



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

BACHELOR THESIS

PPM WebCG

A visual programming environment for PPM

by

Joël Bohnes

Supervisor

Prof. Dr. Ivo F. Sbalzarini

Assistant

Omar Awile

August 14, 2012

Acknowledgments

I would like to thank my supervisor Prof. Dr. Ivo F. Sbalzarini for providing me with the opportunity to work on this project. A very special thanks goes to Omar Awile who provided me with ideas and support in many different areas during the course of this project. He also programmed the job handling part of the server application. I would also like to thank all of the MOSAIC group members for providing a comfortable atmosphere to work in.

Contents

1	Introduction	4
1.1	Motivation	4
1.2	Goal	4
1.3	Related Work	5
2	Overview	6
2.1	Client	6
2.1.1	Project Selection	7
2.1.2	Project Editor	7
2.1.3	Job Management	8
2.2	Server	9
3	Client Architecture	10
3.1	Modularity	11
3.2	Graph Data Structure	12
3.2.1	Graph Node	12
3.2.2	Attributes	12
3.2.3	Graph Edges	12
3.2.4	Dynamic Flow	13
3.3	Code Generation	13
3.4	Server Communication	13
3.4.1	Input Validation	14
3.4.2	Polling	14
3.4.3	Error Handling	14
3.4.4	Synchronized Data Structures	14
3.5	Application Controller	14
3.6	HTML User Interface	15
3.6.1	Views	15
3.6.2	Project Selection View	15
3.6.3	Property Editor View	15
3.6.4	Block List View	16
3.6.5	Job List View	16
3.7	SVG User Interface	16

3.7.1	Widget	17
3.7.2	Block Widget	18
3.7.3	Wire Widget	18
4	Server Architecture	19
4.1	Client Management API	19
4.2	Job Management API	20
5	Results and Future Work	21
5.1	Showcase	21
5.2	Results	23
5.3	Future Work	23

Chapter 1

Introduction

1.1 Motivation

Numerical simulations is of great significance in the scientific method. Advances in hardware development have lead to a huge increase computational power, allowing simulations to become more complex and sophisticated. This also leads to more challenges in how to program the simulations to efficiently use the resources.

The paper [7] discusses these challenges and presents a set of abstractions for formulating simulations. The parallel particle-mesh library (PPM), first presented in [8], is discussed as an implementation of these abstractions.

The PPM Client Generator (PPM CG) is a recent project to facilitate the workflow of building simulations (PPM Clients) based on the PPM Library. The PPM CG project provides a Domain Specific Language (DSL) to ease the programming of PPM Clients. It also provides tools to build, compile and run them.

With these powerful tools available we are always interested in how we can lower the barrier for creating high-performance simulations even further. This project investigates a different approach in how to create and manage a PPM client.

1.2 Goal

The goal is to provide an easy to use web interface that allows the creation, compilation and running simulations using the tools provided by the PPM Project. The interface exposes the PPM abstractions as individual blocks that can be connected together to form an PPM Client. The interface targets the DSL used by the PPM Client Generator. The interface should allow the user to build and compile the simulations and also run them. Furthermore it should provide options to retrieve the output generated by a run of a simulation.

1.3 Related Work

There exists a number of established, visual programming environments. *MathWorks Simulink*[1] is a framework for modeling and simulating different time-varying systems. It provides a graphical interface where the user can add blocks to a canvas, arrange, connect and change block properties to define the simulation. *National Instruments LabVIEW*[2] also provides a similar visual programming environment.

Web based programming environment are not widely spread at this time. There are some interesting projects that target the web browser. *Blockly*[3] is a general visual program editor. The program elements are visualized as blocks that can be snapped together and some blocks can be edited to customize their behavior. The editor uses SVG to render the blocks. A more specialized project is *ThreeNodes.js*[4]. Which is a visual dataflow editor for creating 3D scenes. It uses HTML and CSS for the user interface and can use WebGL to render the scene inside the browser.

Chapter 2

Overview

We separated our project into a server and a client. This allows the client to use create and run ppm programs without any installation. The only requirement from the user perspective is a web browser and a server that runs our server application. An HTTP server is used to serve the client application to the user's web browser. It is also used to act as gateway between the client and the server.

2.1 Client

Our client allows the user to create a ppm client by visually assembling building blocks. As such we require a high amount of interactivity between the user and the application. We target modern web browsers for easy deployment and platform independence. We programmed the client in javascript as it is the main language when targeting a web browser. The user only needs to point the browser to a URL where an HTTP server would provide the necessary files to run the client.

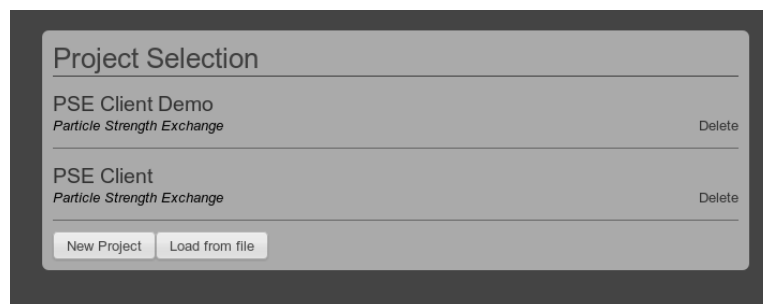


Figure 2.1: The project selection view.

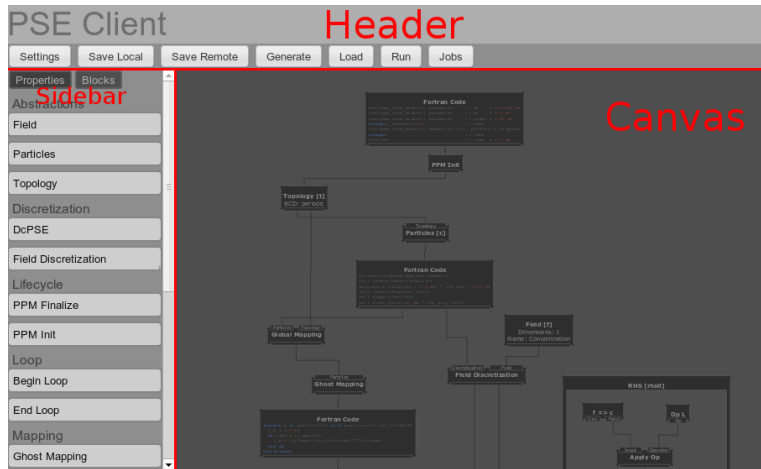


Figure 2.2: The project editor view.

2.1.1 Project Selection

The project selection view, as depicted in Figure 2.1, shows a list of stored projects on the server. It allows the user to load a project in the project editor by clicking on its entry. The user can also delete a project. The “New Project” button opens up the project editor with a new project. The “Load from File” buttons allows the user to load a stored project that has been saved locally. The list of projects is periodically updated from the server. Changes to the list of projects are highlighted.

2.1.2 Project Editor

The project editor, as seen in Figure 2.2, is the main area where the user creates a ppm client and performs various operations on them. The editor split into three areas. The header, located at the top, shows the project name and a list of buttons corresponding to different actions. To the left is the sidebar, which contains the property editor and the list of building blocks. The rest of the screen is an interactive canvas which contains the visual representation of the PPM Client. A building block is added to the canvas by dragging it from the block list to the canvas.

The Canvas

The canvas is the place where different building blocks are placed and connected together. A connection is also called a wire.

The canvas supports a number of features commonly found in similar programs, such as:

Panning

Moving the mouse while pressing the middle mouse button pans the viewport.

Zooming

Scrolling up or down causes the viewport to zoom in resp. zoom out.

Selecting

Clicking on a block causes it to be selected. By holding shift multiple blocks can be selected.

Rectangle Select

Dragging on the canvas using the left mouse buttons creates a rectangle that selects all blocks or wires it intersects.

Deleting

Pressing the delete key deletes all currently selected edges.

Dragging

Dragging a block with the left mouse button causes all selected blocks to be moved.

2.1.3 Job Management

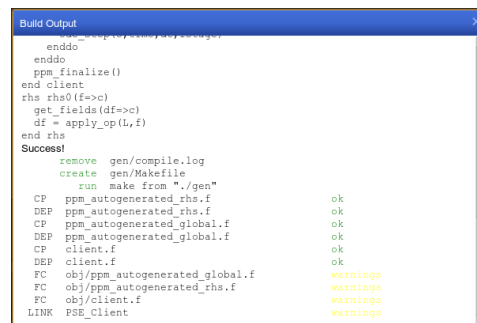
A screenshot of a 'Build Output' window. The window has a blue title bar with the text 'Build Output' and a close button. The main area is white and contains text showing the compilation process. It starts with 'Success!' and lists several steps: 'remove gen/compile.log', 'create gen/Makefile', and 'run make from "/>

Figure 2.3: Build output

A job is a execution of a binary generated by the PPM Client Generator. The binary is created by pressing the “Generate” button which displays the generated code and compilation status, as seen in Figure 2.3. A job is started when the performs the “Run” action in the project editor. The user can view a list of all jobs by pressing the “Jobs” button. This list includes information about when the job has been started, when it has ended and what its current status is. A running job may be terminated early by performing the “Kill” action on the job list. Additionally the output (stdout and stderr) generated by a Job may be viewed. Both the list of jobs and the output of a job are

periodically polled to keep it synchronized with the server and changes are highlighted.

2.2 Server

The server is a web application written in the Ruby programming language using the Sinatra framework. It provides a RESTful API to the Client and offers the following functionality:

- Creation, Loading and Storing of PPM Projects
- Listing stored PPM Clients
- Building executable binaries using the PPM Client Generator
- Running the built executable (called a job)
- Listing currently running jobs
- Killing a running job
- Viewing the output generated by a job

All storage is done using a simple, file-based database structure.

Per default the server launches with Ruby's webrick HTTP server, which allows easy testing of serve without further configuration. Sinatra uses the Rack webserver interface. This allows the application to interface with web servers using common protocols such as cgi or fastcgi.

Chapter 3

Client Architecture

The client is fully written in javascript. We use the following technologies offered by the browser to implement the client functionality:

SVG 1.1[5]

We use SVG to render the visual representation of the ppm components (blocks) and their connections (wires).

HTML

The user interface, with exception of the canvas, is constructing using HTML.

CSS

CSS is used to style the user interface. Both HTML and SVG can use CSS. In HTML we use it to specify colors, fonts as well as placement. In SVG we use it to declare fonts, stroke and fill properties.

AJAX

For non blocking network communication we use AJAX.

DOM

In the browser the user interface is represented as tree. For our application this tree consists of HTML nodes and SVG nodes. DOM is the interface that allows the javascript to manipulate to tree data structure.

JSON

The JavaScript Object Notation is a widely used serialization format which we use for exchanging data with the server and storing data. We use the native ECMAScript 5 JSON encoder/decoder.

ECMAScript 5

This is the programming language we use.

We depend on two third-party javascript libraries:

YUI

Provides various utilities to facilitate javascript development. This includes a framework for modularization, an event system, DOM manipulation and basic support for object oriented programming.

CodeMirror

Provides support for syntax highlighting when editing fortran/ppm code.

The user interface targets modern web browser. We separated the user interface from the application logic. This makes it possible to reuse the code in future projects.

3.1 Modularity

Javascript doesn't have any concept of modularity, that is splitting source code into smaller, self-contained files. Still modularity is of great importance in software development and thus many javascript libraries provide their own framework to facility that. We use the YUI Loader framework to split our source code into multiple files. Each file represents a module. A module is defined via a `YUI.add` call containing the following arguments.

Module Name

An arbitrary name, should not conflict with any other module name.

Function

A function that is executed once after all the dependencies have been loaded, the function gets a pseudo-global YUI object where the module can export its functionality. Using the pseudo-global object avoids using the global namespace, allowing multiple instances of an application.

Dependencies

A list of names of modules that should be loaded before this one

Version

Version number, currently not used.

These modules can then be loaded via YUI. The dependencies are automatically loaded - in a browser this is done by inserting script nodes into the DOM tree.

3.2 Graph Data Structure

We have chosen a dataflow like approach to represent the PPM Client. The dataflow is represented as a graph which consists of two parts. The first part is a set of nodes representing data of transformation on data. The second part is edges representing the flow of data from one node to another. The graph data structures provides event when nodes are added/removed or edges are added/removed. This is used by the user interface to synchronize with the data structure.

3.2.1 Graph Node

A graph node is a logical element representing a data source or a manipulation of data. A node contains terminals which represent endpoints of edges. A terminal is either a input terminal or an output terminal. The number of terminals is arbitrary and can be changed even after the node has been added to the graph. This allows a node to dynamically adjusts its terminals based on attribute values or graph structure.

For example, a **Field Node** represents a PPM Field data structure. It has no inputs as it represents a data source. It has one output which represents the field itself. An example of a node that transforms data is the **Field Discretization Node**. It takes two inputs, a discretization and a field. It has one output representing the discretized field.

3.2.2 Attributes

Each node has a set of attributes. These attributes specify the behavior or initialization parameters of node node. For example the **Field Node** has an attribute dimensions, which specifies how many dimensions the field should have. When defining attributes a programmer can specify default values, type constraints and value constraints of the attribute value. Currently only three types are defined: integer of integral values, number for floating points values and string for arbitrary strings.

Reference

Certain graph nodes need to be referenced in other parts of the graph or in ppm code provided by the user. For this purpose a name can be specified for each node which is called a reference. If needed, this name is automatically generated.

3.2.3 Graph Edges

An edge is directed and contains two endpoints called the **source** and the **destination** endpoint. Each endpoint is identified by to strings: the id of

a node and a terminal name. Edges also contain meta data, which is a list of key value pairs. Edges fire events when meta data has been changed.

3.2.4 Dynamic Flow

The dynamic flow mechanism of the graph allows for immediate changes in the graph based on the data flow. In default behavior of a node it copies the meta data from the edges that end in an input to the edges that are on the output. This behavior pushes meta data along the graph in direction of the flow. A node representing a source of data sets a reference to itself on its output edges.

An example usage of this mechanism is the behavior of the **ODE Node**. This node has inputs where discretized field should be connected and an input for connecting a **RHS Node**. It determines the discretized fields that are connected to its input nodes by inspecting the edge meta data of connected edges and configures the **RHS Node** if one is connected to it.

3.3 Code Generation

The code generator is responsible for transforming the graph into ppm code. Most nodes correspond to exactly one line in the ppm code. A node without input usually turns into an initialization of some data structure. For example the **Field Node** generates code in the following form

```
{ref} = create_field({n_dim}, {name})
```

The curly brackets are replaced by values taken from the node, ref is the name of the node reference and the arguments n_dim and name are taken from the node attributes. Nodes are processed in topological order. That means a node is only visited if all its inputs have been visited. When multiple candidates are available when choosing the next node the one with the lowest Y coordinate is taken.

3.4 Server Communication

Server communication is done over HTTP using the RESTful web API provided by the server. In order for the user interface to remain responsive, all requests have to be done asynchronously. Some interactions between client and server may require more than one request. An example of that is saving a project which, in order, creates a project, writes the settings and lastly writes the client. To ease dealing with asynchronous tasks the client uses an uniform interface called a job. A job either completes with a job-specific return value or errors with an error message or object. The server communication occurs via asynchronous HTTP requests.

3.4.1 Input Validation

Generally the data returned from the server is serialized using JSON. During deserialization the data structure is validated using type and value checking. This minimizes the risk of getting the application in an invalid, unpredictable state when faced with erroneous data.

3.4.2 Polling

Certain data structures such as the list of projects, the list of jobs or the output of a job are dynamic, that is they change without interaction of the current user. To keep the client in sync with the server these structures are periodically polled from the server. When updating a data structure with the latest version received from the server the data structures calculates a list of changes. For a collection like the list of projects or the list of jobs these include new entries, deleted entries and changed entries. The data structure then notifies listeners that have subscribed to the change event. This allows the UI to convey how the structures changed to the user.

3.4.3 Error Handling

If the server is unreachable, responds with an error or returns malformed data a job is considered to have failed. In general the client does not handle error conditions but allows the user interface to display the error. A periodic poll will continue even if a poll has failed thus allowing the client to recover from temporary failures.

3.4.4 Synchronized Data Structures

Most of the data structures inherit of the `Data.Entry` or `Data.Collection` class. The `Data.Entry` class represents a data structures that is composed of key-value pairs where the value is a javascript primitive (boolean, number or string). A `Data.Collection` class represents a collection of `Data.Entry` instances, where each entry has an unique id. Both classes offer serialization and deserialization to resp. from JSON. They also offer an update function so an instance can be updated to a newer version. All modification operations fire an event which details exactly what has been changed, allowing the user interface to visually highlight new or modified information.

3.5 Application Controller

The application controller marks the main entry point for the browser. It is responsible for initialing some application global data structures and deciding which view should be presented based on the URL. The YUI application

module provides the functionality for routing URLs to the different views. We currently declare the following routes:

```
Route /  
    Shows the project selection view.  
  
Route /project/new  
    Opens the project editor with a new project.  
  
Route /project/load  
    Prompt to load a project from the users hard disk.  
  
Route /project/:id/edit  
    Loads the project with given id and show it in the project editor.  
  
Route /project/:id/delete  
    Prompt to delete the project with the given id.
```

3.6 HTML User Interface

3.6.1 Views

A view is a reusable piece of the HTML part of the user interface. A view is responsible for providing a user interface for some given data. All views are based on the YUI component `Y.View`. A view general takes a data structure and a DOM node as initialization parameters. The DOM node specifies the point where the view inserts its own elements.

As an example the view defined in module `View.ProjectList` takes a `Data.ProjectList` data structures. Initially it constructs a HTML list which one list element for each project containing the project name, description but also a link to delete the project. It also keeps the data structure synchronized by subscribing to its change event. Changes to the list are highlighted using CSS animations.

3.6.2 Project Selection View

The project selection view (Figure 3.1) shows a list of projects, provides options to load and delete a project as well as creating a new one. This view uses the global project list polling manager. If the polling manager last poll was an error, the error is shown to the user. The view subscribes to the project list's change event.

3.6.3 Property Editor View

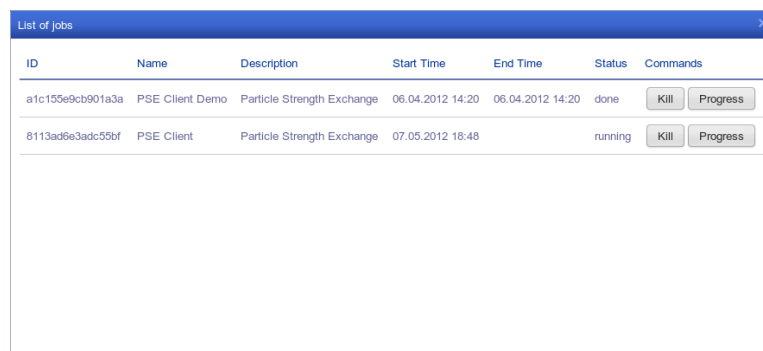
The property editor allows the user to edit the attributes of a node. The listens for node select events on the current project. If the selected node

has been changed the editor constructs an edit widget for each attribute. The node attributes are synchronized with the values in the property editor, changes in either side are immediately reflected in the other.

3.6.4 Block List View

The block list view shows all registered blocks that have a category. The blocks are lexicographically by their category and name. They are then rendered in that order with a header for each new category. Blocks are draggable so they can be dragged to the canvas.

3.6.5 Job List View



ID	Name	Description	Start Time	End Time	Status	Commands
a1c155e9cb901a3a	PSE Client Demo	Particle Strength Exchange	06.04.2012 14:20	06.04.2012 14:20	done	Kill Progress
8113ad6e3adc55bf	PSE Client	Particle Strength Exchange	07.05.2012 18:48		running	Kill Progress

Figure 3.1: List of jobs

This view visualizes a joblist data structure. It creates a table and provides buttons to kill a job or view its output. The project editor creates a floating window containing this view when the jobs button has been pressed, an instance can be seen in Figure 3.1.

3.7 SVG User Interface

In web programming SVG it is often used for visualization of data and many good javascript libraries exist for that purpose. There is however a lack of libraries when it comes to use it for user interfaces. Also many libraries that provide DOM manipulation utilities do not support the SVG DOM. The SVG DOM has some slight differences due to providing support for animation in the language itself. We modified the DOM manipulation utilities of YUI to support the SVG DOM. We also provide a custom widget rendering engine.

3.7.1 Widget

A widget is a interactive visual element. Our rendering engine uses a box model which is commonly used in user interface libraries. A SVG Widget has a parent widget and a (potentially empty) list of child widgets. The parent widget is responsible for rendering its child widgets and it does so by calling the render function on the child with a rectangular, axis aligned box as parameters. If a widget gets destroyed it will first destroy its children before finally removing itself. In the browser SVG is part of the DOM tree, which means that rendering a widget is done by inserting and updating nodes in a tree. Each widget has a container node, which is usually a SVG group node. The life cycle of a SVG widget is as follows:

Initialization

Creating a widget object causes its **initializer** function to be called. This allows the widget to set some initial state. Initialization does not cause any visible effects

UI Preparation

UI Preparation is done at most once and is done before the widget is rendered the first time. It is an error if the parent widget has not been set at this point. For readability the preparation is split in three phases.

UI Initialization

SVG Nodes are created and added to its container node. The container node is added to the container node of the parent widget.

Binding

The widget registers different event handlers. This includes listening for user input (mouse/keyboard) and change events on the data structure the widget displays.

Synchronization

Sets the state of the user interface elements to match the state of the widget.

Render

The widgets **render** function is called whenever the place or size of the widget should be updated. The widget can indicate how much space it needs in its **size** function which the parent may call to determine its optimal size.

Destruction

Destroying a widget is done by calling its **destroy** function. It will first destroy all its children. After that it will free any resources, unbind any event handlers and finally remove itself from the DOM tree.

3.7.2 Block Widget



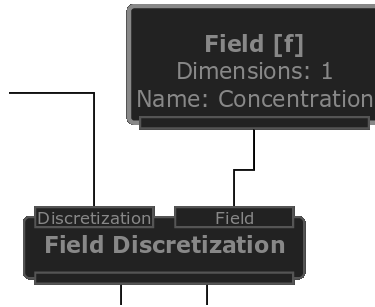
(a) A field block



(b) A field discretization block

The block widget is the visual representation of a node. A block consists of two major components. First there is a horizontal box layout that contains the title of its associated node and its attributes. Second there is a flow widget on the top and the bottom of the block. The flow widget represent the input resp. the output terminals of the node. Output terminals listen for drag events, allowing wires to be dragged out of it. Input terminals listen for drop events to allow wires to be connected to.

3.7.3 Wire Widget



Wires are special in that they don't fit in the box model, but are drawn between two endpoints. The wires have information about the bounding boxes of their endpoints. We provide a simple auto routing routine for the wires so they don't intersect with the bounding boxes of their endpoints. Wires are rendered as a SVG Path and are composed of orthogonal line segments.

Chapter 4

Server Architecture

The server is written in the Ruby programming language. It offers a RESTful API that the client can use to perform various tasks. We use web application framework Sinatra[6] to implement the API. We chose a simple, file based storage system to offer persistent storage. The server focuses on operations on the following two items:

Client Stores, builds and runs ppm clients

Job Information about currently running ppm clients or past runs.

4.1 Client Management API

GET /client/list

Returns a JSON list containing name and description of stored clients.

POST /client/new

Create a new client and returns the id of the created client.

{GET,PUT} /client/:id/settings

Returns or sets settings of client identified by the given id.

{GET,PUT} /client/:id/client

Stores or retrieves the client in the format used by the WebCG client.

PUT /client/:id/generate

Uploads a ppm client, compiles it and stores the executable as part of the client.

POST /client/:id/run

Runs the client executable, creates a job for it and returns its id.

DELETE /client/id

Deletes the client

4.2 Job Management API

GET /job/list
Lists all jobs.

GET /job/:id/progress
Retrieves combined stdin/stderr of a job.

POST /job/:id/kill
Kill a job if the job is still running.

Chapter 5

Results and Future Work

5.1 Showcase

We built a simple particle strength exchange (PSE) client using the visual editor to demonstrate its functionality. Figure 5.1 shows the visual representation of the client. It consists of three major parts: The initialization of the data structures (Field, Particles and Topology) and operators, the ODE/RHS definition and the timeloop.

The PSE client simulates a diffusion and therefore we declare a field named concentration. We also declare some particles which we use to discretize our particles. We still need the ability to use raw fortran/ppm code and provide a fortran code block with syntax highlight. In the example client we use them to initialize some constants and the apply some perturbation to the particles. As we are simulating a diffusion we need to declare a Laplace operator. We discretize this operator on the particles we created using the DcPSE method.

The right hand side (RHS) is connected to the ODE and contains blocks representing the defined operators and the fields that are connected with the ODE block. The RHS for a PSE client is simple and consists of applying the operator L (Laplace) to the field f (Concentration).

For the timeloop we use separate blocks for begin loop and end loop. These are the equivalent to the fortran do/enddo statements. Data that is modified in the loop is connected to the begin block and creates outputs representing the data. There is no strong visual relationship between begin and end loop - in fact the relationship is only established at code generation time. The timeloop is used to call the ode timestep method as well as preserving the boundary conditions by doing a ghost mapping.

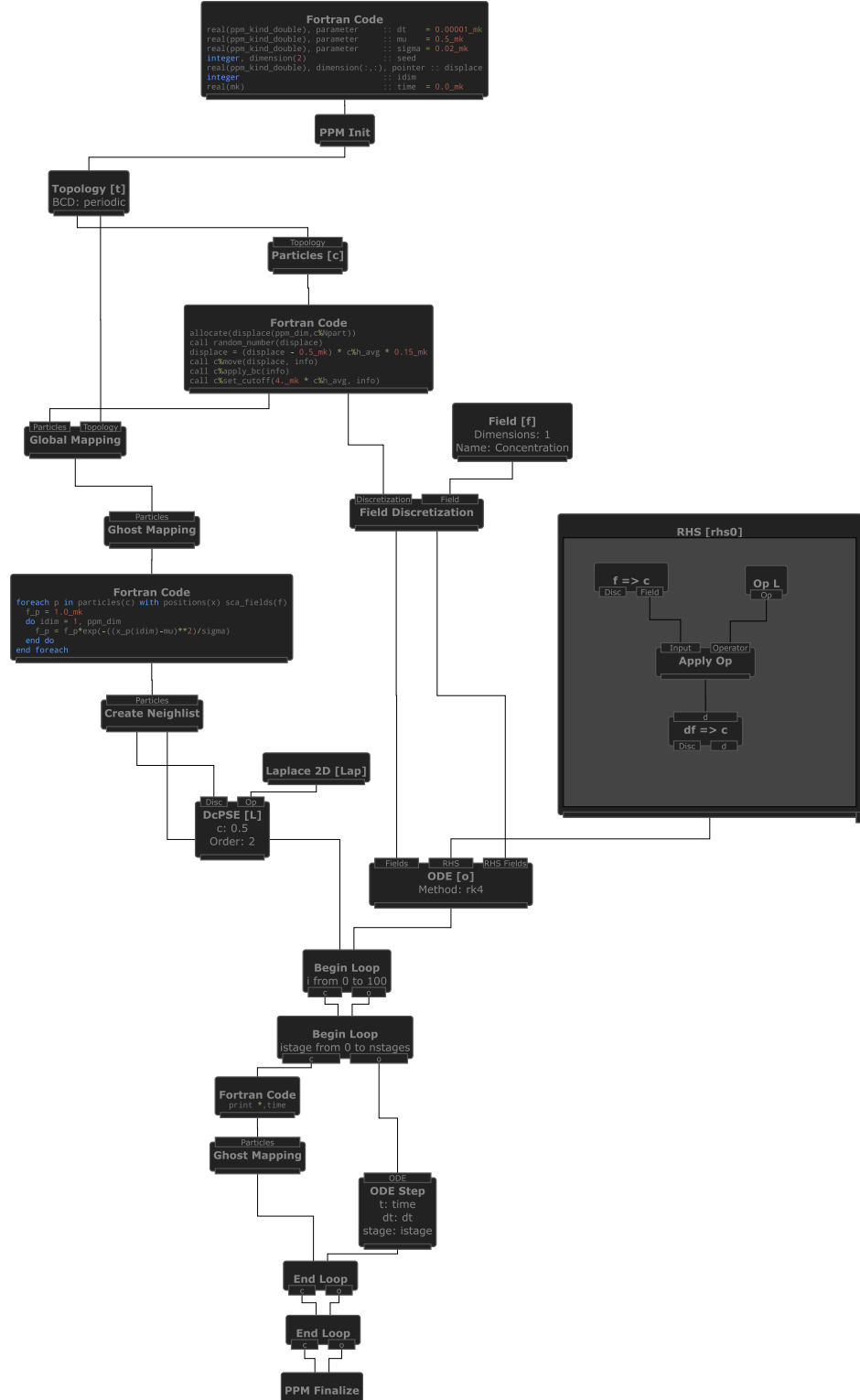


Figure 5.1: Particle strength exchange client

5.2 Results

Our application is a first attempt at making a visual programming environment for PPM. We created a working example client and are able to build, compile and run the client and retrieve its output. All the user interaction is done solely through the web browser requiring no prior configuration.

We do not provide building blocks for all the functionality offered by the PPM Client Generator. Currently we do not support the retrieval of arbitrary simulation output and we do not offer a interface to configure the different runtime settings.

5.3 Future Work

The general dataflow approach needs to be reviewed by people that are familiar with PPM or people that are familiar with other visual programming tools. The details of which blocks should be exposed to the user and what their semantics are need to be specified.

The graph built by the user does not check its correctness. Mistakes are only noticed when compiling the generated source code which is not user friendly as error messages will correspond to lines in code rather than blocks on the canvas. If we had strict constraints on the graph structure we could inform the user about mistakes during editing.

The server currently only allows retrieval of stdout/stderr of a simulation. This should be extended to allow retrieval of all generated output files and the client should offer a way to download them to the user's computer. Also the server should be extended to allow the simulations to be run on a cluster architecture.

SVG allows great flexibility over the visual representation, but also comes with a couple of drawbacks. One drawback is that SVG is rather new and it is unclear how much priority it is given by the browser developers. Another drawback is the increased complexity in having to maintain the custom widget rendering. One might consider switching to HTML and CSS which browsers are heavily optimized for and is easier to maintain.

Bibliography

- [1] <http://www.mathworks.ch/products/simulink>.
- [2] <http://www.ni.com/labview>.
- [3] <http://code.google.com/p/blockly>.
- [4] <https://github.com/idflood/ThreeNodes.js>.
- [5] <http://www.w3.org/TR/SVG>.
- [6] <http://www.sinatrarb.com>.
- [7] I. F. Sbalzarini. Abstractions and middleware for petascale computing and beyond. *Intl. J. Distr. Systems & Technol.*, 1(2):40–56, 2010.
- [8] I. F. Sbalzarini, J. H. Walther, M. Bergdorf, S. E. Hieber, E. M. Kotsalis, and P. Koumoutsakos. PPM – a highly efficient parallel particle-mesh library for the simulation of continuum systems. *J. Comput. Phys.*, 215(2):566–588, 2006.