An Extensible Object-Oriented Executor For The Timeliner User Interface Language

by

Steven M. Stern

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Computer Science and Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 18, 2005 [June 2005]

Author ....
Department of Electrical Engineering and Computer Science
May 18, 2005

Certified by ....
Dr. Robert Brown
Charles Stark Draper Laboratory Thesis Supervisor

Certified by ......
Professor Robert Berwick
M.I.T. Thesis Advisor

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

BARKER

THIS PAGE INTENTIONALLY LEFT BLANK

An Extensible Object-Oriented Executor For The Timeliner User Interface Language

by

Steven M. Stern

Submitted to the
Department of Electrical Engineering and Computer Science
May 18, 2005
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Computer Science and Electrical Engineering

ABSTRACT

Timeliner is a time-oriented scripting language and execution environment used on the International Space Station. This project describes the creation of a new version of the executor, using Java technology. This executor is more modular, extensible, and easier to maintain than the existing Timeliner system, which is written in Ada. This executor, in conjunction with a previously developed Java compiler, completes the Next Generation Timeliner system. This system can now compile and execute many test scripts, including a self contained simulation.

Thesis Supervisor: Dr. Robert Brown
Title: Charles Stark Draper Laboratory Thesis Supervisor

Thesis Supervisor: Professor Robert Berwick
Title: M.I.T. Thesis Advisor

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I would like to thank Robert Brown, Joseph Bondi, and the staff of the Charles Stark Draper Laboratory. Without their support, this project would not have been possible. I would also like to thank Professor Robert Berwick for his advice and guidance. Thank you to my family for your continual support.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under NASA Contract Number 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.

....

Steven M. Stern

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

A system as complex as the International Space Station must be thoroughly tested and monitored in order to insure the safety of the astronauts on board. Even the slightest mistake could ruin a mission. Such a mistake could cost hundreds of millions of dollars to repair and retry the mission, or worse yet, endanger the lives of those on board the station.

Performing these tests and monitoring the International Space Station could itself prove to be extremely costly. It is very expensive to bring someone on board and sustain them. The more time they spend testing and monitoring the station itself, the less time they have to perform their intended mission. Therefore, it is ideal to have an automated system perform these actions. Timeliner is the system currently used for this purpose on the International Space Station. This system has also been in use on the United States space shuttles since 1982 [3].

Before Timeliner, manual crew procedures were used for the testing and monitoring of the space shuttle systems. These procedures were written by ground control operators. The Timeliner User Interface Language (UIL) was written to closely mimic the language of the manual procedures, allowing operators to quickly learn the new computer system. At first, Timeliner was used in simulations for the space shuttle [3]. Since then, NASA chose to use Timeliner as the procedure executor for payload and core systems on the International Space Station.

Timeliner scripts can be used for almost any task on board the International Space

Station. This allows for greater productivity of the mission operators and on board crew, as well as providing greater reliability and safety since many tests and monitors can be created without sacrificing mission time.

However, Timeliner can be made even more extensible. The existing Timeliner system is written in Ada code. Extending the system is possible, but requires a significant amount of development work. Future NASA systems, as well as other systems, would benefit from a new, more modular and extensible Timeliner system.

1.1 Objective

The future of Timeliner is the Next Generation Timeliner. This new system will provide all of the functionality present in the existing Timeliner design, allow for more extensibility, use modern technologies and design patterns, and provide greater assurances of reliability. Previously, a compiler for this new system has been developed [2]. The objective of this project is to create the design, framework, and initial implementation of the executor for the Next Generation Timeliner.

This executor will complete the initial version of the Next Generation Timeliner. The executor and compiler will be usable such that scripts written for the current version of Timeliner can be compiled and executed, with results similar to that of the existing Timeliner system. However, this new system must be modular and extensible, allowing for new functionality not easily available in the existing system.

1.2 Motivation

One of the primary motivations for this project is to create a version of Timeliner that is easier for software developers to maintain. The current version of Timeliner is written in Ada, a language that is not as commonly used as it once was. The new system will be implemented in Java. This allows developers to make use of the tools and support offered by a popular, widely used language. The object oriented nature of Java also allows for the new system to be highly modular. Inheritance allows for the elimination of duplicate code. Finally, Java is platform independent, allowing for Timeliner to be easily used on any target system.

The existing Timeliner system is highly procedural. When executing a statement in the Timeliner User Interface Language, the Ada executor employs a large switch statement. There is no set definition or interface that the Timeliner UIL statements must conform to, which makes extending the system difficult and time consuming. Even small changes can have adverse effects on many parts of the existing system. The Next Generation Timeliner system will provide a fixed interface that Timeliner UIL statements must conform to. Each statement will be treated as a separate object, with abstract classes providing the duplicate functionality. This will make extending the Timeliner system significantly easier.

1.3 Scope

The Next Generation Timeliner consists of a compiler and an executor. This project only deals with the executor. In particular, the scope of this project is to design the framework for the new executor, and create an initial implementation within this framework. This design provides for ease of both unit testing and integration testing. Testing to show basic correctness and the viability of this design are within the scope of this project, as well as the framework for conducting more complex tests; however, the extensive testing NASA would require before this system can replace the existing system is outside the scope of this project.

The primary goal of this project is to create a working version of the Next Generation Timeliner that can immediately be used with great success in testing, simulation, and other systems that are not mission-critical.

1.4 Organization

The remainder of this thesis is presented in five chapters. In Chapter 2, the Timeliner User Interface Language and the existing Timeliner system are described in greater detail. Chapter 3 covers the design and implementation of the Next Generation Timeliner Executor, as well as the rationale behind some of the design decisions. Chapter 4 demonstrates the extensibility of this design by detailing how several changes to the Timeliner UIL can be implemented in this system. Chapter 5 explains how this system meets the goals described previously, including the successful execution of a suite of Timeliner scripts for a greenhouse system. Finally, Chapter 6 explains possibilities for future work, and what will be necessary for the Next Generation Timeliner to replace the existing Timeliner system.

Chapter 2

The Timeliner System

2.1 Overview

The Timeliner system was developed by The Charles Stark Draper Laboratory to automate procedures that require precise sequencing to achieve autonomous control of real-time processes. Timeliner is in use on the International Space Station, and is marketed commercially as part of the TLX family of products offered by the Auspice Corporation. Timeliner has provided significant mission cost savings and productivity gains.

The Timeliner system can be used for many tasks. On the International Space Station, Timeliner has been used for both routine operations as well as operational work-arounds for problems in the flight hardware and software. Timeliner's versatility is a combination of a simple to learn Timeliner User Interface Language, a compiler for this language, and an executor that interprets and performs the actions of the compiled scripts.

The Timeliner User Interface Language is a scripting language with a focus on time-oriented execution. The language was designed by examining how mission control operators would give procedural instructions to astronauts on board the space shuttle. A language that parallels these instructions was desired so that mission control operators, who generally have little knowledge of software development, could easily learn to write Timeliner scripts. The result is that Timeliner commands have a strong correlation to procedural instructions given by human operators. Timeliner scripts are very easy to create, yet still powerful enough to provide the same functionality that other scripting environments provide.

Most Timeliner statements are time oriented. The EVERY statement repeats a task periodically. The WHEN statement blocks until its condition evaluates to true. The more traditional constructs, such as IF ... ELSE are still available, but the time oriented statements are used significantly more often for describing procedures in a flight mission environment.

The statements in Timeliner also follow a very simple pattern. Every statement is of the form:

KEYWORD <argument> [<optional arguments>]

This makes it very easy for mission control operators to learn the Timeliner syntax.

Multiple scripts may be executed in parallel. This allows for handling triggers that do not necessarily have a predefined order: for example, separate actions taken when the temperature and the altitude reach various thresholds. Also, completely separate parts of a system can be controlled from the same Timeliner executor. For example, periodic checks to the International Space Station itself can be performed in parallel to a specific experiment being conducted on board. This gives greater power to the script developer, but also imposes additional complexity on the executor. Timeliner developers do not want to deal with resource allocation or contention; the system must simply work as intended.

The Timeliner system itself is divided into two main components: the compiler and the executor. There are several reasons for this division. Primarily, these systems execute at physically different locations. The compiler runs at ground control, while the executor runs on the target system. This physical division requires that there be a data transmission standard between the components. A minor change to the compiler located at a NASA ground control center should not require a change to the executor on the International Space Station. So long as this minor change does not break the standard, the existing executor version can remain in use.

This division also allows for greater modularity. Compiled scripts can be transmitted to external components for testing and validation. For example, the existing Timeliner system has a viewer for the compiled files, allowing the operator to visually examine exactly how the Timeliner compiler parsed and interpreted the original script. Minor changes to both the compiler and executor do not require changes to this external viewer.

The remainder of this chapter begins with a more detailed explanation of the Timeliner User Interface Language. This is followed by a description of the Timeliner compiler, followed by an introduction to the Timeliner executor. Each of these components refer to the existing version of Timeliner. The Next Generation Timeliner, and how it differs from the existing Timeliner, is described in detail in the subsequent chapters.

2.2 Language Hierarchy

The Timeliner User Interface Language is a hierarchical, block structured language. Each procedure in Timeliner is known as a SEQUENCE, and related sequences combine to form a BUNDLE. Procedures are loaded and unloaded from the Timeliner system at the bundle level at any point during execution. For this reason, sequences within a bundle are generally part of a common execution theme.

Within a sequence is a list of statements, with one statement per line. White space is important in the Timeliner UIL since a carriage return denotes the end of a statement.

Each statement is broken into "components". For example, the statement:

SET X = Y + Z

contains five components: the initial keyword, the three variables, and the addition operator (the equality operator is implied by the SET keyword component). These components have a very strong correlation to the Keyword Handlers described later in Section 3.3.

```
BUNDLE EXAMPLE
     DECLARE X NUMERIC
     SEQUENCE ONE
          WHEN X > 1
               SET X = 0
          END WHEN
     CLOSE SEQUENCE ONE
CLOSE BUNDLE
```

Figure 2-1: Example script in the Timeliner User Interface Language

Figure 2-1 provides an example of a simple script that uses these constructs. This script will check the value of X and wait for it to be greater than 1. Once this happens, the value of X is set to 0 and the script terminates. If the WHEN statement were replaced by a WHENEVER statement the script would repeatedly check the value of X and change its value to 0 every time it becomes greater than 1.

For a more detailed description of the Timeliner User Interface Language, there are references available [3].

2.2.1 Bundles

The highest level of the Timeliner UIL hierarchy is the bundle. A bundle can contain only four Timeliner constructs: sequences, subsequences, declare statements, and define statements. Every bundle must contain at least one sequence for it to be valid. Declare and define statements are for variables that can be accessed from any of the sequences or subsequences within the bundle.

Individual bundles can be installed and removed at any point during Timeliner operation. For this reason, the most logical division of sequences is according to what part of the mission they are needed. If sequences are divided into "launch", "mission" and "landing" categories, bundles will only be loaded into the system when they are needed.

2.2.2 Sequences and Subsequences

Sequences and subsequences are almost identical. They contain a list of statements that are generally executed linearly, with two exceptions. First, conditional statements can modify execution, though to a limited extent. The absence of both a "goto" instruction and a loop "break" instruction keeps execution control very simple. Second, sequences can be paused, stopped and started both from the human operator or by commands sent by another sequence in the same bundle. By default, a sequence begins as "inactive" and requires a start command for it to begin. Alternatively, it is possible to declare a sequence as active upon install in the script source file.

Subsequences, however, do not require start, stop, or pause commands. They can only be executed through a call from another sequence or subsequence, resulting in their execution being directly tied to their parent. If the calling sequence is stopped, so is the execution of the subsequence it called. In general, multiple sequences can call the same subsequence simultaneously. However, a subsequence can be declared as single-execution only. In such a case, a call to execute the subsequence will block if it is currently being executed by another subsequence. The call continues to block until the subsequence is free and can be executed. More than two simultaneous calls are handled in a first-called, first-handled manner.

Each sequence contains a set of statements, with one statement per line. In general, these statements are very straightforward. However, the control statements are what separates the Timeliner UIL from other scripting languages.

2.2.3 Control Statements

The three main control statements in Timeliner are WHEN, WHENEVER and EVERY. What makes these control statements unique is that each of them pauses execution until the conditional is true. For example, the statement:

WHEN TEMPERATURE $>$ 50

will block until the temperature value reaches 50.

```
WHEN X > 1 WITHIN 10
     SET X = 0
OTHERWISE
     MESSAGE "X <= 1 FOR 10 SECONDS"
END WHEN
```

Figure 2-2: Example portion of code that uses secondary control statements.

There are also optional secondary control statements BEFORE and WITHIN. These are used to break out of the condition check if the primary conditional does not evaluate to true before the secondary conditional. For example, the statement

WHEN TEMPERATURE > 50 WITHIN 60

will only wait for 60 seconds to see if the temperature reaches the threshold. If that happens within 60 seconds, execution immediately proceeds within the loop. If not, execution proceeds outside the loop.

Finally, there is also an optional OTHERWISE keyword. This keyword declares a block of statements to execute only if either BEFORE or WITHIN was triggered. OTHERWISE blocks must be declared just before the END statement for a loop that contains a secondary control statement. For example, the code shown in Figure 2-2 will check the value of X for 10 seconds. If during that time the value is greater than 1, execution proceeds to the SET statement. If ten seconds elapses, the MESSAGE command is executed. In either case, execution then proceeds after the end of the loop.

Timeliner also has IF, ELSE, and ELSEIF statements. These control statements execute in a more traditional manner: the condition is checked once, which immediately determines which block of code to execute next.

Each of these constructs will be described in greater detail.

WHEN statement

If the conditional is true, execution immediately proceeds to the statements within the code block. Otherwise, execution pauses at the conditional statement until it evaluates to true. After the statements within the WHEN code block are executed, execution proceeds to the statements following the WHEN block.

This statement is not a loop at all, other than waiting for the conditional itself. In fact, in the absence of any secondary conditionals, it does not matter whether the statements that follow it are within the WHEN block or below the WHEN block. Once the conditional evaluates to true, every statement below the WHEN block is executed linearly.

WHENEVER statement

A WHENEVER statement is a loop version of the WHEN statement. Execution pauses until the condition evaluates to true, just as with the when statement. However, after executing the statements within the WHENEVER code block, execution returns to the primary conditional. This is an "edge-triggered" conditional, meaning that the conditional must evaluate to false at least once in between two successive iterations of the loop.

If there is no secondary conditional, the WHENEVER statement and its code block constitutes an infinite loop. Any statement below this loop is unreachable.

EVERY statement

The EVERY statement is also a loop, but the condition must be a unit of time instead of a boolean result. For example, "EVERY 5" executes the statements within the loop every five seconds. Time spent within the loop will count towards the next iteration: that is, if the loop takes four seconds to run, execution will only wait one additional second in between successive loop iterations. If the example loop takes six seconds to run, there won't be any wait time between successive loop iterations.

If the primary condition of an EVERY loop is not a constant, the value is only calculated when the EVERY statement is first executed. The same value is stored and reused for successive iterations.

BEFORE statement

BEFORE statements are used to break execution when the secondary conditional becomes true. For the WHEN loop, this only applies if the secondary condition evaluates to true before the primary condition. However, the WHENEVER and EVERY loops repeat indefinitely. The BEFORE check happens immediately before the primary condition is checked. Therefore, if both would evaluate to true at the same time, the BEFORE condition takes precedence and the loop will break.

If there is an OTHERWISE statement associated with the loop, it will be executed any time the condition of the BEFORE statement triggers.

WITHIN statement

This is similar to the BEFORE statement, except that the secondary condition here is a unit of time. The WITHIN statement is very similar to the EVERY statement in that the unit of time is only calculated the first time the condition is evaluated. That same value is used every subsequent iteration of the loop.

Time is counted from the first execution of the condition. For example, consider the statement:

```
WHENEVER X > 1 WITHIN 10
```

Execution will break ten seconds after initial execution, regardless of what value X was during those ten seconds. If ten seconds is reached during execution of statements within the loop the remainder of the statements will finish executing before execution breaks out of the loop. Note that there is no way to break the execution of statements within a loop early.

2.3 The Compiler

The compiler provides several key functionalities for the Timeliner system. The primary and most straightforward functionality is converting an ASCII text file into a compiled file representing the abstract syntax tree. The compiler executes in the

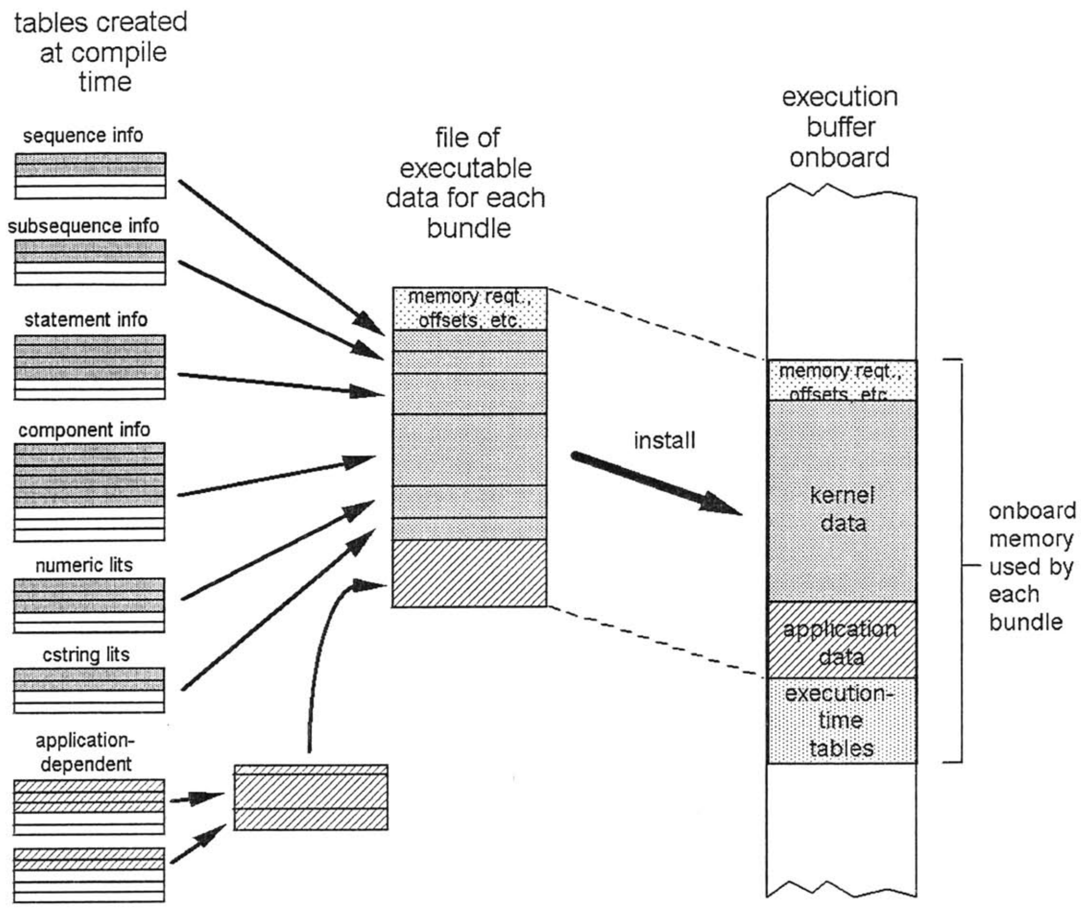

Figure 2-3: Table describing output file for script in Figure 2-1

ground control system and the output files are transmitted to the executor, which is located on the flight system.

There are several key steps involved in compiling a Timeliner script. In order to integrate with the target system the compiler must read a Ground Database (GDB) file which represents the variables and commands specific to the target environment. The compiler then performs syntax checks, reports any errors, and creates two output files. First is the compiled script. Note that this must be interpreted by the executor; it is not machine code, and so it cannot be run natively on any system. The structure of this file is described by the tables in Figure 2-3. The second file generated is a listing file for the mission control operator. The listing file that corresponds to the script in Figure 2-1 is shown in Figure 2-4.

```
—— TIMELINER EXECUTABLE DATA FOR BUNDLE 'SCRIPT' IS AS FOLLOWS:

——  BLOCK/STATEMENT/COMPONENT INFORMATION AS FILED:
——
——  block  stat  comp  type             dat1 dat2 dat3 dat4 dat5
——  ----   ---   ---   -------------    ---  ---  ---  ---  ---
——  bundle  1     1    BUNDLE_STATEMENT      1    8    1    6    7
——                           13
——            8    NUM_INT_VAR        1    8    8    1
——        2   13   DECLARE_STATEMENT     8
——  seq 1   3   15   SEQ_STATEMENT       3    7   14   16   1
——                           0    1    0
——           24    NUM_NTGR_LIT        1
——           26    BOOLEAN_COMBO         1    8    6   24
——        4   31   WHEN_STATEMENT        6    0   26
——           35    NUM_NTGR_LIT        2
——        5   37   SET_STATEMENT       8   35
——        6   40   END_STATEMENT        4
——        7   42   CLOSE_STATEMENT       3
——        8   44   CLOSE_STATEMENT       1

——  NUMERIC LITERALS:
——
——  1   1.00000000000000E+00  0.00000000000000E+00

——  CHARACTER LITERALS:
——
——  1   SCRIPTEXAMPLEONE
```

Figure 2-4: List file for script in Figure 2-1

As shown in Figure 2-3, the compiled script is composed of several tables. Each table describes a certain, key element for the target script. The first tables describe the sequence and subsequences, which are the highest level within a bundle. Next, the statements are described. Each sequences and subsequence has pointers to this table, since sequences are composed of statements. Following this is the component information which is likewise connected to the elements in the statement table. Finally, some components can be subdivided into literals, represented in the final tables.

In the Next Generation Timeliner system, each of these elements are known as Keyword Handlers. Instead of being represented in a table, from highest level to lowest level, they are represented in a tree structure. This is covered in greater detail in the next chapter.

The list file, shown in Figure 2-4, is much simpler. This is designed to be read by the human operators. Messages, warnings and errors from the Timeliner system are reported as statement numbers, shown in the second column of the figure. The remaining information ties each statement to its associated entries in the tables from

Figure 2-3.

2.4 The Executor

The executor in the Timeliner system interprets and processes these scripts. Compiled bundles are sent from ground control to the executor, which is located on the target system. The executor parses these files and recreates the abstract syntax tree. Keeping track of which sequences and bundles are active, the executor schedules the various active tasks and performs the action of the scripts. The executor must provide values for the system variables and perform the correct action for a system command. The executor must also keep track of the timing of the system since that is such a critical component of the Timeliner system.

While performing these operations the executor sends messages back to ground control in human readable format. This allows the operators to inspect the system and insure that everything is operating as expected. Generally this is just to insure there is no human error such as not setting a sequence to "active" status.

A representation of the executor can be seen in Figure 2-5. The "Development Environment" box on the left represents the compiler, which produces two output files. The list file is sent to the mission control operator, represented by the oval on the bottom. The executable data file is sent to the Timeliner executor itself, generally located on the target system. Each of the operations of the executor is listed in the various boxes in the center. The ovals on the right represent the target system that the executor is interfacing with.

The functionality shown in Figure 2-5 is implemented in the Next Generation Timeliner. The details of this implementation, including the differences between the two Timeliner systems, are described in detail in the subsequent chapter.

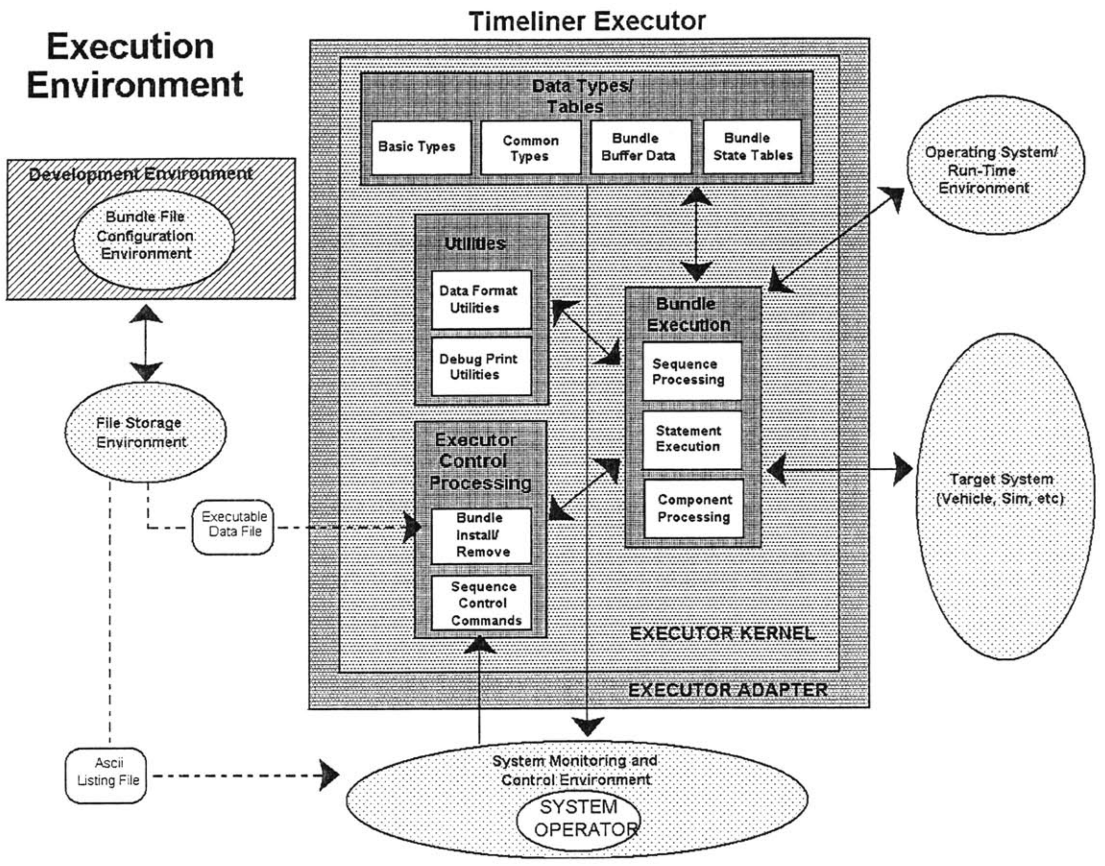

Figure 2-5: Visual representation of the executor components

Chapter 3

Executor Implementation

The primary goal for this project is to design and implement a Timeliner executor in the Java language. This executor, once finished and fully tested, will be part of the Next Generation Timeliner, which will replace the current Ada Timeliner system. In particular, this component must implement all of the functionality seen within the "Executor Kernel" of Figure 2-5.

There are several key goals for this system. First, the system must be very modular. There should be very little modification necessary to add or change the functionality of a command in the Timeliner User Interface Language. The scheduling algorithm should also be a separate module that can easily be changed or upgraded as the need arises. Furthermore, the system must be very easy to test, both for unit tests and for integration tests, since reliability is a paramount concern.

The Next Generation Timeliner is comprised of two primary components: the compiler and the executor. The compiler began development at the Charles Stark Draper Laboratory in 2001, also as a thesis project [2]. The compiler already brings the Timeliner system away from what was described in the previous chapter. The most significant change is that the compiler generates a single output file in XML format, representing the abstract syntax tree of the input script.

The Executor module builds on top of this compiler by providing a system that reads this XML file and executes the script represented by this abstract syntax tree. When a compiled script is received by the executor it is first passed to the "Keyword

Factory". This component generates a set of concrete classes representing each node in the abstract syntax tree of the script. Each of these concrete classes meets the "Keyword Handler" interface definition. The scheduler then decides which order to execute these Keyword Handlers. The entire system is mediated by the Executor module, which handles commands and messages sent between sequences, or between the human operator and a sequence.

The following sections describe each of these components. First, the XML file structure is explained. This leads to the Keyword Factory's implementation to parse this file and create Keyword Handlers, which are also explained in detail. The executor, which ties each of these components together, is then explained. This is followed by a description of the scheduler module, a part of the execution module. Each of these modules together comprise the executor for the Next Generation Timeliner.

3.1 XML Structure

Before any discussion of the executor could begin, a brief overview to the structure of the XML file is necessary. The XML file acts as a tree structure with many areas that are handled as "black boxes". For example, take the XML tag for a sequence. There is some typical data specific for a sequence, as shown in the first few lines of Figure 3-1. However, the tag <DeclStatementList> begins a list of arbitrary statements that appear within the sequence. Furthermore, by examining the full XML code in Appendix A, there are further instances of this for the condition of the WHEN statement and the statements within the WHEN block.

This clearly suggests a modular approach to the problem of parsing and executing this data. For example, the command:

```
SET X = 0
```

could appear both as a child of the <CondStatement>, as it does for the code represented in Appendix A, or it could just as easily appear as a child within the <DeclStatementList> tag shown in Figure 3-1. The same executor code should be

```
<SeqDecl>
     <line>3</line>
     <id>ONE</id>
     <status>INACTIVE</status>
     <DeclStatementList>
          <CondStatement>
               <line>4</line>
               <condWord>WHEN</condWord>
               ...
          </CondStatement>
     </DeclStatementList>
</SeqDecl>
```

Figure 3-1: XML data for the Sequence declaration seen in Figure 2-1.

running in either instance, suggesting that the code for a SET statement should be completely independent of its parent. With very few exceptions, any Timeliner statement could appear in many different places just as with the SET statement, and so every Timeliner statement should be independent of its parent.

Furthermore, the SET statement above is rather simplistic, but it could be more complicated, such as:

SET X = (Y + Z) / 2

In such a case, the arithmetic on the right side of the equation should already be agnostic of its parent, as was explained in the previous paragraph. That implies the SET statement should also be agnostic of its child. Any series of operations that return an integer value are acceptable children.

This description leads to the design of the Keyword Handlers, described in Section 3.3. Every component is an independent Keyword Handler. In the above example, the highest level handler is for the sequence. It has one child – the WHEN keyword handler. It, in turn, has two children: the conditional, and the SET statement. Finally, the SET statement has a child for the expression to evaluate for the assignment.

In the case of the original example in Figure 2-1, the expression to evaluate is just a constant, which itself is a very simple Keyword Handler. In the case of the more complex SET statement described earlier, there is a further tree of Keyword Handlers. The SET statement has a reference to the highest element in the arithmetic, which is the last expression to evaluate. In this case, it is the division operator. The division operator has two Keyword Handler children: the numerator and denominator. In this case, the denominator is a constant Keyword Handler, and the numerator is an addition Keyword Handler. The addition handler also has two children, each of which are Keyword Handlers that represent variable values.

In this way, the tree structure of the XML file is converted to the tree structure of Keyword Handlers. The details of this conversion are explained in the subsequent sections of this chapter.

3.2 Keyword Factory

In order to meet the goal of modularity, a factory design pattern is the most logical choice. This factory must create the varied functionality that can be found in a Timeliner script, yet still present it to the system in a generic way. In particular, the Keyword Factory returns objects that meet the interface "Keyword Handler". This meets the goal of modularity in two ways. First, every other component of the system, including other Keyword Handlers, must only deal with the very simple interface of a Keyword Handler. Second, if an element of the Timeliner UIL is modified, the associated Keyword Factory is the only component of the executor that must be modified to understand the change.

3.2.1 Keyword Factory Operation

The Keyword Factory operates in a loop, reading one element from the compiled XML file each iteration of the loop. A stack is also maintained of all the Keyword Handlers currently being processed.

For the first iteration, the stack is empty and so the Keyword Factory expects the next XML tag encountered to create a new Keyword Handler. The factory maintains a mapping of XML tags to concrete object names, allowing the factory to determine which Keyword Handler to create. Only XML tags that begin a new Keyword Handler are stored in this mapping.

For the second iteration the stack is not empty – the Keyword Handler created in the first iteration is still on the stack. The factory will know when this handler is finished since closing XML tags must match the opening XML tag. When the factory receives the next XML tag, it doesn't use the lookup table. Instead, the factory asks the Keyword Handler at the top of the stack if it recognizes this tag.

If the Keyword Handler returns true, it receives the tag for processing and the factory continues to the next tag. If the Keyword Handler returns false, the next tag must be the opening tag to a child Keyword Handler. The factory goes back to its lookup table and creates a new Keyword Handler to place on the stack.

Again, the factory knows when a Keyword Handler on the stack is finished when it encounters a closing XML tag that matches the tag that originally created the handler. A handler is popped off the stack when it has finished. Then, a reference to the newly completed handler is passed to the Keyword Handler that is next highest on the stack. In this manner, Keyword Handlers are always passed a reference to any child handlers they may have.

3.2.2 Adding Keyword Handlers to the Keyword Factory

This design could potentially allow for an operator to add or change Timeliner statements while the executor is running. The data table used by the Keyword Factory could easily be made mutable through commands from the mission control operator. In order to create a new Timeliner statement, the following steps are necessary. First, either the compiler must be modified to understand the new statement, or the new statement could be added to an existing XML file "by hand". Second, a new object must be written for this statement that meets the Keyword Handler interface. This object must be compiled and loaded into the executor system. Finally, the data table in the Keyword Factory must be modified with the beginning XML tag and concrete class name of the new Timeliner statement.

Of course, modifying the data table during mission operation is a potentially risky change. This would not be performed by NASA, but future users of the Timeliner system could potentially have use for this "hot-swap" functionality. However, the simplicity in modifying the Keyword Factory to understand new Keyword Handlers also extends to more controlled additions of Keyword Handlers.

3.3 Keyword Handler

The Keyword Handler interface is designed primarily to allow communication between different Keyword Handler implementations. For example, take the keyword handlers needed to execute the script in Figure 2-1. The WHEN statement has an associated keyword handler. This handler must communicate with the keyword handler for the greater-than operation in order to examine the result of the inequality. Next, there must be communication between the WHEN handler and the SET Keyword Handler so the latter knows when to begin execution.

The most significant methods implemented in Keyword Handlers fall into two categories: creation and execution. The creation methods are used by the Keyword Factory on a bundle install and are never called again once the bundle has finished installing. Conversely, the methods in the execution category cannot be called during a bundle install. In addition to these two categories, some Keyword Handler implementations also implement the Bundle, Sequence, Declare or Returnable interfaces. Each of these categories will be explained in further detail in this section.

3.3.1 Special Keyword Handlers

Bundle and Sequence

The handlers for a sequence and bundle are a special case. There are circumstances in which each of these are different from every other keyword handler, as well as different from each other. Specifically, there are commands that can come from either the human operator or other scripts which are unique to either a bundle or a sequence. Examples of these include starting, stopping, or pausing a sequence, or uninstalling or halting a bundle. Only keyword handlers that also extend one of these two interfaces will have methods for accepting these special commands.

Furthermore, most modules will not make use of these interfaces. In particular, only the Executor and Scheduler modules will make use of the Bundle interface. Those same two modules, plus any Bundle implementation, will make use of the Sequence interface. Therefore, these two interfaces can be ignored when dealing with interactions between all other keyword handlers. In particular, the commands mentioned above are implemented as a Command Keyword Handler. Even this handler does not need to know these interfaces; It uses the executor as the intermediary between it and the target sequence or bundle.

Returnable

A more interesting special instance of keyword handlers is the Returnable interface. This interface allows for a keyword handler to pass a value along to another keyword handler. For example, the Addition Keyword Handler must pass a value along to another keyword handler to return the result of its execution. Similarly, the Addition Keyword Handler must be connected to two other keyword handlers that both implement the Returnable interface. Through this interface, it can know what values to add together.

Certain keyword handlers know about the Returnable interface. In each of these situations it is a parent-child relationship. While the Keyword Factory is creating these keyword handlers the parent has the ability to insure some or all of its children are Returnable, and an error can be raised if this condition is not met. However, such an error should never arise since it would imply an error in either the compiler or the Keyword Factory.

An example of a keyword handler which requires one child to be Returnable is the WHEN statement. Typically, only one of this statement's many children must be Returnable. The children in the code block do not necessarily have to be Returnable. The XML tags will specify which child this is. In the case of a second conditional, as described in Section 2.2.3, there will be two children that must be Returnable and both will have appropriate XML tags. The Addition Keyword Handler is an example of a handler in which every child must be Returnable.

Declare

The Declare interface is a subset of the Returnable interface. The Declare interface is used for variables, which necessarily must also be Returnable. When a keyword handler must read the value of a variable, it simply views it as a Returnable object. However, with a keyword handler such as SET, it must also write the value which happens through the Declare interface.

As with the Returnable interface described previously, a keyword handler can insure its child is a Declare upon construction if such a condition is necessary.

The Executor and the Environment modules also handle objects at the Declare level. The decision to create a new interface for this, instead of just using Returnable, had to do with future development. When dealing with values declared by the underlying system that Timeliner is interfacing with, added complexities can arise. These complexities can be handled by the Declare interface without unnecessarily complicating the simple Returnable interface. Many keyword handlers implement the Returnable interface, while very few also implement Declare.

3.3.2 Creating Keyword Handlers

The primary creation methods of a Keyword Handler implementation are "processTag" and "childListener". There are of course other minor methods, such as "setLineNumber". This section will step through the creation of two Keyword Handlers to illustrate how they are created and linked together.

Initial XML tag

The Keyword Factory must know the initial tag that creates a new Keyword Handler, but with the help of the processTag method the factory doesn't need any additional information. For example, the factory must know that the XML tag <CondStatement>

refers to a concrete class for a conditional. However, the factory does not need to know about the tags <Condition>, <CodeBlock>, and over fifteen other tags used within a conditional.

See Section 3.1 for a more detailed description of the structure of the XML file.

Instead, the Keyword Factory retains a stack of keyword handlers currently being created. When the tag <CodeBlock> appears, the factory calls the processTag method of the keyword handler at the top of the stack. This method returns a boolean value indicating whether the tag was accepted or rejected. In this example, the conditional handler will accept the tag as its own. The factory then processes the next XML tag.

Creating children of current Keyword Handler

Inside the <CodeBlock> tag there will be other keyword handlers. For the script in Figure 2-1, a <Set> tag will appear. The Keyword Factory passes this tag to the keyword handler at the top of the stack, which is still the When Keyword Handler. The tag will be rejected, causing the Keyword Factory to try and create a new keyword handler from this tag. Note that only when the handler at the top of the stack rejects a tag (or the stack is empty) will the factory try to create a new handler. This solves the potential ambiguity of two handlers having identical tag names: one as the initial tag, and the other as an inner tag. The following section covers this ambiguity in greater detail.

This is where the tree structure comes in, as opposed to a linear creation of keyword handlers. The When Keyword Handler will first have a conditional, indicated by the <CondStatement> tag. Within this tag, another keyword handler will be created and will finish. As stated in Section 3.2.1, the Keyword Factory knows when a keyword handler has finished. The factory remembers the start tag and so the handler is complete when the corresponding XML close tag is reached. After this, the factory knows to give the next XML tag to the parent keyword handler. These XML tags separate each of the children that the target handler may have. Therefore, the When Keyword Handler always knows if the child passed by childListener is a conditional, or the next statement in the code block.

Potential ambiguities

There is yet another potential ambiguity which must be taken into consideration. It is possible to have nested keyword handlers, such that one instance of a particular concrete handler is the parent of a different instance of the same concrete handler. For example, a WHEN statement can appear within the code block of another WHEN statement. If the When Keyword Handler were to always accept <CondStatement> as its own, the second WHEN statement will not begin a new keyword handler. Therefore, special care must be taken for opening XML tags. The When Keyword Handler must reject the second occurrence of this tag since only the first instance is legal for that particular keyword handler.

A second ambiguity, which was mentioned previously, is the possibility of one handler's creation tag matching the inner tag of another handler. In such a case, the keyword handler which has this tag as an inner tag must take care that it does not mistakenly accept this tag when it is meant to be a creation tag for another handler.

The XML structure is designed such that this ambiguity is never a fundamental ambiguity. That is, there will never be two different but correct ways in which to parse a compiled XML file. This is because there is an XML tag to indicate every time a handler must create a child. Whenever this XML tag was the most recently accepted tag, the handler must reject all other tags. The only valid tag would be a close XML tag, indicating the child has completed its creation.

Note that it is possible to handle the first ambiguity in the Keyword Factory; if the next tag to be processed matches the creation tag of the keyword handler at the top of the stack, automatically begin a new keyword handler. However, this alternative was not taken because it does not solve the second ambiguity. Furthermore, any steps taken by the developer to correctly handle the second ambiguity will also satisfy the conditions of the first ambiguity. The first is easier to describe with an example, though the second ambiguity is a generalization of the first ambiguity. Keyword handler developers must take this issue into consideration regardless.

Data values

The Keyword Factory will also pass XML data items through the processTag method. It is the responsibility of the keyword handler to remember the previous XML tag that this data corresponds to. However, the Abstract Keyword Handler described in Section 3.3.4 creates a cleaner interface for this issue. Data items are always passed to the keyword handler at the top of the stack. This is correct since the Keyword Factory invariant states that the handler at the top of the stack either accepted the previous XML tag, or was created by the previous XML tag.

Going back to Figure 2-1, during the creation of the Set Keyword Handler, it will receive a data value indicating that the variable to set is the string "X".

Completing Keyword Handlers

The Keyword Factory can always tell when the keyword handler at the top of the stack has completed. Since the Keyword Factory knows what tag created this object, it can infer the XML tag that will complete it. The factory pops the handler off the stack when it has completed. Then, the Keyword Factory passes a pointer to the newly created object to the handler that is next on the stack using the childListener method.

For example, after encountering the <Set> XML tag, the factory knows to wait for a </Set> XML tag. The current handler can be finished when this tag is reached. A pointer to this completed Set Keyword Handler is given to the When Keyword Handler, which is now at the top of the stack again. The When Keyword Handler must keep a pointer to each of its children so it can execute them when appropriate.

In this way the original abstract syntax tree is again rebuilt. This time, however, each node in the tree has specific methods that allow the executor to walk the tree and execute along the way.

3.3.3 Executing Keyword Handlers

Once the keyword handlers have been created and pieced together by childListener calls, the execution happens with the "execute" method. The details of this method will be very different for each implementation of a keyword handler since this is where the actual Timeliner functionality takes place.

For example, consider the When Keyword Handler. Upon calling the execute method, it will first check its conditional. This is done by asking for the value from one of its child keyword handlers. During creation, the handler insured this child implements the Returnable interface. If this child is a Greater-than Keyword Handler, as in Figure 2-1, it will ask for the value from its two children, both of which implement the Returnable interface. If greater-than returns true, the parent When Keyword Handler can then begin executing its list of child keyword handlers. Upon completion of the block of child statements, the When Keyword Handler will complete its execution. This execution began because a parent, such as a Sequence Keyword Handler, called the execute method of the When Keyword Handler. The parent can then continue its own execution, or pass execution on to another one of its children.

If the Greater-than Keyword Handler returns false, the When Keyword Handler must pause execution here. This is done by returning a special value to its parent that indicates execution must pause until the next time slice. This special value is passed up the tree, back to the scheduler, which will then know that the remaining time slice for this execution can be used elsewhere. The following time slice, each node in the tree knows to give execution back to the same child that had it the previous time slice. The Greater-than Keyword Handler is checked again, and the process repeats.

Keyword Handlers also receive a unit of time in the execute method. This unit of time is how long the handler is allowed to execute. Currently, the time is measured in an arbitrary unit of "steps", with each component deciding for itself how many steps it consumes as it executes. This allows for the system to be usable as a simulation environment for the current Timeliner system; each component can declare a number of steps related to the actual time it would take when executing on a particular system with the current version of Timeliner. Then the execution of this system would mimic the target system exactly. The need for an external simulation environment further shows the extended modularity offered by the Next Generation Timeliner. However, in the future, the Next Generation Timeliner will be real-time. See future work in Section 6.1.1 for a more detailed description.

Note the distinction between execute and the Returnable interface, described earlier in Section 3.3.1. The decision to separate these two methods was for added reliability. An alternate design decision would be to have no Returnable interface, and instead have "execute" return a value in some keyword handlers, or signal that it has no return value. This decision would provide flexibility in allowing keyword handlers that sometimes return a value when executed, but return nothing other times. This also simplifies the overall design.

However, this pushes a key error from process time to execution time. Specifically, if a keyword handler requires a value from one of its child keyword handlers, such as the Addition Keyword Handler, it is possible for it to not have a value at execution time. Since the Timeliner User Interface Language does not allow for such a construct, this could only result from a compiler error or an XML parse error. Reliability is a paramount concern, and so this error must be reported when the XML file is being parsed, not at execution time. Furthermore, due to the simple nature in the Timeliner UIL, it is not feasible that a construct would later be introduced that sometimes returns a value, and other times does not. Therefore, the decision to separate these two functionalities provides for greater reliability, does not hinder any functionality, and only adds a small level of complexity.

During execution, the Executor produces messages in human readable format so the operator can inspect the system. The messages sent for a sample script appear in Appendix C. This sample script is described in greater detail in Section 5.1.1.

3.3.4 Abstract Keyword Handler

There is, of course, much overlap among different keyword handlers. An Abstract Keyword Handler has been built to simplify the development of individual keyword handlers.

Primarily, much of the work in building keyword handlers has been simplified by this abstract class. Every subclass must implement a method, "acceptableTag", which returns true if the target tag is accepted by this particular keyword handler. Also, every subclass must implement "processData". These construction methods abstract away the common aspects of keyword handler construction that are integral to the XML file format, while simplifying the few details that must vary from handler to handler.

In this way, keyword handlers can be written very easily. The acceptableTag method is often implemented as a series of "or" operators, checking the target string against each of the tags that can be accepted. A more complex keyword handler can perform more work on a particular tag, such as to prevent the ambiguity described in the previous section, but most handlers only perform work in the processData method. The abstract class maintains a stack of the accepted tags and therefore can figure out for itself how to handle close tags; each implementation never sees a close tag. On the processData method, the keyword handler can call a method in the abstract class that returns the tag at the top of the stack in order to take the appropriate action.

The Abstract Keyword Handler also handles any operation that is identical across keyword handlers. For example, the "line" tags and associated data are never passed to an implementing keyword handler since it is always treated the same. The methods for reading the line number from a keyword handler are also handled in the abstract class. Creating a pointer to the Executor for command issuing is also handled at the abstract level.

3.3.5 Testing Keyword Handlers

As was described earlier, reliability is a paramount concern for the Timeliner system. Therefore, each keyword handler should be unit tested thoroughly. The modular design allows for this to be very simple. Each keyword handler has a very concise interface with other keyword handlers: they can call execute on each other, and with

Returnable Keyword Handlers they can ask for a return value. Testing keyword handlers is very straightforward with this simple interface.

To test a keyword handler, create a test module and stub keyword handlers. The stubs simply pass messages back and forth between the test module and the handler to test, allowing the test module to create any environment around the target keyword handler. The target cannot distinguish this sand-box operation from an actual execution. The test module passes values through the Returnable interface in the stubs, checks values if the target keyword handler is Returnable itself, calls execute on the target, and reads when the target calls execute on the stub keyword handlers. Through this, the test module can simulate any test case and insure the correct functionality from the target keyword handler. This allows for an automated test suite of each keyword handler.

For integration testing, a very similar scenario can be used. The keyword handlers to test should be linked together correctly. From this collection of handlers, there will be several that must interact with the outside world. Connect those to stub keyword handlers, and test just as described previously.

In fact, the converse of this type of test is also possible. Stub handlers can be written that mimic bundles or sequences. These stubs do not have any actual children; they merely pause in order to mimic the execution of an actual bundle or sequence. In this way, the Executor and Scheduler can be unit tested in any simulated environment. This can further be used for stress tests. The stub bundles and sequences can occasionally take an unusually large amount of time in order to insure the Scheduler can handle this scenario without other stub bundles experiencing resource starvation. Other scenarios, or even randomly chosen scenarios, can be tried in order to test the Executor and Scheduler in any situation.

3.4 Executor

The executor module ties the system together. For the most part, it is a very simple component. On a bundle install, the Executor just passes the XML data to the

Keyword Factory, and then retains a handle to the completed bundle. The Executor also, with the direction of the Scheduler, decides which keyword handlers should receive processing time and how much time they get. The Executor also handles any command request, either between mission operators and scripts, or from one script to another. In doing so, the Executor can perform permission checking, though the current version does not do so. Timeliner currently runs in extremely controlled environments, such as the International Space Station, and so there is no need to check permissions in order to protect against malicious scripts.

3.4.1 Scheduler

The Scheduler is a subcomponent of the Executor. Through using method calls in the Executor, the Scheduler can get a high level view of the installed system. In particular, the Scheduler can examine the bundles and sequences installed, including which bundle each sequence is in. With this information, the Scheduler decides how to divide processing time.

Currently, the system mimics the existing Timeliner scheduling algorithm. This algorithm is to divide processing time equally among bundles without regard to the number of sequences within the bundles. Each bundle is then executed in the order in which they were installed. If any bundle completes its execution early, its remaining time slice is divided evenly amongst every bundle that was installed after it; bundles that already executed and completed do not benefit from this extra time.

A discussion of future work related to improving the scheduler can be found in Section 6.1.2.

3.5 Environment

The Environment of the current Next Generation Timeliner is merely a stub. This was done for two reasons. First, an integral part of any Environment interface is the Ground Database, which the current Next Generation Timeliner compiler does not yet implement [2]. Second, the author is not familiar with the finer details of the space systems that Timeliner interfaces with. This aspect of the project is better left to full time employees at the Charles Stark Draper Laboratory.

Instead, the system was designed to make it very easy to integrate with absolutely any environment. There are stub methods which allow for reading or writing environment variables, and sending commands to the environment. This environment is general enough to provide the correct functionality in systems Timeliner currently powers, and future systems that can benefit from Timeliner.

With this abstraction layer in place, the remaining executor system could be designed and built without the need to know the details of the underlying environment.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Extending the Executor

The primary goal of this project was to create an easily extensible Executor for the Next Generation Timeliner system. To demonstrate this extensibility, the steps necessary to make two possible changes to the Timeliner User Interface Language are described in this chapter. First is the addition of priority levels for sequences in order to demonstrate changing an existing construct. Second is adding a new construct for waiting until a specific time before executing a block of code. Finally, a discussion about modifications during the execution of Timeliner follows.

For both of these constructs, it is assumed that the compiler has already been modified to accommodate these changes. A detailed description of the changes necessary for this are available in Chapter 4 of the associated thesis [2].

4.1 Sequence Priority Levels

This modification is significant in that it demonstrates how separating the compiler and executor can allow for iterative development. If the compiler has been modified to accept priority levels from sequences, the executor can still interact with the new compiler without any modification. Of course, the priority levels would not be honored since the executor hasn't been programmed accordingly, but otherwise the executor will still function correctly. A warning will be given when the XML file is parsed indicating the presence of new data that it does not understand, but otherwise, execution will proceed as if the compiler had not changed at all.

In order to make this change, the first component that must be changed is the Sequence Keyword Handler. First, the interface must be modified so there is a way to read the sequence's priority level. Second, the acceptableTag method, described in Section 3.3.4, must be modified to accept the new priority tag. Then, the processData method must be modified to store the data found in this tag. Each of these changes are very simple and would require very few lines of code.

Finally, either the Scheduler or the Bundle Keyword Handlers must be modified, depending on the functionality desired. If priority levels are to apply system wide, the Scheduler should be modified to give time slices to the sequences directly, instead of giving priority to bundles for them to allocate to their own sequences. Otherwise, if priority levels apply only within a bundle, the Bundle Keyword Handler must be modified to give time to its own sequences according to priority. In either case, very few lines of code would need to be modified for this functionality.

4.2 New SWITCH Keyword Handler

Currently, Timeliner does not have a construct for the switch statement. The first modification for this is to write the keyword handler itself. It would be a straightforward extension of the Abstract Keyword Handler described in Section 3.3.4, and so the methods described there would have to be implemented.

Next, it must implement the childListener method, described in Section 3.3.2. The compiler must generate an XML tag which defines each of the three types of children a switch statement can have: the expression to check, possible values for the expression, and statements to execute if the expression evaluates to the target value. There also must be a tag that defines the "default" statements to execute if no other value is triggered. The childListener method can call a method in the abstract class that indicates which tag is at the top of the stack, letting it know how to handle the newly created child. For the expression and the target values, the method must insure that the child passed implements the Returnable interface. If it does not, an error must be raised. The handler must maintain a mapping of values to a list of statements, plus a special list for the default list of statements if any should exist.

Finally, the handler must implement the "execute" method. This method will first ask for the value of the expression through the Returnable interface. Next, the handler must go through each mapping it has stored and ask for the current value through the Returnable interface. There are possibilities for optimizations here if it were possible for a Returnable Keyword Handler to say if it has a constant value; currently, there is no such functionality. When a value is found, or if a default value exists and no other value matches, the handler has a list of associated statements for that condition. The handler must pass execution to each of those statements, followed by returning execution to its own parent.

Note that during each step of this execution, care must be made to not exceed the time given for execution. The way in which this can be done varies based on how the time is counted: steps, actual time, or something else. In a possible future modification, this won't be necessary at all by making Timeliner multi-threaded, and allowing the Java Virtual Machine to handle timings with the guidance of the Executor.

Once the Switch Keyword Handler has been implemented, the Keyword Factory must be modified to understand the opening XML tag for this statement, and to create the appropriate object. Currently, that is two lines of code per keyword handler.

No other modifications must be made, as every other keyword handler will operate fine with this new keyword handler. This shows the power of modularity and using the factory design pattern: a new component can be created, and only two lines of code in the existing system needed to be altered to introduce the new component.

4.3 Modifications During Execution

The modularity described here allows for even more possibilities. Particularly, it is possible to modify modules during execution and have the modifications take effect immediately. This was not done yet in the project since the National Aeronautics and Space Administration (NASA), the primary user of Timeliner, would never take advantage of this functionality as it introduces some level of risk to the system. However, Timeliner is being marketed for other systems as well, and future users could potentially want this functionality.

Particularly, it would be a simple change to have the Executor have a method that allows for changing the scheduling module during execution. Once the current time slice has been finished by the current scheduler, it will be unloaded, the new scheduler will be loaded, and it will take over for the next time slice.

Similarly, it would be simple to have the Keyword Factory provide a method for introducing new keyword handlers and modifying existing keyword handlers. The two lines of code that map an XML tag to a keyword handler object can be in a simple data structure that is modifiable by this new method.

Of course, modifying an existing Timeliner statement will not effect the bundles currently loaded. Since those were created by the Keyword Factory before it was modified, they still use the previous keyword handler. Bundles loaded after this change will make use of the new functionality. However, if modifying the functionality of the existing bundles during execution was a high priority, it is still possible with a little additional work. It is still possible to have the Keyword Factory create the new version of the Timeliner statement by passing it just the applicable piece of the XML file. Then, the Executor must find the target keyword handler and replace it with the new keyword handler by relinking the parents and children. The complexity, however, is in finding the keyword handler, and knowing the details of it enough to re-link the existing objects. To make this cleaner and easier, modifications would have to be made to the Keyword Factory interface.

Chapter 5

Results

The goal of this project was to create a modular design and implementation of an executor for the Timeliner User Interface Language. Correctness is one clear measure of success. Other measures of success are modularity, extensibility, and ease of testing.

Note that efficiency is not a metric used here. Creating a complete executor is a huge project. In the time frame allowed for this system, a finished product was not feasible. In discussions with the staff at the Charles Stark Draper Laboratory, it was determined that the primary goals were the design itself, and enough of an implementation to execute the various sample scripts that the Charles Stark Draper Laboratory has for testing and demonstrations. Insuring the keyword handlers were highly efficient would have restricted how many could be implemented. This is just the beginning of a long term project, and so efficiency will be revisited at a later date.

This project meets all of its goals. The Executor correctly runs various test simulations available at the Charles Stark Draper Laboratory, as well as other test suites created specifically for the Executor and associated keyword handlers. The modularity of the system clearly allows for extensibility and ease of testing. Some of these topics have been covered previously since these ideals are so integral to the design of the system. The remaining results are described in the remainder of this chapter.

5.1 Correctness

Throughout development of the Next Generation Timeliner Executor, testing was a regular concern. Many scripts were written to test the components of the Executor as well as the various keyword handlers. These tests checked the basic functionality as well as boundary cases. A test driver was written which would automate these tests, allowing for them to be run periodically as new changes are made to insure completed components are not adversely affected by newer changes. However, due to the highly modular design of this system, the addition of one component very rarely affected any other component.

There are also various scripts available at the Charles Stark Draper Laboratory that are used for testing and demonstration. The Next Generation Timeliner system has been used to run these scripts with great success. First the scripts were compiled with the Next Generation Timeliner's Java compiler. The resulting XML files were loaded into the executor and the execution was monitored appropriately to insure correct functionality. Of course, the way in which to monitor a script varies greatly from script to script, depending on functionality. An example of such a script, and the associated tests for correctness, is described below. The Timeliner code for this script is in Appendix B.

5.1.1 Plant Simulator

The Plant Simulator set of scripts shows an example of how Timeliner can be used to control a greenhouse environment. The system provides certain variables, such as carbon dioxide and oxygen levels, pH and nutrient readings, and a handful of other values. These variables are referenced and several pumps are switched on or off depending on the levels. Furthermore, the pumps themselves are monitored: if they don't turn on when the command is given, a warning is given to the operator.

What makes this script especially useful is an accompanying graphical representation of the system. This program has a very simple interface for reading the values through Timeliner. Previously, it would be used with the existing version of Time-

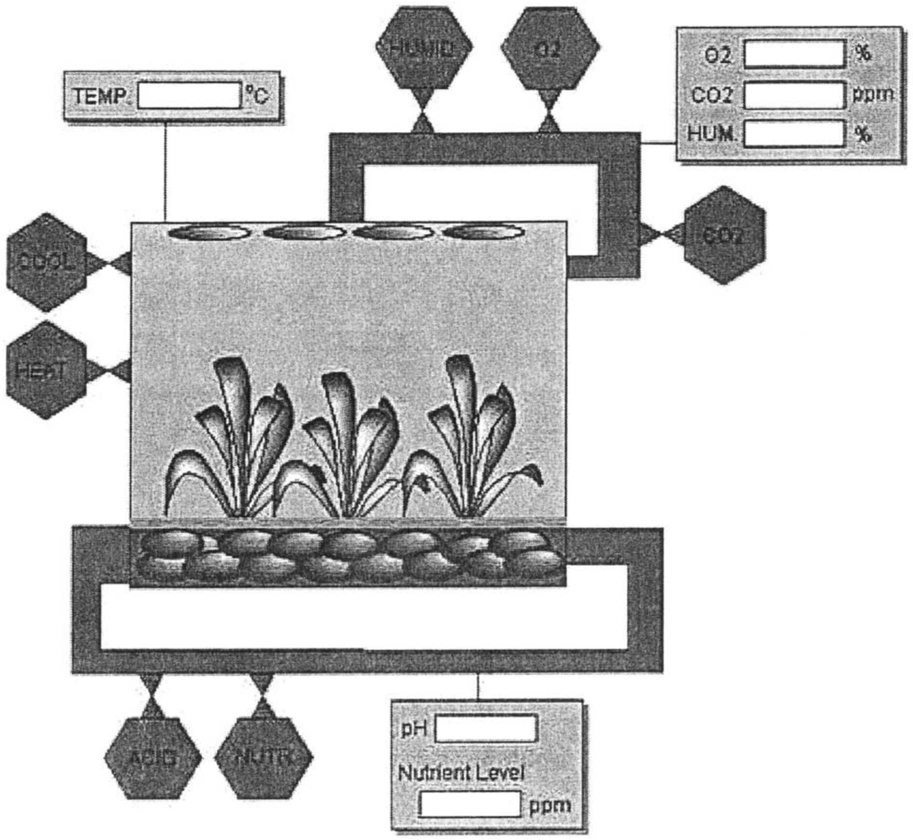

Figure 5-1: Graphical representation of the Plant Simulator environment.

liner, though with a few modifications it can now integrate with the Next Generation Timeliner.

The graphical representation is shown in Figure 5-1. Each of the hexagons represents the status of a pump. A gray hexagon represents a pump that is off, while a pump that is on will have its hexagon in color. Each of the elements monitored also has its value shown directly to insure that the pumps are turning on and off when expected.

Of course, without an actual greenhouse, this set of scripts is useless. In order to use the full functionality of these tests, a new set of scripts were written to augment the Plant Simulator. First, a "greenhouse" script was created, which acts as the pumps themselves. For example, when the carbon dioxide pump is turned on this script will increase the value for the variable representing the carbon dioxide level.

Second, a "plant" script was created which consumes the resources in the system. Periodically, the plant would consume a little of its resources, such as depleting the levels of nutrient and carbon dioxide while increasing the level of oxygen.

When these Bundles are all loaded into the Next Generation Timeliner system, they execute flawlessly. The graphics program shows the level of the various values moving accordingly and shows the status of the various pumps by having them appear in color or in gray. An artificial pause was introduced in order to watch the system and judge its correctness, but it operates identically without the pause. For example, the carbon dioxide level will slowly decrease while the pump is gray. Once the level reaches the threshold for turning on the pump, the graphical pump will become colored and the carbon dioxide levels will quickly increase. If the artificial pause is made long enough, it is possible to see the carbon dioxide level increase a large jump, then decrease a little. This shows that while the pump is increasing the level the plant is still slowly decreasing the level.

This system was left running for over 24 hours for a longevity test. While it was impractical to examine it constantly during that time, at the conclusion of the test the system was still operating exactly as it had at the beginning. No warnings or error messages were generated. A sample of the Executor output while running these bundles is in Appendix C.

5.2 Modularity and Extensibility

The system clearly is extremely modular. The test system, described in Section 3.3.5, is a primary example of the executor's modularity. As a comparison, unit testing of individual statements is not possible in the previous version of Timeliner. Each part of the Timeliner system can only be tested by testing the system as a whole. Furthermore, each module described in Chapter 3 can be tested in such a way. The system is divided into very clearly defined modules, each with a very simple and straightforward interface to the other modules. This design allows for a test system to connect to a module. Once connected, the test system provides correct data through this interface and reads the results passed back. With such a simple interface, writing the tests and insuring correctness is also very simple and straightforward.

Furthermore, the ability to "hot-swap" components, as described in Section 4.3, shows the benefits of a highly modular design. The existing Timeliner design is extremely procedural and tightly integrated. The ability to modify any part while an existing part is executing is simply not possible.

Finally, a third measure of success here is Chapter 4. As a comparison, when implementing a new Timeliner statement in the existing system, it took several months of developer time to write, integrate, and test the new statement. With the Next Generation Timeliner, this time can be cut down by an order of magnitude due to the modularity of the system.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Discussion

The goal of this project was to create the design, framework, and initial implementation of an extensible, object-oriented executor for the Timeliner User Interface Language.

The nodes in the abstract syntax tree of a Timeliner script are each a separate object. Each of these nodes implements the Keyword Handler interface. Part of this interface requires that the node can execute on its own, interacting with only its parent and children in the abstract syntax tree. These objects provide a highly extensible, object-oriented implementation of the Next Generation Timeliner Executor.

Two example extensions to the Timeliner UIL have been described, along with the steps necessary to implement those changes. Each of these changes were very simple in nature, and required minimal modifications to the existing system. This further demonstrates the extensible nature of this system.

Most importantly, the Next Generation Timeliner system can correctly compile and execute a set of scripts representing a hypothetical target system. While this project can immediately be used in a simulation environment, there is still some existing work that must be done on this system before it can replace the existing version of Timeliner. The majority of this work is testing and validation; NASA has an extremely rigorous software reliability standard. Once these changes are made and testing is complete, users of Timeliner will be able to make use of the many improvements made possible by this new system.

The remainder of this chapter describes the next steps needed to make the Next Generation Timeliner system ready to replace the existing Timeliner system.

6.1 Future Work

There is still some work to be done on this system. The remaining tasks necessary are making the system real time, improving the scheduling algorithms, and replacing the environment stub with an actual target system's environment. In addition to this, a full test suite must be written to insure the reliability of this new system.

6.1.1 How Time is Determined

The final version of Timeliner must keep track of time in a very accurate manner. Currently, as described in Section 3.3.3, time is counted in an arbitrary unit of steps. This was done in order to simplify the development and testing of this new framework. However, this must be changed to actual time elapsed before the Next Generation Timeliner can be used in the systems that the previous version of Timeliner currently powers.

This isn't a trivial change either. One solution is to introduce multi-threaded environments. However, this must be done with care. There is much research on the topic of embedded, real time Java systems. This research must be consulted when making this change to insure that Timeliner scripts are executed at the time they are expected to execute [4]. In particular, overhead concerning the Java Virtual Machine, including the automated garbage collector, must be taken into consideration.

6.1.2 Scheduling Algorithm

The current scheduling algorithm, described in Section 3.4.1, is not optimal for certain situations. In addition to these problems, a priority system is an ideal future step to help Timeliner script developers better control which sequences receive extra execution time. A rough outline of the steps required to introduce this priority system is given in Section 4.1.

A priority system would also help in another situation. As described in Section 2.2.2, subsequences can be declared single-execution only. There could be a scenario where a critical subsequence must be declared as such, and is called by many sequences. The high priority, emergency sequences should receive priority in executing the subsequence. In the most extreme cases, it may be beneficial to roll-back the existing execution of the subsequence so that the highest priority sequence can immediately take control of the execution. Each of these scenarios can be implemented within the design and framework of the Next Generation Timeliner.

Even without a priority system, the existing scheduling algorithm may not be optimal. One example is a system with two bundles, one with very many sequences, and the other with very few sequences. Each sequence in the former bundle will have significantly less execution time than each sequence in the latter bundle. This is because execution time is divided evenly at the bundle level.

One trivial solution to this problem is to divide time at the sequence level. However, for certain uses of Timeliner, there is potential for much smarter scheduling algorithms. This design provides a framework so that a developer can create a new scheduling algorithm and easily integrate this new component with the rest of the system.

6.1.3 Environment

The environment is currently implemented as a stub. This was done for a number of reasons. First, the Next Generation Timeliner Compiler does not currently handle the ground database file, an integral component in any environment integration [2]. Second, the integration with a space flight system, such as the space shuttle or the International Space Station, is a time intensive task. The Environment interface does have the necessary functionality to provide this integration. This was verified by employees of the Charles Stark Draper Laboratory who work with the existing Timeliner system.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A

Sample XML Input File for Figure 2-1

```
<?xml version="1.0"?>
<Bundle>
   <bundleName>EXAMPLE</bundleName>
   <endName>EXAMPLE</endName>
   <line>1</line>
   <user_info>null</user_info>
   <fsl><FieldAndSeqDecl><line>2</line>
      <fl><LastFieldDeclList><line>2</line>
         <head><NumFieldDecl><line>2</line>
            <id>X</id>
         </NumFieldDecl></head>
      </LastFieldDeclList></fl>
      <sl><LastSeqDeclList><line>3</line>
         <head><SeqDecl><line>3</line>
            <isSeq>true</isSeq>
            <id>ONE</id>
            <endID>ONE</endID>
            <status>INACTIVE</status>
            <userInfo>null</userInfo>
            <dsl><DeclStatementList><line>4</line>
               <sl><LastStatementList><line>4</line>
                  <head><CondStatement><line>4</line>
                     <endWord>WHEN</endWord>
                     <ch><CondHeader><line>4</line>
                        <condWord>WHEN</condWord>
                        <secondCondWord>null</secondCondWord>
                        <condition><OpExpr><line>4</line>
                           <op>10</op>
                           <left><IdExpr><line>4</line>
                              <id>X</id>
                           </IdExpr></left>
                           <right><IntExpr><line>4</line>
                              <value>1</value>
                           </IntExpr></right>
                        </OpExpr></condition>
                     </CondHeader></ch>
                     <slist><LastStatementList><line>5</line>
                        <head><SetStatement><line>5</line>
                           <id><IdExpr><line>5</line>
                              <id>X</id>
                           </IdExpr></id>
                           <value><IntExpr><line>5</line>
                              <value>0</value>
                           </IntExpr></value>
                        </SetStatement></head>
                     </LastStatementList></slist>
                  </CondStatement></head>
               </LastStatementList></sl>
            </DeclStatementList></dsl>
         </SeqDecl></head>
      </LastSeqDeclList></sl>
   </FieldAndSeqDecl></fsl>
</Bundle>
```

Appendix B

Plant Simulator Bundle

```
BUNDLE PLANTSIM

DECLARE TRYING_TO_COOL_SYSTEM BOOLEAN
-------------------------------------------------------
SEQUENCE STARTUP ACTIVE

    SET TRYING_TO_COOL_SYSTEM TO FALSE
    COMMAND RESET
    WAIT 1
    COMMAND LIGHTS, NEW_STATE=>ON
    WAIT 1
    COMMAND BATH, NEW_STATE=>ON
    WAIT 1
    START PH_MONITOR
    WAIT 5
    START NUTRIENT_MONITOR
    WAIT 2
    START CO2_MONITOR
    START OXYGEN_MONITOR
    WAIT 5
    START TEMP_MONITOR
    START HUMIDITY_MONITOR

CLOSE SEQUENCE
-------------------------------------------------------
SEQUENCE PH_MONITOR
    WHENEVER FARM_DATA.PH <= 6 THEN
        COMMAND ACID, NEW_STATE => ON
        WHEN FARM_DATA.NUTRIENT >= 6.3 THEN
            COMMAND ACID, NEW_STATE=>OFF
        END WHEN
    END WHENEVER
CLOSE SEQUENCE
-------------------------------------------------------
SEQUENCE NUTRIENT_MONITOR
    WHENEVER FARM_DATA.NUTRIENT <= 800 THEN
        COMMAND NITRATE, NEW_STATE=>ON
        WHEN FARM_DATA.NUTRIENT >= 1200 THEN
            COMMAND NITRATE, NEW_STATE=>OFF
        END WHEN
    END WHENEVER
CLOSE SEQUENCE
-------------------------------------------------------
SEQUENCE CO2_MONITOR
    WHENEVER FARM_DATA.CO2 <= 1200 THEN
        COMMAND DIOXIDE, NEW_STATE=>ON
        WHEN FARM_DATA.CO2_PUMP = ON WITHIN 10
            WHEN FARM_DATA.CO2 >= 1500 THEN
                COMMAND DIOXIDE, NEW_STATE=>OFF
                WHEN FARM_DATA.CO2_PUMP = OFF WITHIN 10 THEN
                    WAIT 1 -- DO NOTHING
                OTHERWISE
                    WARNING "CO2 PUMP NOT TURNING OFF --"
                    WARNING "INSPECT PUMP AND RESTART CO2 MONITOR"
                    STOP CO2_MONITOR
                END WHEN
            END WHEN
        OTHERWISE
            WARNING "CO2 PUMP NOT TURNING ON --"
            WARNING "INSPECT PUMP AND RESTART CO2 MONITOR"
            STOP CO2_MONITOR
        END WHEN
    END WHENEVER
CLOSE SEQUENCE
-------------------------------------------------------
SEQUENCE OXYGEN_MONITOR
    WHENEVER FARM_DATA.OXYGEN >= 25 THEN
        COMMAND OXYGEN, NEW_STATE=>ON
        WHEN FARM_DATA.O2_FILTER = ON WITHIN 10 THEN
            WHEN FARM_DATA.OXYGEN <= 20 THEN
                COMMAND OXYGEN, NEW_STATE=>OFF
                WHEN FARM_DATA.O2_FILTER = OFF WITHIN 10
                    WAIT 1 -- DO NOTHING
                OTHERWISE
                    WARNING "02 FILTER NOT TURNING OFF --"
                    WARNING "INSPECT FILTER AND RESTART 02 MONITOR"
                    STOP OXYGEN_MONITOR
                END WHEN
            END WHEN
        OTHERWISE
            WARNING "02 PUMP NOT TURNING ON --"
            WARNING "INSPECT FILTER AND RESTART CO2 MONITOR"
            STOP OXYGEN_MONITOR
        END WHEN
    END WHENEVER
CLOSE SEQUENCE
--------------------------------------------------------
SEQUENCE TEMP_MONITOR
    EVERY 1
        IF FARM_DATA.TEMPERATURE >= 25 THEN
            SET TRYING_TO_COOL_SYSTEM TO TRUE
            COMMAND COOLING, NEW_STATE=>ON
            WHEN FARM_DATA.TEMPERATURE <= 23
                SET TRYING_TO_COOL_SYSTEM TO FALSE
                COMMAND COOLING, NEW_STATE=>OFF
            END WHEN
        END IF
        IF FARM_DATA.TEMPERATURE <= 21 THEN
            COMMAND HEATING, NEW_STATE=>ON
            WHEN FARM_DATA.TEMPERATURE >= 23
                COMMAND HEATING, NEW_STATE=>OFF
            END WHEN
        END IF
    END EVERY
CLOSE SEQUENCE
--------------------------------------------------------
SEQUENCE HUMIDITY_MONITOR
    EVERY 1
        IF FARM_DATA.HUMIDITY >= 80 THEN
            COMMAND COOLING, NEW_STATE=>ON
            WHEN FARM_DATA.HUMIDITY <= 75 THEN
                IF NOT TRYING_TO_COOL_SYSTEM
                    COMMAND COOLING, NEW_STATE=>OFF
                END IF
            END WHEN
        END IF
        IF FARM_DATA.HUMIDITY <= 70 THEN
            COMMAND HUMIDITY, NEW_STATE=>ON
            WHEN FARM_DATA.HUMIDITY >= 75 THEN
                COMMAND HUMIDITY, NEW_STATE=>OFF
            END WHEN
        END IF
    END EVERY
CLOSE SEQUENCE
-------------------------------------------------------
SEQUENCE EMERGENCY_OPS
    EVERY 1
        IF FARM_DATA.TEMPERATURE <= 0 THEN
            WARNING "CHECK HEATING/COOLING UNIT AND TEMP SENSOR"
            WARNING "TEMPERATURE BELOW 0 DEGREES C"
            START SHUTDOWN
        END IF
        IF FARM_DATA.TEMPERATURE >= 50 THEN
            WARNING "CHECK HEATING/COOLING UNIT AND TEMP SENSOR"
            WARNING "TEMPERATURE ABOVE 50 DEGREES C"
            START SHUTDOWN
        END IF
        IF FARM_DATA.HUMIDITY <= 10 THEN
            WARNING "CHECK HUMIDIFIER AND HUMIDITY SENSOR"
            WARNING "HUMIDITY BELOW 10%"
            START SHUTDOWN
        END IF
        IF FARM_DATA.HUMIDITY >= 95 THEN
            WARNING "CHECK HUMIDIFIER AND HUMIDITY SENSOR"
            WARNING "HUMIDITY ABOVE 95%"
            START SHUTDOWN
        END IF
        IF FARM_DATA.CO2 <= 100 THEN
            WARNING "CHECK CO2 PUMP AND CO2 SENSOR"
            WARNING "CO2 BELOW 100ppm"
            START SHUTDOWN
        END IF
        IF FARM_DATA.CO2 >= 3000 THEN
            WARNING "CHECK CO2 PUMP AND CO2 SENSOR"
            WARNING "CO2 ABOVE 3000ppm"
            START SHUTDOWN
        END IF
        IF FARM_DATA.OXYGEN <= 10 THEN
            WARNING "CHECK O2 FILTER AND O2 SENSOR"
            WARNING "O2 BELOW 10%"
            START SHUTDOWN
        END IF
        IF FARM_DATA.OXYGEN >= 30 THEN
```

```
            WARNING "CHECK O2 FILTER AND O2 SENSOR"
            WARNING "O2 ABOVE 30%"
            START SHUTDOWN
        END IF
        IF FARM_DATA.NUTRIENT <= 500 THEN
            WARNING "CHECK NITRATE PUMP AND NUTRIENT SENSOR"
            WARNING "NUTRIENT LEVELS BELOW 500ppm"
            START SHUTDOWN
        END IF
        IF FARM_DATA.NUTRIENT >= 3000 THEN
            WARNING "CHECK NITRATE PUMP AND NUTRIENT SENSOR"
            WARNING "NUTRIENT LEVELS ABOVE 3000ppm"
            START SHUTDOWN
        END IF
        IF FARM_DATA.PH <= 5 THEN
            WARNING "CHECK ACID PUMP AND pH SENSOR"
            WARNING "pH BELOW 5.0"
            START SHUTDOWN
        END IF
        IF FARM_DATA.PH >= 8 THEN
            WARNING "CHECK ACID PUMP AND pH SENSOR"
            WARNING "pH ABOVE 8.0"
            START SHUTDOWN
        END IF
        IF FARM_DATA.LIGHTS = OFF THEN
            WARNING "LIGHTS SHUT OFF"
        END IF
        IF FARM_DATA.BATH = OFF THEN
            WARNING "NUTRIENT BATH SHUT OFF"
        END IF
    END EVERY
CLOSE SEQUENCE
-------------------------------------------------------
SEQUENCE SHUTDOWN
    MESSAGE "SHUTTING DOWN THE SYSTEM"
    STOP EMERGENCY_OPS
    STOP CO2_MONITOR
    STOP HUMIDITY_MONITOR
    STOP NUTRIENT_MONITOR
    STOP OXYGEN_MONITOR
    STOP PH_MONITOR
    STOP TEMP_MONITOR
    COMMAND HUMIDITY, NEW_STATE=>OFF
    COMMAND HEATING, NEW_STATE=>OFF
    COMMAND COOLING, NEW_STATE=>OFF
    COMMAND NITRATE, NEW_STATE=>OFF
    COMMAND DIOXIDE, NEW_STATE=>OFF
    COMMAND OXYGEN, NEW_STATE=>OFF
    COMMAND LIGHTS, NEW_STATE=>OFF
    COMMAND BATH, NEW_STATE=>OFF
    COMMAND ACID, NEW_STATE=>OFF
CLOSE SEQUENCE
-------------------------------------------------------
CLOSE BUNDLE
```

Appendix C

Executor Output When Executing Script in Appendix B

```
----------------------------------------------------------------------
    NO BUNDLES INSTALLED
----------------------------------------------------------------------
```

```
----------------------------------------------------------------------
    NO BUNDLES INSTALLED
----------------------------------------------------------------------
```

```
----------------------------------------------------------------------
 BUN/SEQ NAME             STATUS     LAST INSTRUCTION
----------------------------------------------------------------------
 GREENHOUSE               INACTIVE
----------------------------------------------------------------------
```

```
----------------------------------------------------------------------
 BUN/SEQ NAME             STATUS     LAST INSTRUCTION
----------------------------------------------------------------------
 GREENHOUSE               ACTIVE
     PUMPS                ACTIVE     IF NITRATE.NEW_STATE = ON THEN
----------------------------------------------------------------------
```

```
----------------------------------------------------------------------
 BUN/SEQ NAME           STATUS     LAST INSTRUCTION
----------------------------------------------------------------------
 GREENHOUSE             ACTIVE
     PUMPS              ACTIVE     SET FARM_DATA.O2_FILTER = FALSE
 PLANTSIM               ACTIVE
     STARTUP            ACTIVE     WAIT 1
     PH_MONITOR         INACTIVE
     NUTRIENT_MONITOR   INACTIVE
     CO2_MONITOR        INACTIVE
     OXYGEN_MONITOR     INACTIVE
     TEMP_MONITOR       INACTIVE
     HUMIDITY_MONITOR   INACTIVE
     EMERGENCY_OPS      INACTIVE
     SHUTDOWN           INACTIVE
----------------------------------------------------------------------
```

```
----------------------------------------------------------------------
 BUN/SEQ NAME           STATUS     LAST INSTRUCTION
----------------------------------------------------------------------
 GREENHOUSE             ACTIVE
     PUMPS              ACTIVE     IF OXYGEN.NEW_STATE = ON THEN
 PLANTSIM               ACTIVE
     STARTUP            ACTIVE     WAIT 2
     PH_MONITOR         ACTIVE     WHENEVER FARM_DATA.PH <= 6 THEN
     NUTRIENT_MONITOR   ACTIVE     WHENEVER FARM_DATA.NUTRIENT <= 800
     CO2_MONITOR        INACTIVE
     OXYGEN_MONITOR     INACTIVE
     TEMP_MONITOR       INACTIVE
     HUMIDITY_MONITOR   INACTIVE
     EMERGENCY_OPS      INACTIVE
     SHUTDOWN           INACTIVE
----------------------------------------------------------------------
```

```
----------------------------------------------------------------------
 BUN/SEQ NAME           STATUS     LAST INSTRUCTION
----------------------------------------------------------------------
 GREENHOUSE             ACTIVE
```

```
    PUMPS                 ACTIVE     IF DIOXIDE.NEW_STATE = ON THEN
PLANTSIM                  ACTIVE
    STARTUP               INACTIVE
    PH_MONITOR            ACTIVE     WHENEVER FARM_DATA.PH <= 6 THEN
    NUTRIENT_MONITOR      ACTIVE     WHENEVER FARM_DATA.NUTRIENT <= 800
    CO2_MONITOR           ACTIVE     WHENEVER FARM_DATA.CO2 <= 1200 THEN
    OXYGEN_MONITOR        ACTIVE     WHENEVER FARM_DATA.OXYGEN >= 25 THEN
    TEMP_MONITOR          ACTIVE     IF FARM_DATA.TEMPERATURE >= 25 THEN
    HUMIDITY_MONITOR      ACTIVE     IF FARM_DATA.HUMIDITY <= 70 THEN
    EMERGENCY_OPS         ACTIVE     IF FARM_DATA.CO2 <= 100 THEN
    SHUTDOWN              INACTIVE
-----------------------------------------------------------------------
```

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Eyles, Don. "A Time-Oriented Language for the Writing of Procedures to Sequence the Operation of a Spacecraft and its Systems." The Charles Stark Draper Laboratory Report P-3006, Presented at the AIAA Computing in Aerospace Symposium, October 1991.

[2] Liu, Frank Tien-Fu. "An Extensible Object-Oriented Compiler for the Timeliner User Interface Language." Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2001.

[3] NASA, The Charles Stark Draper Laboratory. "The Timeliner User Interface Language (UIL) System for the International Space Station." http://timeliner.draper.com/docs/971106_ISS_TL_WRITEUP.pdf

[4] Potratz, Eric. "A Practical Comparison Between Java and Ada in implementing a Real-Time Embedded System." Department of Computer Science, University of Northern Iowa, 2004.