Integrating Timeliner and Autonomous Planning

by

Daniel Reed Swanton

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2006

Author .....
Department of Electrical Engineering and Computer Science
May 26, 2006

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

Certified by.....
.....
Prof. Leslie. P. Kaelbling
Professor, MIT
Thesis Supervisor

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

Integrating Timeliner and Autonomous Planning

by

Daniel Reed Swanton

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2006, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Timeliner is used to automate tasks in a target system. Timeliner is capable of automating complex sequences of actions, but the desired actions must be planned out and understood in advance by human script-writers. This thesis presents the design, implementation, and demonstration of a prototype autonomous system built on top of Timeliner. It uses the existing Timeliner system for plan execution and execution monitoring, but adds a deliberative layer for autonomous mission planning. The increased autonomy of this system has the potential to lower operational costs and increase functionality.

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

Thesis Supervisor: Prof. Leslie. P. Kaelbling
Title: Professor, MIT

Acknowledgments

First, I would like to thank my father for putting me in a position to succeed, and for always pushing me to be more.

Second, I would like to thank the entire Timeliner Development group at Draper Laboratory. Robert Brown was a terrific advisor and was both encouraging and supportive. Joseph Bondi was always willing to drop what he was working on to help to me. Rick Brunet was instrumental to the implementation of my design.

Finally, I would like to thank Leslie Kaelbling. I could not have asked for a better advisor.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under Contracts NNJ06HC37C and NAS9-01069, 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 exchanged and stimulation of ideas.

Daniel R. Swanton

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Timeliner

Timeliner was developed in 1982 by The Charles Stark Draper Laboratory, Inc., to automate procedural task execution [8]. These tasks would normally have been executed in a well-defined sequence by a human operator. For example, a human operator may have a list of actions to execute, based on environment conditions, in order to control a fluid exchange onboard a satellite. It may be necessary to execute this fluid exchange many times, requiring the human operator to repetitively execute actions on the list by issuing commands to the satellite. By using Timeliner, the human operator can avoid this repetitive execution. Instead, the human operator encodes the sequences of actions necessary for the fluid exchange into a Timeliner script, which is then automatically executed. In this way, Timeliner automates task execution onboard the International Space Station and Orbital Express Satellite.

Timeliner can automate complex sequences of actions. For example, a Timeliner procedure might require a satellite to perform a fluid exchange, transition to a new state, and then perform another fluid exchange. The complexity of the procedure that Timeliner is capable of executing is limited only by the ability of the human operator to devise and encode the procedure in Timeliner scripts.

1.2 Autonomy

The existing Timeliner system removes the need for a human operator to manually execute operational procedures; however, human interaction is still necessary to devise procedures and plan when they should be executed. An autonomous planning agent adds the capability to determine when these procedures should be executed based on high-level mission goals. Autonomous planners take information about the current world state, a goal state, and available actions as input, and then output a plan to achieve the goal state using the provided actions. This reasoning process works by predicting the new state that would result from a hypothetical action, and then trying to combine these actions in a way that will achieve the goal state. A plan is simply a sequence of actions to take to achieve the goal. Because the plan may not always execute as expected, it is also necessary to identify execution failures and create new plans in response. This thesis presents the design and implementation of an autonomous system that adds high-level mission planning to the existing Timeliner execution system.

1.3 Objective

The objective of this project was to design, implement, demonstrate, and document a prototype system that interfaces the existing Timeliner system with an autonomous planner. The work consisted primarily of designing a framework to support this integration, and then providing the glue to enable the system components to interact. Specifically, the system must:

- Represent Timeliner action sequences in a way the planner can reason about.
- Communicate available Timeliner action sequences to the planner.
- Communicate current world state information from Timeliner to the planner.
- Encode plans in a form the Timeliner executor can execute.

- Install and execute plans.

- Identify plan execution problems and handle them.

The planner will create task plans to achieve high-level mission goals using Timeliner sequences. This functionality should be demonstrated in a proof-of-concept simulation.

1.4 Motivation

Increasing the autonomy of Timeliner should result in a system with lower operational costs and increased functionality. Operational costs can be lowered because human intervention would no longer be necessary to create plans. In addition, an autonomous planning agent may be capable of creating plans that are too complex to be feasibly created by human operators. Finally, the ability to dynamically create new plans can allow the Timeliner system to function effectively in new situations where the environment conditions and the correct course of action cannot be predicted in advance.

1.5 Scope

This thesis presents only a prototype autonomous planning system. The purpose is to demonstrate the feasibility of integrating Timeliner with an autonomous planning agent. The long-term goal for increasing the autonomy of ground operational systems is to integrate Timeliner with a more robust autonomous planner, such as the existing All-Domain Execution and Planning Technology (ADEPT) at the Charles Stark Draper Laboratory [3]. This integration is outside the scope of this project, although the framework developed in this thesis should lay the groundwork for future integration.

1.6 Organization

The remainder of this thesis is presented in six sections. First, the Timeliner section outlines the existing Timeliner system at a high-level and provides motivation for why an autonomous system benefits from using Timeliner. Second, the Related Work section briefly reviews relevant research in autonomous planning systems. Third, the Design section discusses the architecture of the system, as well as design choices. Fourth, the Implementation section explains the construction of a prototype system combining Timeliner and autonomous planning, and then presents a demonstration of the system. Fifth, the Limitations and Future Work section presents problems that the current system would have in various domains, and what steps would be necessary to correct them. Finally, the Conclusion section summarizes the design, implementation, and results of this thesis.

Chapter 2

Timeliner

This chapter presents an overview of Timeliner. First, the different components of the Timeliner system and how they interact will be discussed. Special emphasis will be placed on the aspects of Timeliner that will be used later in the integrated system. This chapter will end by presenting the advantages of using Timeliner in an autonomous system.

2.1 Overview

At a high-level, Timeliner consists of a specialized scripting language and a system for executing these scripts in a target system. An overview of Timeliner is presented in Figure 2-1.

First, a human produces a Timeliner script (TLS) encoding a sequence of commands to be executed. Next, the Timeliner Compiler compiles this TLS script file into an executable TLX file. Finally, the Timeliner Executor installed in a target

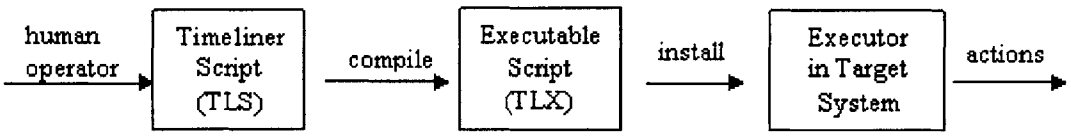

Figure 2-1: Timeliner Overview

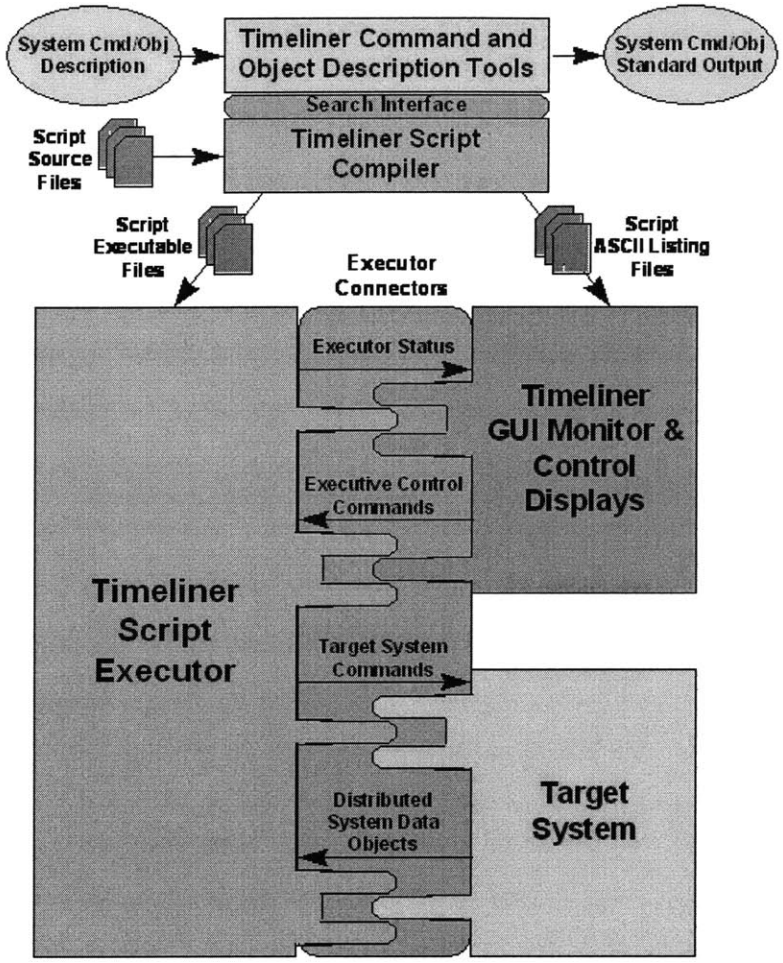

Figure 2-2: Timeliner System Components [8]

system executes these TLX files, resulting in actions.

An overview of the Timeliner system components is presented in Figure 2-2. At a high-level, the Timeliner system can be divided into sequence creation and sequence execution. Sequence creation involves creating Timeliner script files for the Timeliner Compiler. The Timeliner Compiler outputs executable versions of these scripts, and auxiliary ASCII Listing files. The executable scripts are provided to the Timeliner Executor, which interacts with both a target system and external monitoring applications. The Timeliner Executor controls the target system by issuing action commands and monitors the target system by receiving system data. The Timeliner Executor also interacts with external graphical interface applications that display execution status and can issue commands to the Executor. The rest of this chapter will present

```
SEQUENCE THERMOSTAT
      WHEN HEATER.TEMPERATURE > 75
            COMMAND HEATER, NEW_STATE => OFF
      END WHEN
      WHEN HEATER.TEMPERATURE < 70
            COMMAND HEATER, NEW_STATE => ON
      END WHEN
CLOSE SEQUENCE
```

Figure 2-3: Example Timeliner script for controlling a thermostat

these Timeliner system components in more detail.

2.2 Timeliner User Interface Language

Timeliner scripts are written in the Timeliner User Interface Language (UIL) [9], which is a time oriented scripting language similar to English that makes it easy to encode sequences of actions. For example, a Timeliner script for controlling a thermostat is demonstrated in Figure 2-3:

This simple thermostat sequence controls a heater by automatically turning the heater off when the temperature is above 75 degrees Fahrenheit and turning the heater on when the temperature is below 70 degrees Fahrenheit. The Timeliner scripting language is well structured and easy for human operators to understand. This example also demonstrates the relationship between Timeliner scripts and target systems: Timeliner scripts can monitor environment data, such as observing temperature, and issue commands that control a target system, such as turning the heater on or off.

Timeliner scripts are organized into sequences and bundles. Each Timeliner script file has exactly one bundle, and the bundle name is the same as the Timeliner script filename. For example, a Timeliner script named `HOUSE.TLS` would contain a bundle named `HOUSE`. Inside a bundle there can be any number of sequences. The example in Figure 2-3 contained a sequence for controlling a thermostat, but a `HOUSE` bundle could also contain other sequences for controller lights or speakers. Figure 2-4 presents an overview of the Timeliner Script Organization. Subsequences are subroutines that can

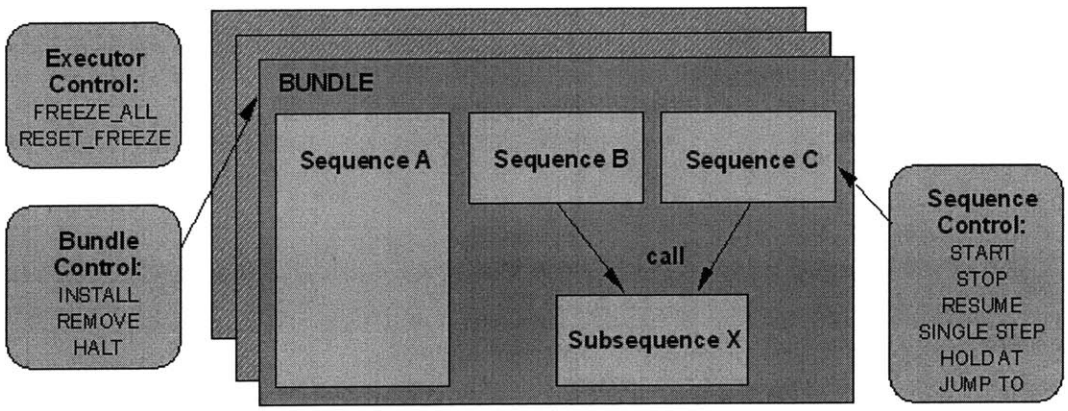

Figure 2-4: Timeliner Script Organization [8]

be called from sequences. The Timeliner Executor has robust control over sequence execution, and is able to control sequences externally through `START`, `STOP`, `RESUME`, and a number of other commands.

Timeliner sequences can contain powerful conditional control expressions. The Timeliner control clauses include `EVERY`, `BEFORE`, `WITHIN`, `WHEN`, `OTHERWISE`, `WHENEVER`, `IF`, `ELSEIF`, `ELSE`, `THEN`, `END`, `WAIT`, and `CALL`. These control expressions function as expected from the English names. Sequences can start and stop other sequences, and define and control subsequences. In addition to control expressions and sequence manipulation, sequences can also read state information from internal and external system variables, and send commands to control target systems. These sequence features can be combined to create powerful action scripts.

2.3 Ground Database

Timeliner was designed to be a modular system that could easily adapt to different target systems. At a high-level, Timeliner consists of two parts: the kernel, which contains core Timeliner language features, and the adapter, which contains target system specific code. The kernel remains the same between different target systems, while the adapter is changed to accommodate the target system. In the thermostat example, the kernel is responsible for basic language features such as `WHEN`, `END WHEN`, `>`, `SEQUENCE`, etc., while the adapter is responsible for the interface with `HEATER`, includ-

ing both reading the HEATER.TEMPERATURE variable and issuing the COMMAND HEATER, NEW_STATE=>ON command. The separation between adapter and kernel allows Timeliner to be easily deployed on different target systems. Furthermore, Timeliner script writers do not have to be concerned with details such as how the heater works in a particular thermostat. Instead, they can write a generic Timeliner script, which can be used to control any heater for which a suitable adapter exists.

The primary way that Timeliner adapts to different target systems is through Ground Database (GDB) definitions. The Ground Database is defined primarily by an `attributes` file and an `instances` file.

The `attributes` file defines which variables will be used by the target system and Timeliner. For example, TEMPERATURE would be defined in the Ground Database `attributes` file for for HEATER. Variables defined in the `attributes` file can have a variety of types including BOOLEAN, INTEGER, or LONG_FLOAT and a variety of properties such as READ_WRITE or READ_ONLY.

The `commands` file defines which commands can be issued by Timeliner to the target system. For example, the Ground Database `commands` file for HEATER would provide definitions for a command that could modify the heater state to a new boolean.

The Ground Database definition files are processed by the Timeliner `gdb` executable, which outputs a compiled version of the commands and variables that are used to adapt Timeliner to the target system.

2.4 Compiling and Mapping

After a Timeliner script (TLS) is created and finalized, it is compiled into a machine-readable TLX file. This TLX file contains a binary representation of the Timeliner script that is minimal in size. This reduction in file size is necessary to load Timeliner scripts onto embedded target systems with constrained memory sizes. The TLX file also contains information for assisting with execution of the script.

After compilation is completed, the Timeliner script is mapped, which generates a file relating the Timeliner script to the Ground Database. Mapping is necessary

for the Ground Database references in the Timeliner script to be handled correctly during execution.

2.5 Executor

Independently of script writing, the Timeliner Executor is installed on a target system. The Executor can receive commands to `INSTALL`, `HALT`, and `REMOVE` bundles. When a bundle is installed, all the sequences in that bundle become available to the Executor.

Sequences containing the `ACTIVE` keyword in the Timeliner script are started automatically. Otherwise, sequences can be started and stopped manually, both by commands to the Timeliner Executor or by other active sequences. The Timeliner Executor steps through each line of a sequence as dictated by the control clauses, reading sensor information from the target system, and activating commands in the target system. The Timeliner Executor is capable of executing different sequences in parallel. During execution, the Timeliner Executor updates the status of each installed sequence to reflect whether it is finished, active, never started, or in a variety of other states.

2.6 Central Value Tables

Timeliner variables are shared across bundles and sequences through Central Value Tables (CVT). CVT acts as a storage location where Timeliner scripts can read and write variable values. Figure 2-5 presents a simplified overview of CVT. In this figure two Timeliner scripts share a `TRYING_TO_COOL_SYSTEM` variable through CVT. Timeliner scripts can treat CVT as global memory for all bundles and sequences.

A CVT Monitor program exists to view and change values in CVT.

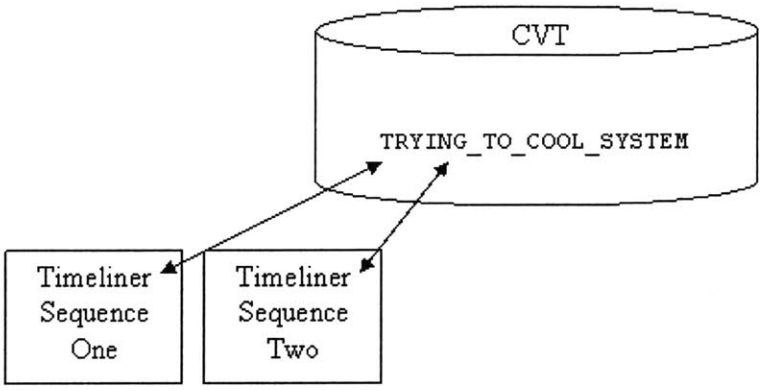

Figure 2-5: Central Value Tables (CVT) Overview

2.7 Timeliner Toolbar

This section presents the Timeliner Toolbar (TLToolbar), which is a graphical interface to the Timeliner system. This interface will later be modified to include an autonomous planner. For now, looking at graphical representations of the TLToolbar, CVT Monitor, and Timeliner Executor should help in understanding the Timeliner system.

The TLToolbar has options for creating ground databases, compiling and mapping Timeliner scripts, viewing compiled TLX files, and starting the Test Environment. This toolbar is presented in Figure 2-6.

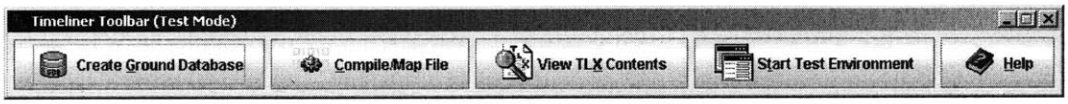

Figure 2-6: The Timeliner Toolbar

The `Start Text Environment` interface launches both the `CVT Monitor`, an interface the Central Value Tables, and the `Timeliner Display`, an interface to the Timeliner Executor.

The CVT Monitor graphical interface is presented in Figure 2-7. This interface displays all variables in CVT as well as their current values. The CVT Monitor edit mode allows the variable values to be changed directly. The CVT Monitor also presents logs of all updates to CVT.

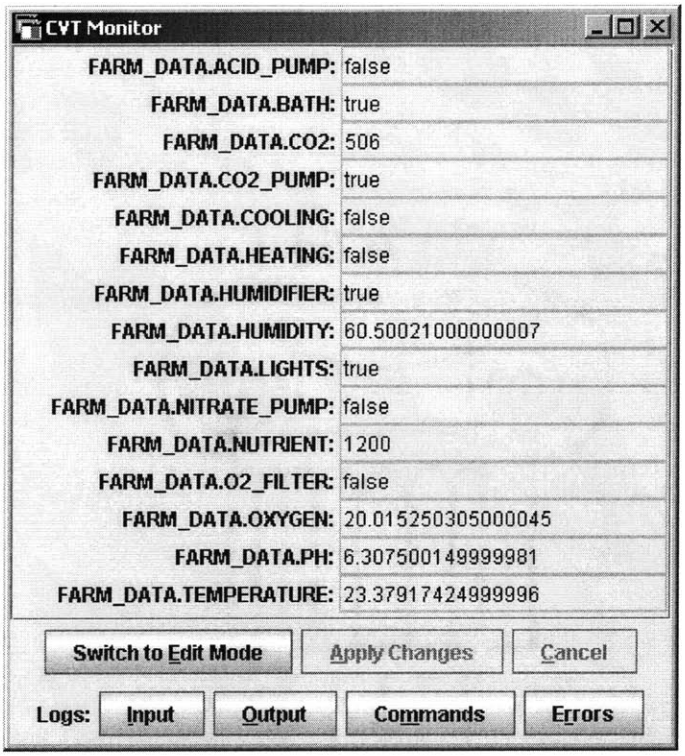

Figure 2-7: The CVT Monitor

The Timeliner Display interface to the Timeliner Executor is presented in Figure 2-8. This interface displays all installed bundles, sequences, and all text messages that have been output by the Executor. As well as displaying information about the current execution status, this interface also allows commands to be issued to the Executor. Bundles can be installed, removed, and halted. Sequences can be started, stopped, resumed, and canceled.

2.8 Timeliner Advantages

This section discusses how an autonomous system benefits from using Timeliner. In short, Timeliner provides a robust and abstract interface with a target system. The primary advantages of using Timeliner in an autonomous system are the ease in writing powerful control scripts, the ease of adapting Timeliner to different target systems, and the robust safety features of the Executor.

The Timeliner User Interface Language was designed to make control sequences

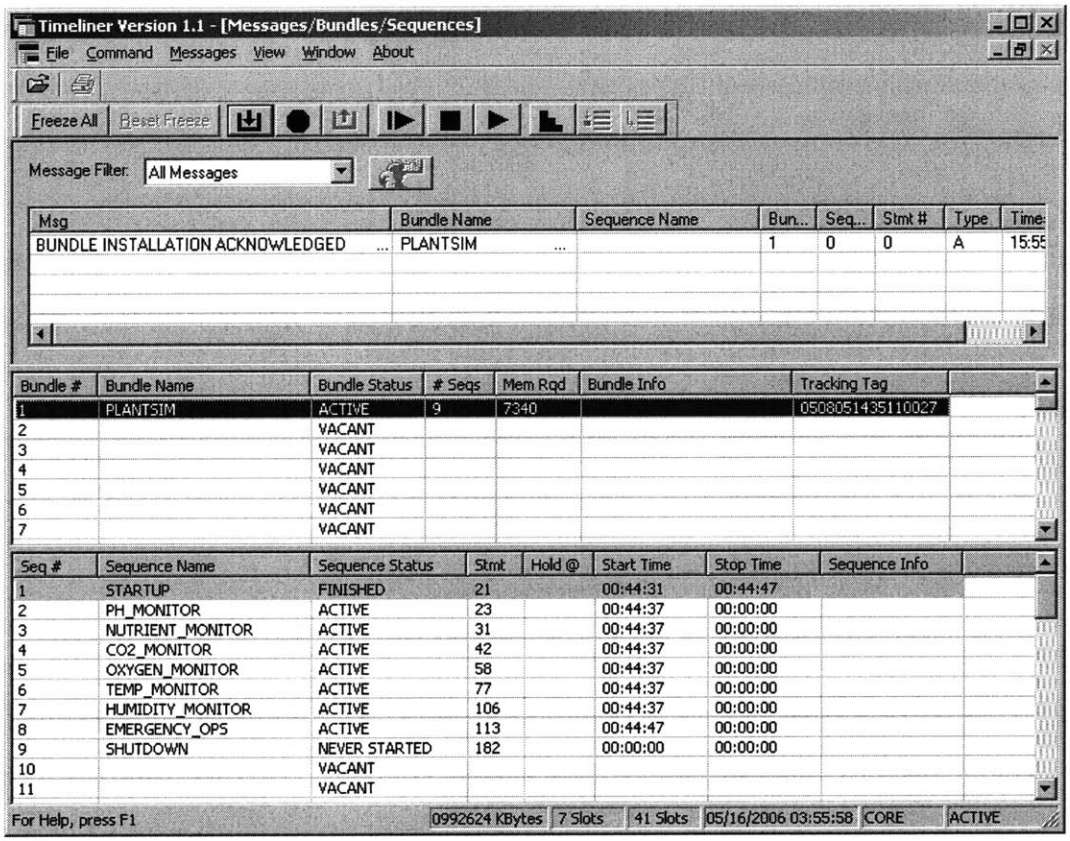

Figure 2-8: The Timeliner Display: an Interface to the Timeliner Executor

easy to write. Timeliner scripts are written in a language similar to English, and Timeliner has many built in control clauses and features. The Timeliner Integrated Development Environment is a graphical editor for Timeliner scripts that makes development even easier by providing continuous compilation, integrated help, database browsing and searching, and integration with external Timeliner tools [4].

Timeliner is easily adaptable to different target systems using different Ground Databases. Timeliner shields the details of connecting to different target systems from the script writers, providing a more abstract interface. This means that a Timeliner script written for one target system has the potential to be reused in a different target system with minimal changes to the script itself. Only the adapter and Ground Database must change.

The Timeliner Executor is reliable enough to be used in mission critical applications such as the International Space Station. Furthermore, it was designed to be easily monitored and to have its execution return to human control if desired. Timeliner is a powerful and robust system for automated sequencing.

Chapter 3

Related Work

This chapter presents related work in creating systems combining sequencing and autonomous planning. First, the Three-Layer Architecture will be presented as an existing solution architecture. Second, an example system using the Three-Layer Architecture will be discussed.

3.1 Three-Layer Architectures

In 1998 Erran Gat proposed the three-layer architect for autonomous robots, which organizes control algorithms by the amount of internal state that they use [5]. Systems are organized into three layers:

- *Control Layer* - Algorithms in the control layer map sensors directly onto actions, with little or no internal state.
- *Sequencing Layer* - Algorithms in the sequencing layer manage sequences of activities, using internal state, but not performing search.
- *Deliberative Layer* - Algorithms in deliberative layer are time-consuming, such as searching and planning.

Requirements of the sequencing layer are of particular importance to this thesis because Timeliner is a sequencing language. Gat explains that the sequencer must be

capable of responding conditionally to any situation it finds itself in. The conditional control structures required are significantly more complex than the control constructs of typical programming languages such as `if` statements.

After presenting the Three-Layer Architecture, Gat describes a case study of controlling a robot. This robot explored a room filled with obstacles. The control layer contained simple procedures such as turning in place, as well as more complex algorithms for wall-finding, wall-alignment, wall-following, obstacle avoidance, and wandering. The sequencing layer contained algorithms that used these procedures to achieve more complex goals such as exploring the entire room. To do this, the sequencing layer kept an internal state of where the robot had been. The deliberative layer had simple time-consuming exhaustive search algorithms for path planning between locations. Organizing the algorithms in this way enabled the robot to exhibit a robust set of behaviors with minimal effort from the programmers. [5]

3.2 3T: An Example Three-Layer Architecture

This section presents 3T, which used the Three-Layer Architecture to design and organize algorithms for autonomous control of an advanced life support system (ALS). The goal of this system was to monitor and control an environment to support a human crew. Primitive actions and behaviors of the controlled system were encapsulated in the control layer. The sequencing layer contained higher-level sequences that used these primitive behaviors to accomplish specific tasks. The deliberative layer created plans to achieve system goals in terms of the sequencing and control layer algorithms. The deliberative layer encoded a plan by placing actions on the agenda of the sequencer. 3T used the Reaction Action Packages (RAPs) system for sequencing, the AP task net planner for planning, and IPC message passing for communication within the system [1].

This autonomous system was tested in two different scenarios and was found to have reduced human work-load significantly [1].

Chapter 4

Design

This chapter presents the design of a system combining Timeliner and an autonomous planner using the Three-Layer Architecture. This chapter will present the architecture of each component necessary to make this integrated system.

4.1 Timeliner and Three-Layer Architectures

The Three-Layer Architecture was used as an organizational tool to integrate Timeliner and an autonomous planner. In this system, algorithms from the control layer are either primitive target system actions, or trivial Timeliner sequences. Algorithms from the sequencing layer are simply Timeliner scripts. Algorithms from the deliberative layer include autonomous planning and monitoring. The proposed system is an implementation of the Three-Layer Architecture.

4.1.1 Control Layer

The existing Timeliner system is already designed to handle simple commands. Once the Ground Database and Timeliner adapter have been configured for a target system, Timeliner can execute simple target system actions directly through the `COMMAND` keyword. A small Timeliner script with little or no internal state can also be thought of as an element of the control layer under in the Three-Layer Architecture. Standard

Timeliner already provides control layer functionality.

4.1.2 Sequencing Layer

The existing Timeliner system is also already designed to handle robust action sequences. These sequences are defined in Timeliner scripts, compiled into Timeliner executable files, and then executed in sequence by the Executor. The sequencing layer requires powerful conditional control structures, which Timeliner already provides. In addition, Timeliner scripts are easy to write and can be executed safely.

4.1.3 Deliberative Layer

The existing Timeliner system does not currently have a deliberative layer, nor does it have any control algorithms that would belong in the deliberative layer. Instead, Timeliner sequences are currently written by human script-writers. Although these sequences can be complex, they perform no search or planning. The decision of when to start and stop sequences is also made my humans. In some cases sequences start other sequences based on conditional situations, but the root decision of which scripts to execute at which time is made manually.

Adding deliberative algorithms to the Timeliner system will require creating a new deliberative layer. This deliberative layer will take the form of an autonomous planner and an autonomous monitor. The autonomous planner will be responsible for constructing task plans to achieve high-level mission goals. The autonomous monitor will be responsible for verifying that plans are executed correctly and initiating replans when necessary.

4.2 Design Goals

The primary goal of this design is to reuse as much of the existing Timeliner system as possible. In particular, special care has been taken to reuse the existing Timeliner communication infastructure when interfacing with the deliberative layer. Reusing

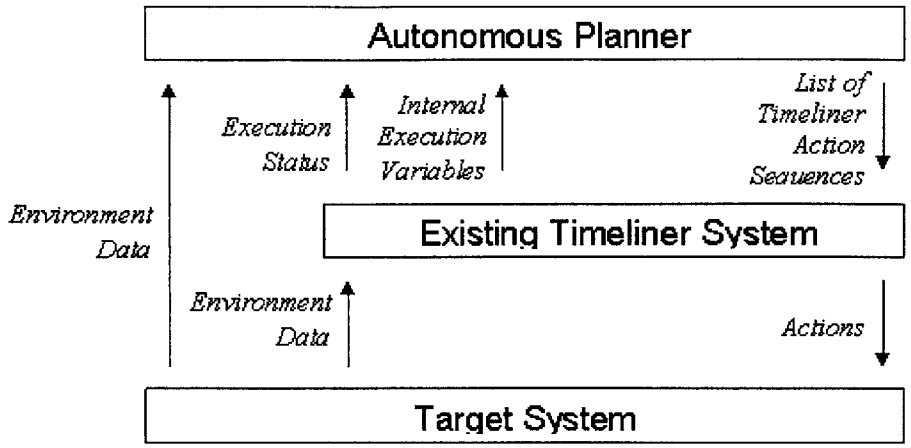

Figure 4-1: Interface Overview between the target system, the existing Timeliner system, and an autonomous planner

existing components makes the resulting system simpler and more elegant, with fewer places for error.

4.3 Design Overview

This section introduces the design of a system integrating Timeliner and autonomous planning. At a high-level, this system is composed of an autonomous planner that interfaces with Timeliner to provide a greater amount of flexibility and functionality. An overview of the interface between the autonomous planner, Timeliner, and a target system is presented in Figure 4-1.

The existing Timeliner system is responsible for executing action sequences in the target system. As input, the autonomous planner receives environment data from the target system and execution status reports from the Timeliner Executor. As output, the autonomous planner is responsible for producing a plan consisting of Timeliner action sequences for the existing system to execute.

4.4 GraphPlan

In this prototype, the autonomous planner is an implementation of GRAPHPLAN. GRAPHPLAN provides the needed planning functionality without being unnecessarily complex.

4.4.1 Autonomous Planner Requirements

As a proof of concept implementation, it is not necessary to select the perfect autonomous planner. In fact, the planner simply needs to provide basic planning functionality. Specifically, given the current world state, a goal state, and a list of available actions as input, the planner must be able to generate a plan to reach this goal using the available actions, provided that the goal state is reachable in a reasonable number of steps. As long as the autonomous planner is capable of generating simple plans, it will be able to demonstrate the integration of Timeliner and a planning system. The autonomous planning component can be replaced in the future if different functionality is required.

GRAPHPLAN was chosen as the autonomous planner for this implementation. GRAPHPLAN is an autonomous planner that is already well understood [7] [6] [2]. It provides the required plan generation functionality without being unnecessarily complex. A number of other autonomous planners would also satisfy the system requirements, and these can be investigated in future work.

4.4.2 Advantages of GraphPlan

GRAPHPLAN has the following advantages as the choice for the autonomous planner in this system:

- *Provides required planning functionality* GRAPHPLAN satisfies the basic requirement of the autonomous planning component by being capable of generating plans to reach simple goals. In fact, GRAPHPLAN always generates plans with the shortest possible number of steps to reach the goal [7].

- *Generates partially ordered plans* GRAPHPLAN can generate partially ordered-plans in which multiple actions are executed in parallel at one step of the plan. This takes advantage of the existing ability of the Timeliner system to execute sequences in parallel.

- *Simplicity* GRAPHPLAN is a relatively simple planner that is already well-understood in the academic community.

4.4.3 Disadvantages of GraphPlan

Depending on the domain the planner will be used in, there are a number of limitations of GRAPHPLAN. A few of the important disadvantages of the chosen version of GRAPHPLAN are presented below:

- *Only booleans* This implementation of GRAPHPLAN considers only boolean variables. This means that numeric values will need to be converted into booleans to be used. For example, if an integer variable named `STATE` ranged in value from `0` to `4`, this variable could be factored into five booleans: `STATE0`, `STATE1`, `STATE2`, `STATE3`, `STATE4`, and `STATE5`. This would be the only way for GRAPHPLAN to plan using this variable, because it can only consider booleans. Factoring numeric variables is an expensive process, especially if the variables can take a wide range of values.

- *No disjunctions* GRAPHPLAN cannot handle `or` statements in boolean expressions [6]. For example, GRAPHPLAN cannot handle an action that has the effect of either turning on fan one or fan two. Each action must have a definite result.

- *Limited plan depth* The GRAPHPLAN planning process takes exponential time in the number of steps of the plan. The unoptimized version of GRAPHPLAN used in this prototype can only construct plans ten steps deep.

While these disadvantages do limit the functionality of this prototype system, GRAPHPLAN was selected as the autonomous planner because of its numerous advantages and because this implementation is only a proof-of-concept.

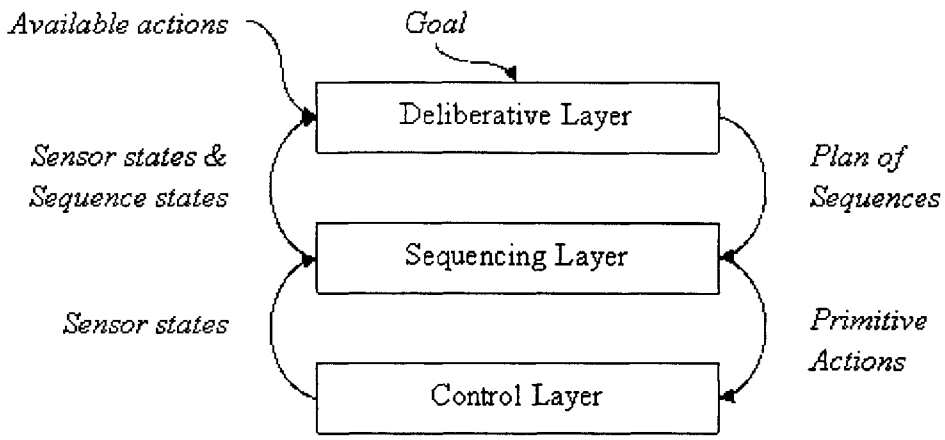

Figure 4-2: Overview of information flow

4.5 Integrating Timeliner and the Deliberative Layer

One of the key design aspects of the integration of Timeliner and an autonomous planner is how information is passed between the different layers of the system. Figure 4-2 presents an overview of the information flow.

The most straightforward interaction in the system is between the control layer and the sequencing layer. This interaction is already a part of Timeliner. Timeliner reads simple sensor data about the environment into primitive variables, and these variables are shared across Timeliner sequences using CVT. Timeliner sequences can directly initiate commands in target systems and start control sequences.

The interface to the deliberative layer is more complicated and will be the focus of the remainder of this chapter. The autonomous planner in the deliberative layer needs information about the current world state, the available actions, and the goal state. The goal state will be input manually by human operators, but the world state and available actions will need to be communicated through the Timeliner system. Once the autonomous planner has all required information, it will construct a plan. This plan will later need to be formatted in such a way that the Timeliner Executor can understand and execute the plan.

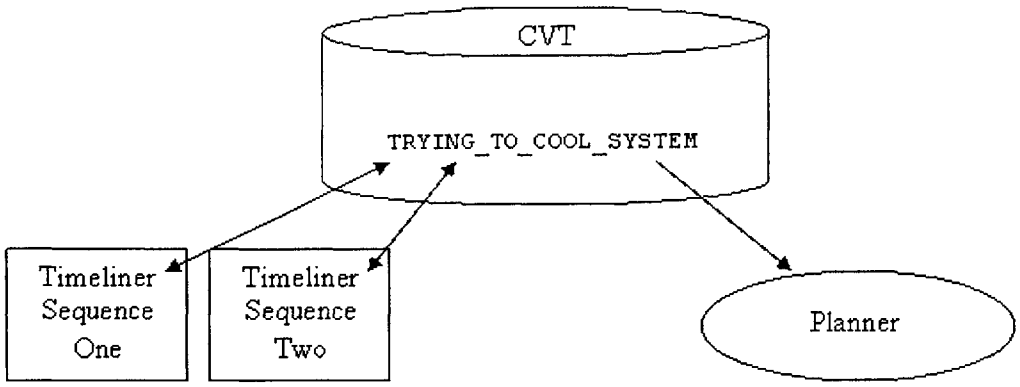

Figure 4-3: Sharing CVT information with the Planner

4.6 Communicating the World State to the Planner through CVT

Timeliner will communicate the current world state to the deliberative layer using the CVT framework described in Section 2.6. Timeliner sequences already use CVT to read and write variable values during execution. In other words, CVT already contains the current world state.

The deliberative layer will access variable values directly from CVT. Figure 4-3 presents a graphical representation of sharing variable values with the planner through CVT. The two Timeliner sequences read and write the `TRYING_TO_COOL_SYSTEM` value, while the planner reads the value. In fact, planner will only read values from CVT, never write them. Although the planner could access the current variable values from CVT at any time, it will only request them when it is constructing a new plan and needs to know the current state.

4.7 Communicating Actions to the Planner

The autonomous planner will use Timeliner sequences as actions in plans it constructs. A target system command can be used as an action by wrapping it in a simple Timeliner sequence that calls the low-level command. Using only one type of action, the Timeliner sequence, simplifies the system design. The planner will only need to

```
SEQUENCE POWER_ON_ASTRO_ORU1
      PRECONDITION NOT ASTRO_ORU1_POWER
      POSTCONDITION ASTRO_ORU1_POWER

            ... (sequence logic) ...
CLOSE SEQUENCE
```

Figure 4-4: Preconditions and Postconditions for POWER_ON_ASTRO_ORU1 sequence

decide which Timeliner sequences to execute and when.

However, in order for the planner to make plans in terms of Timeliner sequences, it will need to understand which circumstances the sequences can be used in and what the effects of executing each sequence would be. This information will be manually specified by human script-writers in the form of PRECONDITION and POSTCONDITION statements.

4.7.1 Extending the Timeliner User Interface Language with Preconditions and Postconditions

The primary change that must be made to Timeliner in order to integrate with an autonomous planner is the addition of PRECONDITION and POSTCONDITION statements to Timeliner sequences. The PRECONDITION statements explicitly specify what the sequence assumes about the state of the world before execution. The POSTCONDITION statements explicitly specify the state the world should be in after a successful execution of this sequence. For example, Figure 4-4 has preconditions and effects statements for the POWER_ON_ASTRO_ORU1 sequence. This sequence has a precondition that the power is off, and executing this sequence has the effect of turning the power on.

The idea behind the PRECONDITION and POSTCONDITION statements tied to sequences is that they are necessary for the autonomous planner to understand when to execute a sequence and what should happen when a sequence is executed. The PRECONDITION and POSTCONDITION statements are used by the integrated Timeliner and autonomous planning system in two distinct ways: at compile time to generate a list of operators and at execution time to verify that sequences are executed as

planned.

4.7.2 Changes to the Timeliner Compiler: Compiled .TLO Files

As described in Section 2.4, when a Timeliner script is compiled and mapped, an executable binary file is generated along with a mapping file. In addition to these files, a new Timeliner Operators file (`.TLO`) will be generated containing details about the `PRECONDITION` and `POSTCONDITION` statements of each sequence in the Timeliner script.

This file will be ASCII plain text and organized as a table of comma-separated values. Specifically, each line starts with a sequence name, is followed by a statement type of either `PRE` for `PRECONDITION` or `POST` for `POSTCONDITION`, and finally contains a boolean expression for the precondition or postcondition.

```
Sequence Name, "PRE" or "POST", Boolean Expression
```

Figure 4-5: .TLO Line Format

When a sequence has multiple preconditions or postconditions, this can either be described by conjoining the statements into one larger boolean expression, or by listing the statements on multiple lines. The effect of having multiple lines with the same sequence name and statement type is that their boolean expressions will be conjoined together with `AND` statements. For example, Figure 4-6 contains the contents of a simple `.TLO` file describing an `ACID_ON` sequence:

```
ACID_ON, PRE, !FARM_DATA.ACID_PUMP
ACID_ON, PRE, !FARM_DATA.NITRATE_PUMP
ACID_ON, POST, FARM_DATA.ACID_PUMP
```

Figure 4-6: Example ACID_ON .TLO File

The `ACID_ON` sequence has preconditions that the acid pump and nitrate pump must be off, and has the effect of turning the acid pump on. This `.TLO` file could have

been generated by either of the following two Timeliner scripts:

```
SEQUENCE ACID_ON
      PRECONDITION NOT FARM_DATA.ACID_PUMP
      PRECONDITION NOT FARM_DATA.NITRATE_PUMP
      POSTCONDITION FARM_DATA.ACID_PUMP
            ... (sequence logic) ...
CLOSE SEQUENCE

SEQUENCE ACID_ON
      PRECONDITION NOT(FARM_DATA.ACID_PUMP) AND NOT(FARM_DATA.NITRATE_PUMP)
      POSTCONDITION FARM_DATA.ACID_PUMP
            ... (sequence logic) ...
CLOSE SEQUENCE
```

Figure 4-7: Two Example Timeliner scripts for generating ACID_ON .TLO

These Timeliner Operator files have been designed to make it easy for the planner to parse. The planner will take the .TLO file and read from it a list of action operators, where each action has a name, a set of preconditions, and a set of postconditions. An alternate design would be to have no external file, and instead for the planner to parse the precondition and postcondition statements directly from the .TLS Timeliner scripts. Both designs would function the same way, as long as the planner was able to parse out the preconditions and postconditions for each sequence. However, the current design makes the parsing process notably easier.

To parse the .TLS file directly would require duplicating functionality in the compiler. Specifically, it would require tracking the sequence context to determine which sequence each PRECONDITION and POSTCONDITION statement belonged to. It would also take longer to parse the file, as there would be irrelevant script information that needed to be ignored. The only cost of creating external .TLO files is a change to the Timeliner Compiler and the generation of another file. Timeliner already generates a number of auxiliary ASCII files such as the .TLL listing files. The .TLO Timeliner Operator file follows the naming convention of a three letter extension starting with TL. The cost of modifying the Timeliner Compiler is minimal. The compiler already needed to be modified so that PRECONDITION and POSTCONDITION statements will become accepted keywords in the Timeliner language. The Timeliner Compiler already tracks sequence context and parses Timeliner scripts, so it can output .TLO files with only a few small modifications. Ultimately, outputting Timeliner Operator

files at compile time and then parsing these files for preconditions and postconditions was easier and more in line with the Timeliner design than parsing Timeliner scripts directly.

Compiled .TLO files are inputs to the autonomous planner describing which actions are available. The planner will consider each sequence specified in a .TLO file to be a possible action and attempt to combine actions in a way that will achieve its high-level goal.

4.8 Converting a Plan to a Timeliner Sequence

The deliberative layer communicates its plan to the Timeliner sequencing layer by encoding the plan in a high-level Timeliner script named PLAN.TLS. PLAN.TLS contains exactly one sequence named ONE, which is specified as ACTIVE, meaning that this sequence will be started automatically when the PLAN bundle is installed.

For example, a toy problem from [2] involves preparing a dinner, wrapping a present, and ensuring that the house is clean. GraphPlan outputs a two-step plan for achieving this goal: step one: cook; step two: wrap and clean. The encoding of this simple plan into a standard Timeliner script is demonstrated in Figure 4-8.

```
BUNDLE PLAN
      SEQUENCE ONE ACTIVE
            START TOY.COOK
            WHEN TOY.COOK.SEQSTAT = SEQ_FINISHED
                  START TOY.CARRY
                  START TOY.WRAP
            END WHEN
      CLOSE SEQUENCE
CLOSE BUNDLE
```

Figure 4-8: Example Plan Script

This plan bundle contains two steps. In the first step, the COOK sequence is started, and in the second step both the CARRY and WRAP sequences are started in parallel. The first step is started automatically, because the ONE sequence starts ACTIVE. Specifically, the first step starts the COOK sequence. When the COOK is finished executing, its

```
BUNDLE PLAN
       SEQUENCE ONE
              -- EMPTY PLAN: goal cannot be reached in search depth
       CLOSE SEQUENCE
CLOSE BUNDLE
```

Figure 4-9: Empty Plan Script

status (SEQSTAT) will be changed to finished (SEQ_FINISHED). The Timeliner Executor will wait until this sequence status change is made before proceeding on to step two. Step two starts two sequences: CARRY and WRAP. In a three step plan, the Timeliner Executor would wait until both sequences had a finished status before proceeding. However, because step two is the last step of this example plan, the script ends.

In some situations GRAPHPLAN will be unable to construct a plan to reach the goal state. This implementation of GRAPHPLAN only consider plans that are up to ten steps deep, and if no path to the goal state is found in this number of steps then it will give up. The prototype system handles this possibility by simply outputting an empty plan. This empty plan is presented in Figure 4-9. Note that the -- symbol in Timeliner creates a comment. A more robust solution would notify a human operator of the problem.

4.8.1 Compiling and Installing the Plan

After a plan has been converted to a Timeliner sequence, it is compiled into a Timeliner executable file, and this file is installed in the Timeliner Executor. This process is straightforward because the plan has already been formatted into a valid Timeliner script. The Timeliner plan script is written to a file named PLAN.TLS, and the Timeliner Compiler is then called on PLAN.TLS to create an executable named PLAN.TLX. PLAN.TLX is a valid Timeliner bundle and can be installed using the existing INSTALL BUNDLE command. Before the PLAN bundle is installed, the system first checks to see if a plan bundle is already installed, in which case the existing plan is halted and removed. Only one plan can be active in the prototype system at one time.

4.9 Monitoring Sequence Status and Replanning

The PRECONDITION and POSTCONDITION statements added to the Timeliner language are also used to verify that plans are executed as expected. GRAPHPLAN constructs only valid plans. Sequences in the first step of the plan have preconditions that are satisfied by the current state of the world. These sequences promise some explicit changes to the state of the world after they are finished executing. Sequences in the second step have preconditions that depend on the first step sequences executing as expected. This process continues until the last sequence in the plan which should produce the goal state.

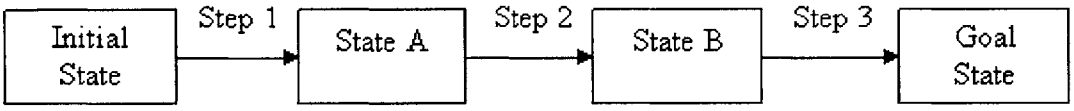

Figure 4-10: Abstract Plan

Figure 4-10 presents an abstract simple plan sequence with three steps and four states. The system starts in the initial state. Step 1 of the plan has a precondition that the system be in the initial state, and has a postcondition that the system be in State A. Step 2 of the plan has a precondition that the system be in State A, and has a postcondition that the system be in State B. Finally, Step 3 of the plan has a precondition that the system be in State B, and a postcondition that the system be in the goal state. Using this plan, the goal state will be achieved as long as the system starts in the initial state and then executes each step in the plan as expected. The plan will fail if any of the steps fails to produce the promised postcondition state, or if an unexpected change to the world state happens that disrupts a precondition.

If the PRECONDITION or POSTCONDITION statements of any step of the plan are violated, then something has gone wrong. A POSTCONDITION error indicates that an action sequence failed to produce the promised change to the world state. For example, the POWER_ON_ASTRO_ORU1 sequence in Figure 4-4 specifies that the ASTRO_ORU1_POWER variable will be TRUE after completion of the sequence. If the sequence finishes ex-

ecuting but this postcondition is not satisfied, then an error occurred that could compromise the remainder of the plan. A PRECONDITION error indicates that some unexpected change has been made in the world that violates assumptions made during plan creation. For example, the steps in the plan may have been executed correctly, but an unexpected hardware failure could have occurred that changed the system state in a way that the plan is no longer valid. Sequence PRECONDITION and POSTCONDITION statements are used to verify that plans are executed as expected.

4.9.1 Changes to the Timeliner Executor

The Timeliner Executor was modified to verify PRECONDITION and POSTCONDITION statements during execution and to update sequence status statements with errors. Before a sequence is executed, each PRECONDITION statement is evaluated to verify that it is satisfied. If any PRECONDITION statement is not satisfied, then sequence execution stops immediately, and the the status of the sequence changes to "ERROR: PRECONDITIONS NOT SATISFIED". After a sequence has finished executing, each POSTCONDITION statement is evaluated to verify that it is satisfied. If any POSTCONDITION statement is not satisfied, then the status of the sequence changes to "ERROR: POSTCONDITIONS NOT SATISFIED". When a precondition or postcondition error is discovered in a sequence, the execution of that sequence is stopped, and the status of the sequence is change to an error statement.

4.9.2 Monitoring Plan Sequences for Errors

The deliberative layer constantly monitors sequence status statements for precondition or postcondition errors. Whenever the Timeliner Executor sends an update packet about the status of its sequences, the autonomous monitor checks the status of every sequence. If a single error is detected, the monitor initiates a replan.

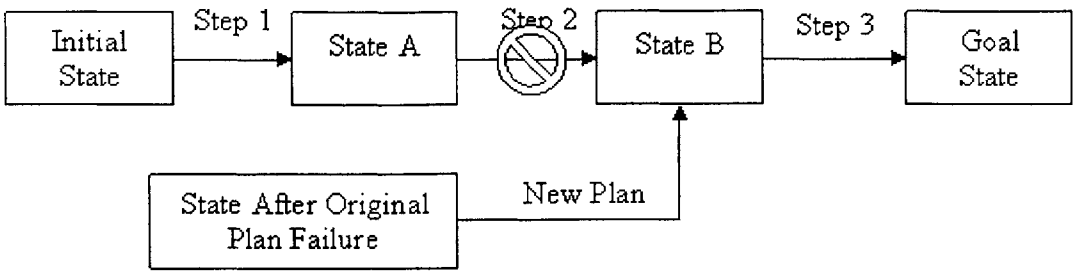

Figure 4-11: Replanning to existing plan

4.9.3 Replanning

In this prototype system, the replan command from the autonomous monitor simply initiates the construction of a new plan from scratch, with the planner still trying to achieve the goal state of the failed plan. The planner takes the current world state from CVT, and constructs a new plan to reach the goal. As long as script-writers are careful to ensure that Timeliner sequences always result in the specified postcondition states, plan failures will result from precondition errors due an unexpected changes in the world.

Another replanning scheme that was considered was to construct a repair plan that attempts to get back on track to the original plan. For example, if a failure occurred in the abstract plan of Figure 4-10 during the precondition of step 2, the system could construct a new plan to get to State A. From here, the old plan could be reused. This idea is depicted graphically in Figure 4-11. This strategy would provide significant savings when constructing full length plans is expensive.

However, because this proof-of-concept prototype focuses on the integration of Timeliner with a planning system, the simplest replanning scheme was chosen. Constructing an entirely new plan during the replanning phase simplifies to the problem that has already been solved of constructing a plan from the current state to the goal.

4.10 Design Review

An overview of how information flows between the control layer, sequencing layer, and deliberative layer is presented in Figure 4-12 below:

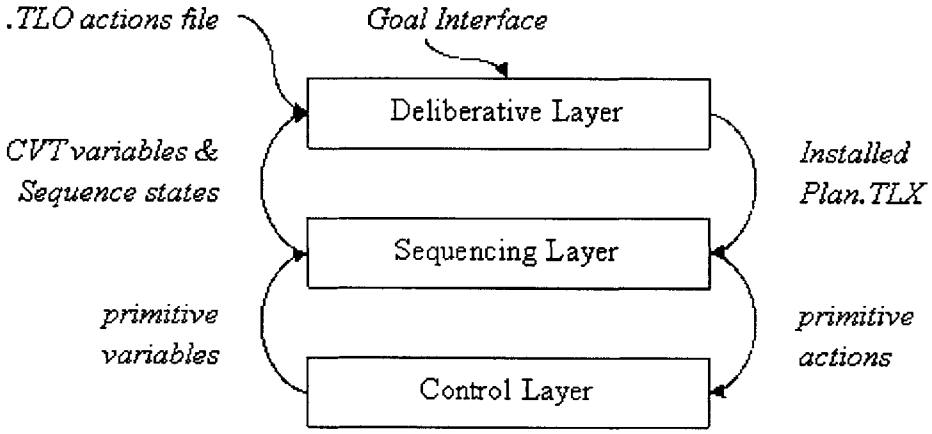

Figure 4-12: Review of information flow

To review, the sequencing layer interacts with the control layer through primitive actions and variables as specified in the original Timeliner design. The deliberative layer receives a high-level goal that is manually specified. It receives a `.TLO` file of Timeliner action sequences generated by compiling the `PRECONDITION` and `POSTCONDITION` statements from Timeliner scripts. It also receives information about the state of the world through CVT. The deliberative layer uses GRAPHPLAN to construct a plan to reach the goal, and then encodes this plan in a Timeliner script. This script is compiled into a `PLAN.TLX` file, which is installed in the Timeliner Executor on a target system. The deliberative layer monitors plan execution by watching Timeliner sequence status statements. If any sequence has a precondition or postcondition error, a replan is initiated that constructs a new plan from scratch.

4.11 Precondition and Postcondition implications

Adding `PRECONDITION` and `POSTCONDITION` statements to Timeliner has implications even outside of integration with a deliberative layer. For example, `PRECONDITION`

statements can be used for increased safety even without the planner. Timeliner is relied on for mission critical applications such as the International Space Station, which might benefit from making verifiable assertions about sequences.

Adding `PRECONDITION` and `POSTCONDITION` statements to Timeliner has an additional cost to Timeliner script writers. The script writers must encode the results of sequences in terms of underlying state variables. However, this extra burden has the potential for increased safety as well, because it will force script writers to verify that all paths to complete the sequence successfully will generate the specified state.

There is currently no automated checking of `PRECONDITION` or `POSTCONDITION` statements. For example, a malicious script writer could write a sequence with a `PRECONDITION` that variable A is both `TRUE` and `FALSE`. This would compile, and a `.TLO` file would be created. However, this precondition error would result in the sequence never being called by the planner, because the precondition statement will never be satisfied. If a script writer makes a mistake and improperly specifies the preconditions or postconditions of a sequence, this is akin to making a logic error in programming the sequence.

Chapter 5

Implementation

This chapter details the implementation changes necessary to create the integrated Timeliner and autonomous planner prototype. This chapter then presents a demonstration that was constructed to showcase the system.

5.1 Extending the TLToolbar with Planner GUI

The Timeliner Toolbar presented in Section 2.7 was modified to include a new `Planner` mode. This mode is a graphical interface to the autonomous planner. The `Planner` mode automatically starts the CVT Monitor, and also starts a simple Java Swing window for interfacing with the planner. This window contains a textfield for inputting the goal state, one button named `Plan`, and a text output area for displaying messages from the planner. When a goal state is specified and the `Plan` button is pressed, the autonomous planner begins to construct a new plan. After the planning is complete, four pieces of information are displayed in the planning window:

1. The initial state as a conjunction of booleans.

2. The goal state as specified in the goal state textfield.

3. A list of action operators available to the planner.

4. The plan to achieve the goal as a Timeliner script.

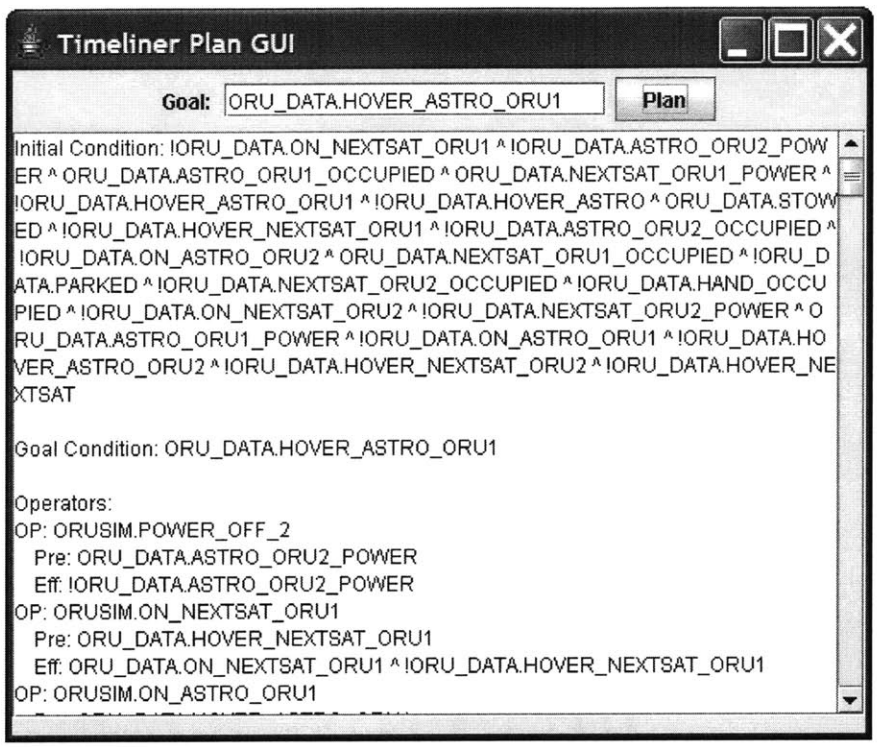

Figure 5-1: The Timeliner Plan GUI

An example of the Timeliner Plan Graphical User Interface after specifying a goal state and pressing the `Plan` button is presented in Figure 5-1.

The Timeliner Plan GUI provides a simple interface for creating new plans and observing the internal state of the planner.

5.2 Interfacing the CVT Monitor with the Planner

As discussed in Section 4.6, the current state of the world will be communicated to the planner through CVT. To implement this, the Timeliner Plan GUI simply stores a pointer to the CVT Monitor that it creates. Whenever the Timeliner Plan GUI needs to know the current state of the world, it calls a `getPUIs()` method that returns a list of every variable in CVT, as well as the type and value of the variables. Next, the non-boolean variables are removed from the list, because the planner used in this prototype considers only booleans. Each remaining boolean in the list is conjoined to create the initial state. For example, if the CVT Monitor contains boolean variable

A which is currently `TRUE`, boolean variable B which is currently `FALSE`, and integer variable C which is currently `17`, then the current state will be `A ^ !B`, where the `!` symbol stands for `NOT`.

The only change to the CVT Monitor code that was necessary for this functionality was to create a single public method for accessing the list of variables, their types, and their values. This list was already present in the CVT Monitor, but as a private variable, because no external program accessed it.

5.3 Parsing .TLO Files

Parsing the `.TLO` Timeliner Operator files is straightforward because the `.TLO` file format has been designed to be easy to parse. As each line of the file is read, the parser first reads out the operator name. The operator is created if it does not already exist. Then, the boolean expression is conjoined to either the precondition or postcondition conjunction of the operator depending on the specified type. This process continues until all lines are read.

5.4 Generating .TLS Plan Files

Section 4.8 specified a way to encode a plan into a Timeliner script. This implementation leaves open the possibility of different types plan formats by defining an abstract `PlanParser` with a single abstract method `parsePlan(Plan p)`. This method takes in a `Plan`, which is a list of steps where each step is a list of actions. The `parsePlan` method returns a String encoding of the plan. The `TimelinerPlanParser` class extends `PlanParser` and encodes plans in the format specified in Section 4.8.

5.5 Compiling and Installing Plans

The `parsePlan` method described in the previous section is used to get a String encoding of a plan in terms of a Timeliner script. This String is written to a file named

PLAN.TLS, which is created if it does not exist. Next, the external Timeliner Compiler and Mapper executables are used on this file to generate the PLAN.TLX executable file. To install the PLAN.TLX executable file, a class called the TimelinerBridge was created.

The TimelinerBridge serves as an interface to the Timeliner Executor. It is based on the backend of the Timeliner Display presented in Section 2.7. The TimelinerBridge has simple methods for sending commands to the Timeliner Executor through sockets. The installBundle method takes a TLX File as input and sends a Command Message to the Timeliner Executor to install this file. The removeBundle method creates and sends a message to remove a specified bundle.

The first time that a plan is generated, the TimelinerBridge is used to first install a bundle containing the Timeliner action sequences available to the planner. Next, the TimelinerBridge is used again to install the PLAN.TLX file. When subsequent plans are generated either through specifying a new goal for the system or through an automated replan, the old PLAN bundle is removed by the TimelinerBridge before the new plan is installed.

5.6 The ORU Satellite Demonstration

The design and implementation chapters until this point have described a complete proof-of-concept autonomous system using Timeliner. This system integrates Timeliner with a deliberative planning layer. The deliberative planning layer receives execution status and sensor observations from Timeliner, and then generates plans in terms of Timeliner action sequences, which are then carried out in a target system by the Timeliner Executor. A demonstration was created to test and showcase this system.

This demonstration is based on a real problem faced by the Orbital Express Satellite. In this problem, two Satellites named Astro and Nextsat are docked. Each satellite has two ORU bays that can either be powered on or off and each ORU bay can either be occupied with an ORU or unoccupied. The Astro satellite has a robotic arm

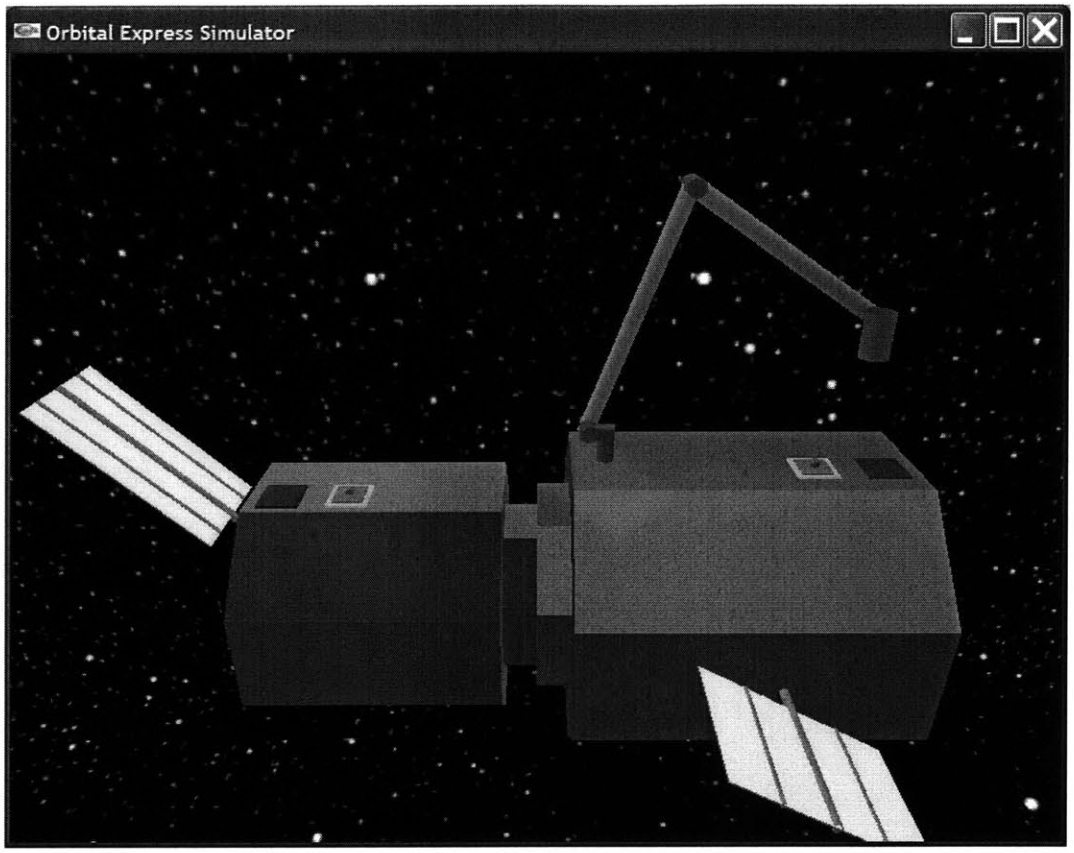

Figure 5-2: ORU Satellite Simulation Frontend

that can move in a predefined ways. The frontend of the demonstration is presented in Figure 5-3. In this figure the robotic arm is hovering over the Astro satellite, each satellite has one ORU bay that is powered and occupied, and the other ORU bay on each satellite is empty and unpowered. Astro is on the right and Nextsat is on the left. This simulation is named the `OruSim`.

When the robotic arm is moved on top of an ORU unit, it can grab the ORU. The robotic arm can then transport the ORU to a new bay and release it. An ORU bay must be powered off before an ORU is removed from it.

Figure 5-3 presents an overview of the actions available in this simulation. The robotic arm can be in one of twelve states. In the Stowed state the arm is folded flat on the Astro satellite. In the Parked state the arm hovers vertically between the satellites. In the Hover Astro and Hover Nextsat states the arm hovers above the respective satellite. In the Hover Astro Oru1, Hover Astro Oru2, Hover Nextsat

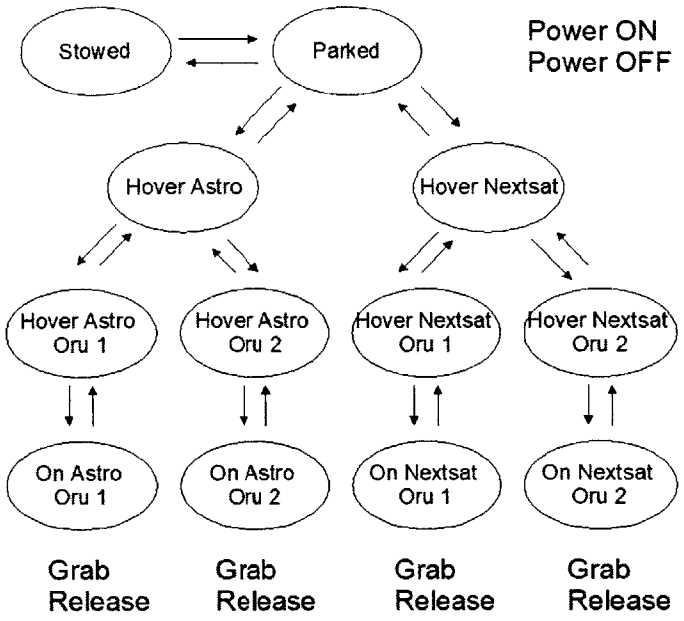

Figure 5-3: ORU Satellite Simulation Action Overview

Oru1, and Hover Nextsat Oru2 states the arm hovers above the specified ORU bay. In the On Astro Oru1, On Astro Oru2, On Nextsat Oru1, and On Nextsat Oru2 states the arm is directly on top of an ORU bay, and will be able to `GRAB` an ORU if the bay is occupied, and `RELEASE` an ORU if the arm is holding one. The valid transitions between arm states are limited, and are depicted graphically in Figure 5-3. The only valid transition from the Stowed state is to the Parked state. From the Parked state, the valid transitions are back to the Stowed state, to Hover Astro, and to Hover Nextsat. Robotic arm transitions not displayed on the diagram are illegal. For example, the robotic arm cannot move directly from the Stowed state to the Hover Astro state. The arm must pass through the Parked state to make this transition.

The `POWER_ON` and `POWER_OFF` commands operate independently of arm positions and can be be executed at any time for any ORU bay.

5.7 Simulation Implementation

The frontend three-dimensional display for the simulation shown in Figure 5-3 already existed before this thesis work began, but it could not be controlled by Timeliner. A Timeliner Simulation backend was created. This backend enables the simulation to function like a target system, receiving commands from the Timeliner Executor and simulating changes to the world state as a result of these commands and the passage of time.

5.7.1 Simulation Ground Database

The Ground Database for the Timeliner implementation of this Simulation was created first to specify how the underlying simulation state would be represented, and to specify which commands would be accepted. The Ground Database for this simulation is named `ORU_DATA`.

The state of the simulation is represented as twenty-one booleans. There are twelve booleans for each of the possible arm states (`STOWED`, `PARKED`, `HOVER_ASTRO`, `HOVER_NEXTSAT`, `HOVER_ASTRO_ORU1`, `HOVER_ASTRO_ORU2`, `HOVER_NEXTSAT_ORU1`, `HOVER_NEXTSAT_ORU2`, `ON_ASTRO_ORU1`, `ON_ASTRO_ORU2`, `ON_NEXTSAT_ORU1`, `ON_NEXTSAT_ORU2`). There are four booleans for whether each ORU bay is powered (`ASTRO_ORU1_POWERED`, `ASTRO_ORU2_POWERED`, `NEXTSAT_ORU1_POWERED`, `NEXTSAT_ORU2_POWERED`). There are four booleans for whether each ORU bay is occupied (`ASTRO_ORU1_OCCUPIED`, `ASTRO_ORU2_OCCUPIED`, `NEXTSAT_ORU1_OCCUPIED`, `NEXTSAT_ORU2_OCCUPIED`). And there is one boolean for whether the robotic arm is occupied (`HAND_OCCUPIED`).

More compact representations of the system state were considered, such as having a single `ARM_STATE` integer where each integer value corresponded to a position of the robotic arm. However, because the preconditions and postconditions of the Timeliner action scripts must be written in terms of the Timeliner system state variables, using a more compact state representation would lead to undesired complexity in the Timeliner `PRECONDITION` and `POSTCONDITION` statements.

Integer State	Action
0	STOW
1	PARK
2	HOVER_ASTRO
3	HOVER_ASTRO_ORU1
4	HOVER_ASTRO_ORU2
5	ON_ASTRO_ORU1
6	ON_ASTRO_ORU2
7	HOVER_NEXTSAT
8	HOVER_NEXTSAT_ORU1
9	HOVER_NEXTSAT_ORU2
10	ON_NEXTSAT_ORU1
11	ON_NEXTSAT_ORU2
12	GRAB
13	RELEASE

Table 5.1: ARM Command Integer State and Actions Mapping

There are six total commands for interacting with the simulation. The `RESET` command restarts the simulation to its initial state. There are four commands for controlling power to the ORU bays: `ASTRO_ORU1_POWER`, `ASTRO_ORU2_POWER`, `NEXTSAT_ORU1_POWER`, and `NEXTSAT_ORU2_POWER`. Each of these four commands takes a boolean `NEW_STATE` parameter. The specified ORU bay will be powered on if a `TRUE` or `ON` value is received, and will be powered off if a `FALSE` or `OFF` value is received. The last command, `ARM`, controls the robotic arm. It has a single integer `STATE` parameter that specifies the desired state of the robotic arm. Each integer value between zero and thirteen corresponds to a different arm action. The mapping between integer `STATE` parameters in the `ARM` command and the effect on the robotic arm is presented in Table 5.1.

5.7.2 Timeliner Simulation Backend

The Timeliner Java Simulation Framework makes it easy to create Timeliner simulations. These simulations act like target systems and allow for testing of Timeliner scripts. Simulations receive command packets from the Timeliner Executor, and can respond to these commands by updating interval variables. Simulations can also update interval variables over time, simulating the changes in a real target system.

Traditionally, simulations made using the Timeliner Simulation Framework have a backend to handle command messages and update the world state, and a Java Swing frontend for graphically displaying the state of the simulation. For this demonstration, the existing external three-dimensional frontend was used to display the system state, so the Timeliner simulation was only responsible for the backend component and for passing messages to the external frontend.

When the simulation backend is started, it initializes the world state to have the robotic arm in the Stowed position, Astro Orul occupied and powered, and Nextsat Orul occupied and powered. Figure 5-4 presents the initial state of the simulation, showing the external frontend on the left, and the state variable values on the right, using the CVT Monitor. Notice that the external frontend and the underlying system state is synchronized: the Stowed state is true, and the arm is drawn stowed. This initial synchronization is achieved by simply setting the underlying world state to be the same as the initial state of the frontend, although the Timeliner simulation will enforce this synchronization over time.

The simulation backend handles the commands specified in the Ground Database by updating the underlying system state and by passing the request on to the external frontend. The external three-dimensional frontend accepts simple commands over the User Datagram Protocol (UDP). For example, if the String "STOW" is sent over UDP to the frontend, it will initiate an arm transition from its current location to the Stowed position. The backend Timeliner Simulation merely needs to accept commands from the Timeliner Executor, and then request the desired state transition from the external frontend through UDP. The backend Timeliner simulation also

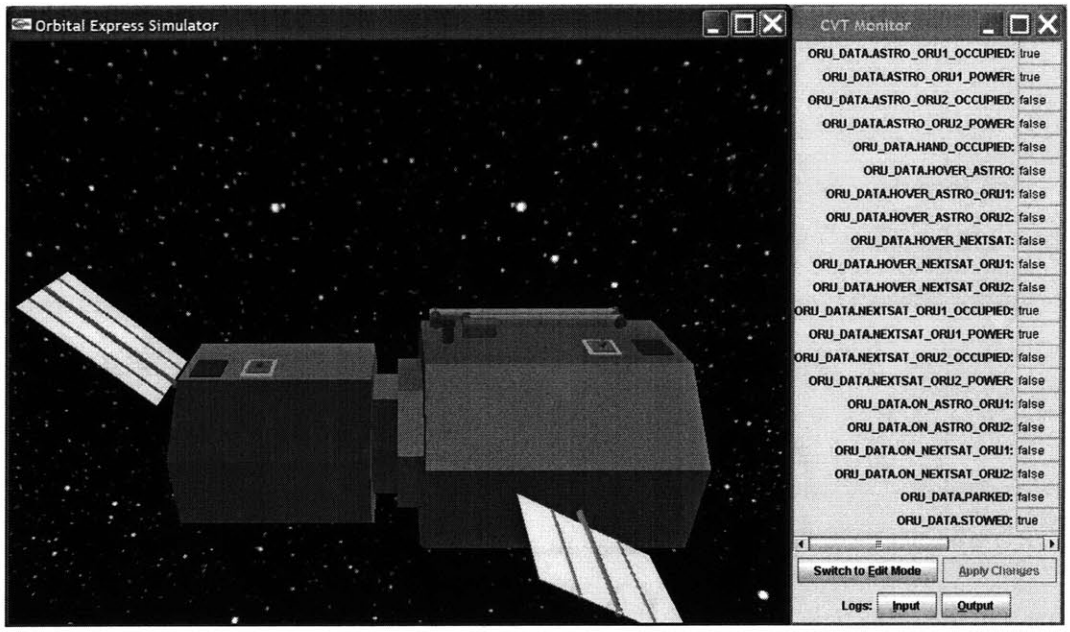

Figure 5-4: Simulation Initial State

updates internal state variables in response to these commands.

5.7.3 Timeliner Action Sequences

The completed Timeliner simulation functions in the same way as a target system. Timeliner scripts can read state variables and issue commands that affect the simulation. Timeliner sequences were constructed to act on this simulation. For example, one sequence named `PARK_1` initiates a transition from the Stowed state to the Parked state. This sequence sends a command to the simulation that results in the robotic arm moving to the parked position, the `STOWED` variable becoming `FALSE`, and the `PARKED` variable becoming `TRUE` after the transition is complete.

Figure 5-5 presents three snippets from the `.TLO` file generated by compiling the Timeliner action sequences for this simulation. The first entry contains the preconditions and postconditions of the `PARK_1` sequence just described. This sequence has a precondition that the arm is Stowed, and postconditions specifying that the arm is Parked but no longer Stowed. It would be easy to forget the second postcondition, that the arm is no longer Stowed, because normal human beings implicitly under-

```
...
ORUSIM.PARK_1, PRE, ORU_DATA.STOWED
ORUSIM.PARK_1, POST, ORU_DATA.PARKED !ORU_DATA.STOWED
...
ORUSIM.POWER_ON_1, PRE, !ORU_DATA.ASTRO_ORU1_POWER
ORUSIM.POWER_ON_1, POST, ORU_DATA.ASTRO_ORU1_POWER
...
ORUSIM.GRAB_2, PRE, !ORU_DATA.HAND_OCCUPIED ORU_DATA.ON_ASTRO_ORU2
ORUSIM.GRAB_2, PRE, ORU_DATA.ASTRO_ORU2_OCCUPIED !ORU_DATA.ASTRO_ORU2_POWER
ORUSIM.GRAB_2, POST, ORU_DATA.HAND_OCCUPIED !ORU_DATA.ASTRO_ORU2_OCCUPIED
...
```

Figure 5-5: .TLO File for Simulation

stand that the arm can only be in one place at once. However, if this postcondition is forgotten, the planner will believe that by using the PARK_1 sequence it can have an arm in both the Stowed and Parked position at the same time. This example highlights the importance of explicit and correct PRECONDITION and POSTCONDITION statements.

Another sequence presented in Figure 5-5 is the POWER_ON_1 sequence that powers on the first Astro ORU bay. The only precondition of this sequence is that the bay is currently powered off, and the only postcondition is that the bay is powered on. A different design of the POWER_ON_1 sequence would have no precondition, and would simply do nothing if the sequence were called with the bay already powered. The main advantage of specifying a precondition for this sequence is that the sequence will not even be considered except when it is useful.

The last sequence presented in Figure 5-5 is the GRAB_2 sequence that will pick up an ORU in the second Astro ORU bay if possible. For this sequence to be considered, the arm must be in the ON_ASTRO_ORU2 position, the bay must be powered off, the hand must be currently unoccupied, and the bay must have an ORU unit occupying it. Executing this sequence will result in the hand becoming occupied, and the second Astro ORU bay becoming unoccupied.

Plan Step	Timeliner Action Sequence
1	PARK_1
2	HOVER_ASTRO_1
3	HOVER_ASTRO_ORU1_1

Table 5.2: Plan for moving the arm to the HOVER_ASTRO_ORU1 state

```
BUNDLE PLAN
      SEQUENCE ONE ACTIVE
            START ORUSIM.PARK_1
            WHEN ORUSIM.PARK_1.SEQSTAT = SEQ_FINISHED
                  START ORUSIM.HOVER_ASTRO_1
                  WHEN ORUSIM.HOVER_ASTRO_1.SEQSTAT = SEQ_FINISHED
                        START ORUSIM.HOVER_ASTRO_ORU1_1
                  END WHEN
            END WHEN
      CLOSE SEQUENCE
CLOSE BUNDLE
```

Figure 5-6: PLAN.TLS script to move the arm to the HOVER_ASTRO_ORU1 state

5.8 Examples

The integrated Timeliner and autonomous planning system was demonstrated by constructing plans in the ORU Satellite simulation. This section presents two example plans. The first plan moves the arm from the initial state to a state where it is hovering over the first Astro ORU bay. The second plan occupies the second Astro ORU bay by removing the ORU unit from the first Astro ORU bay and moving it to the second ORU bay.

5.8.1 Moving the Arm

In the first example, the simulation starts in the initial state, and is given the goal that the arm is in the HOVER_ASTRO_ORU1 state. The deliberative layer constructs a plan to achieve this goal. This plan is presented in Table 5.2. The PLAN.TLS Timeliner script for executing this plan is presented in Figure 5-6.

The plan simply moves the robotic arm through valid transitions to the goal. Each step of the PLAN.TLS Timeliner script starts a Timeliner action sequence. The

Plan Step	Timeliner Action Sequence(s)
1	ON_ASTRO_ORU1, POWER_OFF_1
2	GRAB_1
3	HOVER_ASTRO_ORU1_2
4	HOVER_ASTRO_2
5	HOVER_ASTRO_ORU2_1
6	ON_ASTRO_ORU2
7	RELEASE_2

Table 5.3: Plan to occupy the second Astro ORU bay

execution of this plan is depicted graphically in Figure 5-7.

5.8.2 Moving an ORU

In the second example, the planner is given a goal that the second ORU bay on the Astro Satellite should be occupied. The current state of the world is the result of executing the previous example, so the arm is initially hovering over the first Astro ORU bay. The planner constructs a plan to achieve its goal by moving the ORU unit from the first Astro ORU bay to the second. This plan is presented in Table 5.3. The `PLAN.TLS` Timeliner script for executing this plan is presented in Figure 5-8. The execution of this plan is depicted graphically in Figure 5-9.

Notice that in the first step of the plan, two actions are executed in parallel: the robotic arm moves onto the first ORU bay, and the power is simultaneously shut off. This demonstrates parallel sequence execution using Timeliner. It is possible in this case because the robotic arm and the power control do not depend on each other in any way.

There is more than one way that the planner could have satisfied the goal of occupying the second Astro ORU bay. For example, it could have constructed a different plan where the ORU unit was taken from the other satellite. The reason this plan was chosen instead is that this plan is shorter, because the arm is already closer to the ORU unit in the first Astro ORU bay. Either plan would achieve the

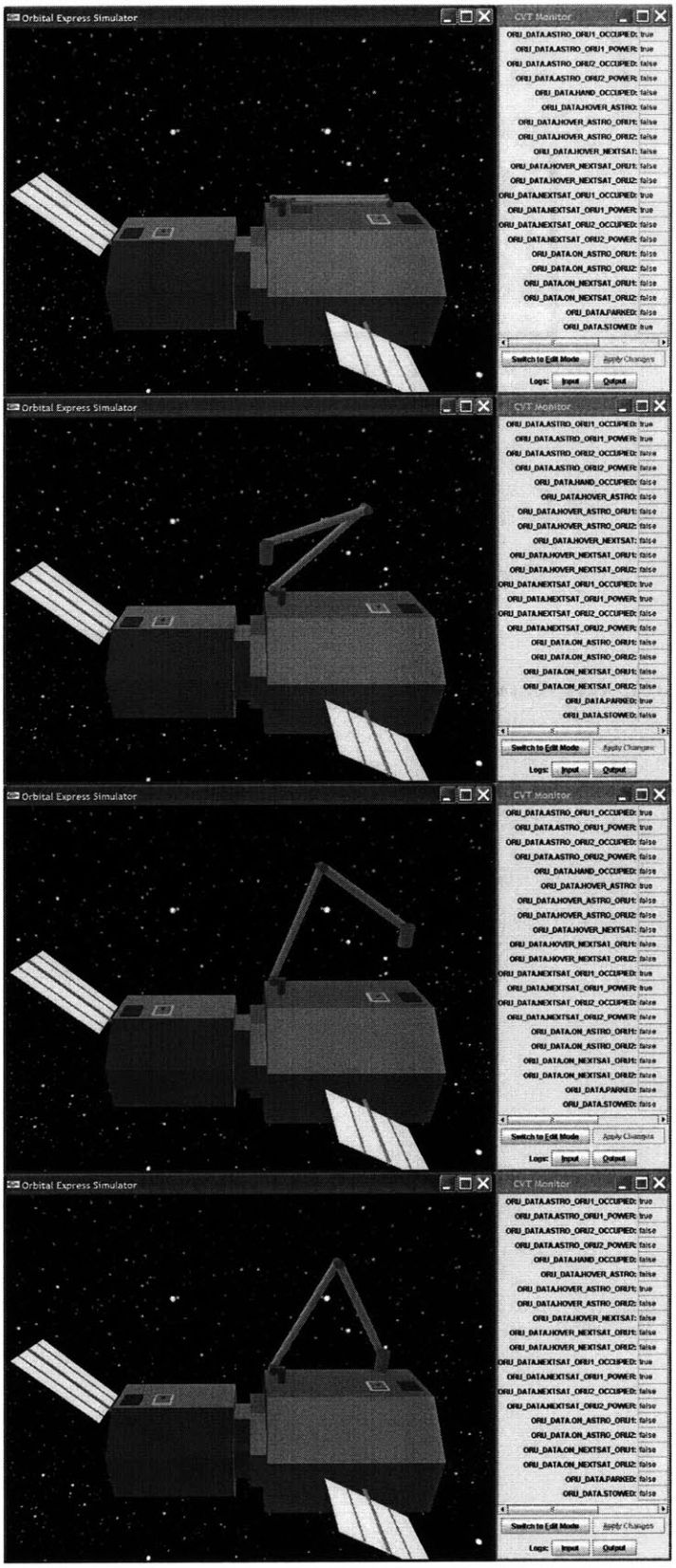

Figure 5-7: Frontend View of Moving the Arm

```
BUNDLE PLAN
      SEQUENCE ONE ACTIVE
            START ORUSIM.ON_ASTRO_ORU1
            START ORUSIM.POWER_OFF_1
            WHEN ORUSIM.ON_ASTRO_ORU1.SEQSTAT = SEQ_FINISHED AND
ORUSIM.POWER_OFF_1.SEQSTAT = SEQ_FINISHED
                  START ORUSIM.GRAB_1
                  WHEN ORUSIM.GRAB_1.SEQSTAT = SEQ_FINISHED
                        START ORUSIM.HOVER_ASTRO_ORU1_2
                        WHEN ORUSIM.HOVER_ASTRO_ORU1_2.SEQSTAT = SEQ_FINISHED
                              START ORUSIM.HOVER_ASTRO_2
                              WHEN ORUSIM.HOVER_ASTRO_2.SEQSTAT = SEQ_FINISHED
                                    START ORUSIM.HOVER_ASTRO_ORU2_1
                                    WHEN ORUSIM.HOVER_ASTRO_ORU2_1.SEQSTAT = SEQ_FINISHED
                                          START ORUSIM.ON_ASTRO_ORU2
                                          WHEN ORUSIM.ON_ASTRO_ORU2.SEQSTAT = SEQ_FINISHED
                                                START ORUSIM.RELEASE_2
                                          END WHEN
                                    END WHEN
                              END WHEN
                        END WHEN
                  END WHEN
            END WHEN
      CLOSE SEQUENCE
CLOSE BUNDLE
```

Figure 5-8: PLAN.TLS script to occupy the second Astro ORU bay

goal, but the chosen plan does it in the minimal number of steps. One way to force the planner to use the ORU unit from the other satellite would to give it a new goal of having both Astro ORU bays occupied. The only way to achieve this would be to retrieve the ORU unit from the other satellite. However, because the valid robotic arm transitions require so many individual steps, this plan would require at least ten steps, which is deeper than the maximum plan depth of the prototype system.

5.9 Sequence Abstraction

One of the difficult design issues when constructing Timeliner action sequences to use with this prototype system is deciding what level of abstraction and complexity to use. For example, in the Satellite ORU simulation, each robotic arm action is a single transition between valid arm positions. Alternatively, a high level action sequence could be constructed that encapsulated several small transitions. For example, a single action sequence could be made that would transition from the first Astro ORU bay to the first Nextsat ORU bay. This single sequence would move from the ON_ASTRO_ORU1 state to HOVER_ASTRO_ORU1 to HOVER_ASTRO to PARKED to

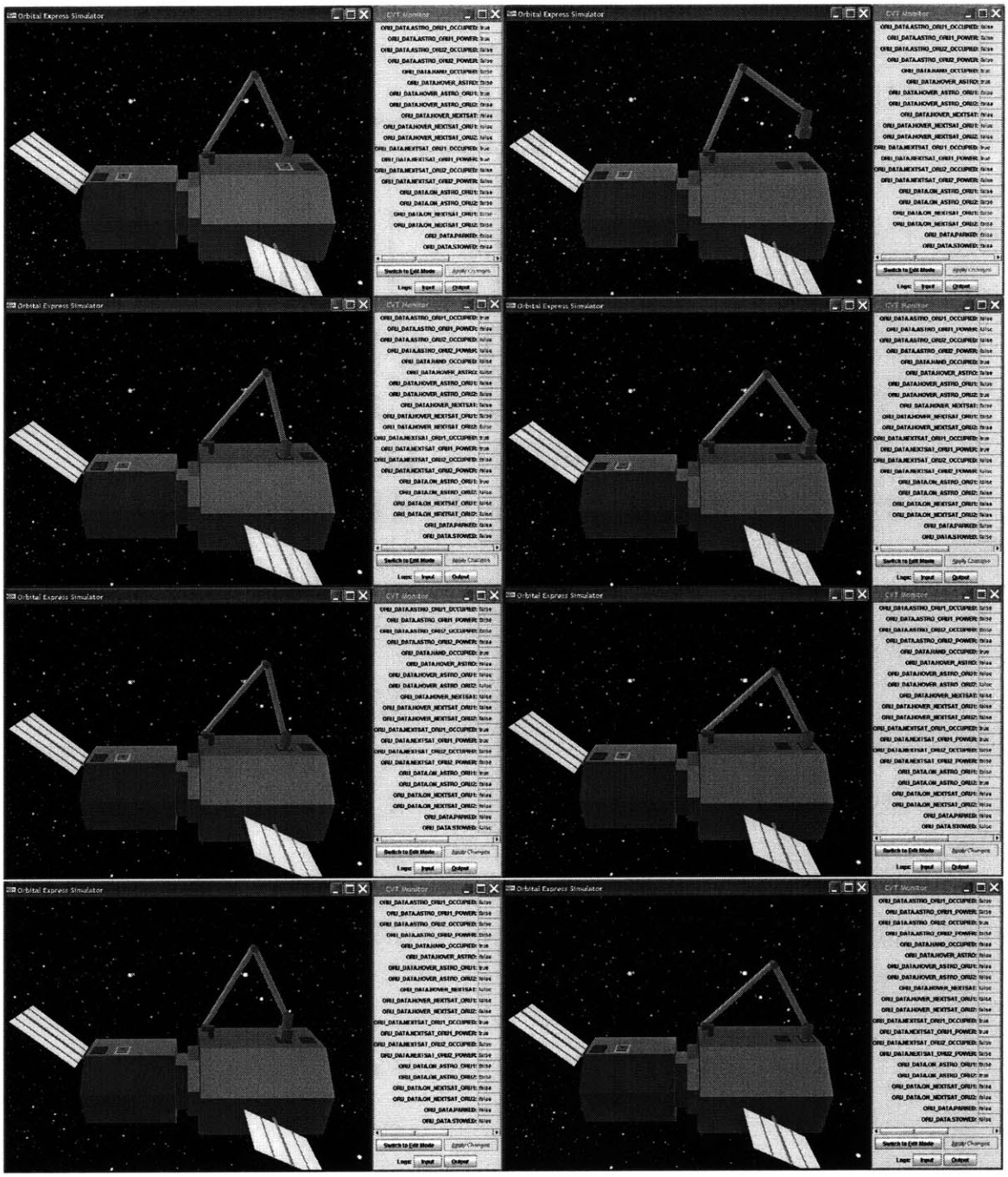

Figure 5-9: Frontend View of Moving an ORU

HOVER_NEXTSAT to HOVER_NEXTSAT_ORU1 to ON_NEXTSAT_ORU1. Thus what would have previously required six steps in a plan would become a single step. This requires additional work from the human-script writers, because they must manually write the script. However, it provides a significant savings in terms of the depth of a plan to move from the Astro satellite to the Nextsat satellite.

Timeliner script writers do not necessarily need to pick a single level of abstraction or complexity. The action sequences could include both simple low-level actions and complex high-level actions, and the planner will select the appropriate actions for achieving its goal in the fewest steps possible. The only additional cost from this approach is an increase in the breadth of actions that the planner must consider at each step in the plan. However, due to the exponential growth nature of the planning problem, it will often be a good idea to include abstract sequences that can shorten the depth of plans that must be generated to achieve the goals of the system.

Chapter 6

Future Work

This chapter discusses the limitations of the prototype system by considering example problems that would not be well suited to the system. The chapter ends by presenting areas for future work in the integration of Timeliner and autonomous planning.

6.1 Limitations

The integrated Timeliner and autonomous planning system was well suited to the Satellite ORU simulation problem. This simulation had an underlying state that could easily be represented with booleans. Although there were a variety of arm position states and transitions, there was only a few actions to consider from any given state. These factors made it relatively simple to construct Timeliner action sequences with `PRECONDITION` and `POSTCONDITIONS` statements. GRAPHPLAN also had little problem creating short plan sequences such as the presented examples of moving the arm or transporting an ORU unit.

The next four subsections discuss areas where the prototype system would struggle.

Plan Depth	Time to Construct Plan
8	0.17 seconds
9	0.28 seconds
10	26.14 seconds
11	810.11 seconds

Table 6.1: Time Required for GRAPHPLAN to construct plans of various depths in the ORU Satellite simulation

6.1.1 Limited Plan Length

The implementation of GRAPHPLAN used in this system only considers plans up to a depth of ten steps. If allowed, it would be capable of considering slightly longer plans, but the planning process grows exponentially longer with each additional step in the plan. For example, Table 6.1 presents the time required by GRAPHPLAN to construct plans of varying depth in the ORU Satellite simulation. Plans with fewer than eight steps are created almost instantly, but by eleven steps the planner requires on the order of minutes. This prototype system would not be well-suited to a domain that required plans with twenty or more steps. Limited plan depth is a problem faced by all autonomous planners, as planning in general requires exponential time.

6.1.2 Only Booleans

Another disadvantage of the implementation of the implementation of GRAPHPLAN used in this prototype system is that it considers only boolean state variables. In order to plan using integer or numeric variables, these variables must be grounded in terms of booleans. As discussed in Section 4.4.3 an integer variable named `STATE` ranged in value from `0` to `4` must be converted into five booleans: `STATE0`, `STATE1`, `STATE2`, `STATE3`, `STATE4`, and `STATE5`. The prototype system is poorly suited to planning problems with underlying states that are difficult to express in terms of booleans. This boolean limitation is not a fundamental limitation of the integrated system, as Timeliner is capable of evaluating boolean expressions containing numeric terms such

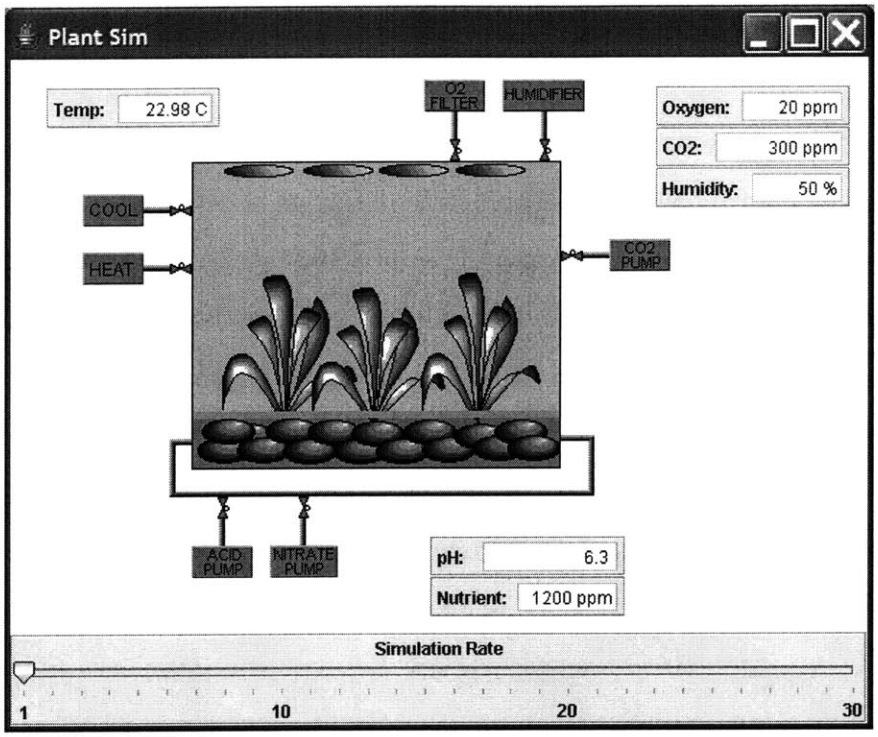

Figure 6-1: The PlantSim Greenhouse Maintenance Simulation

as `STATE > 2`. GRAPHPLAN could be extended to consider boolean variables [6], but this was not attempted in this implementation.

6.1.3 Greenhouse Maintenance Domain

The first simulation problem that the prototype system was considered for involved maintaining a greenhouse. This artificial environment has a number of variables such as carbon dioxide levels, temperature, and pH readings that are maintained. Actions can be taken in the system to affect these variables, such as turning on an acid pump to lower the pH readings or enabling a heater or cooling system.

A Timeliner simulation named `PlantSim` already exists for this problem. This simulation is presented in Figure 6-1. Timeliner action sequences have also already been written for this simulation that attempt to keep each reading in an acceptable range.

This problem would be difficult to adapt to the integrated Timeliner and autonomous planning system. The main challenge is that each maintained variable in

the system is an integer or decimal value. Consider the steps that would be necessary to encode the preconditions and postconditions of a sequence to turn on the acid pump. Suppose the acid pump should be turned on if the pH reading rises above 8, and the effect of turning the pump on is that the pH reading will slowly drop over time. This precondition could be encoded by specifying `PRECONDITION pH > 8`. This expression would need to be grounded for GRAPHPLAN, because it can only handle boolean expressions. In other words, `pH > 8` would become an expression such as `pH8 or pH9 or pH10 or pH11 or pH12 or pH13 or pH14`. Another difficult problem is expressing the concept that the acid pump will decrease the pH level over time. One strategy would be an internal `delta_pH` numeric variable that models how the pH level will change over time. The effect of turning the acid pump on, would be a decrease in the value of `delta_pH`. However, because `delta_pH` is numeric, it will also need to be grounded in terms of boolean expressions. Ultimately, the greenhouse maintenance problem could be handled by the prototype system, but it would require a lot of work because the underlying numeric state is not well suited to the prototype system.

Even though it would be possible to use the prototype integrated autonomous system to control the greenhouse maintenance simulation, it would not provide significant advantages over a Timeliner control script on its own. The greenhouse maintenance simulation has only one type of goal: maintaining a certain range of parameter values. The plans and control sequences in this simple maintenance problem are straightforward. For example when the pH is high, turn on the acid pump; when the carbon dioxide levels are high, turn on the carbon dioxide pump; and when the temperature is low, turn on the heater. This means that it is relatively straightforward for a human script writer to construct sequences that maintain the system state. In other words, planning is not needed in this problem, and integrating with the planner is complex, so the greenhouse maintenance problem is not well suited to the prototype system.

6.1.4 Three Dimensional Movement Domain

Another example problem that this prototype system would be poorly suited for is a three-dimensional motion problem. Imagine a submarine with a latitude, a longitude, and a depth. This submarine has a goal of navigating a three-dimensional space and avoiding objects to reach a set of goal coordinates. The main problem with planning in this domain is that it is hard to encapsulate all the numeric values in booleans. All aspects of this problem such as the position of the submarine or the position of objects or possible movement paths rely on numeric values. The boolean limitation of this implementation of GRAPHPLAN makes a three-dimensional movement problem a bad fit for the prototype system.

6.2 Future Work

The ORU Satellite problem is well-suited to the prototype system because its underlying state can be represented using a small number of booleans, and because a number of different goal states and scenarios are possible. The Limitations section presented a number of situations where the prototype system would perform poorly.

The majority of the presented limitations were the result of this implementation of GRAPHPLAN, such as the required boolean state variables. The majority of the future work in the integration of Timeliner and autonomous planning involves using different planners or replanning schemes with the presented framework. A different planner could be used in this system with minimal modifications. The framework for communicating world state information, action operators, and goals is already present. The new planner would simply be responsible for using this information to construct a new plan, which can then be encoded as a Timeliner script. Similarly, a new replan scheme could be used with few modifications to the overall system.

Timeliner is developed at the Charles Stark Draper Laboratory where the All-Domain Execution and Planning Technology (ADEPT) is also developed. One key area for future work is to use ADEPT as the planning component in an integrated autonomous Timeliner system [3].

Chapter 7

Conclusion

This thesis presented the design, implementation, and demonstration of a prototype system integrating Timeliner and autonomous planning. This system was organized as a Three-Layer Architecture, with the existing Timeliner system used for the control layer and the sequencing layer, and a new deliberative layer added for autonomous planning. This new layer interfaces with Timeliner using existing Timeliner communication infrastructure. The deliberative layer uses an autonomous planner named GRAPHPLAN and a simple replanning scheme that generates new plans from scratch when problems are detected.

An implementation of this system was created, as well as a simulated problem involving the control of a robotic arm on a satellite. The prototype was demonstrated on this simulated problem by providing high-level goals for the system to achieve. The integrated system autonomously created plans to achieve these goals, and then executed these plans using Timeliner. The system was successful in achieving these goals, although there were limitations to the complexity of problems that the system could solve. Ultimately, this thesis presents a proof-of-concept demonstration that Timeliner can be used effectively in an autonomous planning system.

Bibliography

[1] Gautam Biswas, Pete Bonasso, Sherif Abdelwahed, E. J. Manders, David Kortenkamp, Jian Wu, and Scott Bell. Requirements for an autonomous control architecture for advanced life support systems. In *Proceedings 35th International Conference on Environmental Systems*, 2005. SAE Paper Number 2005-01-3010.

[2] A. L. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference an Artificial Intelligence*, pages 1636–1642, Montreal, 1995. Morgan Kaufmann.

[3] Robert A. Brown. Automating space operations using timeliner and adept. PowerPoint Presentation presented at Houston AIAA ATS 2005, March 2005.

[4] Maya Dobuzhskaya. Timeliner integrated development environment. Master's thesis, Massachusetts Institute of Technology, June 2005.

[5] Erran Gat. *Three-layer Architectures*, chapter 8, pages 195–210. Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems. MIT Press, Cambridge, MA, 1998.

[6] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*, pages 123–139. Morgan Kaufmann, 2004.

[7] Stuart J. Russell and Peter Norvig. *Artificial Intelligence A Modern Approach*, pages 395–402. Prentice Hall, second edition, 2003.

[8] The timeliner user interface language (uil) system for the international space station. Technical report, NASA, The Charles Stark Draper Laboratory. http://timeliner.draper.com/docs/971106_ISS_TL_WRITEUP.pdf.

[9] International space station user interface language users guide. Technical Report 306642 Rev 2, The Charles Stark Draper Laboratory, Cambridge, MA, March 1999. http://timeliner.draper.com/docs/306642r2_ug.pdf.