Timeliner Integrated Development Environment

by

Maya Dobuzhskaya

S.B., Massachusetts Institute of Technology (2004)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2005

© Maya Dobuzhskaya, MMV. All rights reserved.

The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis document
in whole or in part.

Author
Department of Electrical Engineering and Computer Science

May 19, 2005

Certified by
Dr. Robert A. Brown
Timeliner Task Leader, Charles Stark Draper Laboratory
Thesis Supervisor

Certified by
Prof. Robert C. Miller
Assistant Professor, MIT
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Students
Abstract

The Timeliner system is used for controlling experiments aboard the International Space Station. Timeliner scripts are typically written in a generic text editor and turned into executable byte code by a command-line compiler. Script writers have no assistance during the development process, resulting in error-prone scripts and a prolonged development cycle. The Timeliner Integrated Development Environment (TIDE) is an environment built on top of the Eclipse Platform that assists developers with their tasks, and allows for any person to easily and quickly develop a Timeliner script. TIDE incorporates existing Timeliner stand-alone tools and contributes an editor with custom language definitions, a database browser, incremental compilation, and integrated help.

Thesis Supervisor: Dr. Robert A. Brown
Title: Timeliner Task Leader, Charles Stark Draper Laboratory

Thesis Supervisor: Prof. Robert C. Miller
Title: Assistant Professor, MIT
Acknowledgments

First, I would like to thank Robert Brown and Joseph Bondi, whom I had the pleasure of working with at Draper Laboratory. Without them this thesis would not have been possible. Thank you for your fantastic support, dedication, and advice.

I would like to thank my phenomenal research advisor, Robert Miller. He always showed great interest, had time to spend with me, and gave invaluable support and guidance. Thank you for giving me the opportunity to brag that I have the best advisor at MIT.

I would like to thank Nidhi Sharma, Philip Rha, Michael Bolin, Victor Costan, Vineet Sinha, Min Wu, and other members of the User Interface Design group at MIT's Computer Science and Artificial Intelligence Laboratory. Thank you for your encouragement and valuable comments.

I would like to thank all the pilot users whose input was helpful in the design of the user interface.

Lastly, I would like to thank my mother. Mom, thank you for always encouraging me to pursue education and being the number one supporter in all my undertakings.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under contract NAS-9-01069, sponsored by the NASA Manned Spaceflight Center, Houston, Texas.

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

(Author's signature)
Contents

1 Introduction 15
   1.1 Timeliner System ........................................... 15
   1.2 Target System Databases ..................................... 16
   1.3 Timeliner Language ......................................... 17
   1.4 Motivation .................................................. 18
   1.5 Users ....................................................... 19
      1.5.1 Timeliner Language Experts .............................. 19
      1.5.2 Timeliner Language Novices ............................. 20
   1.6 Tasks ...................................................... 20
      1.6.1 Create New File ......................................... 20
      1.6.2 Set Up Database ......................................... 20
      1.6.3 Compose Script ........................................... 21
      1.6.4 Lookup Database Variables ............................... 21
      1.6.5 Compile Script .......................................... 21
   1.7 Goals ...................................................... 22
   1.8 TIDE ....................................................... 22

2 Related Work 23
   2.1 Other Real-time Control Languages ......................... 23
   2.2 Language-based Text Editors ................................ 24
   2.3 Development Environments ................................... 25
      2.3.1 History .................................................. 25
      2.3.2 Integrated Development Environments .................. 26
3 Design

3.1 Overview ........................................ 29
3.2 User Interface .................................... 30
3.3 Resource Creation ................................. 32
  3.3.1 New Tide Project Wizard ..................... 32
  3.3.2 New TLS File Wizard ......................... 34
3.4 Editor ........................................... 36
3.5 Database Interface ................................. 38
  3.5.1 Import and Build .............................. 38
  3.5.2 Browse .................................... 39
  3.5.3 Insert Into Script ............................ 41
  3.5.4 Search .................................... 42
3.6 Builder ........................................... 43
  3.6.1 Preferences ................................ 44
  3.6.2 Displaying Errors ........................... 45
3.7 Tool Integration ................................. 46
  3.7.1 Database Importer ........................... 47
  3.7.2 Compiler ................................... 47
  3.7.3 TLX Viewer ................................ 47
3.8 Integrated Help ................................. 48
  3.8.1 Keyword Reference ........................... 48
  3.8.2 Environment Help ........................... 49

4 Implementation ................................. 51

4.1 Choosing a Foundation ........................... 51
4.2 Eclipse Extensions .............................. 52
  4.2.1 Resource Creation ........................... 53
  4.2.2 Editor .................................... 53
  4.2.3 Database Interface .......................... 54
List of Figures

1-1 Database conversion to an internal format .............................................. 16
1-2 Timeliner script monitoring altitude, Event_301, and temperature [9]. ........ 18
2-1 Eclipse plug-in based architecture [16]. .............................................. 27
3-1 TIDE user interface ................. ......................................................... 31
3-2 TIDE user interface customized by a user. .......................................... 33
3-3 First page of Tide Project wizard ...................................................... 34
3-4 Second page of Tide Project wizard .................................................... 34
3-5 TLS File wizard. .................................................................................... 35
3-6 Timeliner editor. .................................................................................... 36
3-7 External database import dialog ............................................................ 38
3-8 Current database user interface ............................................................. 40
3-9 Previous iteration of database user interface ........................................ 40
3-10 Telemetry view showing the properties of a telemetry attribute .............. 41
3-11 Preview of a command insertion in the editor ....................................... 42
3-12 Database Search view. ........................................................................... 43
3-13 Animated progress bar. ........................................................................... 44
3-14 Compiler preference page. ..................................................................... 44
3-15 Compile errors displayed in the editor and Problems view. .................. 45
3-16 TLX file properties seen in an editor. ................................................... 48
3-17 Programmer’s reference of Timeliner keywords with tooltips ............... 49
3-18 Reference view pop-up menu. ............................................................... 50
3-19 TIDE user guide. ................................................................................... 50
4-1  Eclipse extension points that TIDE implements. .................. 53
4-2  TIDE editor, utilities, preferences, and resources classes. ....... 55
4-3  TIDE’s action set classes. ........................................ 57
4-4  TIDE’s view classes. ............................................... 58
4-5  Correctly indented Timeliner script. ............................ 60
4-6  Target system database formats and usage. ..................... 64

5-1  Code that users were asked to write during the user study. ....... 72
5-2  Pilot user interacting with a paper prototype of TIDE. ............ 72
List of Tables

4.1 Categorized Timeliner keywords. .................................. 60
4.2 Ada and Java compiler benchmarks. ............................. 67
Chapter 1

Introduction

This chapter begins by describing the Timeliner system, target system databases, and Timeliner language. It provides motivation for an integrated development environment, and outlines the classes of Timeliner script developers and their tasks. These user distinctions and task characteristics are referenced throughout this paper, as they are a basis for TIDE's user-centered design.

1.1 Timeliner System

The Timeliner system was developed in 1982 by The Charles Stark Draper Laboratory, Inc. to automate procedural tasks. The first release was used to emulate the timelines of on-board crew procedures on the Space Shuttle. The Timeliner system is both a computer language and an execution environment. The language is used to write scripts that provide sequencing and process control. The execution environment provides real-time monitoring and control [22].

Scripts can be written for space applications such as autonomous or interactive vehicle control, performance of preflight subsystem checkouts, handling failure detection and recovery, or stepping through pre-defined procedures of a scientific experiment. Other applications include process control for manufacturing, materials processing, robotics, or any automated procedures that need to be executed reliably. Currently, Timeliner is used on the International Space Station’s real-time core command and
control, and payload control computers [22].

1.2 Target System Databases

Timeliner code interacts with data objects on target systems. Customers of Timeliner, such as NASA and Boeing, have their own definitions of command and telemetry data objects. These definitions compose the external database format in Figure 1-1. The external databases are large, so if the Timeliner compiler were to process scripts with data objects from this database, it would take a long time to look up each data object.

![Diagram](image)

Figure 1-1: Database conversion to an internal format.

An import process converts the external database into an intermediate database format, and can filter data objects to extract a smaller subset. The intermediate database format consists of ASCII text files of commands and telemetry data that can be modified by hand. A telemetry data file contains instance addresses on the target system, called the “real environment.” Timeliner scripts can also be run on desktop platforms, or “test environments.” In this case, the addresses in the telemetry data file are converted to local addresses.

These intermediate files are built to produce a Timeliner internal database. This internal format consists of binary files that the compiler uses, and listing files that are used by the test execution environment and TIDE’s database parser.
1.3 Timeliner Language

The Timeliner language is a procedural language that provides sequencing and command functions. It was designed to be similar to English so that scripts can be composed without in-depth knowledge of Timeliner syntax. The language is case-insensitive, so upper-case characters, lower-case characters, and a mixture of both are all compiled to the same executable file.

Scripts are built from bundles, sequences, subsequences, and statements. Each script contains exactly one bundle and one or more sequences (see Figure 1-2). A script must begin with a BUNDLE statement and end with a CLOSE statement. Sequences are independent threads of execution that contain instructions for monitoring, controlling, and reporting on the operations of a target system. The sequence on lines 11-16 in Figure 1-2 checks temperature, and turns the heater off when the temperature is above 80 degrees. Like all sequences, it starts with a SEQUENCE statement on line 11 and ends with a CLOSE statement on line 16. The keyword SEQUENCE can be replaced by SEQ, as some Timeliner keywords have abbreviated spellings. Sequences contain statements that execute serially except as modified by the operation of conditional constructs such as WHEN, WHenever, EVERY, and IF.

Statements in a sequence are composed of keywords and components. The first word in a statement is a Timeliner keyword. The keyword is followed by components, where most components are of type boolean, numeric or character. A component may be a literal, a constant, a combination, a list, a definition, a built-in constant, a built-in function, an internal variable, or an application defined reference to an external variable. The statement on line 18 in Figure 1-2 contains the keyword COMMAND, and the components TURN_OFF and LAB.HEATER. Some statements, such as those starting a conditional block, must be followed by a closing statement at the conclusion of the block. In Figure 1-2, the WHenever statement on line 13 is followed by an END WHenever statement on line 15.

The Timeliner language allows blank lines and comments. Comments are preceded by two or more adjacent hyphens (--) Everything on a line after the hyphens is
1.4 Motivation

Timeliner scripts are written in a generic text editor and turned into executable byte code by a command-line compiler. A script is written in a text editor such as Notepad and is provided to the Timeliner compiler as an ASCII text file. The compiler checks language syntax and creates an executable file [22].

Script developers have no assistance during the development process, which can result in error-prone scripts and a prolonged development cycle. More specifically, the development process has three major drawbacks. First, a generic text editor provides
no language specific assistance such as keyword highlighting or auto-indentation. This lets syntactic mistakes go unnoticed [11], and slows down novice script developers, since they have to look up or memorize Timeliner keywords. Second, a script writer must know what commands and telemetry data are available. These variables are specific to a target system database, and since a database can contain thousands of variables, it can be tedious searching through files to find what is needed. Lastly, the usability of the Timeliner system could be greatly enhanced if existing tools were integrated into one coherent environment. For example, script creation would be more efficient if the compiler could be invoked from the editor, and errors could be displayed right in the editor [11].

1.5 Users

People who write Timeliner scripts can be classified as Timeliner language experts or novices. This distinction refers to their familiarity with the language.

1.5.1 Timeliner Language Experts

Timeliner scripts are primarily developed by people at NASA’s Johnson Space Center. These developers are very familiar with the Timeliner language and have between average and expert computer expertise. They use text editors such as Notepad and Microsoft Word, and may use development environments such as Microsoft Visual Studio to develop code. These users typically work alone in an office, writing lengthy and elaborate scripts which contain many commands and telemetry data. They might edit multiple scripts simultaneously and work on scripts in multiple sittings. They are focused on getting their work done efficiently, and want fast access to the database variables. These users can type fast and would benefit from shortcuts.
1.5.2 Timeliner Language Novices

A second group of users that write Timeliner scripts are scientists who want to run an experiment on the space station. They are unfamiliar with the Timeliner language, and have between beginner and average computer expertise. Most probably use applications for word processing like Microsoft Word, but do not utilize software development applications. These users typically compose a few short scripts in a single sitting. They are not under time pressure, and want a way to easily look up Timeliner keywords and database variables. These users don’t type fast, and would browse menus for navigation.

1.6 Tasks

Both user classes have the same objective, to develop Timeliner scripts. This objective can be broken down into subtasks of creating a new file and internal database, composing a script, using database variables, and compiling.

1.6.1 Create New File

Users must begin by creating a new script file. This task is performed frequently by experts and rarely by novices. Since the user population is familiar with other software such as Microsoft Word, they will likely look in the File menu to accomplish this task.

1.6.2 Set Up Database

The next step is to set up and internal database of variables that can be used in the script. This task is performed sparsely by experts and is most likely performed once by novices. Experts might build a new internal database every few weeks, and novices have a single internal database that is set up for them in advance.
1.6.3 Compose Script

Having created a new file and internal database, users can start composing a script, which is the main task. Experts know what they want to write and want to accomplish that in minimal time. Novices might not know valid Timeliner language constructs and need to be provided with a detailed help section. Examples and templates would assist novices with this task. Both user groups can benefit from assistance in the editor, such as auto-completion that allows experts to type less and reminds novices of valid keyword choices.

1.6.4 Lookup Database Variables

During script composition, users likely need to include commands and telemetry data from the target system database. They might want to browse the database variables, or they might have something specific in mind that they want to use. Experts have a better idea of what database variables they need, but often do not remember command parameters, or attributes of an instance. Novices need to browse the database variables and see what is available. Both groups perform this task often. Templates would help users input these variables into a script.

1.6.5 Compile Script

Users compile scripts to locate errors and to produce an executable file. Experts often compile many scripts. They would benefit from a compiler than can be launched quickly and can process multiple files simultaneously. Novices might only do this once and need to easily find and launch the compiler. Both of types of users would benefit from catching errors earlier in the development process, so compiler errors need to be clearly visible during script composition.
1.7 Goals

The Timeliner Integrated Development Environment (TIDE) address all of the afore-
mentioned drawbacks with the script development process, and allow for any person
to easily and quickly develop a Timeliner script. For TIDE to be useful, it should have the following features:

- **Usability.** The environment and tools should be easy to learn and use. A consistent user interface should be presented.

- **Extensibility.** The environment should easily accommodate additional tools and changes in the Timeliner language.

1.8 TIDE

TIDE is built on top of the Eclipse Platform and contributes editing functionality, a database interface, incremental compilation, and integrated help. TIDE’s editor is equipped with custom language definitions that assist script developers by providing features such as syntax highlighting and code formatting. The Timeliner compiler is automatically invoked when scripts are saved, and errors are displayed inside the editor, allowing mistakes to be caught sooner in the development cycle. Variables from target system databases are parsed and displayed in TIDE, where their properties can be examined, and they can be easily inserted into scripts. Language specific assistance is available inside the environment. TIDE also seamlessly integrates stand-alone Timeliner tools. The database importer, compiler, and binary file property extractor are launched from, and results are viewed in, TIDE.

The rest of this paper is structured as follows: Chapter 2 describes related work in terms of languages similar to Timeliner, brings up text editing issues, and introduces development environments. TIDE’s user interface design is presented in Chapter 3. Chapter 4 relates how TIDE extends Eclipse and discusses implementation decisions that were made. Chapter 5 evaluates TIDE’s usability through a series of user studies. Chapter 6 suggests future work, and Chapter 7 summarizes TIDE’s contributions.
Chapter 2

Related Work

This section describes languages similar to Timeliner, reviews text editing advances, and examines development environments. It consummates with an account of the Eclipse Platform [16], a popular integrated development environment.

2.1 Other Real-time Control Languages

Similar to Timeliner, the Spacecraft Command Language (SCL) is used for controlling events. SCL is a collection of components that are integrated through a database. It was originally developed as an embedded software system to operate a U.S. Navy satellite for 180 days without ground intervention [18]. Unlike Timeliner, it is not deterministic, since it can respond to events in a rule-based, as well as a time-based manner. The main difference is that SCL is an independently running platform, while Timeliner is meant to plug into a platform.

Also comparable to Timeliner, Supervisory Control and Data Acquisition (SCADA) systems have been in use since the 1960s for gathering and analyzing real-time data. They can remotely monitor and control a plant or equipment in industries such as telecommunications, water and waste control, energy, oil and gas refining, and transportation. Unlike Timeliner, SCADA systems cover large geographical areas where communication is unreliable and open-loop control must be used [19].
2.2 Language-based Text Editors

Since computer languages are highly structured and have clearly defined syntax, editors should not treat them as plain text [11]. Language-based editors exploit their knowledge of the syntax and semantics of a given language to assist users in forming grammatical expressions [4]. Such editors provide better usability since they give users immediate feedback through continuous assistance such as syntax highlighting and auto-indentation [11]. In the 1980s when these language-specific editors were emerging, a question arose about how much discipline to impose on program construction [4].

The choice was between a syntactic approach or a lexical approach. A syntactic approach uses formal grammar and parses text into a tree of well-formed constructs. While this makes code more expressive, it limits the user's freedom to pass through inconsistent intermediate states, since the approach forbids ungrammatical code. A lexical approach is less formal, and treats text as a sequence of flat segments. While this approach gives users complete freedom, it doesn't take advantage of language constructs to guarantee syntactic consistency [15]. Since users switch between perceptions of symbol sequences and character sequences with little conscious effort, a compromise between the two approaches is the best option [4]. The Cornell Synthesizer Generator [21] is a hybrid between the two approaches. It parses text incrementally, allowing users to freely edit code which it subsequently parses into tree nodes [15].

Another language-based customization in editors is content assist. When users type characters, the editor presents possible completions for the word based on known keywords. This technique dates back as far as 1980, when it was used in a study of how users interact with text editors [6]. Today, content assist is used in development environments to make typing more efficient.
2.3 Development Environments

This section outlines the history of development environments, depicts the advantages and problems of modern Integrated Development Environments (IDEs), and explains how the Eclipse Platform overcomes these shortcomings.

2.3.1 History

Software engineers use environments with tools such as editors and compilers to develop code. Earlier environments, such as Unix [10], loosely interconnected the inputs and outputs of tools via redirection. However, these early environments didn't provide for a way to integrate tools, coordinate their activities, or automate common tasks. The earliest vehicles for automating the flow of control among tools provided for a way to describe how tools and data relate, and which tools should be used to process changed data. These vehicles, such as Make [7], enhanced the usability of stand-alone tools, but the resulting environments still only provided loose tool integration [8].

The first significant efforts in producing tightly integrated development environments were Programming Support Environments (PSEs), collections of tools that support coding activities. Earlier PSEs, such as the Cornell Synthesizer Generator, typically provided language-specific editors, compilers, and debuggers. These tools were tightly integrated so that the activities of one tool were reflected appropriately in the other tools. PSEs were extremely useful applications, but were limited because they could only support one software engineering activity and its artifact, or implementation and code [8].

Software Engineering Environments (SEEs) emerged from the need for integrated support of software engineering activities throughout the software life-cycle [8]. For example, Arcadia SEE [1] included tools for requirements specifications, analysis, and testing. From SEEs, two important lines of research emerged: Multi-view Software Environments, and Process-centered Software Engineering Environments (PSEEs). The former allowed multiple views of a given piece of software, where a modification to one view would cause the other views to be automatically updated. The later com-
bined tool support for software artifact development with support for the modeling
and execution of the software engineering process that produced those artifacts [8].

2.3.2 Integrated Development Environments

Modern Integrated Development Environments (IDEs) stemmed from PSEEs, and
include the explicit representation of processes, their products, and their interactions [8]. Early IDEs were highly specific to some content. However, popular IDEs in
use today such as Emacs [20], Microsoft Visual Studio [14], and the Eclipse Platform
allow for adaptation and integration. For instance, they provide support for several
different programming languages.

The two biggest criticisms of current IDEs are that they are too large and too
complicated to use. Since each IDE tries to bundle together all the possible tools a user
might need, some critics claim IDEs are swelling to large proportions [2]. A solution is
to modularize IDEs and have them provide support for independent, integrable plug-
and-play tools which can be reused within multiple IDE frameworks [2, 8]. Other
critics allege that since IDEs encompass many tools, they might be daunting for
novices. However, if the user interface is well designed, IDEs can be simple to use for
both experts and novices.

2.3.3 Eclipse: A Modular and Extensible IDE

The Eclipse Platform, an open-source workbench for the integration of software de-
velopment tools, elegantly meets the aforementioned modularity goal that other IDEs
lack. Eclipse is entirely built from modules, called plug-ins, allowing for the seam-
less integration of independently developed components. Plug-ins range from ones
essential to Eclipse such as Workbench, JFace, SWT, and Workspace, to more spe-
cialized and smaller ones such as the JUnit testing plug-in (see Figure 2-1). Each
plug-in consists of source code, resources such as images, and an XML [3] manifest
file. On startup, Eclipse reads though all the plug-in manifest files and hooks up the
appropriate modules into the system.
Eclipse facilitates the construction and integration of new plug-ins through well-defined Application Programming Interfaces (APIs) and a Plug-in Development module that provides useful building blocks and frameworks [16]. Currently, web directories list over 800 existing Eclipse plug-ins in varying stages of completion being developed by companies such as IBM, Intel, and Macromedia [17].

Eclipse's modular and extensible nature can be attributed to four features: explicit extension points, a multi-level architecture, self-description, and component encapsulation. First, Eclipse has distinct ports to which other plug-ins can connect as long as they adhere to a given interface. These explicit extension points make it easy to see where and how components can be plugged into the platform. Second, Eclipse plug-ins can declare their own extension points, allowing the tool architecture to have multiple levels. Third, plug-ins provide manifest files to allow the platform to be aware of the essential properties of a plug-in before it is loaded. Lastly, Eclipse effectively encapsulates plug-ins. A plug-in can access another plug-in only if it is declared "required" in the manifest. Each plug-in has a class loader, and runs in a container of its own that regulates communication with other components to avoid unintended dependencies between plug-ins [13].
Chapter 3

Design

This chapter discusses the final design of TIDE and tradeoffs that were made. An overview is given, followed by descriptions of the user interface and its main components.

3.1 Overview

TIDE provides a coherent environment for efficient Timeliner script development through an intuitive and easily navigable user interface (UI). TIDE is built on top of the Eclipse Platform, and its interface is modeled after the Eclipse's Java Development Tools (JDT) plug-ins. Building on top of a widely-used IDE gives the benefits of development reuse and external consistency. For instance, Eclipse's code for resource management is reused. Eclipse's UI features have been accepted by millions of users. An example of an accepted feature is an asterisk displayed in front of the name of a resource in the editor window with unsaved changes. By replicating the look and feel of an widely-used IDE, development time has been saved, as the user interface did not have to be rigorously tested and debugged. Furthermore, users who have worked with Eclipse will be familiar with TIDE's UI.

TIDE is designed to assist both Timeliner language experts and novices throughout the development process. Adhering to the requirements of both user groups has been a balancing process. Experts want a variety of advanced features that can be
quickly accessed. Novices crave a simple interface that is intuitive and easy to use, with only basic features initially presented and others hidden away by menus. This tradeoff in TIDE’s design is addressed in this chapter.

3.2 User Interface

TIDE’s user interface consists of an editor in the center, surrounded by a menubar, toolbar, views, and statusbar (see Figure 3-1). Like other software applications, TIDE has a menubar at the top of the screen presenting the user with categorized options. Directly below is a toolbar with icons for frequently invoked features such as file saving and compiling. In the center of the UI is the editor window where scripts are composed. The views surrounding the editor serve to display the state of the opened script file and provide assistance with script composition. On the bottom is the status bar which displays information, such as the cursor position in the editor, and reports on the progress of background running tasks such as compilation.

In the default layout of the views (called a perspective) there are five views surrounding the editor. As other IDEs, TIDE initially presents a large portion of its features to users, and allows them to manipulate the display of these features. While TIDE’s default perspective seems to favor the needs of experts who require fast access to information, it is also the best choice for novices, as most of the views are there to assist them in completing their tasks. For instance, the Reference view in the lower left is for browsing Timeliner language keywords. User studies have proven that users are comfortable with TIDE’s perspective. The UI’s success can be attributed to its aesthetic look, consistency, efficiency, and customizability.

The UI was designed to be aesthetic by keeping clear separation between views and using minimal colors. The two panels of views to the right and left of the editor and equal in width, providing a balance. Similarly, the two views on the left are equal in height. Text size is kept small in the views so the UI is not cluttered despite all the information that is presented. Minimal colors are used in the UI; mostly black, purple, and red. Text in the views and normal text in the editor is black for good
readability on a white background. Timeliner keywords in the editor are in bold font and purple color, to indicate that they are different from normal text. Red is reserved for error indication in the editor and Problems view.

TIDE’s UI is externally and internally consistent. The menubar and toolbar are similar to that of applications in the Windows platform, so most users will be familiar with them. The views surrounding the editor are organized in a tree fashion where branches can be expanded and collapsed. This is consistent with other software applications such as the Microsoft Windows Explorer. TIDE is also consistent within itself. Interactions with views such as the Navigator, Reference, and Database views is done the same way, and the same colors are used throughout.

TIDE helps users increase their productivity with its view layout, shortcuts, and immediate response to user interaction. Since all the views are equidistant from the editor, they can be quickly reached with the mouse. Actions that users often invoke,
such as file saving and compiling, can be accomplished by clicking the appropriate button on the toolbar or typing a keyboard shortcut. The interface also immediately responds to interaction events, such as keyboard events when a user types or mouse events during selection. These events have precedence over background running threads such as the compiler. Consequently, experts who type fast can be efficient since they don’t have to navigate through menus to launch tools, while novices are efficient because assistance is always close by in a neighboring view.

TIDE’s perspective permits fine-grained customizability. The views surrounding the editor can be rearranged and stretched. Multiple editors can be open, and they can be either horizontally or vertically tiled. The toolbar and menus can also be changed to show as many or as little options as the user desires. (Contrast the perspective in Figure 3-2 with the one seen in Figure 3-1.) TIDE presents a default perspective that user studies have proven to be efficient in helping users with script composition, however, each individual can customize TIDE precisely to his or her needs. For instance, experts can close the Reference view, as they are already familiar with Timeliner keywords.

3.3 Resource Creation

TIDE provides two resource creation tools specialized for the Timeliner language: a project wizard and a TLS file wizard. Both of these wizards can be launched from the File menu under the “New” option.

3.3.1 New Tide Project Wizard

The New Tide Project wizard (see Figures 3-3 and 3-4) guides users through the creation of a project, a folder with special properties. Project folders contain script files and internal database files that contain database variables of the target system. The wizard allows users to specify the name and location for a project (Figure 3-3), and optionally import and filter an external database (Figure 3-4). Two pages were used for this wizard to separate the process of creating a folder and importing
an external database. Users have complete control in the wizard; they can proceed forward or backward, and exit the wizard at any time.

The second wizard page (Figure 3-4) has a check-box to import an external database. The default state is unchecked for simplicity purposes, but when it is checked, the input boxes below become active. When a user clicks “Finish” the wizard checks these input boxes for errors. For instance, if only 2 of the 3 necessary fields are specified, the wizard will not exit, and the appropriate error is displayed. This error checking prevents users from making careless mistakes and having to restart the wizard. Values for these input fields that a user enters are saved and filled in next time the wizard is launched, so users do not have to browse for the location of the external database each time. The results of the database import process are output the the Console view. If a user wants to create a project without importing an external database, he or she has the option to click “Finish” on the first page of the
wizard or on the second page without selecting the “Import an external database” check-box.

Projects have a property that indicates if they are associated with a test of real environment, which refers to the domain of memory addresses in the internal database. The test environment has logical memory addresses corresponding to the desktop test platform, and the real environment has physical addresses for a real-time system. When a project is created, by default its environment is set to the test environment variables, since this will be the case for most projects. The environment can be later switched through project properties.

### 3.3.2 New TLS File Wizard

The New TLS File wizard (Figure 3-5) allows users to create new Timeliner script files with default procedure blocks set up. It is called the TLS File wizard because Timeliner script file names have TLS extensions. Users select the project which the new file should be placed in, specify a file name, and can choose to add a BUNDLE and SEQUENCE block. In the “File name” box, users can type a TLS extension or leave it out, in which case it will be automatically inserted by the wizard at file creation time. By default, “Generate BUNDLE block” and “Generate SEQUENCE block” are
checked, as all TLS files contain a bundle and at least one sequence. This automatic insertion feature saves experts the time to type these statements, and provides novices a framework for structuring a new script.

![New TLS File Dialog Box](image)

Figure 3-5: TLS File wizard.

The name of the file is used as the bundle name, and a default sequence name is inserted when the file is created so users can start composing the sequence body right away. Feedback on users' input in the wizard fields is given right in the wizard. If a user forgets to select a project folder or types a file name with an invalid character, the appropriate error will be displayed. Accordingly, the “Finish” button becomes active only when all the fields have valid entries. Users can exit the wizard at any time by clicking “Cancel”. When users click “Finish”, the wizard exits, and a new TLS file is created and opened so that users can start working on it right away.
3.4 Editor

TIDE’s editor comes equipped with Timeliner language definitions and allows users to view and format code according to Timeliner grammar (see Figure 3-6). It is associated with TLS and TLL files, meaning files with those extensions will automatically open with the Timeliner editor instead of the default text editor. The Timeliner editor syntax highlights Timeliner keywords in bold font and purple color. Syntax highlighting gives users instantaneous feedback as they type, so they immediately know if what they just typed is a keyword or if they have misspelled it. Comments are highlighted in green, strings in blue, and errors in red. Consequently, these highlighted items can be easily spotted and identified by visual inspection. The editor will syntax highlight without regard to character case, as the Timeliner language is case insensitive.

![Timeliner editor screenshot](image)

Figure 3-6: Timeliner editor.

Auto completion affords users in-place assistance as they type. When users start typing a word, they can enter CTRL-Space on their keyboard to see a pop-up box with all Timeliner keywords that match what they have started to type. This pop-up box is
filtered as they continue to type, and they can select one of the proposed completions
to insert it into the editor instead of typing out the whole word. In Figure 3-6, a user
has typed the letter “W” and the auto completion pop-up is displaying all Timeliner
keywords which begin with that letter.

Script structure is easier to comprehend and errors are more easily spotted due
to the editor’s formatting capabilities. As users type, lines of code are automatically
indented according to Timeliner grammar. For example, if a user starts a block of
code with a SEQUENCE statement such as that on line 4 in Figure 3-6, the next line
will be indented by a tab when the user presses “return” on the keyboard. When
that user finishes the block with a CLOSE statement, the closing line will be indented
to match the indentation of the SEQUENCE statement line. The editor also provides a
way to format an existing region of code, which is particularly useful for formatting
scripts that have been composed outside of TIDE. This formatter can be invoked
by selecting a region and choosing “Content Format” under the Edit menu. The
algorithms for code formatting are described in detail in the Implementation section
of this paper.

The editor also provides feedback on the state of an opened script. The Outline
view, seen in the top left corner of Figure 3-1, gives users an overview of the script file
by providing a list of its sequences. Users can click on a sequence name to have the
editor window scroll to display that sequence declaration. Another feature is a blue
“T” icon on the editor’s tab. This icon indicates that the tab contains an instance
of the Timeliner editor. If the name on the tab has an asterisk in front of it, that
signifies the file has unsaved changes. These changed regions are indicated by purple
marks in the ruler on the left side of the editor. In Figure 3-6, line 10 has been
changed since the last save operation. Lastly, a blue highlighted line indicates the
line on which the cursor is positioned.
3.5 Database Interface

TIDE has completely integrated the concept of a database into the development environment. Each project is associated with a target system database, and the necessary intermediate and internal database files are stored in the project folder. TIDE lets users build an internal database, browse commands and telemetry data, insert database variables into the editor, and search for a particular variable.

3.5.1 Import and Build

An external database must be imported and filtered to create the intermediate database format. This intermediate database must be built to produce an internal database used by the Timeliner compiler (see Figure 1-1). An external database can be imported when a project is created, or later by choosing “Import ISS stdout database...” from the Run menu (or hitting the corresponding quick-access button on the toolbar). The International Space Station (ISS) external database is currently the only one whose import process is supported in TIDE. The “...” in the menu item indicates that a dialog will be displayed, which follows software conventions for menus. This dialog (seen in Figure 3-7) is similar to the second page of the Tide Project wizard. Just like that wizard, this dialog reports errors, stores the last user-inputted values, and outputs progress to the Console view.

![Import ISS stdout Database dialog](image)

Figure 3-7: External database import dialog.

When an external database is imported, both real and test environment files are
automatically created. The simultaneous creation of both environments is done for performance reasons. While the intermediate database setup process takes a little longer, switching between the test and real environments is very fast, as both already exist. Users can modify the commands and telemetry data ASCII files that are the intermediate database and re-run the build step by selecting “Build internal database” from the Run menu.

3.5.2 Browse

A problem that developers faced when composing scripts was having to dig through the intermediate database’s unformatted text files to find variables. TIDE solves this problem by providing two database views: a view of commands and their parameters, and a view of telemetry data instances and attributes. The current design of the database views (Figure 3-8) can be contrasted with a previous iteration seen in Figure 3-9. The old design combined the search box, commands, and telemetry data all into one view. Since commands and telemetry were in one column, there wasn’t enough vertical space to see many variables at the same time, and users would be forced to scroll through these lists. The new design separates commands and telemetry data into two tabbed views, allowing more entries to appear in the visible region of a view. The second problem with the old design was that since views were bundled together, they could not be moved around or closed individually. The current database UI (as seen in Figure 3-8) decouples the three pieces and provides users with great flexibility as to the positioning of each view in the workbench.

The Commands view displays commands as parent nodes and command parameters as their children nodes. The Telemetry view displays instances as parents and instance attributes as their children nodes. Since each project is associated with its own target system database, when a user selects a project in the Navigator view, the two database views refresh to reflect the variables in the target system of the selected project.

Users can find properties of database variables with little effort. When users right-click on an entry in a database view, they are presented with a pop-up menu of options
(see Figure 3-10). Paper prototyping indicated that most users tried right-clicking in the database views to find more information about selected database variables. The pop-up menu for all database variables has the following two choices: "Insert" and "Open Defining File". The "Insert" option will be described in the next section. The "Open Defining File" option opens up the appropriate file of the intermediate database, which is where variables are defined.

The pop-up menu for command parameters and instance attributes has an additional option to display properties of the selected variable. While commands have numerous properties, Timeliner developers have indicated that type and arrayness (size) are the only ones immediately pertinent to script development, so these two properties are seen inside the Commands view. The properties relevant to attributes
are type, arrayness, and access (read/write permission). Figure 3-10 shows the properties of the attribute FARM\_DATA\_C02. The “Open Defining File” and “Properties” options allow users flexibility in how they find data. Users who are comfortable with the database views can rely on them, while users who like to look at the intermediate database files can use the quick-link shortcut to open the defining files.

Figure 3-10: Telemetry view showing the properties of a telemetry attribute.

### 3.5.3 Insert Into Script

When users locate a database variable that they need, they can insert it into the opened script right from the database views. The right-click pop-up menu described in the previous section has an “Insert” option. As users hover over that option with their mouse, a preview of the insertion is shown in the editor (see Figures 3-11). The variable appears at the current cursor position in italic font and grey color, to distinguish it from the persistent text in the editor.

The insertion is context sensitive, behaving differently for different types of database variables. If users want to insert a command with all of its parameters, they can select “Insert” when the command is selected. After each parameter a “=>” appears, allowing users to easily set the values for each parameter after the command has been inserted (see Figure 3-11). If users wish to insert a command with only some of its
parameters, they can specify those parameters by selecting them in the Commands view while holding down the CTRL key. If users accidentally selects multiple commands, or selects multiple parameters of different commands, nothing is inserted, as this is an invalid combination. In the Telemetry view, if an instance is selected, its name is inserted. If an attribute of an instance is selected, then the instance name is inserted, followed by a period and the attribute name. If multiple database variables are selected, only the first selection will be inserted.

Users who want to quickly insert a database variable without having to right-click and choose a menu option, can double-click on that variable in a database view and have it inserted into the editor at the current cursor position. Expert users found this feature to be extremely valuable, as it saved them the time of having to type out the variable.

### 3.5.4 Search

If users have an idea of what commands or telemetry data they need, but are not sure how to spell it, or if the variable is indeed in the target system database, they can search for it. The Database Search view (see Figure 3-12) provides a simple interface to find database variables of the target system. Users can type a query into
the text box and click on the “Search” button. The search is case insensitive and tolerates spelling mistakes. The Commands and Telemetry views are filtered to show only those variables in the database that are close in spelling to what the user typed. After the desired value is found, users can click “Clear” to clear the search text box and restore the Commands and Telemetry views.

3.6 Builder

TIDE is equipped with a builder that can compile and clean projects and files. The Project menu has a “Build Automatically” option that, if checked, automatically compiles files when they are saved. This feature provides users with instant feedback on the correctness of their scripts. If the option is not selected, then users have complete control when files are compiled. To manually compile the file open in the editor, users can click on the compile button in the toolbar, or select “Compile and map” from the Run menu. To compile several files at the same time, a user can select “Build All”, “Build Project”, or “Build Working Set” from the Project menu. These options give users the flexibility to compile a single project, all the open projects, or a specific subset of projects, respectively. Since the compiler can take up to a few seconds to run, an animated progress bar (see Figure 3-13) is displayed in the lower right hand corner of TIDE to reassure users that their request is being processed, and to let them cancel the operation if they so desire. Projects can also be cleaned, which means all of the files that the compiler generates are deleted.
3.6.1 Preferences

Users can set the compiler options that are applied every time the Timeliner compiler is run. These options previously had to be typed at the command prompt when the compiler was invoked. Users can set these options by going to the Window menu, selecting “Preferences”, and then “Timeliner Compiler”. Figure 3-14 shows the compiler preference page. Novices can move ahead with the default options, while experts have an easy way to customize the compiler.

![Compiler preference page](image)

Figure 3-14: Compiler preference page.
3.6.2 Displaying Errors

Compile errors are presented to users inside the editor and in the Problems view. Whenever the compiler is run, discovered errors are immediately displayed in the editor (see Figure 3-15). The left margin of the editor displays markers, in the form of red circles with an “X”, next to the lines that have errors. The right editor margin corresponds to the editor’s scrollbar and shows red lines next to error locations so that a user can drag the scrollbar thumb to move an error line into the editor’s visible region. Error lines in the editor are also underlined in red so when users visually scan a file they can immediately spot errors.

![Image of the editor and Problems view](image)

Figure 3-15: Compile errors displayed in the editor and Problems view.

While the editor displays the locations of errors, the Problems view (bottom of Figure 3-15), exhibits error descriptions and allows users to navigate to errors in the editor. The Problems view is a table that displays information about errors in all of the scripts in the workspace, even if they are not open. Each table entry contains an error description, the file which the error is in, the name of the project that contains
that file, and the line number of the error. A user can click on an entry in this view and navigate directly to the error. If the file is currently open, the editor will scroll so that the line is visible, and if the file is not open, the file will be opened to display the error. Thus, users can quickly navigate between the errors that exist in the workspace.

Along with the editor and Problems view, users can view errors that were discovered during compilation in the Console view. When the compiler runs from the command prompt, its output and error messages are printed out. In TIDE, this output is redirected to the Console view, so experts who are accustomed to that output can see it in TIDE.

### 3.7 Tool Integration

Timeliner tools participating in the development process that were used separately are incorporated in TIDE. These include the database importer, compiler, and TLX viewer. Previously, these tools had to be launched from the command prompt or from a primitive toolbar written in Swing. This process wasted users' time since they could only invoke one tool at a time and would need to manually supply its input. In TIDE, these tools are launched from within the environment, process multiple inputs concurrently, and look similar to the rest of the application because they are written in SWT.

Users invoke these tools by clicking on their icons in the toolbar or by selecting them from the Run menu. The icons representing these tools are preserved from the Swing toolbar, so experts can easily locate them. TIDE's initial design only had these tools launchable from the toolbar, but user studies proved that novices first looked in the menus and avoided unknown buttons. Consequently, these tools are available under the Run menu, and also as shortcuts for quick invocation. The tools show their progress with an animated icon in status bar (see Figure 3-13) and output results to the Console view.
3.7.1 Database Importer

One of users’ tasks is importing an external target system database. TIDE’s design lets users do this in two different ways. They can import and filter an external database during project creation by selecting that option in the Tide Project wizard, or they can import an external database at any other time by launching the import database tool from the toolbar or Run menu. The dialog for database import is similar in look and feel to that of the former Swing implementation, allowing it to be easily recognized by experts.

3.7.2 Compiler

The Timeliner compiler is integrated into TIDE’s builder framework. The Swing compile tool presented the user with a file chooser dialog to select a file for input. In TIDE, the compiler is launched directly on the file open in the editor. In addition, it can be automatically invoked on multiple files concurrently when a user chooses to build a project. Concurrent compilation and the removal of the need to manually select input files increases users’ productivity.

3.7.3 TLX Viewer

The Timeliner compiler produces binary executable files with TLX extensions. Important properties can be extracted from these TLX files and converted into text format. The former TLX viewer was written in Swing and displayed this text. In TIDE, TLX files can be viewed in the editor window (see Figure 3-16). TLX files are identified by an icon with a brown “X” symbol. When a user double-clicks on a TLX file in the Navigator view, important properties from the file are extracted and displayed in the editor window. Opening the file in an editor makes resource management easier, since users do not have to switch between applications, and can save the text properties file directly from the editor. Additionally, if a TLS file is open in the editor, its corresponding TLX file properties can be opened by clicking the TLX button in the toolbar or choosing “View TLX file” from the Run menu.
3.8 Integrated Help

TIDE provides Timeliner language assistance and help with the environment.

3.8.1 Keyword Reference

When novices compose scripts, they are not very familiar with the Timeliner language. TIDE provides a Reference view where users can browse through Timeliner keywords. Hovering over a keyword displays a tooltip with a brief description and code snippet of how the keyword should be used (see Figure 3-17).

This view has a similar style of user interaction as the database views; keywords can be inserted directly into the editor by right-clicking on them and selecting “Insert” or by a double-click. A preview is shown in the editor when the user hovers over the insert option. The Reference view insertion feature uses templates that correspond to Timeliner syntax. For instance, MESSAGE statements always have the form: MESSAGE ‘string’ where string is what will get printed out when the script is run. The Reference view has a template for the keyword MESSAGE such that after the keyword, quotations are inserted, and the cursor is placed in the middle of the quotations so
that the user can start typing the string contents right away.

User studies proved that users liked the hover tooltip feature, however they did not like reading long tooltips. Consequently, the Reference view has tooltips with a very brief (one or two sentence) description, and an outline of a template how a keyword should be used (see Figure 3-17). Users who want more information about a keyword can choose the “Open Documentation” option on the right-click pop-up menu (see Figure 3-18) which opens up the official Timeliner language documentation.

![Figure 3-17: Programmer’s reference of Timeliner keywords with tooltips.](image)

3.8.2 Environment Help

Help on how to use TIDE is found under “Help Contents” in the Help menu. Since most users do not like to read documentation unless they need help with a specific issue, the help section is organized in a frequently asked questions (FAQ) format. There is a brief section about installing TIDE and how to manage TIDE’s view layout.
The rest of the help section is organized by users' tasks. As seen in Figure 3-19, the FAQ is broken down by the following tasks: resource creation and manipulation, database interaction, script composition, compilation, TLX properties viewing, and search. The TIDE User Guide is searchable, so users can quickly find answers to any specific questions.

![Figure 3-18: Reference view pop-up menu.](image)

![Figure 3-19: TIDE user guide.](image)
Chapter 4

Implementation

TIDE is implemented as a plug-in to the Eclipse Platform. All code is written in Java and SWT is used as the graphical widget toolkit. This chapter explains why building on top of Eclipse is beneficial, characterizes how TIDE extends Eclipse, and discusses implementation decisions.

4.1 Choosing a Foundation

Several different implementation options were considered for TIDE. One possibility was to build an IDE from scratch using the Swing toolkit. Development time was estimated to be 6 man years, since Swing only provides basic widgets such as text boxes, buttons, and lists. The other option was to build on top of an existing open-source environment. Two candidates that were considered are Emacs and Eclipse.

Emacs is a command-based text editor that is implemented in the Emacs Lisp language and comes with a large series of features such as advanced text processing, syntax highlighting, and search. Emacs is efficient, since typing commands is faster than clicking buttons or navigating menus. However, Emacs has poor learnability and memorability. Users new to Emacs have a hard time using the editor, since they must read documentation to find how to do simple tasks like file saving. Furthermore, Emacs commands (such as CTRL-X CTRL-S for file saving) are not obvious, so they are hard to remember, and returning users might find themselves once again reading
help documentation.

Since TIDE needed to provide a wide variety of features, and wanted to support novice users, a direct manipulation interface was the best option. Eclipse was chosen as a base for TIDE because of its modular and extensible nature, as well as its collection of existing components such as a workbench and standard text editor. International Business Machines Corporation (IBM), the company that created Eclipse, spent $28 million developing and testing a rich source of features that TIDE inherits by extending Eclipse. This feature reuse reduced development time from 6 man years to 6 man months. Furthermore, building on top of Eclipse allows TIDE to maintain platforms' native look-and-feel through the use of Eclipse's Standard Widget Toolkit (SWT), a toolkit that delivers native widget functionality.

4.2 Eclipse Extensions

TIDE extends points in Eclipse as seen in Figure 4-1. The Product extension allows TIDE to have branding capabilities such as a custom splash screen and welcome page. The Editor extension point provides support for custom Timeliner language definitions. Eclipse's Builder framework gives a clear interface for the Timeliner compiler. The ActionSet extension point assists with Timeliner tool integration into the environment. Existing Wizard pages are a good base for Timeliner resource creation wizards. The View extension supplies view and controller functionality, letting TIDE classes implement the model. TIDE's orientation of views is easily grouped together and saved through an Eclipse Perspective concept. Preference stores are provided through a PreferencePage interface. Lastly, Eclipse provides a Help extension with HTML pages and a search service that TIDE utilizes to present users with a guide to the environment.

The following sections describe the major functional pieces of TIDE. Figures 4-2 through 4-4 are module dependency diagrams that show how the described classes relate to each other. As seen in the diagrams, the main class in the system is TidePlugin.
4.2.1 Resource Creation

TideNewProjectWizard extends Eclipse’s Wizard and guides users through the process of creating a project. TideNewProjectWizardPage1 is for inputing the name and location of a project, and TideNewProjectWizardPage2 is for importing an external database. Project preferences, such as whether the internal database is in the test or real environment, is set by ProjEnvPrefs, which extends Eclipse’s PropertyPage. TideNewFileWizard allows users to create a new TLS file and pick a project to put it in. Figure 4-2 illustrated these resource creation classes.

4.2.2 Editor

TimelinerEditor is registered as the default editor for TLS and TLL files in TIDE.
(see Figure 4-2 for classes associated with text editing). TimelinerEditor extends Eclipse’s TextEditor and provides custom Timeliner language definitions that are enforced by TimelinerDocumentProvider and TimelinerPartitionScanner classes. TimelinerEditorActionContributor specifies the editor’s text processing actions that are added to TIDE’s Edit menu. One such action is formatting a region of code according to Timeliner grammar by TimelinerFormattingStrategy.

The editor also contains an instance of TimelinerSourceViewerConfig, which dictates how text in the editor is displayed. TimelinerTagScanner is responsible for syntax highlighting. It determines whether a region of code is a Timeliner keyword, comment, error, string, or plain text, and colors the region accordingly. TimelinerTagScanner uses helper classes to accomplish syntax highlighting. TimelinerWordDetector determines if a character is part of a Timeliner keyword, and CaselessWordRule and CaselessScanner look in the keywords defined in TimelinerSyntax to determine if there is a match. TimelinerAutoIndentStrategy is invoked when users typing in the editor hit “return” to move to the next line. This class determines the indentation of the next line according to Timeliner grammar rules. TimelinerCompletionProcessor provides in-place assistance in the editor. As users type, they can request the completion processor, which displays a pop-up box with possible Timeliner keyword completions.

Numerous classes related to text editing rely on TimelinerSyntax, which defines Timeliner keywords by category and uses CaselessHashSet and CaselessHashMap to store groups of keywords for fast lookup.

4.2.3 Database Interface

Target system databases are manipulated by ImportDBAction and BuildDBAction (see Figure 4-3), which extend Eclipse’s ActionDelegate. ImportDBAction launches DatabaseBuilderUI, a dialog that lets users specify a directory where the external database is located and files of commands and telemetry data to use as filters. DatabaseExecutables are used by these action classes. It groups database related processes in CommandExtractor, MemoryExtractor, AddressConverter, and
Figure 4-2: TIDE editor, utilities, preferences, and resources classes.
GroundDBExecutor. Output files from these processes comprise the intermediate and internal database formats. These files are placed inside project folders where they can be read by CompileAction and classes implementing IDatabaseView.

A simple interface to the database is presented to the user by CommandsView and TelemetryView, which extend Eclipse's ViewPart and implement IDatabaseView interface (see Figure 4-4). These two views are implemented as Trees that present data extracted from the internal database by DatabaseParser. These Trees are populated with CommandEntries and their CommandParameters, and TelemetryEntries and their TelemetryAttributes.

SearchView also extends Eclipse's ViewPart, and provides users with an input field and search button to query the target system database. The search allows wildcards, and attempts to fix spelling mistakes by using EditDistanceAlgorithm to find the closest match to the strings entered by users.

### 4.2.4 Builder

Builder extends Eclipse's IncrementalProjectBuilder and provides methods for a full build, incremental build, and clean build of projects (see Figure 4-3). These methods create Visitor classes to visit files in a project folder and perform applicable operations. These visitors in turn create Compiler and Mapper classes that launch the command line Timeliner compiler and mapper in threads and report the results to TIDE's Console view. If the Compiler or Mapper have errors to report, an ErrorDetector is created, which scans the appropriate TLL file for errors, and reports them to the environment. Compiler options are specified by CompilerPrefs, which extends Eclipse's PreferencePage.

### 4.2.5 Tool Integration

Timeliner tools are integrated into TIDE to present a consistent front end to users. The supported actions are importing and building a target system database, compiling a file, and viewing the properties of a TLX file. This functionality is accomplished
Figure 4-3: TIDE's action set classes.
Figure 4-4: TIDE's view classes.
through ImportDBAction, BuildDBAction, CompileAction, and ViewTLXAction (see Figure 4-3). The database and compiler classes were described in previous sections. ViewTLXAction runs the TLX executable in a thread and wraps the output in a TLXEditorInput object. This object is passed to the TLXEditor, which extends Eclipse’s TextEditor.

4.2.6 Integrated Help

ReferenceView and ReferenceTooltips provide a way for users to browse through Timeliner keywords defined in TimelinerSyntax and to insert these keywords into TimelinerEditor. TIDE’s help contents pages are HTML files, and are thus not included in the module dependency diagrams.

4.3 Discussion

4.3.1 Formatting Algorithms

TimelinerAutoIndentStrategy is used by TimelinerSourceViewerConfig to indent code according to Timeliner grammar constructs. When users press “return” at the end of a line as they type, this class determines the indentation of the line just typed and of the next line.

The algorithm looks at Timeliner keywords at the beginning of lines and indents lines according to which categories those keywords fit in. Table 4.1 illustrates the three main keyword categories. The algorithm operates with respect to the categories as follows:

- **Start block keyword.** Indents the next line by a tab.

- **Intermediate block keyword.** These keywords are defined as those that are within a block, but should be indented the same amount as the start and end block lines. The algorithm finds the corresponding start block keyword, indents the intermediate line by that amount, and indents the next line by a tab.
• *End block keyword.* Finds the corresponding start block keyword by scanning the file upwards, and indents the end line by that amount.

• *Any keyword.* Indentation of the previous line is copied over to the next line.

<table>
<thead>
<tr>
<th>Start</th>
<th>Intermediate</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUNDLE</td>
<td>OTHERWISE</td>
<td>CLOSE</td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>ELSEIF</td>
<td>END</td>
</tr>
<tr>
<td>SUBSEQUENCE</td>
<td>ELSE</td>
<td></td>
</tr>
<tr>
<td>EVERY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WHEN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WHENEVER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Categorized Timeliner keywords.

Figure 4-5 shows code indented by the described algorithm. Lines 1, 3 and 4 contain the start block keywords BUNDLE, SEQUENCE and WHEN, so the lines after these, up until the end of the blocks, are indented by a tab. The intermediate keyword OTHERWISE on line 6 is indented by the same amount as the start of that block on line 4. Lastly, the end block keywords END on line 9, and CLOSE on lines 10 and 12 are indented the same as the lines that started those blocks of code.

```plaintext
1  BUNDLE PLANT_SIM
2
3  SEQUENCE CO2_MONITOR
4      WHEN FARM_DATA.CO2_PUMP = OFF WITHIN 10 THEN
5      WAIT 1
6  OTHERWISE
7      WARNING "CO2 PUMP NOT TURNING OFF"
8      STOP CO2_MONITOR
9      END WHEN
10     CLOSE SEQUENCE
11
12     CLOSE PLANT_SIM
```

Figure 4-5: Correctly indented Timeliner script.

`TimelinerFormattingStrategy` is called from `TimelinerSourceViewerConfig` and corrects the indentation of either the selected region in the editor or the whole
document if nothing is selected. It formats code according to Timeliner grammar, taking into account the different categories of keywords shown in Table 4.1. Comment lines are left unchanged so individual users' commenting styles are not hindered. The algorithm starts scanning code line by line from the beginning of the region and keeps track of two things: \texttt{indentsStack} (a stack of indents from start and close block statements), and \texttt{curIndent} (what the indentation of the next line should be). When a line from the editor is read, its first word is extracted and the line is categorized and processed according to the following six cases:

- **CASE 1**: Bundle, sequence or subsequence starter. Indents for these lines are hard coded to be no indent for bundle lines, and a tab indent from the left margin for sequence or subsequence starting lines. The size of this indent is pushed onto \texttt{indentsStack}, and \texttt{curIndent} is set as the line's indent plus a tab so the next line will be indented inside the bundle or sequence.

- **CASE 2**: Start block keyword other than bundle, sequence or subsequence. Set this line's indent to that of \texttt{curIndent}, which should be the proper indent. If \texttt{curIndent} has not been set, then this is the first line in the region, assume it is indented correctly and push the indent on \texttt{indentsStack}. In any case, set \texttt{curIndent} as this line's new indent plus a tab since this is the start of a block.

- **CASE 3**: Intermediate keyword. If \texttt{indentsStack} contains a value on top of the stack, indent this line by that amount and keep the same \texttt{curIndent}. If \texttt{curIndent} has been set, but \texttt{indentsStack} is empty, this intermediate keyword's corresponding starting keyword was not part of the selection. Treat this line as a start block line and push it on \texttt{indentsStack} so the ending block keyword will line up appropriately, then add a tab to \texttt{curIndent}. If \texttt{indentsStack} is empty and \texttt{curIndent} hasn't been set, that means this is the first line in the selected region. Keep the indent of this line, push this indent onto \texttt{indentsStack}, and set \texttt{curIndent} as this line's indent plus a tab.

- **CASE 4**: End block keyword. If \texttt{indentsStack} is not empty, pop the top of the stack to find the indent of the matching start block and set that as this line's
indent. If `indentsStack` is empty, use `curIndent` minus a tab as the indent for this line because that is the best guess. If `curIndent` is not set, then this is the first line in the region, so keep its indent. Always set `curIndent` to the indent of this line, as the indented block has ended.

- **CASE 5: Any keyword.** Set the indent of this line to `curIndent`. If `curIndent` has not been set, then keep this line’s indent and set it as the `curIndent`.

- **CASE 6: Any non-keyword.** Non-keyword lines can be comment lines or lines that are partially selected. If this is a comment line, do not change its indent. If this is a partially selected line, it means this is the first line in the region. Get its indent (which is outside the selection) and set that as `curIndent` so the next lines will be formatted with respect to this line.

Thus, `TimelinerFormattingStrategy` can correct the indentation of any selected region of code by reading the region one line at a time, and establishing its proper indentation according to the six cases described above.

### 4.3.2 Case-less Support

The Timeliner language is case insensitive, so the editor and all operations that rely on Timeliner keywords need to be case insensitive as well. `TimelinerSyntax` is the class that all other classes rely on for Timeliner language keywords and structure. `TimelinerSyntax` defines keywords in hashed sets, grouping them by category. A categorized structure is easy for programmers to read, and supplies a structure for the Reference view. Keywords are also stored in hashed maps, to assist the formatting strategies. These structures provide a mapping between keywords that end blocks with their corresponding keywords that open blocks. While hashed structures provide the benefit of near constant lookup time, the default string hash functions are not case sensitive, so a hash of the string “BUNDLE” is different from that of “Bundle” and that of “bundle”. This posed a problem, as all of these capitalizations are valid.

This problem was solved with data structures that ignore case. `CaselessHashSet` and `CaselessHashMap` are wrappers around the default implementation of `HashSet`
and HashMap. These classes store strings in uppercase, but lookup methods such as contains() and get() ignore case. When these methods are called with strings, they convert the strings to uppercase and then attempt to find the strings in the stored data structure.

4.3.3 Database Building and State

TIDE supports test and real database environments. Each project folder contains files for both environments that are generated when an external database is imported (see Figure 4-6).

An external target system database must first be converted into the intermediate database format. Users specify the location of the external database and the files that have commands and telemetry data that are used as filters. ImportDBAction creates a DatabaseExecutables class which runs a command and memory extraction processes. These processes generate intermediate database files ("commands.txt", "instances.txt", and "attributes.txt") that are placed in a folder inside the project called "gdb.in". The "instances.txt" file now has real addresses (addresses of instances on the target system). The file needs to be different if test environment (local address space) is desired. TIDE makes a copy of this file under the name "instances_real.txt" and then runs an address converter to change memory addresses as is required by the test environment. The resulting file is renamed to "instances_test.txt".

After filtering commands and telemetry data from the external database into an intermediate database, an internal database must be created that the compiler can read. The database builder is run, which takes as input a "GDB_INPUT.SCRIPT" specifying the locations of commands, attributes, and instances files, as well as the location of a "gdb.out" folder to place output files. To create files for both environments, the builder is run twice, once to process "instances_real.txt" and place its output in a "gdb.out_real" folder, and a second time to process "instances_test.txt" and place its output in a "gdb.out_test" folder. Furthermore, the executable writes different "GDB_TESTER_FILE.TXT" and "GDB_PUI_ADDRESS_FILE.TXT" files based on the environment. These two files are renamed so that either "REAL" or
“TEST” is appended to the end of the file name.

Since all the necessary files for both environments exist simultaneously, the compiler needs to know which files it should read. The compiler gets input information from a file called “GDB_PATH.SCRIPT”. This file lists the appropriate “gdb.out” directory and is updated when the user switches the project preference between the two environment options. The correct “GDB_PUI_ADDRESS_FILE.TXT” file must also be given to the compiler. TIDE checks the environment variable in the project’s preference store, and the applicable file is supplied as an argument to the compiler.

4.3.4 Editor Text Insertion

The Reference view and database views let users browse through keywords and variables that can be used in scripts. Providing users with a way to insert this text into the editor directly from the views would save users time and prevent spelling mistakes. The first design iteration simply included the right-click pop-up menu and performed the insertion operation. However, a problem became evident during user testing. While users interacted with these views, the editor lost focus and the cursor was no longer visible. When these users chose to insert into the editor, they could not tell where the insertion would be placed.

TIDE’s current design solves this problem of not presenting adequate system state to the user by switching focus to the editor and showing a preview of the insertion when a user hovers over the insert option. When the right-click menu is created, a Listener of type SWT.Arm is set for the “Insert” item. When a user arms the item by hovering over it with the mouse, the keyboard focus is switched to the editor and the cursor becomes visible. Furthermore, TimelinerEditor.insertPreviewText() is called with a string. The editor inserts this string into the document at the current cursor position and sets the style for that region to be italic font and grey color, indicating its temporary status. When a user moves the mouse to another item in the menu, or when the menu is destroyed, TimelinerEditor.removePreviewText() is called to return the editor to its former state.
4.3.5 Database Search Algorithm

The Database Search view allows wildcards, and attempts to fix spelling mistakes in the search query. `EditDistanceAlgorithm` finds the closest match among the target system database variables to the string entered by the user. The "edit distance" of two strings is the minimum number of character mutations required to change one string into the other. Character mutations include change, insertion, or deletion of a character. A standard implementation of the edit distance algorithm [5] is employed in TIDE. A threshold of 0.2 was chosen, meaning 80% or more of a result string matches the query string.

4.3.6 Compiler Selection

The Timeliner compiler used on TLS files is written in Ada and invoked from the command prompt. Two years ago, a Java version was developed as part of a Masters thesis at MIT [12]. One idea for compilation inside TIDE was to use the Java compiler for checking syntax in the editor and the Ada compiler for producing output files (such as TLL and TLX files). Using the Java compiler would provide a speed advantage, since the Ada compiler runs much slower than the Java one, and users save files frequently when they type. (Table 4.2 compares the speeds of the two compilers as they performed on an Intel Pentium M machine with a 1.3 GHz processor and 256MB of RAM.) The Ada compiler would still be needed for producing output files since the Java compiler only produces an XML syntax tree. Unfortunately, using the Java compiler for syntax checking was not possible because the compiler does not do any error handling. If everything in the input file is correct, it runs to completion. However, if there are any errors in the file, the Java compiler throws an exception and crashes. Thus, TIDE only uses the Ada compiler. When files are saved, the Ada compiler launches in a background thread so users can continue editing files.

When the Ada compiler completes, the generated TLL files are scanned for errors. Error lines in a TLL file are matched up to the corresponding lines in the TLS file, and errors are reported to TIDE's error handling infrastructure (the classes that list errors
in the Problems view and underline them in the editor). On a slow computer, the Ada compiler can take a few seconds to process a large file. This presented a problem, since the user might have modified the TLS file after the compiler was launched. In this case, the generated TLL file would not correspond exactly to the TLS file, as lines could have been inserted, deleted, or modified.

TIDE's solution is to check if the file has indeed been modified, and if so, to re-run the compiler on the latest version, saved or unsaved. When the compilation and mapping processes complete, a UI thread is launched to find errors. It checks to see if the file has changed since the last save operation, if so, it compares the generated TLL file with the current state of the TLS file in the editor instead of the last saved state. If these two files do not match, the compiler is re-run. This comparison is done in a UI thread, unlike the actual compilation which is done in a background thread. A UI thread is utilized so users cannot modify the file as the comparison operation is processed. Since comparison is a fast operation, users do not notice the delay.

### 4.3.7 Concurrent Compilation

When projects are compiled, the files in them are compiled concurrently. Since the Ada compiler can take a few seconds to complete, a sequential compilation operation could take tens of seconds. To make compilation efficient, each file needs to be compiled in its own thread. Compiled files are tracked in the Timeliner executor by a *tracking tag* that is generated by the compiler. When the compiler begins, it gets the current system time (in the granularity of seconds) and uses that as the *tracking tag* for the generated output files. As a result of compilations occurring concurrently, numerous files would start compiling at the same time, and would have the same
tracking tag.

This problem is solved by adding a one second delay to the start of the compiler. A global variable records the last system time when the compiler was launched. Each file still gets its own compilation thread, however, the thread compares the current system time with the last global compile time. If the difference is less than one second, the thread sleeps for however much time it needs until a second has passed, and then attempts to check the time difference again. The subsequent time check is necessary, as a competing thread could have started compiling while this thread was sleeping, so the last global compiler launch time would have been reset. Once the thread gets clearance to run, it sets the last compile time to the current system time. Consequently, concurrently compiling files have unique tracking tags.

4.3.8 TLX Viewer Input

The former TLX viewer was implemented in Swing, and ran an executable to process the binary TLX file and extract important properties. When the TLX viewer was integrated into TIDE, it was possible to implement it as a view or an editor. Since the TLX contents should only be displayed and not modified, a view representation was considered. However, the ultimate decision was to integrate it as an editor, since the contents is a representation of a file. This decision also allows the the contents to be saved as a TXT file right from the editor. Like TimelinerEditor, TLXEditor extends Eclipse's TextEditor class. The contents provided to this editor are in a TLXEditorInput object, a wrapper around the output from the TLX properties extractor that runs in a thread and produces a ByteArrayInputStream.

4.4 Extensibility

TIDE can easily accommodate changes in the Timeliner language and additional tools. All classes that process text for syntax highlighting, auto-completion, indentation, etc. rely on TimelinerSyntax for language definitions. TimelinerSyntax groups language keywords by categories and stores them in structures like CaselessHashSet.
and CaselessHashMap. If a keyword such as SWITCH was to be added to the Timeliner language, only the TimelinerSyntax class would have to be updated. Additional executable tools can easily be added to TIDE by contributing to TIDE’s ActionSet.
Chapter 5

Evaluation

TIDE underwent two rounds of user testing: a paper prototype evaluation and an evaluation of the application. Three paper prototype users and four application users participated in the study. Users were informed that they will be testing an application that is used to develop code. They got a briefing about the Timeliner language and the concept of a target system database. They were educated about the development process: how scripts are written in TLS files and compiled to produce an executable file. They were also informed about how the target system database contains commands that can be called, and telemetry data that can be queried and set.

Each user was faced with the following three tasks:

1. Create a new script file.

2. Write code to check if the temperature is below 21 degrees. If so, turn on a heater and print out “heater is on.”.

3. Compile the script and fix any errors.

The result of these tasks should look similar to the code in Figure 5-1. The tasks were designed to imitate tasks of real script developers. More specifically, they were meant to test TIDE’s resource creation, editing, database interface, and compilation features.
Figure 5-1: Code that users were asked to write during the user study.

```
1 BUNDLE USER-STUDY
2
3    SEQUENCE TEMPERATURE_CONTROL
4        IF FARM.DATA.TEMPERATURE < 21
5            COMMAND HEATING, NEW_STATE=>ON
6            MESSAGE "HEATER IS ON"
7        END IF
8    CLOSE SEQUENCE
9
10    CLOSE BUNDLE
```

Paper prototype users were all students taking a class on usability at MIT. They all liked TIDE's view layout with the editor in the center and various views surrounding it. Figure 5-2 shows a user interacting with the paper prototype. All these users were able to easily accomplish the first task of creating a new TLS file with the help of the wizard under the File menu. These Timeliner language novices liked the BUNDLE and SEQUENCE statements that were automatically inserted into the newly created script.

Figure 5-2: Pilot user interacting with a paper prototype of TIDE.

The three paper prototype users had different opinions as to the interaction with
the database views. All of them tried expanding the tree viewer to see database variables. However, some users tried double-clicking the variables while others tried right-clicking first. As a result of this split, TIDE supports both operations for inserting variables into the editor. Additionally, users did not like the clutter in the database views that was a result of variable properties being displayed in the view. Hence, TIDE's final design initially hides these properties, and they can be viewed through a right-click menu.

All pilot users made the mistake of forgetting to end their IF block of code with an END IF statement. One user simply typed the IF keyword, assuming the language supports it. Another user browsed the Reference view, and as soon as he spotted the keyword, he typed it into the editor. The last user found the keyword in the Reference view and used the right-click menu to insert it. Since all three users made the mistake of not including the END IF statement, TIDE's final design has template insertion in the Reference view. So when a user chooses to insert IF, the closing statement is also inserted into the editor.

For the compilation task, two out of the three users looked to menus for a compile option. The third user looked at the toolbar and hovered over icons to see tooltips. The paper prototype didn't have a compile menu option, but TIDE's final design consequently has a Run menu from where all the formerly stand-alone tools are launched. The paper prototype users also suggested having compile errors displayed in the editor, which is implemented in TIDE's final design.

The application pilot users included a person who is an expert in the Timeliner language, and three people who do not know the language. These three Timeliner language novices had different levels of computer expertise. One was a programmer who had used a lot of IDEs, another was an average programmer, and the third was a non-programmer. The Timeliner language expert was thrilled to try out TIDE, a his current Timeliner development environment consists of Notepad and the Windows command prompt.

All application users successfully created a TLS file. The Timeliner language expert typed a TLS extension in the wizard's file name box, and all three novices did
not type an extension. TIDE’s design allows both variations, if an extension is not entered, it will be automatically inserted. All four pilot users kept the default insert statements checked and liked the automatic insertion of the BUNDLE and SEQUENCE blocks. The SEQUENCE statement was given a default name so that users could start composing its body right away. None of the users initially changed this default name, but the expert and one of the novices changed the name to a more descriptive one after finishing composing the script.

During script composition, all users were able to locate the necessary Timeliner keywords. The language expert typed the IF block directly into the editor. All three language novices found it in the Reference view, and inserted from there by choosing that option from the right-click pop-up menu. This insertion routine revealed a problem. The editor’s cursor was not visible when the insert operation was performed. Thus, when users tried to insert, they were unsure about where in the editor the insertion would appear. As a result, TIDE’s final design shows a preview of the insertion when users hover over the insert option in the menu. When the language expert user was trying out TIDE, this preview feature was implemented. As he brought up the pop-up menu and moved his cursor over the insert option, he was at first surprised by the preview showing in the editor. He then proceeded to try it out on random keywords and expressed enjoyment. Since the Reference view’s template insertion feature automatically inserted the END IF statement, all of these pilot users avoided the pitfall of the paper prototype users.

Two out of the four application users typed characters in all uppercase, and the other two in all lowercase. One of the language novices and the language expert typed all capital letters. This novice was trying to follow the conventions of the already inserted SEQUENCE block. The language expert was typing in capitals out of habit. Moreover, when he accidentally typed the end in lowercase, he deleted it and retyped it in uppercase even though the keyword had been colored purple in the editor. The other two novices cautiously tried typing in lowercase, and as soon as they saw keywords syntax highlighted, they proceeded with confidence.

When the time came to use database variables in the script, all users immediately
turned to the database views. They browsed through the commands until they found one relating to a heater, and chose to insert into the editor with the assistance of the right-click menu. They were thrilled how a whole statement (COMMAND HEATER, NEW_STATE=>) was inserted for them. Novices would not have otherwise known the proper format for setting commands. The expert was able to save time by not having to type out the whole statement. Once the command was inserted into the editor, three of the users asked, “how do I know what value I can set for this command?” Apparently, these users did not notice the “Properties” option on the right-click menu because it was at the bottom of the menu items. Three of the users, after a few second pause, tried right-clicking on the variable and discovered its properties. The last user decided to type a value in the editor, hoping the compiler will inform him if there is a problem. When these users needed to perform a numeric comparison, three users typed in the greater than sign, and one of the novices looked in the Reference view. He was able to quickly navigate to the comparison section through the categories in the view.

For the compilation task, all application users looked to the menus and speedily found the compile option. The users liked how errors were underlined in the editor, and they immediately saw that error descriptions where displayed in the Problems view. The language expert intentionally introduced errors into his script to see what would happen. He was thrilled to see all the red marks in the editor appear, and tried clicking on each error on the Problems view to navigate to the line in the editor window.

All users were able to efficiently accomplish the tasks laid out for them. Moreover, they all voluntarily expressed enjoyment during the exercises.
Chapter 6

Future Work

TIDE's current database handling routines only support importing and filtering the ISS stdout databases. TIDE's goal is to support more customers in the future. To more easily support the import step, TIDE classes that deal with the database can be extracted into a separately deliverable Eclipse plug-in. These classes can then be standardized through interfaces that will allow any database format to be easily integratable with TIDE. Furthermore, intermediate and internal database files are currently stored in project folders. These extracted data files can be more easily maintained, and storage space can be reduced, if the files are stored in their own project folders. "Database projects" could be created such that numerous projects of Timeliner scripts can reference the same database files. Dependencies can then be instantiated, such that when a database project is modified, all the script projects that use variables from that database project are automatically recompiled.

Intermediate database files can be modified by hand, which makes them susceptible to typing mistakes. The commands, instances, and attributes text files that constitute the intermediate database format define database variables to be used in scripts. Script developers can modify these files by hand. However, these files require a particular format that can be hard to get right without careful attention to every comma and parenthesis bracket. Modifying these files can be made easier through a database editor. This editor can have templates for commands and instances. For instance, users can specify that they want to create a command, and pick the number
of parameters. Then they would be presented with a form where names, types, sizes, etc. can be entered for each parameter.

Properties of TLX files can be viewed inside TIDE's editor window. However, these properties are displayed as a plain text file, making it hard to comprehend. An enhancement would be to create an advanced TLX viewer where users can navigate through the properties and be able to easily identify important information.

Lastly, compilation could be faster inside TIDE if a Java compiler was used. Table 4.2 showed that the Java compiler gives a large performance advantage over the Ada compiler. Large files compiled about 7 times faster and small files compiled about 15 times faster with the Java compiler. Unfortunately the current Java compiler does not do any error handling. If the Java compiler were extended to check Timeliner grammar, it could be used for fast, incremental compilation in TIDE.
Chapter 7

Conclusion

TIDE increases Timeliner script developers' productivity with its rich feature set and tool automation capabilities. TIDE integrates stand-alone tools, and provides a custom editor, database interface, incremental compilation, and integrated help.

Developers previously had no language specific assistance during script composition, leaving scripts susceptible to errors. TIDE provides smart editing with custom language definitions that highlight keywords and indent according to Timeliner grammar. These features allow mistakes to be spotted earlier in the development cycle. Compilation occurs continuously in the editor, and is invoked when scripts are saved. Users can thus recognize, catch, and fix errors quickly. Integrated help gives users fast access to Timeliner keywords, eliminating the need to navigate through lengthy help documents.

Looking up target system database variables was formerly a tedious process. TIDE allows users to browse through commands and telemetry data, and easily look up their properties. Users can follow a link to view the intermediate database file where a selected variable is defined. Users can also search for specific database variables. The tedious process of scrolling through lengthy text files to find database variables has been eliminated. Moreover, database variables can be inserted from the browser right into the editor, saving time and preventing typing mistakes.

Lastly, TIDE makes stand-alone tools more efficient. These tools are launched from, and their output is processed, in the environment. The external database
The importer is invoked from a menu, and its output is displayed in TIDE’s console. The compiler processes multiple scripts concurrently, and errors are reported to the editor and a problems view. Compiler options can be specified once and applied to multiple compilations. Properties extracted from TLX files are viewed in, and saved from, the editor.

In conclusion, the Timeliner Integrated Development Environment allows any person to quickly and easily develop Timeliner scripts.
Bibliography


