DylanWorks

Project and Library Management

Scott McKay & Roger Jarrett

1. Introduction

2. Basic Functionality

Roughly speaking, a *project* is a collection of files and libraries, upon which various operations can be performed.

More precisely, a project consists of a set of typed *modules*. Each module has some contents; for example, a Dylan module contains a bunch of Dylan files, a library module points to a library, and a project module points to another project.

Modules are ordered by a set of *dependencies*.

To perform an operation on a project causes the operation to be recursively applied to each module in the project. So loading a project consists of loading each module in the project. Loading a Dylan module causes each file in the module to be loaded. Loading a project module causes each project in the module to be loaded.

3. Projects

3.1 Defining a Project

Contents of a project (sources, documentation, etc.).

Dependencies between the “modules” of a project.

Other attributes - compiler attributes, debugging vs. development, etc.

3.2 Operations on Projects

Compilation

Linking

Loading - downloading across access path, loading derived info into development env.

4. Journalling and Reproducibility

5. Patching

6. Top-Level Controller

This paper describes the tool that controls the overall aspects of a DylanWorks session. The top-level controller provides some simple entry points into the other tools (perhaps via a menu bar), has a “control panel” that allows a user to set various global parameters and preferences, and allows the selection of an “access path” (which is the object that hooks the development environment to a component or application being worked on).

6.1 Principles

The top level controller is the first thing (and perhaps only) thing that a user sees when they boot up DylanWorks. It must therefore function as an “introduction” to new users as well, as a convenient way for an experienced used to begin navigating to the functionality they’re after. User Interface and functionality are both crucial. Here are some goals to strive for:

- Completeness. The user must be able to get anywhere, and at least start to invoke any functionality that DylanWorks has.
- Ease of use. This is the gateway to the rest of the system, including the documentation. This means the top-level controller needs to be usable *without* any documentation.
- Uncluttered. We don’t want to scare the user with too many options at the top level.

6.2 Methodology Assumptions

This section should concur with the Methodology document.

The programmer will be running the development environment and one or more components or applications that he will be inspecting/debugging. Each such component will be running in its own address space. Each of these address spaces may have a number of libraries loaded into it. Although there may be a number of components or applications loaded into virtual memory, the programmer will only be inspecting/debugging one at a time. The connection to one component is via a *debug access path*. The programmer should be able to easily switch between which component is being worked on, which will be accomplished by selecting a new access path to use.

6.3 Basic Functionality

The top level controller should permit access to the following functionality:

- On-line tutorial for both Dylan and the DylanWorks environment.
- On-line Reference manual for both Dylan and the DylanWorks-supplied libraries.
- Access to the project browser, which has a list of known projects. Any of these projects can be loaded up so that the user can get going immediately. This includes being able to get a list of known libraries and load them into any given running component or application's address space.
- Creation of a new project.
- Maintenance of the access paths to the running components or applications. This includes seeing which components are running and be able to pick which one the current access path should point to; getting information about the project connected to any access path; interrupt the process at the remote end of an access path and debug it; terminate it; and restart it.
- Launch the browsers. This might provide a list of browsers, or just one, top-level "meta" browser. This should include a directory/file browser.
- Virtual address space browser: See how much swap space there is on disk, how much is left, what components are in virtual memory. [Maybe there's an OS utility that will do this for us.]
- See and set user options/preferences. These could be for all tools, just the loaded tools, or just the top-level preferences, with each tool having its own user options. [We need a Dylanworks-wide policy here.]
- Quit DylanWorks.

6.4 User Options

Perhaps this should be in a separate document.

Individual users will have some latitude to configure their environment. Deciding what should be an option versus simply left out will be a balancing act between keeping the number of options low and unconfusing versus providing desirable flexibility.

There is a tendency in some systems to give the user a lot of initialization options. While there should perhaps be a number of these accessible to the experienced user, we should recognize that providing too many options represents a failure to come up with a design that pleases most users in most situations. It also makes it hard for one user to use a machine configured for another user. Lots of user initializations frequently cause problems when the user upgrades to a new version of the software that are customized by the initializations. Another problem with initialization options is that the user may want one option in some situations but another in others that they didn't anticipate when they last modified their profile.

Out of the box, the options will default to the “factory settings” which will favor the new user. We should also have a few other sets of settings, say for intermediate and/or expert users or perhaps users with different programming backgrounds (for example, C versus Smalltalk).

We want the capability to change options for a given tool while you’re using the tool, as well as change the options for a given tool starting out from the top level controller. Since we want to minimize the code and inconsistency mistakes that this dual-use design demands, we will need to specify how a programmer can declare the available options in just one way and place so that it can be used both from within the tool and from the top-level controller.

6.5 User Options Strategy

We plan to provide a substrate that allows each user to declare its options. This substrate will be able to create a dialog that allows the user to edit the user options for each tool. The user will be able to access this dialog either from the top-level controller or from within the tool. Each item in the dialog will correspond to an option. The option will be described in a high-level way. The actual gadget chosen to represent the option and the layout of gadgets within the dialog will be generated automatically by the DylanWorks UI substrate. We may want to provide a facility for overriding in special situations.

One implementation strategy might be:

- There is a **<options-collection>** class. Each tool/library will create an instance of this class and fill it with its options.
- There is an **<option>** class. Instances of it will go into an instance of **<options-collection>**. The **<option>** class will have slots such as:
  - its name
  - its type
  - a description of its possible values
  - default values, one for each major setting (for instance, novice user or expert user)
  - a documentation string
  - which option collection it belongs to