A Web-based Application for the Program Algebra-Toolset

Arthur van Assenderp

June 17, 2010

Supervisor(s): Dr.Ir. Bob Diertens, Dr. Alban Ponse
Signed: Dr. Inge Bethke
Abstract

This paper describes the implementation of a web-based application for the Program Algebra-Toolset. The web-application fulfills three basic requirements. Easy extendibility, cross-platform compatibility and security.
Contents

Abstract 3

Contents 5

1 Introduction 7

2 Development process 9

2.1 Approach .................................................. 9

2.2 Requirements .............................................. 10

2.2.1 Overall description .................................... 10

2.2.2 Specific requirements ................................. 10

2.2.3 Functional requirements ............................... 11

2.2.4 Performance requirements ............................. 12

2.2.5 Design constraints .................................... 12

2.2.6 Other requirements ................................... 12

2.3 Design ....................................................... 12

2.3.1 Design Pattern MVC ................................. 12

2.3.2 Design Approach ..................................... 13

2.3.3 User Interface ....................................... 13

2.4 Implementation ........................................... 14

2.4.1 Application framework ............................... 14

2.4.2 Parser ................................................ 16

2.4.3 Projections ........................................... 16

2.4.4 Save and Open files .................................. 16

2.4.5 Simulator ............................................. 17

2.4.6 Server installation .................................... 17

2.4.7 Security .............................................. 17

3 Conclusions .................................................. 19

Acknowledgments ............................................. 21

Bibliography .................................................. 23
Program Algebra (PGA)\cite{12,2} provides an algebraic framework for imperative, sequential programming. In this framework a basic program notation is defined. Its syntax is given by a set of constants, a set of primitive instructions (basic instruction, termination, positive and negative test, and jump) and two composition mechanisms (concatenation and repetition). The primitive instructions receive a set of basic instructions as parameter.

Using this basic notation a hierarchy of program notations is build on top of it. Projections and embeddings are the operations to translate a higher level program notation to a lower one and vice versa respectively. The program notations in this hierarchy are related to one another by these operations. A program in a given program notation can be converted to any other program notation in the hierarchy by applying projections or embeddings repeatedly.

The PGA-Toolset\cite{6} provides a collection of programs that automates projections and manipulate programs in different notations with different sets of basic instructions as input. It includes a generic parser, a generic simulator, a parallel simulator and a bisimulator.

Program Algebra is used in both research and education. Since 2002 it is taught every year to students of different levels in different institutions. The toolset is an excellent support for the students in and outside the classroom but the installation requirements cannot always be fulfilled in each institution, and in the case of the students’ computers it depends on their knowledge and (fully) availability of a computer to install the software needed for the Toolset.

A solution to this problem is to make the toolset available without the need of installing software in the local computer. Given that each institution, and probably each student is provided with an internet connection and that all modern operating systems include an internet-client, a web-based application can solve these problems and make the toolset available for students, teachers and researchers.

The web-application must fulfill three basic requirements. The PGA-toolset is intended for education and research, therefore easy extendibility is crucial for the support of additions to the toolset. A web-application can be used from different operating systems with different internet clients and with default or more restrictive configurations, therefore cross-platform compatibility must be guaranteed. The third basic requirement is security. An (un)intentional bad formed input must not compromise the application or the platform where it is installed.

In this paper we describe the development process of a web-application for the PGA-Toolset. The product includes an easy to extend framework and the implementation of a part of the toolset. This subset contains the programs needed to support the introductory course to Program-Algebra: Wat is een Programma? (What is a program?)\cite{3,11}, namely the generic parser, the projections and the generic simulator.

The development process is in the following section described. The section is divided in three subsections: requirements, design and implementation of the product. The paper ends with some concluding remarks.

To understand the results and working of PGA and the PGA-Toolset, the reader should be familiar with Program-Algebra. Specially with the program notations, projections and Molecular Dynamics as described in \cite{4,3,11}.
In this section the different phases of the development process are described. We begin by introducing the employed methodology and describing the requirements, design and implementation phases that lead us to the final product.

2.1 Approach

The development process can be described as an iterative process of five phases as depicted in Figure 2.1. The overlap between phases shows that they are not strictly separated activities. Sometimes backtracking to earlier phases may occur to fix errors or changes in the specification.

The process begins by the demarcation and analysis of the problem as introduced in section 1. The global requirements are defined and the programs of the toolset that must be available in the web-application are known. We divided the problem in two sets. The first set involves the development of a basic framework and the second the addition of the programs of the toolset to this framework, one program at a time. The framework must be easy to extend, allowing to add new programs or reflect changes made in the toolset. Each of these subproblems is depicted in Figure 2.1 as "web-application part".

The next phase, the requirements engineering, has as a result a precise description of each requirement. This description involves the global application requirements, functionality, performance and design constraints. The requirements posed by and on the environment in which the web-application is going to function are also described. The resulting documentation is used as base for the design phase.

The goal of the design phase is to make a model of the web-application based on the requirements specification. This phase results in a technical specification that serves as starting point for the implementation phase.

The implementation and test phases are the phases with the most overlap in our approach. Each "web-application part" was tested when its code was complete. These tests involve checking the result for different parameters and testing cross-browser compatibility.

The creation and extension of the web-application are the result of one or more iterations of this development process.
2.2 Requirements

The requirements of the web-application as introduced in section 1 are in this subsection described. This description is intended as a starting point for the design phase.

2.2.1 Overall description

Product perspective

The web-application is a bridge between the PGA-Toolset and an internet client (browser). It allows the communication between the user and a PGA-Toolset program. The web-application’s main purpose is to make the PGA-Toolset available for education, while avoiding the need of installing it on the local computer. A typical interaction between the user and the web-application is depicted in Figure 2.2. The user selects a program of the PGA-Toolset and enters data by using the web-application user interface. These data are sent to the web-server according to the HTTP protocol. The web-application processes the data and executes the chosen PGA-Toolset program with the given parameters. The web-application receives the result and display it in the user’s screen. Finally the processed result is sent to the user. The process begins again when the user interacts with the web-application user interface.

Product functions

The web-application provides access to the following programs of the toolset:

- Generic parser
- Projections
- Generic simulator

See section 2.2.3 for a detailed specification.

2.2.2 Specific requirements

External interface requirements

User Interface

The user interface must fulfill the following requirements:

- Switch among PGA-Toolset programs at any moment.
- Show possible errors or messages from the web-application.
- Show toolset program’s results. It can be, for example, console output or a graphic representation.
- Enter program-code and select the parameters needed for each tool.
The web-application receives data from the user in three ways:

1. By entering text direct through an editable text area
2. By uploading a file to the server to insert program-code in the editor (Option Open file).
3. By selecting parameters for the toolset program.

Software interfaces

The web-application relies on a working installation of the PGA-Toolset in the server. It will be installed in a Linux server with the Apache HTTP Server[1] with support for PHP 5.2.

The web-application must be platform independent. It must run in a default installation of any contemporary browser, but also in more restrictive configurations where cookies or JavaScript are disabled. At least, the following browsers have to be supported: Internet explorer ≥ 7.0, FireFox ≥ 3.0, Safari ≥ 3.0.

2.2.3 Functional requirements

Generic Parser

When using this tool, the user enters program-code, selects a program notation and a basic instruction set. The Parser outputs the program-code decomposed into primitive instructions or a message in case of an error. This output is presented to the user in the Console Output that is part of the user interface. The input parameter program-code is a string of characters entered by the user, the program notation and basic instructions set are selected from a list.

Projections

This tool allows the user to enter program-code in a program notation and project it to another program notation in the PGA-hierarchy. The projection outputs the equivalent program in the destination program notation or a message in case of an error. The input consists of three parameters. The program-code is a string of characters entered by the user, the program notation of the program-code and the desired target program notation are selected from a list.

Simulator

The simulator receives as parameter a program-code, a program notation, a basic instruction set and an action. Based on this input, a graphical representation of the memory state of a system can be modeled as a fluid. The possible actions are Step, Undo, Multiple steps and Reset. The simulator offers the possibility to load different programs and to switch among programs in execution. The actions are described as follow:

- **Step**
  This action executes the next available instruction. It returns a graphical representation of a fluid and the program counter pointing to the next instruction. The fluid represents the memory state after executing this instruction. If the last instruction of the program is reached, the actual graphic representation and the program counter are not modified.

- **Undo**
  This action reverts the execution of the last executed instruction.

- **Multiple steps**
  This action gives the possibility to execute more than one step at a time. It adds two extra parameters, ’show updates’ and ’number of steps’. The ’show updates’ option can be true or false. When its value equals true, the graphic result of each step will be shown as an animation. When its value is equal to false, only the result generated by the last executed instruction is shown. When the number of steps to run is higher than
the available steps possible to execute, execution will finish at the last instruction. The number of steps cannot be higher than a certain configurable number. This parameter is of particular importance when the user enters program-code containing an infinite loop. Without a limited number of steps the simulator will run until a time-out is reached or an administrator terminates the process. This situation would lead to an undesired waste of resources and an unexpected behavior of the web-application.

- Reset This action sets the program counter to 0 and removes the graphic representation from the screen.

2.2.4 Performance requirements

The web-application will be mainly used for educational purposes. The scenario where a class of students uses the application from different computers simultaneously has a high probability. How many parallel requests must be handled depends on the class size.

2.2.5 Design constraints

Standards compliance

To maximize cross-browser compatibility all XHTML must be valid. The code must be validated with the w3c validator[7].

2.2.6 Other requirements

Security

The application or the platform where it is installed must not be compromised after an (un)intentional bad formed input.

Maintainability

The web-application has to be easy to extend in order to reflect changes in the toolset or to support other tools that are not required in this document.

2.3 Design

This section describes the architecture of the web-application. The web-application can be divided in three basic components. The most evident is the collection of programs included in the PGA-Toolset. These programs receive input from the user and return an output. The second component is where the input and output data are manipulated. These manipulations include the validation of the input data for security concerns but also the manipulation of the data returned by the PGA-Toolset program to be correctly presented to the user in XHTML code. The third component is the XHTML code self where the user interface is presented to the user with the manipulated output of the PGA-Toolset program. In the following subsections we describe the Model-View-Controller paradigm[8] and our approach.

2.3.1 Design Pattern MVC

Model-View-Controller is a design pattern that consists of three kinds of objects. The model object encapsulates the system’s data and the operations to manipulate those data. Data representation is completely independent from the model. The view object displays the data obtained from the model. The controller object receives and manipulates the user input, defining the way the user interface reacts to it.

In Figure 2.3, the interaction among the user, controller, model and view is illustrated. The arrow from user to controller shows input from the user generated by interacting with the user
interface. The controller processes the input, and this is translated into an action. The action may involve data manipulation or data retrieving from the model (arrows from the controller to the model and vice versa). As a result from this interaction, the state of the model may be changed. When the state of a model changes it notifies its associated views and controllers. The view receives a message from the model when it changes its state; typically the views will retrieve data from the model and update their display accordingly. The controllers also retrieve a message from the model on state change and may change their interaction method depending on the new model state.

2.3.2 Design Approach

We used a simplified approach of the MVC design pattern. The same criteria regarding controllers, models and views are used, but communication among components and behavior on model updates is changed.

Our approach is depicted in Figure 2.4. In our application design, each program of the toolset is represented by a model. Each model provides methods to execute the program with several parameters. The state of the model is represented by the output of the program with the given parameters. Each method in the model returns its state after execution.

A controller is divided into actions. The controller handles the input from the user; this input is verified and sanitized. When the action interacts with one program of the PGA-Toolset, the controller calls the model that represents that program with the parameters given in the user input. Only the controller can access the model. The controller processes the data of the user, the data of the model, and passes the data needed for display to the view.

The view is rendered after execution of the controller action.

2.3.3 User Interface

In Figure 2.5 the design of the user interface is depicted. The top of the interface has a fixed place for the navigation bar to select the program of the PGA-Toolset to be used, and for application messages and errors. The rest of the screen is designated to the tool that is being used. At the
left the console output takes about a fourth of the screen, and at the right the editable area, program options and editor options are shown. In the case of the simulator the editable area is replaced by an image.

This interface applies for the parser, projections and simulator tools but as a result of the separation of the data processing, the execution of the programs of the toolset, and the display of data, it is simple to add or modify elements of the interface for new tools.

2.4 Implementation

This section describes the implementation phase of the process. In Section 2.1, the development process was introduced as an iterative process consisting of five phases. Each iteration has as result a "working web-application part". In the following subsections the implementation of each of these parts is described.

2.4.1 Application framework

A basic framework is implemented in which programs of the PGA-Toolset can be easily added. This framework manages the communication among the different components. The framework is based on four classes: Registry, Router, View and ApplicationController.

Registry

The registry object is used mainly to pass values from the controller to the view. The ApplicationController constructor receives a registry object as parameter. The controller sets values that need to be displayed on a view template into a view object. The view object is passed to the controller in the registry.

Router

The mission of the router object is to instantiate the correct controller, run the given action and trigger the show method on the view object. The controller, the action and the options for the action are given as parameters in the HTTP request. If the controller is not specified, the MainController is instantiated. If an action is not specified, the index action is called. When the controller runs the action, it will optionally run methods of a model object and set variables in the view object kept in the registry.
**View**

The **View** class is used to store variables that must be presented after executing the action of a controller. The **View** class also defines the **show** method. This method will find and present the correct view template for a given controller and action.

**ApplicationController**

The **ApplicationController** class is the parent of all the controllers. The methods defined in this class involves the definition of program notations and basic instructions available in the PGA-Toolset, and the default action **index** that is inherited and probably redefined in the child controllers.

Upon construction, the controller class checks if the **HTTP** request is a common request or an **Ajax** request. **Ajax** (asynchronous JavaScript and XML) is a set of techniques that allows to retrieve data from the server asynchronously in the background. An **Ajax** request results in the updating of a part of a webpage without reloading the entire page. We use **Ajax** requests in this application to load the results from the PGA-Toolset in the webpage. Using **Ajax** for these actions reduces the time to present the result. By updating only parts of the page without reloading it entirely and with short time responses we create the illusion of a desktop-application when in fact it is a web-application.

**MainController**

The **MainController** has two purposes. When a controller is not specified in the parameters, the **index** action of **MainController** is executed. This action redirects the user directly to the Parser. The second purpose is to delegate the action to other controller. The user interface has an **HTML FORM** element where all parameters are kept. Upon submit the form sends all the parameters to the delegate action of the main controller. Depending on the tool and the action selected by the user, the parameters are redirected to the controller that handles the desired program of the toolset. The reason for this construction is that the options and program-code entered on each tool remain available even when the user changes of tool in the web-application. This is an example of a controller where an action has not a view and where there is no communication with a model.

**Single point of entry**

The application has a single point of entry. From this point the right controller is instantiated and an action is called.

The interaction among the different classes is depicted in Figure 2.6. At the single point of entry (step 1) application configuration values are loaded. A **router**, **view** and **registry** object are instantiated in step 2. In step 3, the **router** will instantiate the right controller and call the given action if an action is specified. If the action is not specified the default action is called. Subsequently, the controller will instantiate one or more models in step 4, depending on the parameters received. The model returns the results given by a tool of the PGA-Toolset (step 5), and the controller makes these values available for the view by setting them in the **registry** object in step 6. The last step is executed by the router; it calls the **show** method on the **view** object. This results in including a **view template**, that will render the results stored by the controller in the **registry** appropriately.

Extending the web-application to support PGA-Toolset programs is achieved by adding a **controller** class with the desired actions, a **model** class that executes the program of the toolset, and a **view template** for each action defined in the controller. The following subsections describe the addition of the Parser, the Projections and the Simulator to this framework.
2.4.2 Parser

The controller for this tool consists of one action, its default action index. In this action the parameters needed for the tool are first checked to avoid common security exploits.

The Parser takes as parameter a program-code, a program notation and a basic instruction set. When the parameters pass the checks, the method parse of the ParserModel class is called. It returns the result to the ParserController and it stores the values in the registry. The view templates for the default action index of the ParserController contain the Console Output and the Editor. The last is used to enter code and select Parser options.

2.4.3 Projections

The Projection's controller, model and view are very similar to the ones created for the Parser tool. The projection requires as extra parameter the program notation for the resultant program-code. The view template has therefore an extra option to select the destination program notation. The Editor and Console Output are included.

2.4.4 Save and Open files

The editor has the options to save and open files from the local disk. These options are implemented in the EditorController.

The Save action has as parameter the program-code in the editor and the tool being used at the moment that the user selects this action. The program-code is parsed. If no errors are found it is sent back to the user, otherwise a message is shown with a confirmation dialog. An HTTP header that forces the browser to open a “Save file” dialog is sent.

The Open action receives as parameter an uploaded file, and the tool used at the moment that the option action is called. The option to select a file from the local disk is given to the user to open a file with program-code.

When the file is uploaded and verified the tool that has been used when the Open action was called is displayed with the uploaded file loaded in the editor. A user can open more than one
file per session. The editor has the option to select a program-code to use. Loading multiple files is specially useful to generate a graphic representation with the Simulator tool by executing different parts of different program-codes.

2.4.5 Simulator

The SimulatorController has three actions.

1. Step
   This action receives as parameter the program-code, a program notation, a basic instructions set and a number equal to the instruction to be executed. This action is used by the Step and the Undo options of the user interface. The Undo option sends the same parameters as the Step action but subtracting 1 of the program counter.

2. MultipleStep
   This action receives the same parameters as the Step action and two extra parameters. Namely: the number of steps that must be executed from the given instruction number, and the option "updates". The "updates" option can be true or false. When it is true, the browser receives for each step an image representing the memory state of the program-code at a given instruction. The browser presents the images as an animation using JavaScript, showing the transformation from the first to the last instruction executed. When the updates option is false, only the result of the last instruction is returned.

3. Reset
   This action is used to remove the graphical representation and set the program counter to 0. Both procedures can be done in the local computer with JavaScript without involving communication with the server, but the action is programmed to allow compatibility with clients that have JavaScript disabled.

The SimulatorModel has one method. This method executes a program in the PGA-Toolset that makes a graphic representation after executing the code and returns the number of the next instruction to execute. If the last instruction is executed, it returns -1 and an image is not made.

Images produced by the toolset will have variable size depending on the complexity of the program-code. A complex graphic representation will generate a wide image. Therefore it is necessary to introduce images in the user interface in a container that automatically can scroll horizontally and vertically when the image is bigger than the container.

The generated images have an unique id number per image as name. This id is used by the web-application to retrieve the results of the simulator. Image files are immediately deleted after retrieval. It is possible that some images never get retrieved. This could happens if the browser is closed before all images of a MultipleStep is called. Therefore image files older than a given time must be deleted periodically. This is done every time that an image is retrieved, but could be programmed outside the application if this give performance problems.

2.4.6 Server installation

The application was installed and tested on a Linux server with the Apache HTTP server[1]. The web-application performance relies on the hardware properties of the server, the network connection bandwidth and the configuration of the Apache HTTP server. This configuration must allow the minimal desired incoming requests.

2.4.7 Security

When the web-application receives data from the user it must be validated. Different checks and validations must be done depending of the type of data and what is done with it. We distinguish two uses of the data in the web-application:

1. Data into the HTML code. This is for example the program-code entered by the user.

2. Data that is passed as parameter to a program of the toolset.
When data entered by the user is rendered directly into HTML code it is possible to use Cross Site Scripting techniques. This permits extern code to be executed in the client browser. The use of these techniques can be avoided by converting all special characters (specially '<' and '> ') to HTML entities.

A user can upload a file to the server using the Open option of the editor. We expect a text file to be uploaded but the user can upload any type of file. A binary file is immediately deleted and an error message is shown. When a text file is successfully uploaded, special characters are changed into HTML entities to show the program-code safely in the browser.

Parameters passed to the parser tools need to be escaped in order to avoid executions of other programs in the server. In that way, each parameter is treated as a single safe argument.
We have described the development process of a web-application for the PGA-Toolset. We used a variation of the MVC design pattern described in [8]. This implementation allows us to separate code by application domain, logic and user interface. The first step of the implementation was to build a framework in which different programs of the toolset could be added. The framework handles the communication among the components described in the design pattern and establishes a clear file structure. The addition of new tools to the framework is therefore fast and effortless. The access to the toolset (model), processing of data (controller) and rendering of results (view) are programmed separately. This permits independent maintenance of each component.

To increment cross-browser compatibility the XHTML/CSS code follows the standards proposed by the World Wide Web Consortium (W3C). This results in XHTML documents that are semantically correct, but it is not a guarantee that different browsers will render the elements in the same way. This is a problem particularly in old browsers like Internet Explorer 6.0, in which some CSS directives behave different or are not recognized. We did not make a specific version code for each browser, but made one version using the simplest XHTML structure possible, and CSS recognized by both old and contemporary browsers.

The basic functionality of the application does not rely on a specific configuration of the browsers. Ajax and JavaScript are used to make the illusion that the web-application behaves as a desktop-application by updating a part of the webpage. The application will present the results also when JavaScript is disabled. This strategy is known as “progressive enhancement” [10] and is used to emphasize accessibility.

The web-application as described in this paper makes the PGA-Toolset available to a wide public in an easy and fast way.
My sincere thanks go to Alban Ponse and Bob Dierens for their continuous support, encouragement, kindness and enthusiasm.

I am very grateful to Nuria Rodriguez for the proofreading.

Last, but certainly not least, I am indebted to my dear wife Andrea Visbeek, for all her love, support, understanding and even proofreading during this project.
Bibliography


