Worldview Manager:
An Automated Socratic Interrogation Framework

Scott Aaronson, Leonid Grinberg, and Louis Wasserman
{aaronson,leonidg,lowasser}@csail.mit.edu

September 1, 2009

Abstract

We present Worldview Manager (http://worldview.csail.mit.edu), a prototype for a framework and website that help users uncover hidden inconsistencies in their personal belief systems ("worldviews"). This is accomplished by presenting the user with a series of statements and asking him or her to indicate the degree of agreement or disagreement with the statement. Whenever a "tension" (contradiction) between two or more responses is detected, the website presents the relevant responses, and gives the user an opportunity to revise one or more of them, presenting, whenever possible, the logical reasoning involved. It is our hope that this website will enable users to better understand their political and philosophical views and why they hold to them, while the framework on which it is built will enable developers working on related problems to implement their own, similar programs.

Contents

1 Introduction

We present Worldview Manager (`http://worldview.csail.mit.edu`), a prototype for a framework and website that help users uncover hidden inconsistencies in their personal belief systems ("worldviews"). This is accomplished by presenting the user with a series of statements and asking him or her to indicate the degree of agreement or disagreement with the statement. Whenever a "tension" (contradiction) between two or more responses is detected, the website presents the relevant responses, and gives the user an opportunity to revise one or more of them, presenting, whenever possible, the logical reasoning involved.

In principle, such a general specification is no more than a truth maintenance system[1] with a user-interface built on top. However, there is no requirement that the system parse and understand the statements it presents and then try to determine the tensions automatically. While, in theory, such advanced functionality would work perfectly well with the system, the only main requirement, as far as Worldview Manager is concerned, is that there is some database of statements with explicit instructions as to which combinations are logically inconsistent. Whether the database is generated by human or machine is essentially a matter of implementation.

1.1 About this document

This document is intended to act as definitive documentation of Worldview Manager's internal functionality. It is not intended to be read by users of the program; indeed, much of the internal functionality is intentionally hidden from the end user, with only basic, interface-level instructions presented as necessary. Anyone wishing to understand intimately how Worldview Manager works should read this document. Care has been taken to reference specific code and specific files, so the reader is strongly encouraged to read this document in parallel with the actual source files.

Those wishing to simply install a copy of Worldview Manager without understanding how it works can safely stop reading after section 4, which describes the database structure that Worldview Manager needs.

This document assumes a general knowledge of modern programming and web paradigms and technologies. It is helpful to have a working knowledge of HTML syntax, MySQL use and PHP programming (or at least PHP syntax) when reading this document. Some knowledge of Haskell and Scheme will be helpful for sections 7.2 and 7.3, respectively. Lastly, some basic knowledge of standard UNIX (or, more relevantly, LAMP) servers – including shell scripts and Apache administration – will be very helpful, though not strictly necessary.

All file paths in this document are written relative the root of the source tree.

[1]Technically, a truth maintenance system would include some built-in heuristic by which it would determine what data to accept and reject in the event of a contradition. However, the modification that the contradiction is resolved by the data source, rather than by the system itself, is not a particularly significant one.

1.2 Components

Our prototype consists of the following parts:

- A MySQL database with several tables specifying “topics” (isolated[2] groups of statements and tensions between them), statements, and user responses to statements. We also have a table for user comments which appear on the page for every topic, statement, and tension.

  See section 4 for more on the database.

- A PHP-powered framework for manipulating the database and passing data between different components of the application.

  See section 5 for more on the framework (specifically section 5.2 for the database querying library and section 5.3 for the data-passing library). See appendix A for the API that comes with the framework (specifically appendix A.1 for the database querying API and appendix A.2 for the data-passing API).

- A PHP-powered website that uses the framework to generate user-visible HTML and Javascript. Other, more static files, including CSS and Javascript programs, are served directly by the web server (in our case, Apache).

  See section 6 for more on the website. Special components of the website include:

  - A `.htaccess` file that manages some of the more specific and quirky behaviors of the website. Most importantly, this file contains a configuration for semantic URLs that make the site easier and friendlier to use.

    See section 6.3 for more on the `.htaccess` file.

  - Administration pages, written in PHP, that allow for relatively easy administration of topics. Specifically, they allow for the uploading and deletion of topic files through a web interface, as well as some aggregate statistics about user responses. The pages are not currently intended to be viewed or used by ordinary users, but the code base can be relatively easily adapted to that functionality, e.g. via user accounts.

    See section 6.4 for more on the administration pages.

- A compiler suite for ”topic files” that converts from a user-readable format in a logic-based notation into SQL.

  See section 7 for more on the compiler suite. The suite includes:

[2] See sections 4 (Database structure) and 7.1 (Topic files) for a more thorough discussion of cross-topic tensions.

– `WVMPreprocessor`:

  A preprocessor, written in Haskell, that validates most of the input, makes all logical inferences that it can, translates any LaTeX code to HTML or PNG images via the `texvc` utility, and then translates the logical syntax to a much more verbose syntax.

  See section 7.2 for more on `WVMPreprocessor`.

– `mktopic`:

  A compiler, written in Scheme, that reads the output of WVMPreprocessor and generates a SQL script that, when executed, uploads statements and tensions.

  See section 7.3 for more on `mktopic`.

1.3 Choice of technologies

Since any choice of language and technology for the implementation of a system is the subject of much controversy, we would like to take a moment to defend our particular choices.

Specifically, we chose MySQL for the database system due to its ubiquity in installations, the fact that many different clients and installations can all access the same database, and the excellent support that PHP has for it.

PHP was chosen as a robust, powerful and well-known language, and is a sort of compromise between power on one hand and cleanliness of code and strict adherence to a MVC system (as featured in many other web frameworks such as Django or Rails) on the other.

Apache was chosen due to its convenience as a robust and ubiquitous web server.

Lastly, the compiler suite was written in Haskell and Scheme due to the preference for these languages by the authors as well as due to the fact that performance is a much smaller issue for those programs – the programs only need to be run once per topic file.

2 Related work

The idea of mixing strict logical consistency with a computerized process is hardly novel. As described in section 1, Worldview Manager essentially serves the purpose of a truth maintenance system, albeit in a highly specialized manner. That said, it is this highly specialized purpose of Worldview Manager that is the bulk of our contribution.

One motivation for Worldview Manager is the very large number of political and philosophical tests and surveys that the Internet has to offer[3]. Besides being inconsistent and vague in their questions (something that Worldview Manager probably can also be blamed for), these sorts of websites do not actually attempt to interpret the responses of the user in any way, instead relaying to the user a summary of his or her personality/philosophy/polticial belief system. Some systems, most notably `http://politics.beasts.org`, do make some effort to detect inconsistent responses, but this is done primarily to ensure consistency in

[3]See `http://www.politicalcompass.org`, `http://politics.beasts.org`, and `http://typology.people-press.org/typology/` for a few examples

question interpretation; not only are most questions not subject to these sorts of confirmations, but the inconsistency is never presented to the user and instead silently ignored.

Therefore, it is the primary functional goal of Worldview Manager to ensure a *consistent* set of responses. The meaning of the responses themselves, beyond their relationship to each other, is entirely irrelevant to the system. Thus, Worldview Manager's goals are, in some sense, orthogonal to those of the systems mentioned above; one can imagine a system that not only interprets a set of responses as indicating some specific political leaning, but also insures that such a set of responses is in fact logically consistent.

Similarly, there have been and still are many attempts to teach computers to parse human languages, from projects such as OpenMind[4] to those such as START and Wolphram Alpha[5]. While ostensibly, these sorts of projects have goals entirely independent from those of Worldview Manager, there are a number of similarities, albeit at another level. Specifically, the database component of Worldview Manager is entirely human-generated, which is not only somewhat of a tedious process, but also error-prone. Therefore, one can easily imagine a combination of Worldview Manager and some of the systems outlined here, in which a shared database allows a human language parser to both parse responses as well as generate questions to answer, while Worldview Manager provides a more high-level framework to ensure logical consistency in the responses.

It is this compatibility between Worldview Manager and such a wide variety of projects that serves as motivation for the design choices we made. An open and relatively generic database model, combined with a fairly standard HTTP-based session management system adds modularity not only to the individual components of Worldview Manager, but also to the whole program as a component of some other, even larger system.

3 Installation

This section describes the process of obtaining and installing Worldview Manager.

3.1 License

Worldview Manager is licensed under the Apache Software License version 2.0[6]. It uses several other libraries and packages. For a complete list, see the file `NOTICE`.

3.2 Software requirements

This section provides a complete list of the software Worldview Manager needs in order to properly run.

[4]See `http://openmind.media.mit.edu`

[5]See `http://start.csail.mit.edu` and `http://www.wolframalpha.com`

[6]See `http://www.apache.org/licenses/LICENSE-2.0.html` or the file `LICENSE` for more details.

PHP 5: The framework and website components of Worldview Manager are all written in PHP. Several features used require PHP 5. Though Worldview Manager has not been tested on PHP 6, it should work, perhaps with minimal modifications. See `http://www.php.net`.

MySQL 5: The database querying library relies on MySQL being present to obtain data. Version 5 is necessary for some of the functions. See `http://www.mysql.com`.

Apache HTTP server with `mod_rewrite`: Worldview Manager can run on any web server, as long as the URL rewriting rules as provided. The `.htaccess` file (see section 6.3) provides these, and several other, useful features. The reader is strongly encouraged to look at `.htaccess` if porting to another server is desired. See `http://httpd.apache.org`.

Haskell compiler and libraries: `WVMPreprocessor` (see section 7.2) is written in Haskell; having a compiler or any libraries is not actually necessary both because using the compiler is not necessary and because the compiled version has all required libraries statically linked. However, any modifications to the program will require a compiler and libraries. See `http://haskell.org/` and `http://www.haskell.org/haskellwiki/Parsec`.

The `texvc` utility of MediaWiki: `WVMPreprocessor` (see section 7.2) utilizes `texvc` to translate LaTeX code into HTML or PNG images. The utility is not included with Worldview Manager, and must be downloaded separately as part of MediaWiki. See `http://www.mediawiki.org`.

Bourne Again SHell (BASH): `WVMPreprocessor` (see section 7.2) uses a shell script to invoke `texvc` (see above). BASH is necessary for this script to run, which is necessary if `WVMPreprocessor` is to be used. See `http://www.gnu.org/software/bash/`.

Bigloo Scheme compiler and libraries: The `mktopic` program (see section 7.3) is written in Scheme for the Bigloo compiler; having either the libraries or compiler installed is not necessary only insofar as using `mktopic` is not necessary. The Bigloo libraries are not statically linked, and so using mktopic without them installed is impossible. The program was written for version 3.2a, but later versions should be backwards compatible. See `http://www-sop.inria.fr/mimosa/fp/Bigloo/`.

3.3 Downloading Worldview Manager

The code for Worldview Manager is held in a Git[7] repository. The repository contains the source code to every file used on the website (including all PHP, Javascript, HTML, and CSS files), along with the compiler suite (see section 7) source and binaries, configuration files, `.htaccess` files (see section 6.3), topic files (see section 7.1), the Apache Software License and helper `NOTICE` file (see section 3.1), and the LaTeX source code of this documentation.

[7] See `http://www.git-scm.com` for details.

The repository can be downloaded at `http://www.gitorious.org/worldview`, e.g. with the command "`git clone git://gitorious.org/worldview/worldview.git`".

3.4 Installing Worldview Manager

In order to "install" Worldview Manager, it needs to be placed in a folder that the web server can see. This is a process that depends strongly on a particular setup – for some installations, placing the code into a subdirectory of, e.g. `/var/www`, is enough; for others, a virtual host might be required. Please check with your system administrator for details.

The only part of the website that should be directly accessible to the webserver is the `manager/` directory. However, for the files in `manager/admin/` to work (specifically `manager/admin/topic_list.php`, `manager/admin/upload_topic.php`, and the compiler binaries), it is necessary for the files in the directory above `manager/` to be accessible by the web server (but not the web browser).

Once this is done, the following steps should be taken:

1. The MySQL database should be set up according to the structure described in section 4.
2. The configuration file `conf/dbinfo` should be edited – the first line should contain the database server, the second line the user, and the third the password for the user.
3. The salt text (the value of the variable `$SALT` in `manager/lib/usercookie.php`) should be changed to something very long and hard (read: impossible) to guess. We recommend using a random-password generator.
4. The directory `manager/media/images/topic_explanations` needs to be created in such a way so that the server can write to it. This might involve changing POSIX or AFS permissions.
5. If the compiler suite is used (see section 7), `mktopic` has to be compiled. In the directory `topic_file_parsers/mktopic`, run something along the lines of "`bigloo -o mktopic mktopic.scm database.scm parse.scm`" .
6. **(Optional, but *highly* recommended)** The `manager/admin` directory should be password-protected. The easiest way to do this is to create a `.htpasswd` file[8] and add the authentication directions to `manager/admin.htaccess`.
7. **(Optional)** The `.htaccess` file (`manager/.htaccess`) should be edited to include any server-specific quirks.
8. **(Optional)** Git should be configured. The exact details of this depend strongly on the particular setup, and whether or not you git is going to be used. We recommend setting up a hook on the server that will automatically compile `mktopic` whenever a push is made. See `http://git-scm.com/` for more information.

[8]See `http://httpd.apache.org/docs/2.0/programs/htpasswd.html` for more information.

4 Database structure

Worldview Manager's database uses MySQL. The prototype is hosted by the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL)[9], which operates a centralized MySQL installation. This installation is utilized over other alternatives, including SQLite, which provide portability at the expense of centralization. Since we do not expect our prototype to move to a different server, the use of MySQL was particularly beneficial as it allowed several checkouts of the code, including local development versions on the authors' computers, to access the same database without any problems.

4.1 Tables

The MySQL database is called `worldview` and has the following tables:

- `topics` – A list of topics, each with a unique `topic_id` (a positive integer serving as the primary key), a `topic_name`, and `topic_text` that describes what kinds of statements are in the topic.

- `statements` – A list of statements, each with a unique `statement_id` (a positive integer serving as the primary key), a corresponding `topic_id` of which the statement is a part, the `statement_text`, and a `short_ident` (an internal identifier which is never seen by the user, but which allows for the formatting of tension explanations by the website – see section 7.1 for more on Topic Files and tension explanations).

- `tensions` – A list of tensions, each with a unique `tension_id` (a positive integer serving as the primary key), a `tension_explanation`, and a series of "`statement`" and "`agreement`" pairs. In each pair, the "statement" (labeled `statement_`i) is a `statement_id` from the `statements` table and the "agreement" (labeled `agreement_`i) is either `1` for agreement or `0` for disagreement. At present, the database holds up to four such pairs[10] (so i can range from `1` to `4`). Those rows that don't use all four pairs can hold `NULL` values in some of their cells.

  Note that there are no `topic_ids` because the `statement_ids` from `statements` are globally unique. This means that, at least on the database level, cross-topic tensions are pefectly acceptable. If the database does contain cross-topic tensions, the website will essentially behave appropriately, though some odd behavior might be encountered in some places. See section 7.1 (Topic files) for more about cross-topic tensions.

  As an example, the table row

tension_id	statement_1	agreement_1	statement_2	agreement_2	statement_3	agreement_3	statement_4	agreement_4	tension_text
17	42	1	17	0	91	1	NULL	NULL	Foobar

[9]See `http://csail.mit.edu`.

[10]See Appendix B for information about how to properly modify this value to something other than 4.

shows a tension with a `tension_id` of `17` and the (somewhat unhelpful) explanation "`FOOBAR`". It is triggered when the user agrees to the statements with the `statement_id`s `42` and `91` and disagrees with the statement with a `statement_id` of `17`. There is no fourth statement involved in this tension.

- `users` – A list of "users". In our prototype, a user simply has a numeric `user_id` (a positive integer and the primary key in the table), a time and date indicating the `last_action` performed, and a `name`. The `name` for most users is `NULL` and is only generated when the user makes a comment (see the bullet for the "`comments`" table as well as section 6.2.8 for more on comments).

- `user_responses` – A list of responses. Each response has a unique `response_id` (a positive integer serving as the primary key), with a corresponding `user_id` (indicating the user who made the response), `statement_id` (indicating the statement to which the user responded) and `response` (a floating point number from `-1` [complete disagreement] to `1` [complete agreement]). There is also a `time` column, indicating the time and date when the response was made.

- `responses_history` – This table is identical to `user_responses`. Its purpose is merely for the gathering and analysis of statistical data – users whose sessions has expired are moved from `user_responses` to `responses_history`. See section 6.1 for more on sessions.

- `deferred_tensions` – A list of `user_id`s and `tension_id`s, as well as a `time` column that indicates when the tension was deferred. See section 6.2.6 for more on deferred tensions.

- `comments` – A list of comments. Comments appear on Topic, Statement, and Tension pages. See section 6.2.8 for more details on comments as well as sections 6.2.2 (Topics), 6.2.3 (Statements), and 6.2.4 (Tensions) for more information about the respective views. Each comment has a unique `comment_id` (a positive integer that serves as the primary key), a `user_id` indicating the author of the comment, a `time` indicating the time the comment was written, an `area` (either "topic", "statement", or "tension") and a `target_id` indicating the relevant ID (either `topic_id`, `statement_id`, or `tension_id` as indicated by `area`).

5 The framework

The bulk of Worldview Manager is a PHP-based framework that faciliates the manipulation of the database (see section 4) to provide data for a variety of views (see section 6.2) – in our prototype's case, for a website. The framework provides a wide array of functions and several variables, all of which are documented in the API (Appendix A). Roughly, they can be divided into three groups – a database querying library, a data passing library, and a collection of miscellaneous features and utilities.

5.1 Variables

Several (global) variables are provided by the framework. As a policy, all are written using capital letters:

`$USER_ID` Stores the user ID of the current user. In our prototype, it is first defined in `lib/usercookie.php`. However, this is mostly a matter of convenince, since in our prototype, the user ID is dictated solely by the contents of a cookie. In another implementation, this may not be the case.

`$TOPIC_ID` Stores the current topic ID of the current user. In our prototype, it is also first defined in `lib/usercookie.php`, and this is also mostly a matter of convenience – one can easily imagine a different prototype where the current topic is defined by something other than a cookie.

`$MAX_ARITY` Stores the maximum number of statements that can be involved in a tension. This is used primarily in the database querying library to generate SQL queries. This variable is currently set to `4`, and is read from the file `conf/max_arity`. The process of seamlessly changing it is somewhat involved; see Appendix B for details.

5.2 The database querying library

The Worldview Manager database querying library is intended to be a centralized collection of every function involved in retrieving and processing data. Unlike the data passing library (see section 5.3), it is used specifically for data retrieval and processing, rather than for communication between different views.

As a policy, *every* direct call to the database is written as a function in the database querying library; the intention of this is to decouple and abstract the process of data retrieval from the rest of the website as much as possible. That said, our prototype's views do assume the use of MySQL, as they rely on the fact that database querying functions all return the results of calls to `mysql_query()` (which can then be processed via `mysql_fetch_row()`, `mysql_fetch_array()`, etc). It is our hope that the system may someday be refactored so as to abstract these details even further into the library.

The database querying library is in many ways more complicated than the data passing library – not only is it larger, but it is also arguably more ambitious in scope, while also being considerably more low-level. Furthermore, like the data passing library, it has its own set of security issues to deal with, mostly having to do with SQL injection attacks. Care has been taken to avert such security issues, in part by taking advantage of the fact that since the majority of data used is numeric (and integral), running `intval()` on the data or checking that `is_numeric()` returns true before processing it, would remove malicious requests while preserving benign ones. In cases where these sorts of tricks do not work, standard functions such as `mysql_real_escape_string()` are used. However, we attempt to minimize this usage in part due to its inconsistent behavior with and without Magic Quotes.

The bulk of the database querying library is used for data retrieval. These functions all return data via the result of a `mysql_query()` in some specified format, which is described on a per-function basis in the API (appendix A.1).

There are also several functions in the library that are designed to add data to the database. These functions accept data in some specific format described in the API and upload it to the database. In some cases, a return value is given.

Lastly, there are a few functions in the library whose goal is to *process* data. The two most important examples of this are `pick_statement()` and `find_disagreements()`. For a complete list, please see the API. These functions utilize some of MySQL's more complicated features to quickly process the data given – in other words, rather than simply obtaining a list of responses and tensions and looping through each in PHP to find tensions, the library performs this entire process on the database level. While this does cause the program to be somewhat less portable, it also adds a significant speed boost, and means that porting some of the more complicated aspects of the library to another language (although not another database model) is relatively straightforward.

5.3 The data passing library

The data passing library is a large collection of functions intended to centralize the exchange of data between different HTTP requests. Like Sessions (see section 6.1), it is intended to bypass the limitations posed by HTTP's statelessness. On a more nebulous level, it is intended to centralize all direct interactions with HTTP, so no direct references to HTTP variables (including `GET`, `POST` or cookies) need to occur from within the code used to generate views. If Worldview Manager were to be ported to a different platform that doesn't use HTTP (at least not in the same way), the actual code in the data passing library would have to be modified, but the API would not be changed. Thus, this library can be seen, in a way, as "glue" for the website and the framework components of Worldview Manager.

Like the database querying library (see section 5.2), the data passing library is divided into several files, roughly aligned with the various MySQL tables (or by an alternate interpretation, with the different website views).

It is not the intention of this section to describe the functions in the library, neither from the standpoint of an API (for that, see Appendix A.2) nor from that of a description of functionality (for that, see the code). Instead, several conventions used within the library are discussed, with an overall emphasis on general purpose, rather than specific functionality. Again, the reader is strongly encouraged to see Appendix A.2 for the API, which includes a complete description of all functions in the library.

As a side note, since the majority of the data passed between files in Worldview Manager is numeric, the data passing library adds an extra layer of security by forcing the data it operates on to be numeric, when appropriate – either by checking if `is_numeric()` returns `true` before operating on data or simply by overwriting the data to its `intval()` representation. That said, developers wishing to use the data passing library are still encouraged to perform their own security checking on data.

5.3.1 Getting data

The majority of views in Worldview Manager – indeed in any site with user interaction as the primary feature – need to read data generated earlier in another HTTP connection. In many cases, this data comes from the database, in which case it is provided by the database querying library (see section 5.2). However, some data is either too new to have yet been written to the database (e.g. the response to a statement as it is being passed to a function whose purpose is to *record* it in the database rather than to *read* it) or is not the sort of data that is written to the database in the first place (e.g. which link a user clicked on to load a particular view). This sort of data might be passed in a number of ways including writing to a file, communicating through a pipe, etc. In the case of our prototype, however, we limited ourselves to the somewhat more humble functionality of HTTP `GET` and `POST` variables.

The names of functions whose job it is to read data from a HTTP variable are, as a rule, prefixed with "`get_`". These functions do not take any arguments because they read from an external source (the HTTP variables). All of them return some data that is referenced by the values held in the relevant variables, though the exact data (either the value itself or the data referred to by that variable) depends on the specific function. These functions have no side effects.

5.3.2 Sending data

Since the functions that read data do so primarily through HTTP `GET` variables, which, in practical terms, are almost inseparably tied to URLs, it is very important that URLs are formatted correctly so that the data can be read properly. The data passing library attempts to centralize and abstract this process by a series of functions whose job is to "send" data. As a rule, the names of functions whose job it is to send data begin with `send_`; they take an argument, which is the data to send, formatted in some way (see the API in Appendix A.2 for specific details) and return a string, the URL of the data. These functions also do not have any side-effects.

5.3.3 Miscellaneous functionality

There are several functions in the library that have jobs that do not fit into the generalizations outlined above. One is `redirect()`, which in our prototype's case is a simple wrapper for the HTTP `Location:` header. The other is `make_deferrals()` which is a wrapper for the `defer_tension()` function from the database querying library. It has a side effect but is in the data passing library primarily to follow through on the promise of centralizing all HTTP `GET` and `POST` functionality. Please see Appendix A.2 for the API, which contains more details on these (and all other) functions.

5.4 Miscellaneous framework features

Besides the two main components of the framework – database querying (see section 5.2 and data passing (see section 5.3) – several other miscellaneous features are provided for the purposes of abstraction, centralization, and convenience. These are outlined in the sections below.

Note that the file `manager/lib/usercookie.php`, while being fairly tightly coupled with the functions in the framework, is not, strictly speaking, part of the framework itself. Instead, it is the glue that connects the Sessions feature of the website (see section 6.1) with the rest of the library.

5.4.1 Templates

Templates are simple ways to mix static and dynamically-generated code. A template file is simply an HTML file with extra tags, stored in the `manager/templates/` directory. The template system is inspired by that of Django, but is considerably simpler.

A template is loaded by a PHP file by creating an instance of the `Page` class (as defined in `manager/lib/templates.php`). The constructor for `Page` objects takes two arguments – `$template`, the template filename (relative to `manager/templates/`), and `$tags`, an associative array with the keys being template variable names and the values being the values (which must be strings) that those variables hold.

Template variables are unique strings enclosed in double-curly-braces (i.e. {{ `varname` }}). Their value is also a string, and when a `Page` object is created, each instance of {{ `varname` }} in the code is replaced by `$tags["varname"]`, if it is set. The resulting code is placed in the public variable `$this->output` (where `$this` is a pointer to the specific instance of the `Page` class).

Note that unlike in Django, the template system is not a self-contained, minimalistic language. There is no way, for instance, to use a loop within a template, and one cannot use the template languate to access member variables or methods.

Note also that, in many cases, the data passed to a template will actually be generated by another template. As an illustration of this, several "generic" templates are used throughout the website. For instance, the template in `manager/templates/generic/header.html` has three variables, one for text to go between the HTML `<title></title>` tags, one for the navigation bar code, and the third to act as the "Loc" text (see section 5.4.2 below for more on the navigation bar and Loc). The function `header_template()`, as defined in `manager/lib/util.php` returns the output that is generated by this template, and can be used in other templates.

5.4.2 Navigation bar and Loc

There are two methods of global site navigation within Worldview Manager – the navigation bar and Loc.

The navigation bar has links to the main sections of the website. They are defined by two variables in the function `navbar()` as defined in `manager/lib/util.php`. The first variable is the associative array `$urls` which matches PHP files to semantic URLs (see section 6.3 for more on semantic URLs as defined in the `.htaccess` file). The second variable is the associative array `$pages` which matches semantic URLs to the text of the link in the navigation bar page. When calling the `header_template()` function, the third (optional) variable is called `$page`. It is passed as an argument to the `navbar()` function, and checked against each key of `$urls`, and the resulting value, if it exists, is checked against each key of `$pages`. If one matches, the HTML code for the link is given the additional HTML class "`curr`", which allows the CSS (specifically `manager/media/css/header.css`) to style the link appropriately.

Loc is a system completely decoupled from the navigation bar. Its goal is to generate a small text of links near the top of pages that simulate a sort of tree-like structure to the webpage. Each link is of the form `<span ...><a href="...">...</a></span>`. Together, they are all surrounded by `<div id="loc"></div>` tags.

Loc defines a new class, called `PathElement`, which is constructed with a mandatory `$text` variable (between the `<a></a>` tags), an optional `$span_special` variable (usually holding either `false` or `class="separtor"`) that can go inside the `<span>` tag, and an optional `$link` variable, which can hold the value that goes inside of the `href` argument of the `a` tag.

Loc provides three functions. The first, `make_path()`, takes either a single `PathElement` or an array of them and produces a new array, `$path`, which has each of the `PathElements` in the same order, but with a special "separator" element between each one and a "root" element followed by the separator element prepended. The root element is an instance of `PathElement` with `$text` set to "Worldview Manager", no `$span_special` and a `$link` pointing to "`/worldview/home`". The separator element is an instance of `PathElement` with `$text` set to "`&raquo;`" (the HTML symbol for ≫), `$span_special` set to '`class="separator"`', and no `$link`. This `$path` variable can then be passed to two other functions, one called `make_title()`, which returns the `text` member variable of all the `PathElements` passed to it (i.e. just the text, no `<span>` or `<html>` tags), and the other called `make_loc()`, which returns the `html` member variable of all the `PathElements` passed to it. The strings returned by `make_title()` and `make_loc()` can then be passed as the first and second arguments, respectively, of `header_template()`.

6 The website

The website component of Worldview Manager consists of several PHP files which utilize functions from the framework to generate relatively simple and clean HTML, CSS, and Javascript[11] files. Most of the data used to generate these files is stored in a MySQL database (see section 4). As written, the website is designed to be used with Apache, though porting to another web server should be relatively straight-forward as the only Apache-specific features that are are used are in the `.htaccess` file – see section 6.3).

[11]Including some AJAX, when the browser supports it. See section 6.2.8 on the Comments view.

6.1 Sessions

The prototype has no concept of user accounts, though adding them as an extra feature would be relatively straight-forward, to the point that much of the website and framework were written specifically with the intention that such an addition could be easily accomodated. Instead, we track users using "sessions". Since HTTP is a stateless protocol, the system simulates the concept of a session using HTTP cookies.

As described in section 4 (Database structure), the system keeps track of "users" via user IDs. A user ID is stored in a HTTP cookie called `worldview_uid`. In order to prevent malicious users from tampering with the cookie and thus being identified as a different user (something that is a serious security issue with user accounts and at best a glaring oversight without them), there is a second cookie called `worldview_hash`. This cookie is a string of secret "salt" text (hardcoded in the PHP code) along with a user ID, all encrypted with the SHA-256 algorithm. Before identifying the user with a particular user ID, the program encrypts the user ID against the salt and compares the hash values. If they match, the user is identified with the user ID, and if not, a new user ID is given, and both cookies are overwritten.

There is also a third cookie, called `worldview_topic_id`, which contains the Topic ID of the user's "current topic". This feature is the main reason that having cross-topic tensions is not entirely supported. However, it exists because it allows for a natural way for a user to independently work on several different topics, and to allow the system to present him or her with a summary of only the statements from the topic he or she just completed (though a way to show a summary of all statements responded to this session is also available). If there is no current topic, the cookie holds the value "`0`".

All three cookies are available from the root of the server (necessary due to the semantic URLs – see section 6.3) and expire a day after they are set (which happens every time a new page is loaded).

6.2 Views

Following accepted web design practices, Worldview Manager is organized around a MVC structure. The Model is encapsulated in the database and the queries made to it (see section 4 fore more on the Database structure and section 5.2 for more on the the database querying library). The controller is encapsulated in the data passing system (see 5.3 for more). The views are encapsulated in a custom HTML template system, along with statically-written CSS and Javascript files, as well as PHP scripts that generate further HTML, CSS, and Javascript code dynamically.

6.2.1 response.php

The file `manager/response.php` is used heavily by the system. Its job is to parse responses to statements and redirect to proper views.

Before doing anything else, it checks to see whether the previous action was to "defer" a tension (see section 6.2.6 for more on deferred tensions), and if so, it makes the deferral. After doing this, it checks to see whether a response was given, and if so, records the response in the database.

It then determines the next view to redirect to in the following order:

- If the function `find_disagreements()` in `manager/lib/db/query/query_tensions.php` returns a tension, the function redirects to the Tension view (section 6.2.4) with the tension information.
- Otherwise, the function checks to see if the `pick_statement()` function in `manager/lib/db/query/query_statements.php` returns a statement ID. If so, it redirects to the Statement view (section 6.2.3) with the statement information. This happens whenever there are statements *in the current topic* that the user has not yet responded to.
- Otherwise, the function checks to see if the user has made any responses to the current topic at all. If so, it redirects to the Summary view (section 6.2.7), and if not, it redirects to the Topic view (section 6.2.2).

All redirections made by this file are done using the function `redirect()` (as defined in `manager/lib/data_passing/data_passing_util.php`), which is a wrapper for the HTTP `Location:` header. This adds some overhead to page loading time, but we believe that the benefits provided by this arrangement outweigh the costs. Specifically, the system allows complete decoupling of the framework from the rest of the website. This sort of decoupling is stronger than the kind that could be achieved by having a single centralized file that dispatches functions, because the components are closer to being self-contained, complete programs.

6.2.2 Topics

The Topic view is handled by `manager/topic.php`, with the templates `manager/templates/topic/topic.html` and `manager/templates/topic/topic_page.html`, and styled by the CSS file `manager/media/css/topic.css`. There are actually two different Topic views – one with links to all the different topic pages and the other being the topic pages themselves.

The former is launched if `topic.php` is launched with no `topic_id` HTTP `GET` variable or with a value that is not a valid topic ID. The view prints a list of links to all the topics, with the URL being `/worldview/topics/[topic_id]` and the text of the link being the `topic_text`. The URL is then rewritten by the `.htaccess` file to `/worldview/topic.php?topic_id=[topic_id]`. A list of all topic IDs that have been completed (i.e. all statements within them responded to) by the current user is obtained, and all links to these completed topics are given the extra HTML ‘`class="done"`’ which is styled to produce a link written in a grey font color and with a line through it. If there are any such links, an extra explanation about what such links mean is added. When this view is loaded, the `worldview_topic_id` cookie is set to `0`.

Topic pages are the other kind of Topic view. This view is launched when the `topic_id` `GET` variable contains a valid topic ID. When this view is loaded, the `worldview_topic_id` cookie is set to the topic ID of the topic, and some generic instructions about how the website works are presented along with the specific `topic_name` and `topic_explanation` and a link back to the main Topics page. Also, the system detects the "status" of the topic – whether the user hasn't responded to any of the statements, whether he or she has responded to some of the statements, or whether he or she has responded to all of the statements. The specific status is displayed along with a link that allows the user to either "begin", "resume", or "clear answers and start over" the process of responding to statements. In the "resume" status, an extra opportunity to clear old answers before resuming is presented, and in the "clear answers and start over" status, an opportunity to view a summary of the responses given is presented (see section 6.2.7 below for more on the Summary view). In all cases, the link of the action points to `/worldview/topics/start` (with the exception of the "clear answers and start over" link which points to `/worldview/topics/start?done_topic=[topic_id]`). The `.htaccess` file rewrites this URL to `/worldview/start_topic.php` (preserving the GET variable, if present), which then redirects to `response.php` (see section 6.2.1 above for more on `response.php`).

6.2.3 Statements

The Statement view is somewhat more complicated than the Topics view. It is handled by `manager/statement.php` and uses a collection of templates located in `manager/templates/statements/`. This view (as well as the Tensions view – see section 6.2.4 below) attempts to use a modified version of the "Slider" package, written by Erik Arvidsson and distributed under the Apache Software License 2.0 at `http://webfx.eae.net/dhtml/slider/slider.html`. The Slider package is essentially a collection of static Javascript (in `manager/media/js/slider/`), CSS (in `manager/media/css/slider/`), and image (in `manager/media/images/slider`) files intended to be called from HTML code that uses them. However, since Worldview Manager uses templates, and, especially in the Tensions view, the file needs to be generated mostly dynamically, this process is somewhat more involved than the standard method that is described in the Slider package's documentation.

Specifically, each template creates an instance of the Javascript `Slider` class called `s`x where x is the number of the slider on the page. For the Statements view, there will only ever be one slider, so x is always 1. The `Slider` class is defined in the Javascript file `manager/media/js/slider/slider.js`. Internally, the Slider system uses a hidden `<input>` for each slider, with the name `si`x. Additionally, each slider is followed by a `<noscript></noscript>` block which contains a simpler HTML form if Javascript is disabled. It is worth noting that this is different from the documentation's suggested use, which is to create a text input that is hidden only by javascript; we create a hidden input because we have the more complicated `<noscript></noscript>` system for browsers that don't support Javascript.

Note also that while the database and framework assume that user responses are encoded as floating point numbers from `-1` to `1`, the Statement view internally represents them as integers from `-10` to `10`. This is done because the Slider package does not support fractional increments. All conversions are done by `manager/statement.php` and helper functions.

Besides presenting the slider, it is the responsibility of the Statement view to present generic instructions for the user, as well as the actual statement text, and a button labeled "respond", that, when pressed, submits the actual response (along with the statement ID, which is a hidden field), via `HTTP`'s `POST` method to `response.php`. Also, if there are any deferred tensions (see section 6.2.6 below), a link that allows the user to address the tensions is provided.

The URL for the Statement view is of the form `/worldview/statements/[statement_id]`. This is translated by the `.htaccess` file to `/worldview/statement.php?[statement_id]=0`. The reason for this is that, since the statement has not yet been answered, the slider will be initialized at position `0` ("completely neutral"). The Tension view (see section 6.2.4 below) uses a similar URL structure, except that the sliders are not initialized at `0`.

6.2.4 Tensions

Though it appears as a separate view to the user, the Tensions view is actually a separate mode of the Statements view. In it, multiple statements are presented side by side in a table (at most two per row). Additionally, the view is responsible for providing (somewhat different) generic instructions, a link to "defer" the tension, and a link that opens the tension explanation (if available) in a popup window (the tension explanation is stored as `tension_text` in the database).

Each slider is initialized to the response given to it via the URL (in much the same way as they are given in the Statements view, except the response is set to something other than `0`. The same "Respond" button is provided and has the same internal effect – the response to each of the statements involved in the tension is updated, and after which the next view (which might be again a tension, possibly the same one) is loaded.

6.2.5 Tension explanations

The tension explanation view is provided by `manager/tension.php`. It is meant to be opened in a popup window. Internally, it uses the data passing library (see section 5.3 above) to retrieve the tension text and format it in a readable manner. This is performed by the function `handle_explanation()` in `manager/lib/explanation_parser.php`. For more on how this function works, see the end of section A.3 (miscellaneous functions of the API).

6.2.6 Deferred tensions

In some cases, a user might be reluctant to change their response to a statement, even if they are presented with an explanation as to why it is logically inconsistent. In this case, the user may choose to "defer" the tension. Deferred tensions are never again presented to the user, except that the Statement, Tension, and Summary views each have a link mentioning that there are still unresolved (i.e. deferred) tensions that ought to be addressed.

Each of these links points to /worldview/tensions/deferred which is rewritten by the .htaccess file to /worldview/deferred_tensions.php. This file generates a view which is essentially a list of “tension summaries” – each contains the involved statement texts and the responses given to each (“agreement with” or “disagreement with”), along with a link to the tension explanation (if one exists), and a link to resolve the tension (which simply points to the appropriate Tension view – see section 6.2.4 above).

6.2.7 Summaries

The Summary view, also called the “Done” view, is generated by manager/done.php and uses the manager/templates/done.html template. Its job is to inform the user that the topic is completed and to present him or her with the responses that were given. It has two modes of operation – topic-specific and holistic.

In the topic-specific mode, only the responses to statements in the specific topic are shown. The topic is determined first by a HTTP GET topic_id variable and then, if one is not provided, by the HTTP worldview_topic_id cookie. If neither exist, the system switches to holistic mode. In topic-specific mode, the system checks if there were responses to a statement in a different topic, and if so, provides a link to the holistic mode.

In holistic mode, a summary of every response that was made “this session” (that is, by the current user ID), is shown. If the worldview_topic_id cookie is nonzero, a link to the topic-specific mode for that topic is provided.

In both modes, a link back to the topics page is displayed at the top of the page. Also, if there are any deferred tensions, a link to the Deferred tensions view is provided in bold, red font.

The actual summaries are generated as a table. Each statement text is shown next to a graphic representation of the response (generated by tiling images in manager/media/images/done_summary) and a textual representation of the response.

6.2.8 Comments

The Comment view is a special view that is embedded in other views – specifically in the Topic, Statement, and Tension views. Comments are simple ways for users to communicate their thoughts and objections to the data on the relevant pages. Comments are handled by the manager/post_comment.php program, which accepts comment text via HTTP POST, saves them to the database, and, depending on whether the ajax variable was set to true in the data stream, either prints the comment encoded in HTML (which is then displayed on the page via AJAX through a small Javascript program in manager/media/js/comment.js) or redirects back to the target area which, when printed, calls the make_comment_area() function which uses the manager/templates/comments/comment.html template to display all of the comments written on a page.

Besides the comment text, comments provide an opportunity for a user to enter a name, which is then associated with the user ID in the database and presented automatically in all comment forms for that user ID. The comment template displays the name of the poster before each comment – the name in the database if one exists, and “Anonymous” otherwise.

6.2.9 Other views

The program has a number of other views which are essentially static pages – they use a single template (except the generic templates which all pages use), and a PHP script that loads and prints the template. This is done instead of serving a static HTML file specifically so that generic header and footer templates could be used, maintaining a consistent look throughout the website.

6.3 The `.htaccess` file

Worldview Manager is hosted on an Apache web server and is configured using an Apache `.htaccess` file. Some of it is needed due to quirks on the system that our implementation is hosted on. Specifically, we explicitly denote `.php` files as `php5-script` because the default on our server is `php4-script` and we use features specific to PHP 5. Also, we explicitly *enable* Magic Quotes via the “`php_value magic_quotes_gpc On`” directive because the system uses the PHP function `stripslashes()` to bypass its effects instead (the motivation for this is to accomodate systems where disabling Magic Quotes is impossible).

The majority of the file, however, is a collection of URL rewriting rules using `mod_rewrite` that together create readable semantic URLs for Worldview Manager. We generally have the following policy:

- Ignore case in all URLs (via the `[nocase]` flag).
- Ignore trailing slashes in all URLs (via the `/?` regular expression).
- Allow backward compatibility for all PHP scripts (via a regular expression that redirects to the script).

Please see the actual file (located in `manager/.htaccess`) for more on exactly which rewriting rules are used.

6.4 Administration pages

There are a number of administration pages that come with Worldview Manager. They are not intended to be accessible to users and therefore should be password-protected on a public server. However, if Worldview Manager were to be extended to include user accounts, much of their functionality could be ported for the purpose.

6.4.1 Topic Manager

The primary feature of the administration pages is to act as a front-end to the compiler suite (see section 7 below). Specifically, the file `manager/admin/topic_manager.php`, which uses the `manager/templates/admin/topic_manager.html` template generates a list of the current topics with a "delete" link next to each one, and provides a simple web-based interface to upload another topic file (see the "Topic files" section below).

6.4.2 Statistics

Another page not intended for users is the Statistics view. The file responsible for it is `manager/admin/statistics.php`, but, like all other views, many functions from the framework are used. Unlike other views, there is no Statistics template – the generic header and footer templates are loaded but the rest of the code is generated and `print`ed directly. The reason for this is that due to the potentially large amount of data to print, the page is likely to load slower than other pages, and this method prints data on the screen as it is generated.

The view has two main features. The first is tension suggestions: if the system notices unusually large correlations in responses (e.g. the vast majority of people who answered negatively to statement `a` also answered positively to statement `b`, but there is no explicit tension in the database between `a` and `b`), the system will display this in another table. The second feature is a summary of every statement – a table with statement ID, statement text, the percentage of negative, neutral, and positive responses, and the total number of respondents is printed.

7 Compiler suite

The Worldview Manager framework provides a flexible and rich means to manipulate the database and extract data from it; meanwhile the website provides a usable and clean interface for the user. However, as it stands, there is no easy method to actually *populate* the database – there is a specification for what data in the database should look like (see section 4) but no specification about how the data should be placed there.

In a sense, this is intentional, because it means that Worldview Manager can be fairly seamlessly paired with other projects, including data inference systems and other information databases. However, we believe that even a prototype would not be complete without some means to upload topic files and tensions (beyond directly adding data through some MySQL client). Hence, the "Topic file compiler suite".

7.1 Topic files

Topic files are, as the name implies, files that contain the contents of a Worldview Manager topic – that is, a list of statements and a list of tensions between them. The former is done via a simple list of "statement declarations" and the latter via a fairly robust system of "implications". There is also support for comments.

Topic files do not contain the topic name or topic description. These are passed as arguments to `mktopic` (see sections 7.3 and 6.4).

The files are written in a plain-text format. Since the text can get somewhat dense, however, we have found it useful to use some sort of specialized editor. Therefore, we have provided `wvm-mode.el`, an Emacs major mode for Worldview Manager topic files.

The topic file structure is arguably the biggest bottle-neck to having cross-topic tensions, because there is currently no way to reference statements defined in one topic file from another. Thus, if the compiler suite is not used, and the database is populated with cross-topic tensions in some other way, the system should work fine.

An example topic file is located in `topics/testFile.wvm`.

7.1.1 Statement declarations

Any line that begins witha block of only alphanumeric/underscore characters (ignoring whitespace) is interpreted as a statement declaration. That first block is known as the "statement identifier" and the rest of the line after the first block of whitespace after the identifier is treated as the "statement text". The statement ID (that is, the numerical identifier for statements as stated in the database) is not in any way encoded in the topic file. It is generated by MySQL and handled by `mktopic` (see section 7.3).

7.1.2 Implications

Worldview Manager works with tensions as tuples of statements and response directions. Consider, as an example, two statements with the identifiers `FOO` and `BAR` (see section 7.1.1 above for more on statement identifiers; identifiers are stored in the database but not used for anything but the concepts described in this section). If agreeing with `FOO` and disagreeing with `BAR` was logically inconsistent, the system would have to be told this as the tuple (`FOO + BAR -`) (this is in the spirit of, but not quite how the database actually stores tensions. For that, see section 4, specifically the `tensions` bullet). This is somewhat awkward for a human to think about – it is much more intuitive to say "`FOO` implies `BAR`" than to say "Agreeing with `FOO` and disagreeing with `BAR` is a logical contradiction".

Therefore, rather than storing *tensions* directly, we store *implications.* All tensions are written in parentheses. Any line that begins with an open-parenthesis (again, ignoring initial whitespace) is treated as an implication. Statements are referenced by the statement identifiers given in statement declarations (see section 7.1.1) and the actual implications using standard logic notation symbols:

- "`=>`", "`==>`", "`->`", "`-->`", or "`IMPLIES`" for implications
- "`<=`", "`<==`", "`<-`", "`<--`", or "`IF`" for reverse implications
- "`<=>`", "`<==>`", "`<->`", "`<-->`", or "`IFF`" for equivalencies
- "`&`", "`/\`", or "`AND`" for conjunctions

- "|", "\/" or "OR" for disjunctions
- "^" or "XOR" for exclusive-or
- "!" or "NOT" for negation

Note that the word forms of all logical operators are case-insensitive.

The system also supports parentheses for grouping but in most cases, this is not necessary because grouping operators (conjunctions, disjunctions, exclusive-or) take precedence over relations (implications, reverse implications, equivalencies). The main exception to this is that the "NOT" keyword requires grouping. Thus, to write that agreeing with both FOO and BAR is logically inconsistent, we would write "(FOO IMPLIES NOT(BAR))". This does not apply to the "!" symbol, so the same could be written as "(FOO IMPLIES !BAR)" or "(FOO => !BAR)" (word and symbol notation can be mixed arbitrarily).

Note also that the relation operators (implications, reverse implications, equivalencies) can generally be used on other arbitary statements. So lines such as "(FOO => (BAR <=> BAZ))", "(!(FOO => BAR))", etc. are perfectly valid and have the expected meanings.

After the final close-parenthesis, there can be an optional tension explanation. It must be separated by at least one whitespace character from the close-parenthesis, after which everything to the end of the line is written as the tension explanation. Note, however, that some constructs are effectively short-hand; for instance, writing "(A | B => !C)" will actually be expanded internally as two separate tensions – "(A + C +)" and "(B + C +)". The tension explanation given to the short-hand logical notation will be copied to both, which may not be the intended effect. Please remember this when using grouping symbols and writing tension explanations.

7.1.3 Comments

The system supports "comments" in topic files. These are ignored by the system but can be useful for remarks and explanations for people reading the topic file. Comments begin with "// " (two comments followed by at least one whitespace character) and end at the end of a line. Comments take precedence before everything else, so if "// " is seen *anywhere* in a line, the rest of the line starting from the "// " will be ignored. Please note that the whitespace character (which doesn't need to be a space, and can be a TAB character, etc.) *must* be present; this is to allow links in tension explanations (or else everything after http:// would be interpreted as a comment).

7.1.4 wvm-mode.el

Topic files are written and encoded as plain text files, but it can be useful to have a specific editor for them. In lieu of this, we provide an Emacs major mode – wvm-mode.el, located in topic_file_parsers/wvm-mode.el. When loaded, the mode associates all files ending in .wvm with itself. It treats statement identifiers as functions (colors them blue and prints them in bold font, by default) when used in the beginning of statement declarations, and treats them as keywords (colors them orange, by default) when used in implication lines. Comments are written in the comment font (red, by default). Logical operators – both in symbolic and (capital-case only) word form – are treated as keywords (cyan and bold, by default).

Note that, due to a glitch in the Emacs mode system, comments are resolved after everything else, so if a comment contains parentheses or any keyword, it will not be colored as a comment. Similarly, if the statement text contains any symbol or text in parentheses, they will be colored as described above. For this reason, `wvm-mode.el` only recognizes the word form of logical operators when written in all uppercase. If they are not written in uppercase, they will not be colored, though the rest of the system will interpret and understand them perfectly.

7.2 `WVMPreprocessor`

`WVMPreprocessor` is charged with the following tasks, which are undertaken roughly in the order indicated.

1. Eliminate comments and other formatting-only components of the topic file.
2. Parse potentially intermixed statements and tensions, including parsing the logical expression for each tension
3. Report a warning for apparent typos (statements referenced only once that lack an explanation) and for boring statements (statements with no tensions referring to them).
4. Process LaTeX code in statement and tension explanations, replacing it with HTML when possible and images otherwise.
5. Infer new tensions of length up to 4, constructing readable composite explanations.
6. Eliminate duplicate tensions, or tensions for which a strict subset of the conditions also form a tension.
7. Output a new tension file.

At each step, `WVMPreprocessor` may also encounter errors or warnings which should be passed in a reasonable way along to PHP. Design decisions and algorithm outlines are presented below.

7.2.1 Parsing

The processing begins by eliminating spaces at the beginning of lines, and any part of a line that occurs after the string "//" *followed by a space character.* (This allows the use of `http://` in HTML, for instance.) [12]

[12] All parsing is done using the Haskell Parsec library. For details, see `http://hackage.haskell.org/package/parsec`.

The next nontrivial piece of the program is identifying statements and tensions. The problem of intermixed statements and tensions is eliminated by keeping the collections of statements and of tensions totally separate until after parsing is complete, and by adding a simple alternative. The processing is done on an specialized error monad transformer, holding an error type designed to handle the specific errors errors encountered by `WVMPreprocessor`, wrapping a `Parser` that maintains in its state

1. A `Map` from `String`s to `String`s, here used to match statement identifiers to their explanation.

2. An `Expression`. `Expression`s hold an entire Boolean expression and explanations for each component, and are generally described and defined in section 7.2.4. This `Expression` is a *necessary condition* for a set of responses to be considered noncontradictory, and will later be assumed sufficient.

This tuple is held in a datatype designated `TopicFile`.

Statements are recognized by the beginning of a line with a *variable*, which is restricted to a string formed of alphanumeric characters and underscores. After the variable comes at least one but possibly more space characters – tabs and spaces, but no newlines – and the remainder of the line becomes the statement explanation. Since we have already eliminated spaces on both ends of each line, as well as any comments, this suffices. If a statement is successfully parsed, the `TopicFile` is updated with the new statement. **Note:** If two lines with the same statement ID are parsed, *the explanations are concatenated.* This allows multi-line statement explanations to be handled naturally. *

Tensions are recognized by a Boolean expression in parentheses, followed by any number of spaces (possibly zero) and an explanation. The parsing of Boolean expressions is discussed in section 7.2.3, and the explanation is assumed to continue until the end of the line. In particular, the full meaning of the Boolean expression is parsed and then that expression is tagged with the specified explanation. **Note:** If the expression has arity greater than MAX_ARITY, a warning is returned and the tension is ignored. Otherwise, if the previous `Expression` held X, and x was parsed from this tension, then $x \wedge X$ becomes the new `Expression`. **Note:** that the Boolean expressions in the original topic file should express positive statements about a *consistent* set of responses. * *

7.2.2 Statement/tension processing

After this point, we check for missing or boring statements. A *missing* statement is referred to by at most one tension but no explanation for this statement was given; in this case, it seems likely that this reference was in fact a typo. A *boring* statement was given an explanation but did not appear in any tensions. If either type of statement exists, an appropriate warning message is returned but the processing continues.

Next, we process LaTeX in statement and tension explanations. We compile a list of all LaTeXsnippets in the topic file, including both statement and tension explanations. As part of this process, we do some significant parsing of the LaTeX, in particular identifying commands and giving them an empty argument set if no arguments were previously given. We then separate each snippet by spaces and collect a list of distinct LaTeX pieces. All of this extra work is done with the objective of minimizing the number of distinct pieces of TeX to be compiled, because texvc, MediaWiki's TeX-to-HTML converter, is painfully slow. After compiling the full list, we export that list into `vcrunner`, a small bash script that runs texvc on each piece of code with the appropriate parameters. We assemble the output of `vcrunner` into a mapping between TeX words and compiled HTML. Finally, we take a second pass through statement and tension explanations, replacing any TeX with the HTML code. **Note:** if your topic file is taking too long to compile, you may have TeX code that should have more spaces separating distinct pieces. *

After resolving and simplifying tensions (covered in (7.2.4)), we check for contradictions. Any clause of the form x, $\neg x$, or a vacuous clause (implying False) is considered a contradiction, as the spirit of Worldview Manager is not to dictate responses to any user. A contradictory clause is handled with a warning.

Finally, we display the compiled topic file. Statements are reproduced in lexicographical order of their identifier, followed by tensions. The details of how tensions are shown are covered in (7.2.4).

7.2.3 Boolean expression parsing

An expression can be any of the following, in descending order of precedence. Let p, q be *factors*, to be defined later. (All binary operators are infix.)

- $p \wedge q$. Valid conjunction symbols include "`&`", "`and`", and "`/\`"
- $p \vee q$. Valid disjunction symbols include "`|`", "`or`", and "`\/`".
- $p \Rightarrow q$. Valid implication symbols include "`=>`", "`==>`", "`->`", "`-->`", and "`implies`".
- $p \Leftrightarrow q$. Valid equivalence symbols include "`<=>`", "`<==>`", "`<->`", "`<-->`", and "`iff`".
- $p \Leftrightarrow \neg q$. Valid exclusive-or symbols include "`^`" and "`xor`". (Parsing this is considerably more efficient than parsing $p \Leftrightarrow \neg q$.)
- $p \Leftarrow q$. Valid reverse-implication symbols include "`<=`", "`<==`", "`<-`", "`<--`", and "`if`".
- p.

A factor is defined as follows:

- Parentheses around an expression.
- The negation of a factor. This can be "`!`" followed by a factor with optional spaces, "`not p`" where `p` is a factor, or "`not(p)`" where `p` is a factor.
- A variable, which must take the form of a statement identifier – composed of alphanumerics and underscores.

7.2.4 Boolean expression manipulation

Expression manipulation is done in the module `CNFExpression`, so named because it encapsulates the manipulation of expressions held internally in conjunctive normal form. In particular, a *clause* – a collection of assignments from variables to values, at least one of which must be true – is held in a simple `Map String Bool`, and an expression is held as merely a list of clauses, which are intuitively being $\wedge$'d together. In this context, computing $X \wedge Y$ is as simple as list concatenation; $X \vee Y$ becomes

$$X \vee Y = \bigwedge_{\substack{x \in X, y \in Y \\ \neg taut(x \vee y)}} x \vee y$$

where $taut(x \vee y)$ indicates whether or not $x \vee y$ is a tautology, which is true if and only if x is satisfied by l and y is satisfied by $\neg l$ for some literal l.

$\neg X$ is computed by noticing that if every literal l is replaced by $\neg l$, we essentially get the negation of X in *disjunctive* normal form. This is as simple as replacing each value in a clause with its negation, treating the entire expression as if it were in disjunctive normal form, and then converting it back.

These three operations are sufficient to parse a Boolean expression. Now, we can get into the fun part: the resolution algorithm. Here are some definitions:

```
type LogicClause = Map String Bool      -- maps variables to assignments
                                        -- any one of these assignments would
                                        -- satisfy the clause
type Clause = (LogicClause, Explanation)
type Literal = (String, Bool)
```

The resolution engine is probably the most algorithmically sophisticated piece of the preprocessor. It tracks the following:

- `Seq Clause`: a queue of clauses waiting to be resolved into the overall expression.
- `Map LogicClause Explanation`: a mapping associating each distinct logical clause to its explanation, ensuring that no duplication of clauses is permitted. The conjunction of these clauses is the current value of the expression.
- `Map Literal (Set LogicClause)`: the critical piece of the resolution engine, this maps variable assignments to every clause that would be satisfied by that assignment.

Consider now the task of `resolveClause`: given a clause C and an explanation, return all resolutions of this clause with the current value of the expression. In particular, given a clause

$$C = \vee_i l_i$$

where each l_i is either a v_j or a $\neg v_j$, we seek clauses in the main expression of the form

$$C' = \bigvee_j l'_j \vee \neg l_k$$

for some k, because then $C \wedge C'$ implies

$$\bigvee_{i \neq k} l_i \vee \bigvee_{j} l'_j$$

Of course, if any of the l_i are negations of any of the l'_j, this statement is trivially true, and therefore useless to us. We therefore seek clauses which contain *exactly* one $\neg l_k$.

The key to the algorithm is that the `Map Literal (Set LogicClause)` tells us exactly which clauses contain each literal, so:

1. For each literal (association pair) in our original clause, we look up the negation of that literal to get a `Set LogicClause`. (We now have a list of these `Set`s, perhaps one per literal.) Let the set associated with $\neg l_i$ be S_i.

2. From each S_i, construct a `Map` with the associations $\{(C', [v_i]) \mid C' \in S_i\}$. That is, we map each clause C' from S_i to the singleton list of the variable of l_i, a variable on which C' differs from C. (Note also that for *each* variable v of difference with any given C', there is a `Map` associating C' to $[v]$.)

3. We take the union of all these maps, combining values with list concatenation. We now have a `Map` associating each C' in our original expression to a list of all variables on which it differs from C.

4. Now, the clauses we can resolve with are *exactly* the clauses mapping to a list of a single element, which is the name of the variable v we want to unify on! For each C' mapping to only a single variable v, we may simply take the union of these two logical clauses (literally a `Map.union`) and delete the assignment corresponding to v.

5. In addition, we must combine their explanations. Combination of explanations is done according to the following heuristic: if $l_i = v_j$, that is, it makes a positive assignment to a variable, then the explanation for C precedes the explanation for C', otherwise the explanations are reversed.

The final piece to the algorithm is relatively simply implemented: the simplification algorithm, whose objective is simply to test for clauses that imply one another. Implication is handled almost trivially by the function `isSubmapOf`. If one map is a submap of another map, the first map is an implication of the second (since the second is more general).

7.2.5 Files

`WVMPreprocessor` is located in `topic_file_parsers/WVMPreprocessor/`. This directory contains both source and binary files of the program, as well as a file called `buildWVM`, which can be used to easily compile the program, and a file called `vcrunner` which is used by the program to run `texvc`.

7.3 `mktopic`

`WVMPreprocessor` (see section 7.2 above) takes a topic file (see section 7.1) and generates it into a much more verbose syntax. It is the job of `mktopic` to take this newly generated data and, as the name implies, "make it into a topic", that is, upload it to the MySQL database. This is a three-part operation – insert the new entry into the `topics` table, insert the new statements into the `statements` table, and insert the new tensions into the `tensions` table (see section 4 for more on the database structure).

Furthermore, note that the data that `WVMPreprocessor` generates does not include anything about statement IDs because there has not yet been any interaction with MySQL – thus, it is also the job of `mktopic` to remember the numeric statement IDs that MySQL associates with each statement and, when uploading tensions, translate the identifers into these statement IDs.

`mktopic` is written in Scheme and compiled using the Bigloo compiler. It must be called with several command-line arguments in the following fashion: "`mktopic topic-name topic-description data-file`".

Descriptions of design decisions follow:

7.3.1 Reading data

The program needs to be launched with three command-line arguments (excluding the call to `mktopic` itself. The first argument should be the name of the file (presumably generated by `WVMPreprocessor`), which is read via a function called with `call-with-input-filename`. This produces a list of lines, which are first parsed (see section 7.3.2) and then uploaded to the database (see section 7.3.3).

7.3.2 Parsing data

Each line in the list is parsed either as statement or tension, depending on whether or not it begins with a "`(`".

When parsing a line as a statement, the program uses a simple regular expression to extract the statement identifier (`statement-id`) and statement text (`statement-text`) portions of the statement . All apostrophes in `statement-text` are escaped with a backslash (this is to prevent SQL injection), and the pair is added to the associative list `statements`.

When parsing a line as a tension, the program works in several steps: First, it extracts `tension-definition` (the parentheses and everything between them) and `tension-text`. Next, the tension information is read as an s-expression (this is done via `read`ing the result of calling `open-input-string` on `tension-text`). Finally, the result is added as an entry to the associative list `tensions`.

7.3.3 Uploading to the database

Due to a lack of a portable SQL library for Scheme, `mktopic` instead calls `mysql` directly. It does this once, giving it one large file with a series of MySQL commands.

The difficulty with doing this is caused by the fact that the tensions given to `mktopic` use "statement identifiers" while the database stores numerical statement IDs. In order to bypass this problem, `mktopic` creates a local SQL variable for each statement, with the name `@$IDENT` (where `IDENT` is the statement identifier). After all the statements are thus added, tensions are added in the expected way, with the only caveat being that apostrophes in tension explanations are escaped at this stage.

The entire script is then loaded to MySQL. This stage of the process is responsible, also, for reading relevant configuration files (and coming up with reasonable default values) for the database.

7.3.4 Files

`mktopic` is located in `topic_file_parsers/mktopic/`. It comes only with source code. It was tested using Bigloo version 3.2a[13] but should be reasonably simple to port to another Scheme implementation.

[13]See http://www-sop.inria.fr/mimosa/fp/Bigloo/.

A API

Worldview Manager contains a framework that provides a number of functions for developers who wish to create their own websites and programs that use the Worldview Manager database. This API contains all the functions intended for this use, along with the arguments they take, what they return, and, in some cases, how they work. Please note that not every function that is defined in the source files are here – some are helper functions intended only for internal use, and they are not listed here.

A.1 Database querying Library

Please see section 5.2 for more information on the general goals and scope of the database library. All of the functions are in various files in the directory `manager/lib/db/query/`, which are all included by the file `manager/lib/db/query/query.php`.

A.1.1 Topics

These functions are defined in `manager/lib/db/query/query_topic.php`

`topic_detail($tid)` Takes a topic ID and returns the corresponding topic name and text.

`delete_topic($tid)` Takes a topic ID and deletes all database entries for the topic, statements and tensions in the topic, responses to statements in the topic, and comments on the topic and statements and tensions in the topic.

`get_all_topics()` Returns an array of associative arrays, where each associative array has the `topic_id`, `topic_name`, and `topic_text` of a topic.

`topic_id_by_name($topic_name)` Returns the topic ID with the given name, if one exists.

A.1.2 Statements

These functions are defined in `manager/lib/db/query/query_statements.php`.

`statement_detail($sid)` Returns a query with at most one row, containing the statement text and topic id associated with the given statement.

`statement_details($sids)` Takes an array of SIDs, and returns a query returning SIDs, statement texts, and topic ids.

`statement_ident_detail($tid, $ident)` Takes a topic ID and a short statement identifier, and returns the statement ID and statement text associated.

`pick_statement($uid, $topic_id)` returns a query returning at most one statement ID to ask the user about, subject to the following constraints:

- Only returns statements with the specified topic ID.
- No statement that has already been responded to is allowed.
- Minimizes the number of tensions with statements already responded to.
- Maximizes the number of tensions relating to this statement.

Any ties after these ordering constraints are broken randomly.

A.1.3 Tensions

These functions are defined in `manager/lib/db/query/query_tensions.php`.

`get_tension_text($tension_id)` Given a valid tension, produces a public-HTML version of that tension explanation. In particular, this involves passing it to `handle_explanation`, which identifies every distinct statement referred to in the tension explanation, labels it with letters, and produces fully formatted explanations.

`find_disagreements($uid, $topic_id)` Given a user ID and a topic, finds any disagreements between the user's responses. This is pretty much the most crucial function in Worldview Manager. A disagreement is defined as when the user expressly violates the logical implications set forth in the topic file. The extent of a disagreement depends on how many statements are involved. Each statement's disagreement level is scored according to the formula $va/\sqrt{|v|}$ where v is the user's agreement with the statement, scaled between -1 and 1, and a is 1 if the response isn't supposed to be positive and -1 otherwise. The sum of these scores for each statement involved in a tension yields the overall tension score on that tension. The threshold for the total score needed to create a disagreement, depending on the number of statements involved, is held in the array `$TOLERANCES`, which currently has a minimum of 1.25 for a two-statement tension, 2.2 for a three-statement tension, and 3.1 for a four-statement tension.

`defer_tension($uid, $tension_id)` Defers the specified tension for later consideration.

`deferred_tensions($uid)` Returns a query listing tension ids deferred by this user.

`hit_tensions($uid, $sids)` Given a user ID and an array of statement IDs, returns any deferred tensions related to those statement IDs.

`alleviate_tensions($uid, $topic_id)` Deletes any deferred tensions from this user that have since been resolved.

`address_tension($uid,$tension_id)` Deletes the specified tension from the deferred list.

`address_tensions_with_statements_in_topic($uid, $topic_id)` Deletes any tensions with statements in the specified topic from the deferred list.

A.1.4 User/Sessions

These functions are defined in `manager/lib/db/query/query_user.php`

`get_agreement($statement_id)` Looks at the global $USER_ID variable and returns the response the user has provided to the given `$statement_id` (which will be `0` if the user hasn't responded to the statement yet).

`report_name($uid, $name)` Associates the given user ID with the given name. There is no return value.

`get_users_responses($uid, $topic_id=0)` Returns all the responses that the current user has provided. If the topic ID is not `0`, it will only limit to responses in that topic.

`get_name($uid)` Returns the name that is associated with the user ID, if any.

`report_response($uid, $responses)` Takes a user ID and an associative array of responses, with the keys being statement IDs and the values being the response values (from `-1` to `1`.

`first_response($user_id, $topic_id)` Returns `true` if the user hasn't responded to any statements in the topic yet and `false` otherwise.

`clear_responses($uid)` Removes all of the user's responses to all topics and moves them to `responses_history`.

`clear_responses_to_topic($user_id, $topic_id)` Removes all of the user's responses to the provided topic, and moves them to `responses_history`.

`topics_done($uid)` Returns an array of topic IDs of all topics completed by the user.

`other_responses($user_id, $topic_id)` Returns `true` if there are responses left in the given topic and `false` otherwise.

A.1.5 Comments

These functions are defined in `manager/lib/db/query/query_comments.php`

`comment_on_topic($user_id, $topic_id, $comment)` Records the given comment text and returns the new comment's ID.

`comment_on_statement($user_id, $statement_id, $comment)` Records the given comment text and returns the new comment's ID.

`comment_on_tension($user_id, $tension_id, $comment)` Records the given comment text and returns the new comment's ID.

`get_comments_on_topic($topic_id)` Returns an array of associative arrays, where each associative array has a key for the comment ID, comment text, and user ID of the commenter.

`get_comments_on_statement($statement_id)` Returns an array of associative arrays, where each associative array has a key for the comment ID, comment text, and user ID of the commenter.

`get_comments_on_tension($tension_id)` Returns an array of associative arrays, where each associative array has a key for the comment ID, comment text, and user ID of the commenter.

`get_comments_by_user($user_id)` Returns an array of arrays of comment text, area, ID that the given user made.

`get_comment_by_id($comment_id)` Returns an associative array corresponding to a comment row with the given commment ID.

A.1.6 Statistics

These functions are defind in `manager/lib/db/query/query_statistics.php`

`get_statements_with_agreements()` Returns a list of statement IDs, text, and relative responses for every statement in the database.

`suggest_tensions()` Returns an array of arrays of statement/agreement pairs, along with their percentages and total agreements.

A.2 Data passing library

Please see section 5.3 for more information on the general goals and scope of the data passing library. All of the functions are in various files in the directory `manager/lib/data_passing/`, which are all included by the file `manager/lib/data_passing/data_passing.php`.

A.2.1 Topics

These functions are defined in `manager/lib/data_passing/data_passing_topic.php`.

`get_topic()` Checks that the HTTP `GET` `topic_id` variable is set and is a numeric value; if it is not, returns `false`. Otherwise, if the value is `0`, the function returns the array `(0, "", "")`, and if it is nonzero, it passes the data to the `topic_detail()` function as defined in the Database querying library (see section A.1); if it gets back a topic name and description, it returns them in the array `($topic_id, $topic_name, $topic_text)`, otherwise, it returns `false`.

`get_done_topic()` Checks that the HTTP `GET` `done_topic` variable is set and is a numeric value. If so, it returns the value, and otherwise returns `false`.

`send_topic($topic_id)` Converts `$topic_id` to `intval($topic_id)` (which, in the intended use of the function, doesn't make any change, but prevents dangerous input), and returns the string "`/worldview/topics/$topic_id`".

`send_done_topic($topic_id)` Converts `$topic_id` to `intval($topic_id)` (which, in the intended use of the function, doesn't make any change, but prevents dangerous input), and returns the string "`/worldview/done_topic.php?topic_id=$topic_id`".

A.2.2 Statements

These functions are defined in `manager/lib/data_passing/data_passing_statement.php`.

`get_statements()` Goes through the variables in HTTP `GET` and builds an associative array from them. For every key/value pair where both key and value are numeric, the function adds them to the array (the intention being that the key is the statement ID and the value is the response). Also, if the `GET` variable `tension_id` is set, the key "`tension_id`" is associated with "`intval($_GET["tension_id"])`". The associative array is then returned (empty if nothing was added to it).

`send_statements($statements)` The function operates on an associative array stored in `$statements`. If the array contains only one pair, the function creates a variable `$suffix = intval($key)` and returns the string "`/worldview/statements/$suffix`". Otherwise, the variable `$tension_id = intval($statements["tension_id"])` is created and `$statements["tension_id"]` is subsequently unset. The function then returns "`"/worldview/tensions/$tension_id/" . http_build_query($statements)`".

A.2.3 Tensions

These functions are defined in `manager/lib/data_passing/data_passing_tension.php`.

`get_responses()` Builds and returns an array of arrays, where each element array contains a response ID from HTTP `POST` (`response1`, `response2`, ...), and a "response" (either `1`, `0`, or `-1`).

`get_tension_string()` Builds an associative array `$strings` to which every numeric HTTP `GET` variable with a numeric value is added. The function then returns "`http_build_query($strings)`".

`get_tension()` Checks that the HTTP `GET` variable `tension_id` is set and is numeric. If it is not, returns `false`; otherwise the function runs `get_tension_text()` on the tension ID; if the call successfully returns the tension text, the function returns `array($tension_id, $tension_text)`; otherwise, it returns `false`.

send_tension($tension_id) Sets `$tension_id = intval($tension_id)` and returns "`"/worldview/tension.php?" . http_build_query("tension_id" => $tension_id)`".

send_defer_tension($tension_id) Sets `$tension_id = intval($tension_id)` and returns the string "`/worldview/tensions/defer/$tension_id`".

send_deferred_tension($tension_id) Sets `$tension_id = intval($tension_id)` and returns the string "`/worldview/tensions/defer/$tension_id/show`".

A.2.4 Comments

This function is defined in `manager/lib/data_passing/data_passing_comment.php`.

get_comment() Checks that the HTTP `POST` variables `page`, `id`, and `comment_text` are set, that `id` is numeric, and that `page` is set either to "`topic`", "`statement`", or "`tension`". If any of this is not true, the function returns `false`. Otherwise, the function creates another variable `$name` that is assigned to equal the HTTP `POST` variable `name` if it is set, or to the empty string otherwise. Similarly, the variable `$url_extra` is assigned to equal the HTTP `POST` variable `url_extra` if it is set, or to the empty string otherwise. The function then returns `array($page, $id, $comment_text, $url_extra, $name)`.

A.2.5 Statistics

This function is defined in `manager/lib/data_passing/data_passing_statistics.php`.

get_statistics() Checks that the HTTP `GET` variables `page` and `id` are set, that `id` is numeric, and that `page` is set either to "`topic`" or to "`statement`". If any of this is not true, the function returns `false`; otherwise, it returns `array($page, $id)`.

A.2.6 Utilities

These functions are defined in `manager/lib/data_passing/data_passing_util.php`.

redirect($addr) Returns "`header("Location: $addr")`".

make_deferrals() Checks to see if the HTTP `GET` variables `action` and `tension_id` are set and that `action` is set to "`defer`". If all of this is true, the function sets `$tension_id = intval($_GET["tension_id"])`, imports the global variable $USER_ID (see section 5.1), and runs `defer_tension($USER_ID, $tension_id)`. This function, therefore, has a side-effect.

A.3 Miscellaneous Functions

These are some of the miscellaneous functions provided by the Worldview Manager framework. Please see section 5.4 for more information about the framework's miscellaneous features. The functions are defined in `manager/lib/`.

A.3.1 Navbar

This function is defined in `manager/lib/util.php`. Please see section 5.4.2 for more information about the navbar.

`navbar($page)` Returns the HTML code for a navigation bar. There are two hardcoded associative arrays – `$urls`, which matches PHP files to semantic URLs, and `$pages`, which matches semantic URLs to the text of the corresponding link. `$page` is set to `$urls[$page]` (which might be an unset value), and then checked against each key of `$pages`. If one matches, then the link in the navbar gets the extra HTML '`class="curr"`', which allows the "current" page to be styled differently. The resulting HTML (which is in the form of a `<ul></ul>` list) is then returned.

A.3.2 Loc

These functions and class are defined in `manager/lib/loc.php`. Please see section 5.4.2 for more information about Loc.

`PathElement($text, $span_special=false, $link=false)` This class' constructor takes some text, extra HTML code that goes inside the `<span...>` tag, and a link. All but the `$text` is optional. The constructor creates two member variables, `text`, which simply contains the `$text` passed to the constructor, and `html`, which contains an HTML snippet of the form `<span $span_special><a href="$link">$text</a></span>`, where `$span_special` is set to the empty string if it was passed as `false` to the constructor.

`make_path($ending)` Takes either an array of `PathElements` or a single `PathElement`. Returns `array($root, $separator, $ending_interspersed_with_separator)`, with

- `$root = PathElement("Worldview Manager", false, "/worldview/home")`
- `$separator = PathElement("&raquo;", 'class="separator"', false)`
- `$ending_interspersed_with_separator` is `$ending` with `$separator` between every two elements (or simply `$ending` if there was only one element).

`make_title($path)` Takes the array returned by `make_path()` and returns a string made up of concatenated `text` member variables of the `PathElements` in the array.

`make_loc($path)` Takes the array returned by `make_path()` and returns a string made up of concatenated `html` member variables of the `PathElements` in the array.

A.3.3 Templates

This class is defined in `manager/lib/templates.php`. Please see section 5.4.1 for more information about templates.

Page($template, $tags) This class' constructor takes a template file (relative to manager/templates/) and an associative array with the template's variable names and the corresponding values. Once an instance of the class is initialized, the contents of the HTML generated is in the output member variable.

These functions are defined in manager/lib/util.php.

header_template($title, $loc, $page=false) Constructs a Page class (see above) with the header template (as defined inmanager/templates/generic/header.html) as the template file, and with the title variable set to $title, the navbar variable set to navbar($navbar) (see section A.3.1 above for more on the navbar) and the loc variable set to $loc (see section A.3.2 above for more on Loc). The output member variable of the resulting object is returned.

footer_template() Constructs a Page class (see above) with the footer template (as defined in manager/templates/generic/footer.html) as the template file. There are no variables in this template. The output member variable of the resulting object is returned.

A.3.4 Comments

These functions are defined in manager/lib/util.php. Please see section 6.2.8 for more information about comments.

make_comment_area($page, $id, $name, $url_extra="") Prints to the comment form template as defined in manager/templates/comment/comment.html. $page must be one of "topic", "statement", or "tension". The function obtains the comments posted on the relevant $page by passing intval($id) to the corresponding function from the database querying library (see section A.1) – either get_comments_on_topic(), get_comments_on_statement(), or get_comments_on_tension(). For each comment, it runs encode_comment() on the comment, and creates a string $encoded_comments, which consists of the encoded comments concatenated onto each other. The amount of comments is extracted and formatted in an English sentence, and all the variables are passed onto the template.

encode_comment($comment, $num=0) Prints to the comment template as defined in manager/templates/comment/encoded_comment.html. All data is taken from $comment and $num.

A.3.5 Cookies

These functions are defined in manager/lib/usercookie.php. Please see section 6.1 for more information about sessions and cookies.

`delete_cookie()` Sets each cookie (as described in section 6.1) to the empty string and sets their expiration date to `"1"`, causing them to be deleted.

`set_topic_id_cookie($topic_id)` Sets `worldview_topic_id` (as described in section 6.1) to `$topic_id` with an expiration date of a day from the time the function is run.

A.3.6 Tension explanations

This function is defined in `manager/lib/explanation_parser.php`. Please see section 6.2.5 for more information about tension explanations.

`handle_explanation($topic, $explanation)` Accepts a topic ID and an explanation, which must be valid XML (though in practice, it's just HTML with an extra tag) with the following important caveats:

- There is no `<?xml>` tag, `<!DOCTYPE>`, or root tag. The root tag `<explanation></explanation>` is appended by the function.
- There is no limit to character entities. Anything matching the regular expression `/&([a-z0-9]+);/i` (which is the form of character entities) is replaced by `<entity>$1</entity>`, where `$1` is the first match of the regex (that is, the data between the amperand and the semicolon). On printing, the function converts these back to character element format and hands them off to the browser to parse.

The function generates a "tension explanation" from the data in the database. Specifically, its main job is to expand and format the `<stext>` tags that are generated by `WVMPreprocessor`.

B Changing the Maximum Tension Arity

The process of changing the maximum tension arity (that is, the maximum number of statements that can be involved in a tension) is, unfortunately, not trivial. Worldview Manager is composed of many different components, all of which have their own quirks and requirements. It is the intention of this appendix to explain the process in detail. For best results, the reader is encouraged to follow the instructions in this appendix in the order in which they are presented.

B.1 MySQL

The relational database model is not well-designed to variable amounts of data. Therefore, the first step of adding extra tension arity is to alter the MySQL `tensions` table by adding two columns for each extra level of arity desired. For instance, to add a fifth level of arity, create a column called `statement_5` of type "`INT`" and a column called `agreement_5` of type "`tinyint`". Do *not* set either to be "`NOT NULL`", since most tensions will not have that high a level of arity.

B.2 Configuration file

The file `conf/max_arity` is meant to contain a single numerical value, which is the maximum arity. Much of the PHP website (see below) reads from this file. The first line of the file should contain *just* the number of the arity. Do not add extra whitespace, etc. characters at the end. All subsequent lines of the file are ignored. All programs that are unable to properly read the arity from the file default to a value of `4`.

B.3 PHP

The bulk of Worldview Manager is written in PHP. There are two relevant parts – the framework and the website.

B.3.1 Framework

The framework is written without any arity hard-coded, relying entirely on the `$MAX_ARITY` global variable (see section 5.1). All relevant functions, particularly those in the database passing library (see sections 5.2 and A.1) return all the data from the database up to the relevant arity. For this reason, it is incredibly important that they are given the correct arity (which is generally done by editing the configuration file – see section B.2) after editing the database.

B.3.2 Website

Ostensibly, the website does not have any code that assumes any particular arity. Care has been taken to ensure that all values taken from the framework functions are treated as if they were of a dynamic size. The Tension view, in particular (see section 6.2.4), is generated completely dynamically.

That said, since the website is the portion of the code that is most likely to be changed, if any new code is written, the reader is encouraged to pay careful attention to assumptions made about data returned by the framework – both in size and sometimes in contents. Please especially read the API (appendix A) and take effort so that functions can operate on a dynamically-sized amount of data.

B.4 Compiler suite

The compiler suite, like the PHP framework, attempts to act dynamically based on the configuration file `conf/max_arity`.

B.4.1 WVMPreprocessor

The `WVMPreprocessor` program reads the configuration file `conf/max_arity` and generates all tensions based from that, with a default value of `4` if reading from the file fails. However, since one of its jobs is to make inferences about tensions, it also has a "local max arity", which is equal to the smaller of the value in the configuration file and the maximum arity of any tension in a topic file that it is given. The program will limit all of its inferences to local max arity.

B.4.2 mktopic

The `mktopic` program is, like the PHP framework, designed to act dynamically on the data it is given. However, its job is somewhat more difficult because it is uploading to MySQL, which only accepts a static amount of data. Thus, there is only one part of the program where the maximum arity is explicitly defined. The program first attempts to obtain this value via the configuration file (see section B.2). If it is unable to extract a value from the file, it resorts to a default of `4`. The value is stored in the variable `*DATABASE-TENSION-LIMIT*` as defined in the beginning of `topic_file_parsers/mktopic/database.scm`.