Definition of Projects and Scheduling Blocks from a Scheduling Perspective

Allen Farris
NRAO
Version 1.0, Draft
June 5, 2003

This is an ALMA software discussion document.

1 Introduction

This paper addresses the structural definition and some implementation details of the definition of observing projects and scheduling blocks from the perspective of the Scheduling Subsystem in ALMA. The purpose of this exercise is not to completely define projects and scheduling blocks, but rather to define those features specifically needed by scheduling. A secondary purpose is to allow the Scheduling Subsystem to begin its work on developing a stand-alone simulator package. In addition, it is hoped that this discussion will aid the process of crafting and implementing the actual definitions of ObsProject and SchedBlock in the Observation Preparation Subsystem.

This discussion is also a summary of various points that have been raised in the course of design documents and comments on those documents.

It is intended as a working document. Some portions have been implemented and tested; others are completely undefined. It is undoubtedly incomplete, and additions and changes will be made.

The Java programming language is used to discuss the definition and implementation details. In all of the examples of class definitions, I have omitted “getter” and “setter” methods to condense the code examples.

Throughout this discussion, in the definition of observing projects and scheduling blocks, I have avoided using names that conflict with those in the Observation Preparation Subsystem. So, the names I have chosen are SProject, SUnitSet, and SUnit. These correspond to the ObsProject, ObsUnitSet, and SchedBlock – from a scheduling perspective.

2 Definition of SProject

The definition of the SProject class is presented below.

```
/**
 * An SProject is an observing project as viewed by the
 * scheduling subsystem.
 */
public class SProject {
      private String obsProjectId;
      private String projectName;
      private String PI;
      private STime dateOfSubmission;
      private STime startTime;
      private STime endTime;
      private int totalRequiredTimeInSeconds;
      private int totalUsedTimeInSeconds;
      private int totalUnits;
      private int numberUnitsCompleted;
      private int numberUnitsFailed;
      private Status projectStatus;
      private boolean breakpoint;
      private SUnitSet unitSet;

      /**
       * Construct an SProject from an ObsProject object.
       */
      public SProject(ObsProject project) {
            // Details not implemented.
      }
}
```

Figure 1 – Definition of SProject

In order to define a project we need the observing project's entity id, which ties it to the archive, the name of the project, the PI, and its date of submission. This date is the time it was submitted to the archive as a completely defined project that is ready to be executed. In addition, various fields denote global properties of the project and its progress in being executed. Its starting and ending times are recorded. The total time required to execute the project is the cumulative total of the maximum times to execute all its scheduling blocks. Likewise, the total number of scheduling blocks in the project is recorded. As the project is executed fields are updated, giving the total time used in executing the project, the number of scheduling blocks that have been executed, the status of the project and whether or not the project is in a breakpoint.

An SProject is a hierarchical structure. Contained in the SProject is the top-level SUnitSet.

2.1 STime

The STime class and its companion class, TimeInterval, are special classes used within the Scheduling Subsystem. They have been implemented and tested. The STime class provides for a moment in time accurate to the nearest second[1]. String input and output are in the FITS date-time format. The FITS format is of the form:

YYYY-MM-DDThh:mm:ss

Leading zeros are required if months, days, hours, minutes, or seconds are single digits. The value for months is from "01" to "12". The only departure from the FITS standard is to allow the "T" between the data and time values to be optional in constructing an STime from a string.

2.2 Status

The status class is simply an enumeration of states, defined as follows.

Waiting	Not completely defined or otherwise not ready to execute[2].
Ready	Ready to execute, but execution has not started.
Running	The process of execution has been started, but is not complete.
Aborted	Execution has ceased and one or more portions failed.
Complete	Execution is complete and successful.

As applied to a project, the status denotes the status of the project as a whole. In this case, the “running” state means that the project has been started, not that it has one or more components that are now being executed by the control system.

3 Definition of SUnitSet

An SUnitSet is a hierarchical tree of SUnitSet members whose leaves are SUnit objects. Its definition is complex and is presented below.

[1] The Scheduling Subsystem uses two time-related classes, one that is accurate to the nearest second (STime), sufficient for most purposes, and ArrayTime, which is the time generated by the control system, accurate to the nanosecond.

[2] If a project is in a breakpoint and waiting for a response from the PI, the project status is “waiting”.

```
/**
 * An SUnitSet is a hierarchical tree whose leaves are
 * SUnit objects.
 */
public class SUnitSet {

      // The members of this set: either SUnitSets or SUnits.
      private ArrayList member;
      private int name;

      private Priority scientificPriority;
      private Priority userPriority;
      private String dataReductionProcedureName;
      private Status pipelineStatus;
      private FlowControlExpression[] flowControl;
      private NotifyPI notify;
      private Expression scienceGoal;
      private WeatherCondition weatherConstraint;
      private SystemSetup requiredInitialSetup;
      private int maximumTimeInSeconds;
      private Status unitSetStatus;

      public SUnitSet() {
      }
}
```

Figure 2 – Definition of SUnitSet

The SUnitSet class uses a number of other classes in its definition. These will be discussed below.

The first point to make concerns the definition of the members of an SUnitSet. This is an ordered set. Scheduling sees no reason to restrict these to being all SUnitSets or SUnits. In other words, they could be mixtures of SUnitSet and SUnit objects. There may well be reasons outside the Scheduling Subsystem for making this restriction, such as simplifying the observing preparation tool.

The SUnitSet also has a name that uniquely identifies it within the project hierarchy. This name can be a simple integer that is automatically generated by the Observation Preparation tool. One of its uses is in flow control expressions.

3.1 Priority

The Priority class is a simple class that implements a priority scheme that varies from 1 (lowest) to 10 (highest)[3].

The scientific priority must be specified. If it is not specified at a given level, then its value is inherited from the next higher level, and so on to the top level of the hierarchy. The scientific priority must be specified at the top level of the project and applies to the project as a whole. In principle, this scheme allows one to assign different scientific priorities to different portions of a project. Whether such a feature would actually be used is up to the proposal approval process.

There is also a user-defined priority. Its purpose is to allow the user to designate the relative importance of different portions of the observing project. If it is not specified at a given level, then its value is inherited from the next higher level, and so on to the top level of the hierarchy. It need not be defined at the top level, however. If the user priority is not specified, then all scheduling blocks have the same user priority.

3.2 Initiating Data Reduction

If the data reduction procedure name is not null, then the science pipeline is initiated whenever all members of this SUnitSet have a status of "complete" and all members that are themselves SUnitSets have a pipelineStatus of "complete".

3.3 Expressions

Within the definition of an SUnitSet, flow control, PI notification, science goals, and weather constraints all employ expressions. To accommodate this usage, an Expression class has been defined, implemented and tested.

The Expression class accepts a logical expression in the form of a string, parses and stores it using a stack-based postfix notation. There is also a method to evaluate the expression, which yields a boolean. The logical expression has the following definition.

```
Syntax:
     <logical_expression> :=
           a. '(' <logical_expression ')'
           b. '!' '(' <logical_expression ')'
           c. <logical_expression> '&&'  <logical_expression>
           d. <logical_expression> '||' <logical_expression>
           e. <function_name> <comparison_operator> <numeric_value>
           f. <numeric_value> <comparison_operator> <function_name>
     <comparison_operator> := '==' | '!=' | '<' | '<=' | '>' | '>='
     <numeric_value>  := a valid floating point number
```

[3] This range of priority is arbitrary, but ten levels should be adequate to accommodate most proposal priority schemes.

There are two things that must be configured prior to using this class. The first is to supply the names of external functions that can appear in expressions. The second is to configure method names for evaluating those functions. Failure to properly configure these names will result in a null pointer exception, which is intentional. This two-step operation must be done in this order; setting the function names and then setting the method names.

The function names that are part of the logical expression are static strings defined as part of the Expression class. These must be defined before any expressions are parsed. This is done via the static method setFunctionNames. The Expression constructor parses the strings containing these expressions.

Supplying the method names that correspond to the function names is done via the static setMethods method. This must be done prior to any evaluation of any function. Java reflection is used to execute the functions that correspond to the names.

At present the functions themselves are time-dependent[4] functions that take no arguments and return a float[5]. This works for many cases and can be enhanced if viewed as too restrictive. The following example code fragments will be used to illustrate how the class may be used.

```
// First, set the function names.
String[] names = {
      "temperature",
      "humidity",
      "pressure",
      "windVelocity",
      "windDirection"
};
Expression.setFunctionNames(names);

// Now, we set the methods that correspond to the functions.
String[] methods = {
      "scheduling.EnvironmentalModel.getTemperature",
      "scheduling.EnvironmentalModel.getHumidity",
      "scheduling.EnvironmentalModel.getPressure",
      "scheduling.EnvironmentalModel.getWindVelocity",
      "scheduling.EnvironmentalModel.getWindDirection"
};
Expression.setMethods(methods);

// Create an expression object.
String s = "(humidity <= 10.0) &&
            (temperature <= 10.0) && (windVelocity  <= 3.5)");
Expression p = new Expression(s);
// Evaluate an expression object.
System.out.println("expression " + s + " is " + p.evaluate());
```

[4] The time is taken from a configurable clock that denotes the simulated system time.
[5] The values returned by functions of this type are in standard units as recommended by FITS and the IAU Style Manual.

3.4 Flow Control

Flow control is optional and may be used to constrain the order in which members of the set are executed. If there are no flow control expressions, the members may be executed in any order. The "flowControl" field is an array of FlowControlExpressions. Each FlowControlExpression contains the following fields.

```
private int unitName;
private Expression condition;
private int[] dependentUnits;
private TimeInterval waitTime;
```

The "unitName" field identifies the member unit that is being constrained. The "condition" field is a logical expression that must be true before the member can be executed. The "dependentUnits" field is an array of member units that must be complete before this member can be executed. Finally, the "waitTime" is optional and is the time interval that must elapse between the completion of the dependent units and the execution of this member.

3.5 Notification of the PI

The PI may be notified whenever a unit set is complete and when the PI notification expression is true. The NotifyPI class contains the following fields.

```
private boolean isBlocking;
private Expression condition;
```

The "isBlocking" field denotes whether this is a project breakpoint or not. The "condition" expression is optional and is a logical condition that must be true. If there is no logical condition present, the assumed condition is true.

3.6 Science Goals

Science goals need not be stated at the unit set level. However, it may be useful to state them at this level if they can be crafted in such a manner that they apply to all the members of this set.

The relevant science goals are stated using logical expressions of the type previously described. One of the tasks to be completed is to identify relevant functions that are useful in stating the science goals. These functions would then be evaluated using accumulated data from relevant events generated by the calibration and control subsystems.

3.7 Weather Constraints

Likewise, weather constraints need not be stated at the unit set level. However, it may be useful to state them at this level if they can be crafted in such a manner that they apply to all the members of this set.

Weather constraints are difficult to state and this implementation solution attempts to allow a very general method of stating these constraints. We may well define a set of "default" constraints that are much simpler to implement. Nevertheless, the general case is presented here.

The WeatherCondition class defines a measure of the favorable conditions under which a block may be executed. After being created, the principle method is "evaluate()". This method returns a float that is between 0.0 and 1.0. A value of "1.0" means ideal weather conditions and a value of "0.0" means do not execute the block under these conditions. For intermediate values, the higher the value the more favorable is the condition.

The constructor takes an array of strings. Each string is of the form:

```
<logical_expression> -> <target>
```

where target is a floating point number between 0.0 and 1.0. All the environmental variables have names (humidity, temperature, pressure, windVelocity, windDirection, etc.) and these names may appear in the logical expression.

When the favorable condition 'evaluate()' method is called, the logical expressions are evaluated in the order in which they were defined. The first expression that is true, results in the 'target' that is associated with that condition being returned as the favorable condition indicator.

An example will make this clearer.

Suppose we have the following array of strings as defining the WeatherCondition.

```
"humidity <= 10.0 -> 1.0",
"humidity <= 20.0 -> 0.8",
"humidity <= 30.0 -> 0.6",
"humidity <= 40.0 -> 0.4",
"humidity <= 50.0 -> 0.2",
"humidity >  50.0 -> 0.0"
```

If the current value for humidity is 35.25, the "evaluste' method would return 0.4. This result has the meaning that under the current conditions, the favorability rating is 40%.

3.8 The Initial Setup

At the present time the required initial setup is undefined. This class would characterize the system setup, i.e., receivers, frequencies, correlator setup, etc., needed initially by a scheduling block.

Again, this class need not be stated at the unit set level. However, it may be useful to state it at this level if it applies to all the members of this set.

3.9 Maximum Time

The maximum time is the total of the maximum time required to execute each member of this set.

3.10 Status

The status applies to the unit set as a whole and has the same meaning as in section 2.2.

4 Definition of SUnit

The definition of an SUnit is presented below.

```
/**
 * An SUnit is the lowest-level, atomic scheduling unit.
 */
public class SUnit {
      private String schedBlockId;
      private int name;

      private Priority scientificPriority;
      private Priority userPriority;
      private Expression scienceGoal;
      private WeatherCondition weatherConstraint;
      private SystemSetup requiredInitialSetup;
      private int maximumTimeInSeconds;

      private SkyCoordinates coordinates;
      private boolean isStandardScript;
      private Status unitStatus;

      /**
       * Construct an SUnit from an SchedBlock object.
       */
      public SUnit(SchedBlock sb) {
            // Details not implemented.
      }
}
```

Figure 3 – Definition of SUnit

An SUint object is the lowest-level, atomic scheduling unit. These are the leaves of the hierarchical tree that forms a project. The scheduling block entity id ties the object to the archive contents. In addition it has an assigned name, as discussed in conjunction with SUnitSets in section 3.

4.1 Inherited fields

The scientific priority, user priority, science goals, weather constraints, required initial setup are all the same as those discussed in SUnitSets. Of these, user priority and weather constraints are optional. If there are no weather constraints and none specified at higher levels, the scheduling block is not constrained by weather. Scientific priority, science goals, and required initial setups, if not specified, are inherited from higher levels. It is an error if these are not present at least at some higher level.

4.2 Coordinates

This is simply the sky coordinate that is initially required by this scheduling block. It is presently defined in the SchedBlock class.

4.3 Standard Script

The scheduling system needs to know whether this scheduling block contains a standard observing script or whether it contains a so-called “expert” script. Expert scripts may require special consideration by the scheduling subsystem.

4.4 Status

This status of the scheduling block is the same as is defined in section 2.2. In this case, the “running” state does denote that this scheduling block is being executed by the control system.

5 Questions and Issues

The major questions to be answered are the following.

- Are the concepts discussed here sufficiently well defined to use in crafting a scheduling simulator?

- Do the concepts discussed here adequately implement the ideas discussed in conjunction with ALMA projects and scheduling blocks?
- What concepts and classes need enhancement and more detailed refinement?
- Are the concepts discussed here sufficiently general so that observing projects from other observatories[6] could be mapped into these concepts?
- Is the expression evaluation scheme discussed here adequate for capturing user intentions and sufficiently easy to use?

The major issues to be resolved are the following.

- Define a list of functions needed in expressions of the type discussed here.
- Define the parameters need to characterize the required initial setup.
- Get feedback from the Observation Preparation Subsystem on the adequacy of this approach.

[6] Strictly speaking this isn't necessary, of course. However, this is a good test to see if any vital concepts may be missing, as well as being a test of conceptual generality.