantic significance of regions of text in the source code. While Monarch does this via regular expressions, the Modelica playground actually uses a full blown Modelica parser. Normally, this would probably be considered overkill. But there are two reasons this is reasonable in this case. First, the Modelica Playground doesn't deal with large quantities of code so the extra computational effort required to do a complete parsing of the code isn't really that significant and doesn't really impact responsiveness of the user interface. Second, the particular parser we are using is actually purpose built for these kinds of tasks.

The parser we are using was created using Tree-sitter (Brunsfeld 2022). There are two aspects of Tree-sitter that make it well suited for our purposes. The first is that it was developed specifically as an incremental parser. What that means is that it is designed to parse source code that is constantly changing. The typical use case that an incremental parser would concern itself with is syntax highlighting source code in a text editor. The goal is to quickly re-parse the source code after text has been inserted or deleted in a certain range. The parser itself is designed to reuse as much of the effort from previous parsing passes as possible. Although in a playground context where the source code is small, this is of minimal benefit. But the parser itself could be reused in other contexts with larger files. The other aspect of Tree-sitter that makes it well suited for our purposes is the fact that it compiles down to WebAssembly (WebAssembly Community Group 2022). It does this by first compiling a C language implementation of the parser and then using Emscripten (Empscripten Contributors 2021) to compile that into Web Assembly. The result is near native performance in the browser.

The resulting parser has been open-sourced as Modelica-tree-sitter (Michael M. Tiller 2022b). Because tree-sitter is developed and used by Github, its existence will hopefully lead to a future where Modelica source code is natively highlighted on Github (and perhaps other platforms).

2.2 Error Handling

Error handling comes in two varieties. The first is basic syntax errors. These can be easily detected by the same parser that is used to generate the syntax highlighting. Unfortunately, one of the current limitations in Tree-sitter is a lack of good diagnostic messages from generated parsers. As a result, the decoration of syntax errors in Modelica simply identifies the text where a syntax error occurs but doesn't provide much useful information beyond that. There are several open issues related to this topic associated with the Tree-sitter project and the authors appear to recognize these limitations. Hopefully future versions of Tree-sitter will address these limitations which could translate into better syntax error diagnostics in Modelica Playground.

The other type of error that Modelica Playground handles are compilation errors. These errors are more semantic in nature and are reported back from the OpenModelica compiler (Open Modelica Consortium 2022) used by Modelica Playground to compile the Modelica source code. Fortunately, OpenModelica errors include information about the text range for each error. And, unlike the syntax errors, they include considerable information about the nature of the error. All of this is leveraged by the Monaco platform in providing text decorations over the regions of text that, when hovered over (see Figure 2), elaborate on the nature of the error contained there.

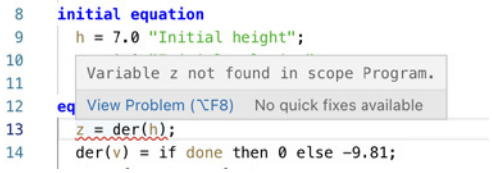

Figure 2. Semantic Error Highlighting

3 Simulation

Of course, editing Modelica code is just the beginning of what is required in order to engage readers with the Modelica language. The next logical component is to enable simulation of those models. For many language playgrounds, code is compiled and then run and the textual output of the program is captured and relayed back to the user. But in this case, we need to compile the code, run a simulation and relay the simulation results back.

3.1 OpenModelica

Let's start with the compiler itself. As previously mentioned, the Modelica Playground runs the OpenModelica compiler to compile code. Architecturally, the Modelica Playground application makes a request to an HTTP API asking that the model be simulated. The model source code is *included in the request*. This kind of an approach admittedly does not scale for dealing with large code bases. But because this is simply a "playground" (dealing with small fragments of code), it is acceptable. The backend is implemented using the Go language (which features its own playground, (Google 2022)). The API writes the source code to a temporary directory, running the OpenModelica compiler and then bundling the simulation results up in the response. Rate limiting is implemented using a worker pool in each server responding to such requests. These servers are themselves deployed as Kubernetes `Deployment` resources and, being stateless, can be scaled up as needed. The default number of replicas is two but a horizontal autoscaler could easily be associated with such a deployment to handle high load situations.

3.2 Results

For the moment, simulation results are handled in a fairly simplistic way. The compilation step requests output in `csv` format and the API parses that output and identifies which signals are constant at every time interval and which ones are not. The results returned in the simulation response segregate the signals accordingly. A better approach would be to output results in a more "sophisticated" output format, like the `dsres` format, that was more space efficient (*e.g.,* leveraging things like alias elimination). But again, the requirements for a playground are not so demanding.

4 Report Editor

A basic proof of concept of the Modelica Playground providing a basic text editor and the ability to request simulation results for Modelica source was put together in a day or two. But providing a high quality user experience takes much more effort. Apart from the syntax highlighting and error handling already discussed, adding functionality that provides attractive visuals and extensibility takes a lot more effort. In this section, we'll talk about how post simulation reports are generated and all the various possibilities the Modelica Playground provides developers of such reports.

5 Report Rendering

If the only purpose of the Modelica Playground were to allow users to compile Modelica code without needing to install tools, then generating simple tables and plots (which is the default behavior when no post-processing report is specified) would be sufficient. But this type of approach limits the kind of narrative that can be associated with a given model.

Since one of the goals of this project was to build a platform for users to create content that told a story about various models (and to stitch them together with some degree of structure), a richer capability was required. Furthermore, previous work has demonstrated that if the platform itself requires the underlying source code for the application to be modified in order to add additional content, this will put considerable constraints on who can add new

content and how it can be added.

For all of these reasons, the Modelica Playground was created with a post-processing report generation capability whose goal was to bring a no/low code approach to creating visual content. While MCP-0033 (Tidefelt and Tronarp 2020) proposes standard annotations for plots, the rendering functionality in the Modelica Playground goes well beyond (and could complement) that capability by opening up vastly more types of visualizations and interactivity. The following subsections will address the moving parts that make this possible.

As shown in Figure 3, the report tab in the Modelica Playground (found on the left) contains the purely textual source code for the post-processing report. On the right the Modelica Playground shows the rendered report (when simulation results are available).

5.1 Markdown

The heart of the report generation process is Markdown (John MacFarlane 2021). Markdown is widely used across the web as an easy to learn format for creating textual content. Various platforms have extended Markdown in different ways but Commonmark represents a fairly standard core which works reliably across different platforms.

Markdown brings standard markup support for text, paragraphs, images, font style, inline HTML, *etc*. This is the foundation for generating the reports, but it is simply the beginning of the transformations that occur. We have chosen the Remark (*Remark* 2021) and Rehype (*Rehype* 2022) tool chains because, as we shall see shortly, they can be quite easily extended via plugins.

These two tools by themselves allow the report textual description to be rendered on the fly in the right pane. This ability to immediately preview a report is not only useful for previewing how the text in the report will appear, it also works with all the extensions discussed in the remainder of this section which means the (report) content creator can preview mathematical equations, tables, plots and animations all in the context of simulation results. Every adjustment made to the report provides an instant preview.

5.2 Math

For rendering of mathematical equations, the Modelica Playground leverages the `remark-math` package (*remark-math* 2022). This provides both a `remark` and `rehype` plugin for parsing the mathematical markup in the Markdown content and rendering these equations using KaTeX (*KaTeX* 2022), respectively.

5.3 Custom Components

Another possibility with the `remark` rendering engine is to define custom components. As mentioned previously, Markdown allows HTML code to appears alongside Markdown syntax. But what the `remark` engine allows us to do is effectively "extend" HTML to introduce new element types and then gives us a hook by which we can render those custom elements.

Using this functionality, we define two additional specialized components. The first is the `<constants>` element. When rendered, the `<constants>` element will be replaced by a table that renders all constant variables found in the simulation results.

Another custom component provided by the renderer is the `<chart>` element. By default, the `<chart>` element will be rendered as a plot (using ECharts (Apache Software Foundation 2022)) containing all time varying variables in the simulation results. However, the `<chart>` element provides a `signals` attribute which, when supplied with a comma separated list of signal names, will display just the signals explicitly listed.

5.4 Templating

So far the rendering has been leveraging functionality that exists in `remark` plugins. But one challenge content creators may face is creating reports that leverage reuseable Markdown code fragments. Another challenge is the injection of information from the simulation into the report. The limitation of Markdown itself is that it doesn't actually provide any kind of templating functionality. So while it is excellent for describing content, it isn't designed at all for *managing* it.

This is where Nunjucks (Mozilla Software Foundation 2020) comes in. This is a templating engine written in Javascript and heavily inspired by the (Python based) Jinja (Pallets 2022) package. Nunjucks is a templating engine that allows us to define macros, variables, expressions and conditional constructs and in this way create reusable "units" of markdown as well as inject contextual information into the rendered report.

Note that the Modelica Playground is written in TypeScript (another language with its own playground, (Microsoft 2022c)) and leverages the React (Facebook 2022) framework. So one might wonder why is *another* system for creating reusable units of markdown required?

Why not simply use React components? This was certainly considered. For example, the MDX (MDX Community 2022) platform would have allowed us to mix React components into our Markdown code. But any solution that involves React *involves code*. Recall that one of the goals here is to have a no/low-code solution. Those creating content for this application should be able to do it easily without having to learn React or modify the source code of the application.

Nunjucks' learning curve was judged sufficiently easy to consider it for this purpose. It doesn't involve "linking" at all with the underlying application code and can be offered up and exposed to end users in a compartmentalized way that insulates them from the underlying application's architecture and technology stack.

Note that the template processing of Nunjucks is applied *before* the markdown processing. In this sense, we are using Nunjucks as a preprocessor.

In general terms, we are using Nunjucks to render a report. The report might make reference to constant values.

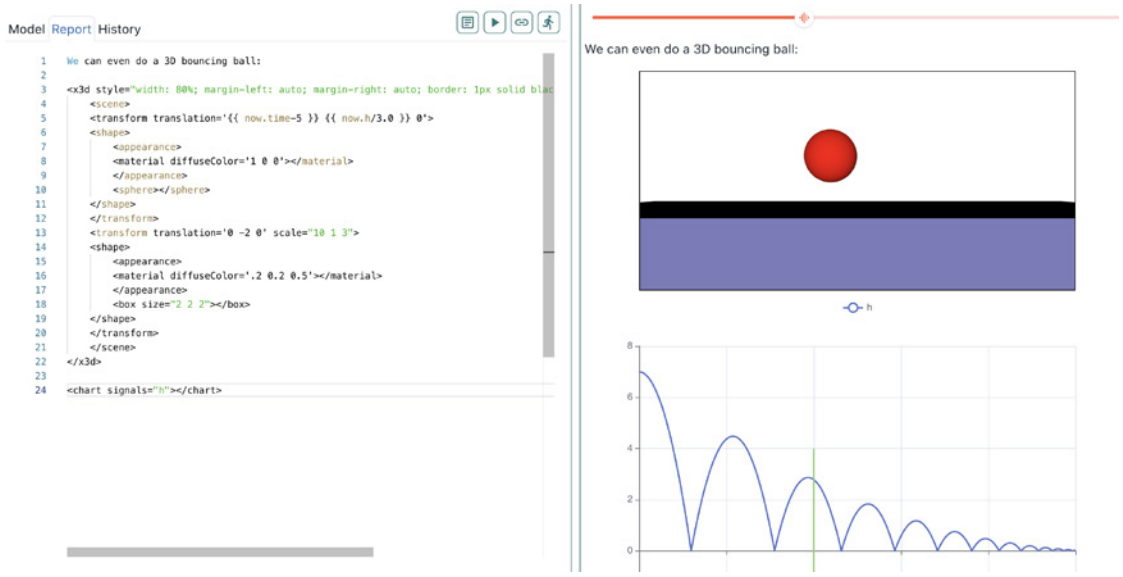

Figure 3. Report textual description vs. rendered report

In such cases, we can refer to those values in Nunjucks expressions by referencing the predefined `constants` object, *e.g.,* `constants.x`. Assuming *x* is an expression in our simulation results, `constants.x` will be replaced, during Nunjucks preprocessing, with the actual simulated value for *x*.

But what happens if the result we want to reference is time varying. Such results have different values at different times. Our Nunjucks preprocessor defines a special object referred to as `now`. So if *y* is a time varying variable, we can refer to the "current value" of *y* as `now.y`. Similarly, there is a built in function in Nunjucks called `at` and we can use that to refer to the value of a time varying signal *at* a particular time *e.g.,* `at(1.2).x`. Keep in mind the `now` objects relies on a notion of what the current time is. But how do we define "current value"?

The Modelica Playground application is equipped with a play/pause button and a scrubber controller. The application itself assumes, by default, that `now` represents the start time of the simulation. But by pressing "play" or dragging the scrubber control around, the value of `now` is automatically updated to the time associated with the current position of the scrubber. Any Nunjucks output that depends on the `now` variable is then automatically re-rendered.

This templating is particularly useful when dealing with the potentially verbose constructs associated with the visualization languages described next.

5.5 2D Visualization

Markdown, like Modelica and HTML, is a declarative approach to rendering. It doesn't involve imperative commands for how to render. Instead, it focuses on a description of what to render and leaves it to the tooling and the platform to perform the rendering task according to the specifications. In order to promote a no/low-code approach for 2D visualization, a similar approach was required. Fortunately, browsers have built-in rendering capabilities for 2D (and 3D) visualizations. In fact, the browsers include *two* such approaches. The first is the Canvas API (*HTML Canvas 2D Context* 2011). The problem with the Canvas API is that it is *not* declarative. Fortunately, the other option, Scalable Vector Graphics (*Scalable Vector Graphics* 2018), also known as SVG, *is* declarative.

For our purposes, it is not sufficient to simply render SVG. A normal Markdown processor can already do that. What is required for the Modelica Playground is that the SVG be rendered *as a function of the simulation results*. In other words, where numerical literals would appear in SVG to describe positions, rotations, scaling and transformations, we require the ability to replace those numeric literals with *simulation results*. This is made possible thanks to the Nunjucks preprocessor described in subsection 5.3.

Note that these numeric literals might arise from constants in our simulation results. But more often than not, they arise from time varying variables in our simulation results. In the former case, we can use the `constants` variable described earlier and in the latter case, we can use the `now` variable. In this way, any SVG figure that references the `now` variable is effectively transformed automatically into an animation.

5.6 3D Visualization

Just as with 2D animation, our requirement for 3D animation depends on the ability to provide a declarative representation of the 3D scene we wish to render and then describing it via our templating capabilities in order to couple it to our simulation results for the purposes of vi-

sualization and animation. However, unlike the 2D case, browsers have no built-in analog to SVG for declarative specifications of 3D scenes.

But this isn't the end of the world. The first thing to note is that browsers *do* have built-in, hardware accelerated, 3D rendering capabilities in the form of WebGL (*WebGL* 2022). Also fortunate for us is the existence of a framework called X3D (*X3D* 2022) that *does* provide a declarative scheme for describing 3D scenes. Even thought X3D isn't built-in to browsers, it can be loaded into any standard compliant browser so it is the next best thing to a built-in capability.

So once again, we can leverage the Nunjucks rendering to inject numeric values into a declarative scene description. And once again, references to the `now` variable automatically translate into animations of our now three dimensional scene and thereby satisfying our requirement for a no/low-code approach to visualization.

5.7 Vega

As mentioned previously, the custom `<chart>` component relies on ECharts for rendering the chart. ECharts is one of many different visualizations libraries available for the browser. Another is called Vega (*Vega: A Visualization Grammar* 2022). An important property of the Vega approach is that it provides a rich visualization grammar. Through this grammar, users are able to describe a wide ranging set of data visualizations going well beyond simple plots as shown in Figure 4.

Just as with SVG and X3D, we have a declarative vocabulary for describing a nearly infinite set of rich visualizations. To support this, an additional custom component was added, the `<vega>` component. This component can be used to delimit a JSON object that conforms to the expected structure of a Vega visualization. In such cases, the custom component will be replaced, during rendering, by the actual Vega visualization.

5.7.1 Safe HTML

Allowing users to define their own markup brings with it some risks. A modern browser is actually quite a powerful platform and it is the platform that is used for lots of other important tasks besides visualizing Modelica models and their results. As such, we need to ensure that the Modelica Playground doesn't expose users to any security risks.

Fortunately, the rendering toolchain for `remark` includes the `rehype-sanitize` package (*reype-sanitize* 2021). While it might seem tempting to blacklist specific HTML elements (*e.g.,* the `<script>` element) in order avoid introducing opportunities for security exploits, it turns out that blacklisting is impractical. There are simply too many ways in a modern browser to give people unwanted access if you allow users to simply type in "code".

For this reason, `rehype-sanitize` employs a whitelisting approach. What this means, in practice, is that it is necessary to identify explicitly all legal elements *and attributes of those elements* and `rehype-sanitize` will remove any references to any non-whitelisted elements. This is a tedious process, but it is one that was followed in producing the Modelica Playground. The result is that the Modelica Playground should be a very safe "sandbox" in which to play around with Modelica code.

6 Content

The default mode for the Modelica Playground is to present the user with a "blank slate" where they can type in any Modelica code they wish and create any post processing report. There is a table of contents that can be accessed that provides a few simple examples as a means of getting started, but the base application is deliberately quite open ended.

However, there is a mechanism by which specific models and reports or even collections of models and reports can be "published" using the Modelica Playground. In this section, we'll discuss how this is accomplished.

6.1 Links for Sharing

As a user, if you develop a particular model (and associated post processing report) that you would like to share with other users, you can click on the "Copy Link to Clipboard" button. Doing so copies a URL to the clipboard (the same URL visible in the browser's address bar, in fact).

This URL can then be emailed to other users. The URL will include query parameters that encode the text of the model and report. As a result, anybody who follows the generated link will be placed in the Modelica Playground with the associated models and report already pre-loaded. The results themselves are too bulky to bundle in with the URL. But since they can be reconstituted simply by pressing the "Simulate" button, doing so means that the link recipient will then see exactly the same visualizations that the original author saw.

Such link sharing could even be the basis for collaboration since participants could each modify the models they receive and send them back to the original developer. Similarly, professors could assign homework to students and request the solutions be done in the Modelica Playground and the students could then copy the link to their solutions into an email and send them back to the professor.

6.2 Lesson Plans

In some cases, users may want to share more than just a single model. Instead, they may wish to share a collection of models that, progressively, tell a story or explain a topic. In the Modelica Playground application, a collection of models and their associated post processing reports is referred to as a *lesson plan*.

Each lesson in a lesson plan can consist of five distinct parts:

- **metadata**: This is used to specify a title for each lesson as well as an ordering for each lesson.

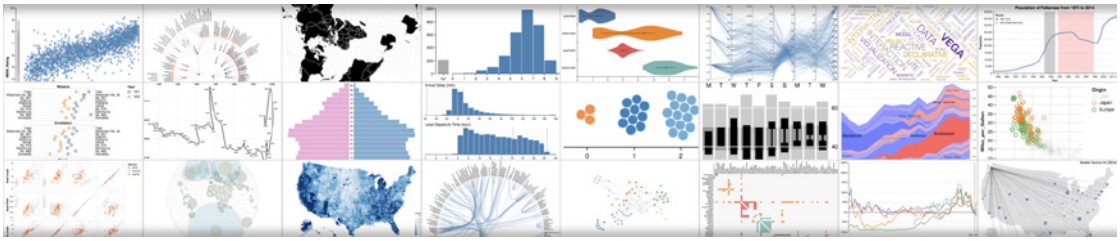

Figure 4. Samples of various Vega visualizations

- **a model**: The assumption is that each lesson will contain exactly one Modelica model.
- **explanation** (optional): The content of the explanation should be written in Markdown and will be rendered just above the model to provide some introductory context. Mathematical equations along with 2D and 3D figures may be used as part of this explanation but they may not reference simulation variables (since nothing has been simulated at this point).
- **a report** (optional): This report should be provided in the format described in Section 4. Unlike the explanation, it may reference simulation results since it is only rendered once simulation results are available.
- **macros** (optional): As mentioned previously, the model text and report text are encoded in each URL. This allows users to start from a lesson and potentially modify it for their own purposes. The creator of the lesson may have included predefined macros to be used in the post processing report. Since these are "static" (users aren't allowed to edit these), they are *not* contained in the URL. Instead, they are referenced as part of a templating "preamble" associated with the lesson itself.

Each lesson is composed of the various parts mentioned. These lessons are then bundled together into a lesson plan. The Modelica Playground expects this complete lesson plan to be bundled as a single JSON file that conforms to the Siren specification (Kevin Swiber 2017). But knowledge of Siren or the expected structure of that bundle are not required for content creators. Instead, they can use the `lessonplan` tool (Michael M. Tiller 2022a) to create such a file. Once created, the file *does not* need to be published on the `modelica.university` domain. Instead, it can be hosted *anywhere* and simply referenced via the `toc` query string parameter.

6.3 Examples

For reference, the following are examples of using the lesson plan functionality to create content. Hopefully, over time, users will start to create more such content.

- Lesson Plan Sample: The previously mentioned `lessonplan` tool used to bundle lessons includes, in its repository, an admittedly simple sample lesson plan. The bundled version of this lesson plan is hosted at `https://raw.githubusercontent.com/mtiller/lessonplan/master/sample.json`.
- Tour of Modelica: A more complete lesson plan is one that is bundled with the Modelica Playground. It can be found by clicking on the "Gallery of Lessons" in the upper right corner of the application. The goal of this lesson is to walk users through some of the basic functionality of Modelica.
- Content Creation Tutorial: This lesson plan is also available in the "Gallery of Lessons". Instead of teaching users about Modelica, this lesson is dedicated to teaching users about the Modelica Playground itself. It presents several examples that demonstrate the features discussed in this paper regarding post processing reports with the hope that, armed with this knowledge and combined with the documentation associated with the `lessonplan` bundler, users will create additional Modelica Playground content.

7 Data Management

As mentioned previously, one of the goals of this project was to avoid having to add cookie consent forms. This can be avoided so long as we avoid GDPR related concerns. Since this site is free and does not generate revenue in any way, we have no interest in tracking individual users. Doing so adds many more complications for absolutely zero benefit.

This has implications for how data associated with the application is managed and it is worth spending at least some time discussing this.

7.1 Analytics

It is quite common for web applications to include Javascript code that contacts some third party server to record the activity of visitors. This by itself is not a GDPR concern. It only becomes a GDPR concern when personally identifying information (PII) is recorded.

While we are interested in how many people utilize the site and what they utilize it for, we have no interest in being able to associate that activity with identifi-

able individuals. For this reason, we wanted to leverage a tracker that *did not* record such information. Fortunately, such analytics tools exist. The first on we tried was from `https://plausible.io`. This worked quite well and we would recommend it as an alternative to Google and other add targeting motivated trackers. Ultimately, we ended up using Cloudflare for our analytics. Like `plausible`, Cloudflare avoids recording PII data. It also has the benefit of being part of the Cloudflare platform which also includes many features related to content distribution, DDoS attack prevention and a whole host of other features.

It is important to note that analytics tools that do not track individual users do not benefit from the revenue associated with tracking users. As such, you should expect to pay for such tools since they do not pay for themselves by selling information about you to third parties.

7.2 History

The most recent addition to the Modelica Playground application is the introduction of user history. Every time a user runs a simulation, a record is made of that simulation. When you return to the Modelica Playground, you can access all your previous models, report templates and results.

Now it might seem like this must be a GDPR concern. But, in fact, it is quite easy to implement such functionality without violating the GDPR guidelines. The reason for this is that the information is not stored server side. Instead, the information is stored *directly in the users browser*. All modern browsers provide something called the IndexedDb API (World Wide Web Consortium 2021). This API allows applications to store data in a relational database directly on the machine that the browser was run from. Because the information *never leaves the users computer*, it doesn't violate the terms of the GDPR.

8 Future Work

Before wrapping up, it is worth some time to discuss potential future work to improve the Modelica Playground even further.

8.1 Improved Link Sharing

As already mentioned, the Modelica Playground allows users to capture their current work in the form of a special link. Such links can then be shared with other users via email, text message, Slack, *etc.*. But these links can be a bit problematic because they can be quite long. While this isn't generally an issue for the browsers (most browsers can tolerate very long URLs), it can be a problem for these various applications used to communicate the links because some applications impose their own limits on URL length. For this reason, users may find a "link shortener" useful.

Of course, there are many existing link shortening services and users are welcome to use those. The Modelica Playground links should work with any such service. But it is slightly inconvenient to visit a third party web site in order to create such a link (unless, of course, you have a browser extension installed that helps with that). An integrated link shortener could help with this.

The complication here is with respect to privacy. This would result in storing more user information. Furthermore, this information would have to be stored server side (unlike our current information which is all stored locally in the browser). Nevertheless, such a system could be made pretty easily GDPR compliant by simply being careful to only store the content but no information about the content creator. Most likely, the link shortener would simply store an association between a content hash and (only) the content.

Note, the same cannot be said for most existing link shortening services. By registering a link with them, you are implicitly opting in to allowing tracking of users who visit the shortened link.

8.2 Support MSL

Another welcome addition would be the ability to reference the Modelica Standard Library (MSL) from within the code written in the Modelica Playground. Although actually loading the MSL in the browser is probably well beyond the scope of practical improvements, it could be loaded server side prior to running the code. This would slow down simulations because loading the MSL takes a non-trivial amount of time. But it is quite possible that references to the MSL could be identified client side and the server could be told *a priori* whether or not loading the MSL was necessary.

Allowing references to the MSL would then allow the Modelica Playground to easily describe more complex models leveraging components available from the Modelica Standard Library. For the foreseeable future, such models would only be represented in their pure text form (*i.e.,* no diagram rendering). But even that should be reasonably intuitive for users.

8.3 Gist support

At the moment, all user work is stored in the browser. But another option for storage would be to store content in a Github Gist. In this case, the content would be available beyond the user's browser. While Gists can be marked "secret", they are still accessible to anybody who has access to the Gist id. So while this is possible, it wouldn't be implemented unless there was significant interest.

8.4 Simulation Caching

The current HTTP API receives the model content as part of the payload. The server could easily cache the simulation results of previous simulations of that particular model. For models presented in lessons (where the model is frequently run, unmodified), such a cache could improve the simulation time as perceived by the user (by avoiding the simulation altogether).

9 Conclusion

Despite being over 20 years old, Modelica remains a compelling technology. It is at least as relevant and useful now as it ever was. The goal of the Modelica Playground is to keep it relevant by making it as accessible or more accessible than alternatives.

By leveraging a variety of open source tools, the Modelica Playground provides a platform not only for exploring the Modelica Language online but also for creating compelling content showcasing Modelica along side interactive 2D and 3D visualizations.

Acknowledgements

This project would have been impossible without the availability of open source tools like the ones mentioned in this paper. This application is truly built on the shoulders of giants.

I'd also like to thank my daughter, Alisha Tiller. Her freshman year at Purdue has helped rekindle my passion for math and physics and I created this tool in large part so that students like her would have a Modelica based alternative to tools like VPython.

References

Apache Software Foundation (2022). *ECharts*. Version 5.3.2. URL: https://echarts.apache.org/en/index.html (visited on 2022-03-31).

Bruce Sherwood (2022). *VPython*. Version 7. URL: https://vpython.org/ (visited on 2022-04-24).

Brunsfeld, Max (2022). *Tree-sitter*. URL: https://tree-sitter.github.io/tree-sitter/ (visited on 2022-03-03).

Empscripten Contributors (2021). *Emscripten*. Version 3.1.9. URL: https://emscripten.org/ (visited on 2021-11-22).

Facebook (2022). *React*. Version 18.1.0. URL: https://reactjs.org/ (visited on 2022-04-26).

Google (2022). *Go Playground*. Version 1.18. URL: https://go.dev/play/ (visited on 2022-04-15).

HTML Canvas 2D Context (2011). URL: https://dev.w3.org/html5/2dcontext-LC/ (visited on 2011-05-24).

John MacFarlane (2021). *Commonmark*. Version 0.30. URL: https://commonmark.org/ (visited on 2021-06-19).

KaTeX (2022). Version 0.5.13. URL: https://katex.org/ (visited on 2022-04-13).

Kevin Swiber (2017). *Siren*. Version 0.6.2. URL: https://github.com/kevinswiber/siren (visited on 2017-04-27).

MDX Community (2022). *MDX Playground*. Version 2.1.1. URL: https://mdxjs.com/playground/ (visited on 2022-03-31).

Michael M. Tiller (2022a). *Lessonplan*. URL: https://github.com/mtiller/lessonplan (visited on 2022-04-30).

Michael M. Tiller (2022b). *Modelica-tree-sitter*. URL: https://github.com/mtiller/modelica-tree-sitter (visited on 2022-04-30).

Microsoft (2022a). *Monaco*. URL: https://microsoft.github.io/monaco-editor/ (visited on 2022-04-24).

Microsoft (2022b). *Monarch*. Version 0.33.0. URL: https://microsoft.github.io/monaco-editor/index.html (visited on 2022-02-03).

Microsoft (2022c). *TypeScript Playground*. Version 4.6.4. URL: https://www.typescriptlang.org/play (visited on 2022-04-28).

Microsoft (2022d). *Visual Studio Code*. URL: https://code.visualstudio.com/ (visited on 2022-04-24).

Mozilla Software Foundation (2020). *Nunjucks*. Version 3.2.2. URL: https://mozilla.github.io/nunjucks/ (visited on 2020-07-20).

Open Modelica Consortium (2022). *Open Modelica Compiler*. Version 1.19.0-dev.beta1. URL: https://openmodelica.org/ (visited on 2022-04-20).

Pallets (2022). *Jinja*. Version 3.1.x. URL: https://jinja.palletsprojects.com/en/3.1.x/ (visited on 2022-04-28).

Rehype (2022). Version 12.0.1. URL: https://github.com/rehypejs/rehype (visited on 2022-01-29).

Remark (2021). Version 14.0.2. URL: https://remark.js.org/ (visited on 2021-11-18).

remark-math (2022). Version 5.1.1. URL: https://github.com/remarkjs/remark-math (visited on 2022-04-24).

reype-sanitize (2021). Version 5.0.1. URL: https://github.com/rehypejs/rehype-sanitize (visited on 2021-12-08).

Scalable Vector Graphics (2018). URL: https://www.w3.org/TR/SVG2/ (visited on 2018-10-04).

Tidefelt, H. and O. Tronarp (2020). *Modelica Change Proposal MCP-0033 Annotations for Predefined Plots*. Tech. rep. Modelica Association.

Vega: A Visualization Grammar (2022). Version 5.22.1. URL: https://vega.github.io/vega/ (visited on 2022-03-25).

WebAssembly Community Group (2022). *WebAssembly Specification*. Version 2.0. URL: https://webassembly.github.io/spec/core/ (visited on 2022-04-27).

WebGL (2022). URL: https://www.khronos.org/webgl/ (visited on 2022-04-24).

World Wide Web Consortium (2021). Version 3.0. URL: https://www.w3.org/TR/IndexedDB/ (visited on 2021-10-06).

X3D (2022). URL: https://www.web3d.org/x3d/what-x3d (visited on 2022-04-24).