The Kepler Science Operations Center pipeline framework extensions

Todd C. Klaus*[a], Miles T. Cote [b], Sean McCauliff [a], Forrest R. Girouard [a], Bill Wohler [a], Christopher Allen [a], Hema Chandrasekaran[c], Stephen T. Bryson[b], Christopher Middour [a], Douglas A. Caldwell [c], Jon M. Jenkins [c]

[a]Orbital Sciences Corporation/NASA Ames Research Center, MS 244-30, Moffett Field, CA, USA 94035-1000

[b]NASA Ames Research Center, MS 244-30, Moffett Field, CA, USA 94035-1000,

[a]SETI Institute/NASA Ames Research Center, MS 244-30, Moffett Field, CA, USA 94035-1000;

ABSTRACT

The *Kepler* Science Operations Center (SOC) is responsible for several aspects of the *Kepler* Mission, including managing targets, generating on-board data compression tables, monitoring photometer health and status, processing the science data, and exporting the pipeline products to the mission archive. We describe how the generic pipeline framework software[1] developed for *Kepler* is extended to achieve these goals, including pipeline configurations for processing science data and other support roles, and custom unit of work generators that control how the *Kepler* data are partitioned and distributed across the computing cluster. We describe the interface between the Java software that manages the retrieval and storage of the data for a given unit of work and the MATLAB algorithms that process these data. The data for each unit of work are packaged into a single file that contains everything needed by the science algorithms, allowing these files to be used to debug and evolve the algorithms offline.

Keywords: Kepler, pipelines, distributed processing, cluster computing, photometry, pipeline, framework

1. INTRODUCTION

The *Kepler* SOC[1] is primarily responsible for analyzing raw pixel data from the *Kepler* spacecraft for up to 170,000 stars with the goal of identifying those stars that have possible transiting exoplanets for further follow-up research[2]. The *Kepler* SOC has developed a number of distributed software pipelines to perform this analysis. These pipelines are built using a pipeline framework[3] that provides a generic, extensible platform for building pipelines. Some of the *Kepler* pipeline modules actually perform this analysis, while others produce products in support of this analysis or the data collection (for example, generating the on-board target and compression tables). Additional pipeline modules support automated testing of the code base, including generation of simulated test data. The *Kepler* pipeline configurations are described in more detail in section 2. The *Kepler* pipelines are implemented using a combination of Java and MATLAB. Java classes store and retrieve data to/from a relational database and the Kepler DB[4], a custom time series database, while the algorithms that process the data are written in MATLAB. The Java/MATLAB interface is described in more detail in section 3. Algorithm inputs and outputs are passed between Java and MATLAB using a custom binary file format. The science algorithm input files are completely self-contained and self-describing. This allows scientists and developers to execute the science algorithms outside of the pipeline without access to the cluster or data store.

2. KEPLER PIPELINES AND UNITS OF WORK

Before discussing the *Kepler* pipelines themselves, it is important to understand how the unit of work is defined for each pipeline module. Raw *Kepler* data consist primarily of 30-minute (known as long cadence) pixel samples for up to 170,000 stellar targets collected from 84 CCD output amplifiers (referred to as a CCD channel or a module/output), resulting in nearly 20 GiB of raw, long cadence pixel data received by the SOC per month. In addition, *Kepler* collects 1-minute (short cadence) samples for up to 512 targets for an additional 4 GiB of raw pixel data per month.

These data sets as well as the intermediate data sets produced by the pipeline must be partitioned into smaller units of work for distributed processing across the cluster. Pipeline module algorithms have data dependencies which impose certain constraints on how data sets can be partitioned. For example, the presearch data conditioning (PDC) module[5] needs data from all stars for a given module/output in order to create ensembles of targets for purposes of detecting and removing systematic errors, but each month of data can be processed independently. In contrast, the transiting planet search (TPS) module[6] can operate on a single target at a time, but needs as many time samples as possible (back to the beginning of the mission) in order to maximize the detection of periodic transits.

Resource utilization imposes another set of constraints on partitioning of data sets. Memory typically sets an upper bound on the size of the dimensions of the data set. CPU cores set an upper bound on the number of units of work (data set partitions) that can be generated. Small units of work result in more efficient load balancing across the cluster than large units of work. However, some algorithms produce better results with a larger data set, so often a compromise must be struck between improving the results and staying within the hardware constraints. For example, outlier detection will work better with more samples.

To this end, the pipeline framework allows the unit of work type to be customized for each pipeline module. The framework allows the application (in this case, *Kepler*) to customize the unit of work by creating what are referred to as unit of work generators[3]. Unit of work generators are responsible for generating a set of unit of work descriptors. Unit of work descriptors are application objects that describe the dimensions of the data set to be processed by each pipeline job. For example, the unit of work descriptor might contain the module/output to process and/or the time range to process. The framework creates a pipeline job for each unit of work descriptor created by the generator. Generators can query the data store for products produced by the previous module. This allows generators to partition data sets differently depending on the previous module's results. For example, the units of work for the *Kepler* data validation module[7] include only the targets for which the transiting planet search produced a positive result. This approach allows the unit of work descriptors produced to be dynamic and data dependent.

The framework allows each module in the pipeline to use a different unit of work generator, but for *Kepler* several modules share the same unit of work type. Several of the unit of work generators developed for *Kepler* and how they map to pipeline modules are described below.

2.1 Kepler Unit of Work Generators

Listed below are some of the custom generators used by the *Kepler* pipeline modules. This is not an exhaustive list, but it does include all of the generators used by the main science processing pipeline modules.

Time Index Range

This generator sub-divides a specified interval of absolute time index numbers (referenced to the start of the mission) into sub-intervals of a configurable size. For example, if a 90-day interval was specified and the maximum size per sub-interval was set to 30-days, then 3 descriptors would be generated. This generator can optionally be configured to further sub-divide these descriptors such that no unit of work descriptor straddles a quarter boundary (required by the algorithms). Attributes include the start and end absolute time index numbers.

Module/Output and Time Index Range

This generator creates one unit of work descriptor for each of the 84 CCD module/outputs. It can be configured to include or exclude specific module/outputs. Attributes of the descriptor include the module number (2-4, 6-20, 22-24), the output number (1-4), and the start and end absolute time index numbers. The generator first creates a descriptor for each time index interval, and then sub-divides each of those descriptors into a descriptor per module/output. For example, if a 90-day interval with 30-day sub-intervals was specified, a total of 252 descriptors (3 * 84) would be generated. Attributes include the start and end absolute time index numbers and the module/output numbers.

Observed Targets

This generator queries the database for the complete list of observed targets, sorts them by target ID, then partitions this set into a configurable number of chunks. Attributes include a start and end target ID.

Planetary Candidates

This generator queries the database for the unique set of observed targets where a significant transit event was detected by the most recent run of the transiting planet search module. These targets are partitioned first by sky group (sets of stars that all fall on the same module/output during any given quarter), then into chunks of a configurable size. Attributes include a start and end target ID and the sky group ID.

Single

Always generates a single task with no attributes. This generator is provided by the framework.

2.2 Science Processing Pipelines

There are three main science processing pipelines, one for long cadence (30-minute) data, one for short cadence (1-minute) data, and one to perform the transit search and subsequent validation of those results.

The long and short cadence pipelines are run monthly, after each spacecraft downlink. The monthly results are analyzed and the results are used to fine-tune the algorithms and parameters prior to the quarterly reprocessing run.

The transit search pipeline runs on all long cadence mission data collected to date to identify targets with possible planetary candidates. This pipeline is run quarterly, after the quarterly long cadence pipeline completes. It may also be run as needed, for instance after a software or parameter change is made to TPS and/or DV.

The unit of work generator may vary from module to module, depending on the nature of the science algorithms. Figure 1 shows the sequence of modules for each pipeline, along with the unit of work generator configured for each module. The arrows between the modules indicate the type of transition; solid for a synchronous transition and dashed for an asynchronous transition[3].

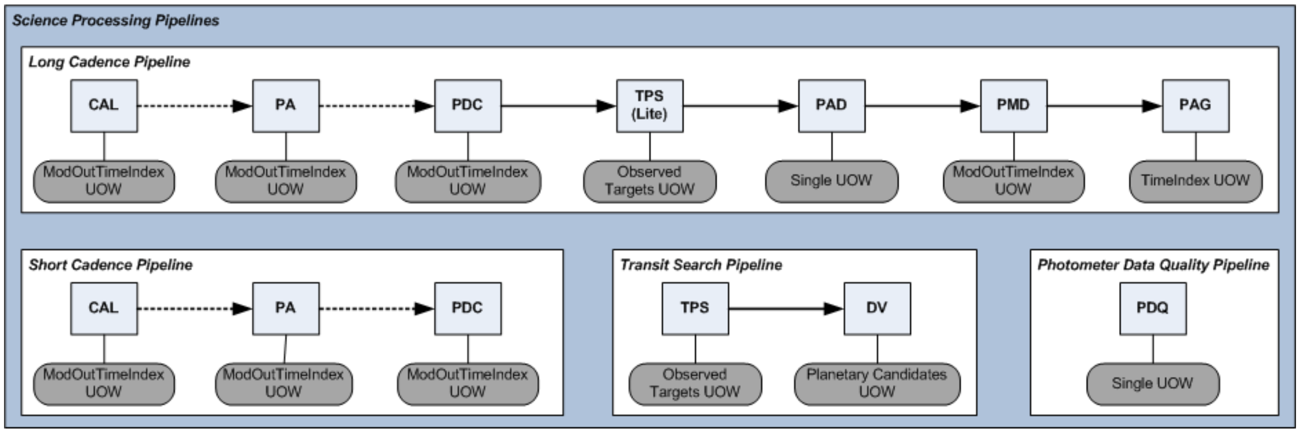

Figure 1: Science processing pipelines

The following pipeline modules perform the primary analysis of the raw data downlinked from the *Kepler* spacecraft.

- **Pixel-level Calibration (CAL)[8]:** Corrects the pixel data for instrument effects such as electronics noise, smear due to the lack of a shutter, and optical vignetting. The unit of work generator for CAL is Module/Output and Time Index Range. CAL needs all pixels from a full module/output at the same time because of the nature of the collateral data used for calibration and to enable calculation of certain metrics at the module/output level. The time index range is typically no more than a month because of the memory requirements of the algorithm.

- **Photometric Analysis (PA)[9]:** Estimates and removes background and computes flux and centroid time series for all targets. Like CAL, unit of work generator for PA is Module/Output and Time Index Range. PA needs a full module/output because of the way the background is estimated (with a polynomial that covers the entire module/output) and in order to calculate the pixel to sky coordinate mapping for each time index (known as motion polynomials), which requires an ensemble of stars.

- **Pre-Search Data Conditioning (PDC):** Removes systematic errors from the flux time series for all targets. The unit of work generator for PDC is also Module/Output and Time Index Range. PDC needs an ensemble of stars from the same module/output in order to detect and remove systematic errors.

- **Transiting Planet Search (TPS):** Searches for periodic transit-like signatures in the corrected flux time series for targets of interest and estimate precision of the flux time series on transit time scales. The unit of work generator for TPS is Observed Targets. TPS can operate on a single target at a time, so the unit of work generator is typically configured to evenly divide the targets across the available workers in the cluster. TPS can run in 'lite' mode, where it only computes photometric precision metrics for the targets, or in 'full' mode, where it searches for transits.

- **Data Validation (DV):** Performs a series of validation checks on the candidates identified by TPS in support of the follow up program. Like TPS, DV can operate on a single target at a time, but DV is the most timeconsuming module in the pipeline, so the number of targets per task is kept small (< 10) to improve the efficiency of the cluster and to prevent one bad target from preventing the entire unit of work from being processed. The unit of work generator is Planetary Candidates.

- **Photometer Attitude Determination (PAD)[10]:** Uses centroids and catalog coordinates to determine the pointing of the spacecraft over time. The unit of work generator is Single since PAD needs an ensemble of stars across the entire field of view.
- **Photometer Metrics Determination (PMD)[10]:** Calculates various metrics on the data to assist scientists in assessing the performance and health of the instrument. The unit of work generator is Module/Output and Time Index Range since the metrics are computed on a module/output basis.
- **Photometer Metrics Aggregator (PAG)[10]:** Aggregates the metrics computed by PMD into a single report covering the entire field of view. The unit of work generator is Time Index Range since the goal of PAG is to generate a single report that covers all module/outputs for a specified time index interval.
- **Photometer Data Quality (PDQ)[11]:** Performs analysis on a small subset of the data that is downlinked every four days (vs. once per month for the full set) for the purpose of assessing the near-term performance and health of the instrument. The unit of work generator is Single because PDQ generates a report for the entire field of view.

2.3 Development and Test Pipelines

There are also several pipeline configurations used for automated testing and generation of simulated data for testing. These pipelines contain modules that perform many of the tasks that are normally performed manually by the SOC operators due to the various checks and balances that make up the operations procedures. In a test environment, however, these steps can be automated by the pipeline. This approach allows us to fully automate the end-to-end processing done by the SOC for testing purposes. These tests start with an empty data store and perform the following steps.

1. Initialize the database schema and clean out the data store.
2. Generate simulated test data with our end-to-end model of the *Kepler* photometer.
3. Ingest the simulated test data.
4. Process the simulated test data through the science pipelines
5. Export the pipeline products to FITS and XML files
6. Validate the exported products to verify that the data flowed correctly through the pipeline

We use these test pipelines to run a small data set through the science pipelines on a nightly basis as a regression test, allowing us to detect problems introduced by changes to the code or parameters as soon as possible in each development cycle. Developers and scientists also use these pipelines to troubleshoot problems or to experiment with changes to the algorithms prior to releasing the changes to the full cluster for processing of flight data.

3. ANATOMY OF A PIPELINE MODULE

The pipeline framework and data management software elements of the *Kepler* pipeline are written in Java, while the science algorithms are written in MATLAB. The Java classes that wrap these algorithms and manage their inputs and outputs are referred to as the module interfaces. The module interface code is responsible for retrieving the inputs for a given unit of work descriptor from the data store, passing these inputs to the algorithm, and storing the resulting outputs in the data store once the algorithm completes.

This section describes pipeline modules that include a MATLAB algorithm component, but not all pipeline modules follow this model. Pure Java modules integrate with the pipeline framework the same way that MATLAB modules do, they just don't invoke the serialization and execution services of the framework.

The diagram below shows the main steps involved in executing a pipeline job, starting with the invocation of the module interface code by the framework in the worker process (step **2** in figure 2).

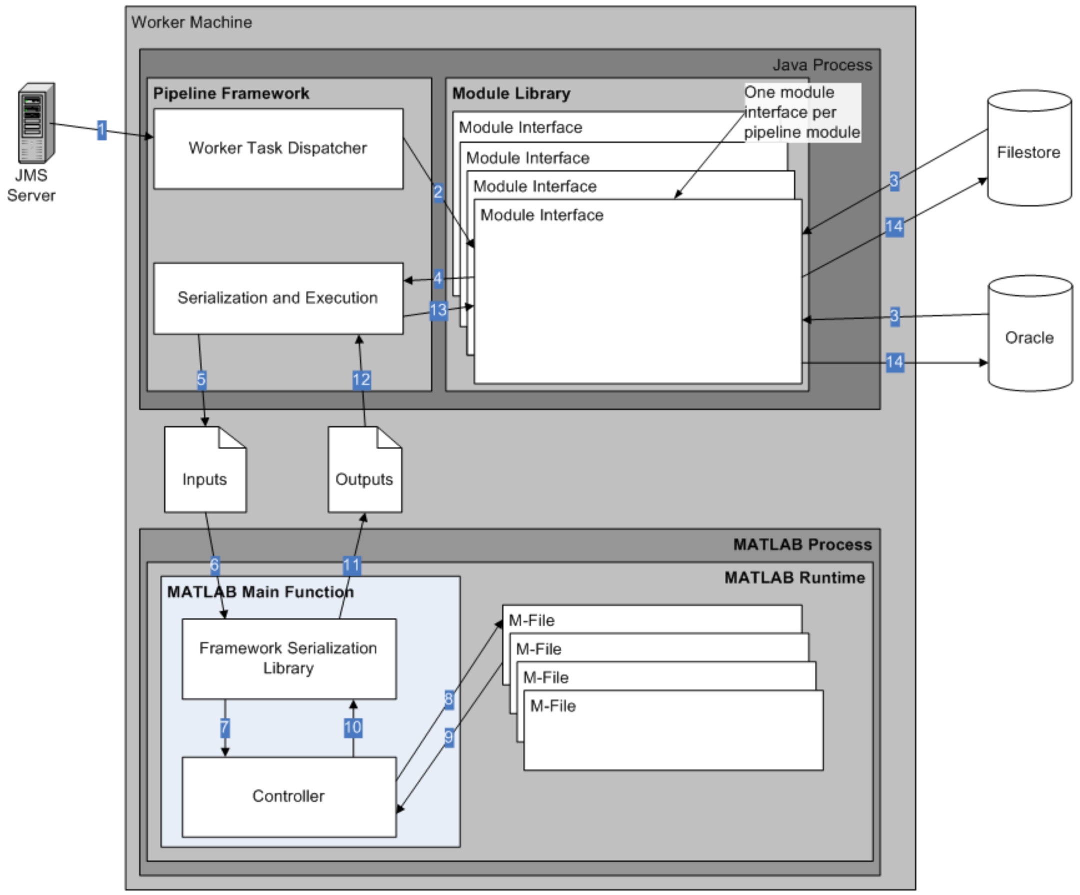

Figure 2: Module interface data flow

The module interface class must first retrieve the inputs specified by the unit of work descriptor from the data store (**3**). It then aggregates these inputs into an object tree and passes the tree to the serialization and execution services of the pipeline framework (**4**).

The framework includes a serialization component that can stream the object tree containing the algorithm inputs to a temporary file (**5**). The execution component of the framework then launches the MATLAB process, passing it the path to the temporary file (**6**). The serialization and execution components of the framework are described in more detail below.

The main function (entry point) of the MATLAB process first invokes the framework serialization component on the MATLAB side to reconstitute the temporary file created in step (**5**) into a MATLAB struct (**6**), then invokes the MATLAB controller for the module (**7**). The controller then invokes the various algorithms needed to process the job (**8, 9**), then returns control to the main function (**10**) along with a MATLAB struct containing the outputs of the module. The main function then serializes these outputs to a temporary file (**11**) and the MATLAB process exits.

Once the MATLAB process completes, the serialization component of the framework reconstitutes the outputs into a Java object tree (**12**) and returns it to the module interface class (**13**), where the outputs are stored in the database and/or the Kepler DB[4] (**14**).

Because all access to the data store is done from the Java process, the serialized inputs file contains everything the algorithm needs to process the unit of work, so the algorithm has no external dependencies at runtime other than the inputs file. This also means that the algorithm can be executed on a different machine (like a developer or scientist workstation) with only the inputs file. This allows activities such as debugging a problem found during pipeline execution, experimenting with algorithm or parameter changes, or data analysis to be performed without access to the cluster hardware where the inputs file was originally created.

The components that support these steps are described in more detail in the sections below.

3.1 Module Interface Classes

This section describes the main classes involved in the execution of a pipeline module. This includes classes that are part of the framework as well as application-specific classes written by the module developer. The diagram below shows the relationships between these classes. The classes in the top two boxes are implemented in Java, and the classes in the bottom two boxes are implemented in MATLAB. We show the TPS module in figure 3 below as an example.

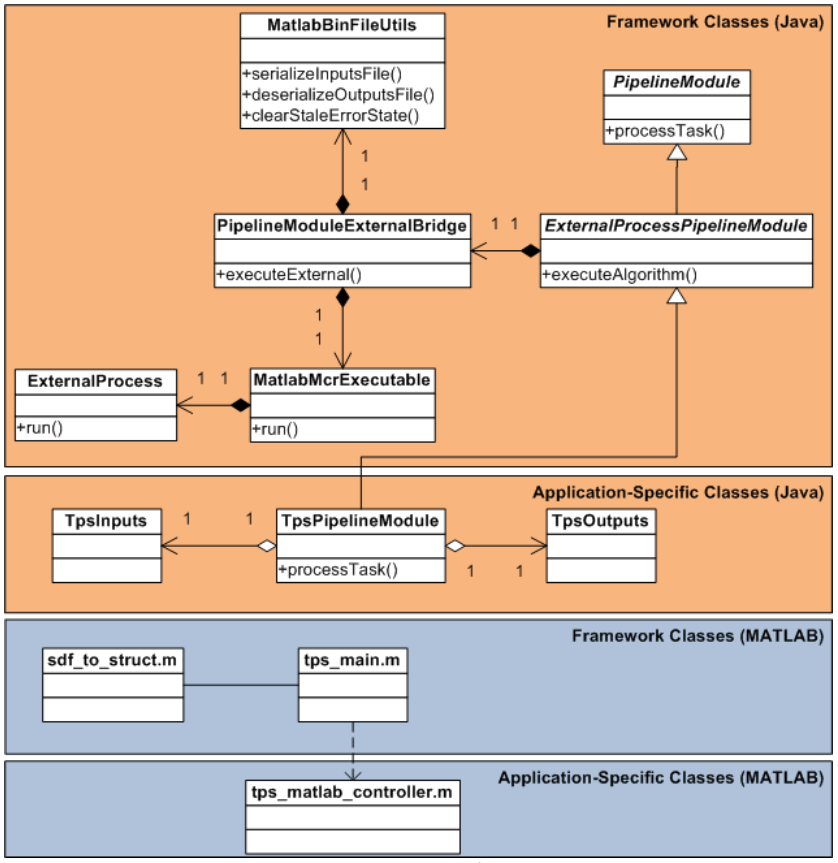

Figure 3: Module interface classes

3.2 Application-specific Java Classes

The `PipelineModule` abstract class is the main interface between the pipeline framework and the module interface code. All pipeline module classes must extend this class or a sub-class. Typically, modules that include a MATLAB algorithm extend the `ExternalProcessPipelineModule` abstract class (which extends `PipelineModule`). `ExternaProcessPipelineModule` provides easy access to the serialization and execution services of the pipeline framework via its `executeExternal` method. Pipeline modules that do not include a MATLAB component (Java-only pipeline modules) typically extend `PipelineModule` directly.

The main role of this class is to act as an entry point for the framework and to fetch the algorithm inputs from the data store and store the algorithm outputs to the data store.

Input and Output Classes and the `Persistable` Marker Interface

The inputs and outputs classes for the module interface contain the inputs and outputs of the MATLAB algorithm that are serialized by the framework for passing between the Java and MATLAB processes. These classes, and any contained classes, must implement the `Persistable` marker interface. The serialization library only supports `Persistable` classes, and `Persistable` classes must contain only Java primitives, `Strings`, `Enums`, `Dates`, the standard Java collection classes `List`, `Set`, and `Map` (containing supported types), and other `Persistables`. These restrictions ensure that the contents of the input and output object trees can be properly represented in MATLAB structs using built-in MATLAB types.

3.3 Serialization Services

As mentioned above, the framework uses files to pass the MATLAB algorithm inputs and outputs between the Java and MATLAB processes. These files are created and accessed using serialization classes that are part of the framework. In order to simplify MATLAB framework code that reads and writes these files, a simple serialization method was chosen over existing serialization methods, such as Java object serialization. Because the data files can be very large even for a single unit of work, we chose a binary format over a text format like XML.

`PersistableInputStream` and `PersistableOutputStream` are abstract framework classes that support serialization of a Java object tree. These serialization classes do not define the external serialization format themselves; instead they delegate the actual reading and writing of the field values to implementation classes via the abstract methods shown in figure 4. The combination of the abstract serialization classes and the sub-classes that implement the abstract methods provides a complete serialization solution.

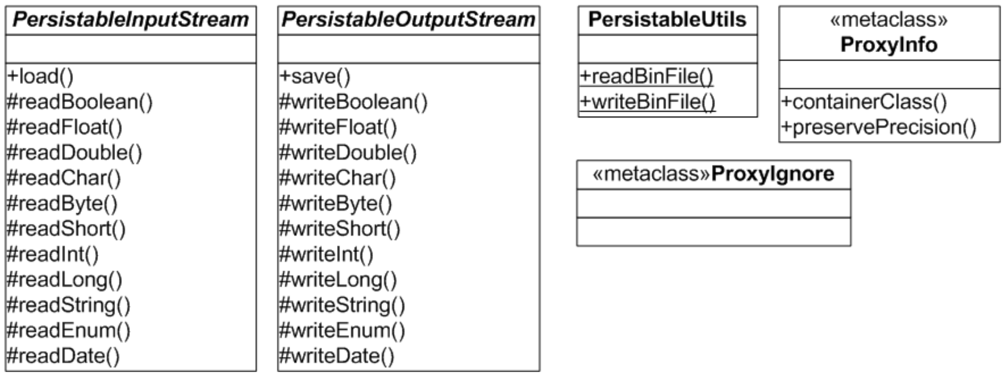

Figure 4: Framework abstract serialization classes

The `Persistable*Stream` serialization classes use the Java Reflection API to access the individual fields of the objects to be serialized. They then feed these field values to the subclass for storing in the external format via the `readX` and `writeX` abstract methods shown in figure 4. When a field that contains an object of type `Persistable` is encountered, these classes use recursion to feed the fields of that class to the subclass. For example, consider the case where the object tree contains two objects of types A and B, defined as follows.

```
public class A implements Persistable {
        int a1;
        B a2;
        float a3;
}
public class B implements Persistable {
        String b1;
        double b2;
}
```

In this case, the fields would be fed to the sub-class in the following order:

```
a1,b1,b2,a3
```

The serialization implementation class determines the external format used. In theory, any external representation could be used (XML, other text or binary formats, etc.) by sub-classing these two abstract classes.

A `ProxyIgnore` annotation class allows the developer to control the behavior of the serializer on a field-by-field basis. Fields that are tagged with the `ProxyIgnore` annotation are not fed to the implementation class.

The framework currently uses the `SdfPersistableInputStream` and `SdfPersistableOutputStream` implementation classes (described below) to serialize the module input and output object trees in order to pass them between the Java and MATLAB processes. These implementation classes read/write the fields from/to a file with a .sdf extension using the native machine format. This file is known as an SDF file. The endianness of the platform is used, which means that the binary files are not portable between hardware with different endianness.

`ClassWalker`

The `ClassWalker` framework class provides a static analysis of a class tree in support of the SDF serialization components described below. Unlike the `Persistable*Stream` framework classes, which walk an object tree and feed field values to an implementation class, `ClassWalker` walks a class tree and feeds class and field descriptors to a registered listener that implements the `WalkerListener` interface.

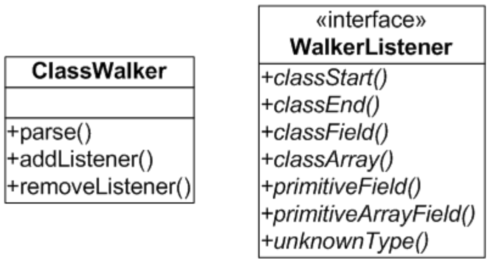

Figure 5: ClassWalker Classes

The `ClassWalker` class uses the Java Reflection API to identify all of the fields of a specified class. Each time a previously-unseen class is encountered, `ClassWalker` fires a `classStart` event, followed by events for all of the fields of that class, followed by a `classEnd` event. The `classStart`/`classEnd` event pairs will be nested to match the static structure of the classes. Classes are only described once, even if they are referenced multiple times in the class structure.

Every SDF file contains a header with a data dictionary that describes all of the classes and their fields that are used in the payload of the file. The SDF implementation uses the `ClassWalker` classes to generate this data dictionary, and it extends `PersistableInputStream` and `PersistableOutputStream` to serialize the object tree for the payload of the SDF file.

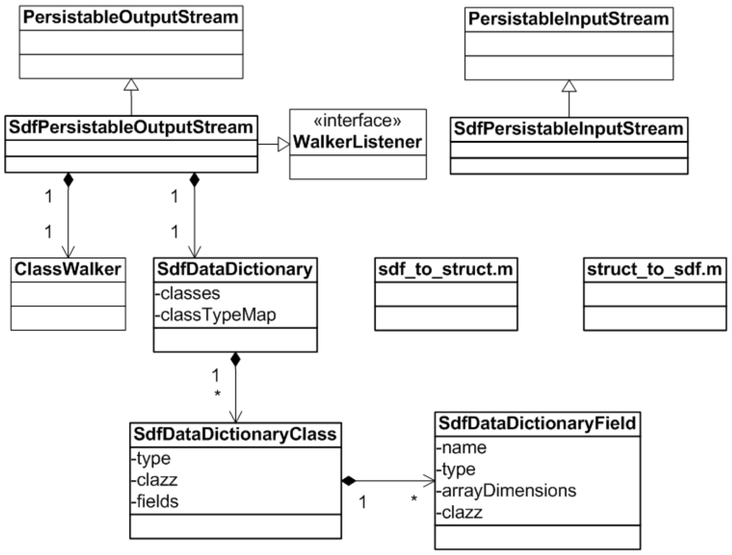

Figure 6: SDF Classes

3.4 Execution Services

The framework provides several classes on the Java side to manage the lifecycle external MATLAB process, including passing inputs and outputs.

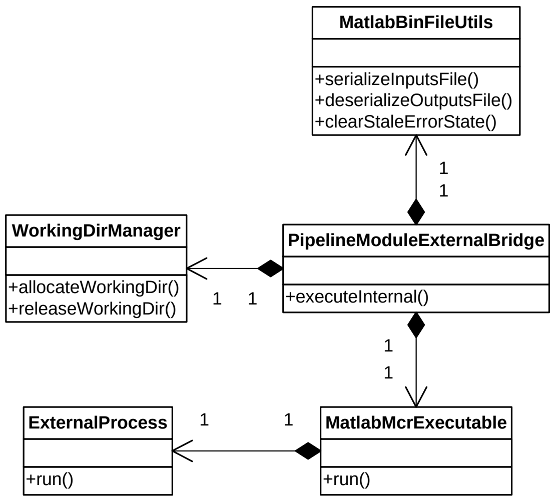

Figure 7: Framework External Process Management Classes

On the Java side, the PipelineModuleExternalBridge class acts as the coordinator for the entire process of serializing the inputs to a SDF file, launching the external MATLAB process and waiting for it to complete, and deserializing the outputs produced by the MATLAB process. The following sequence diagram shows the main steps of this process.

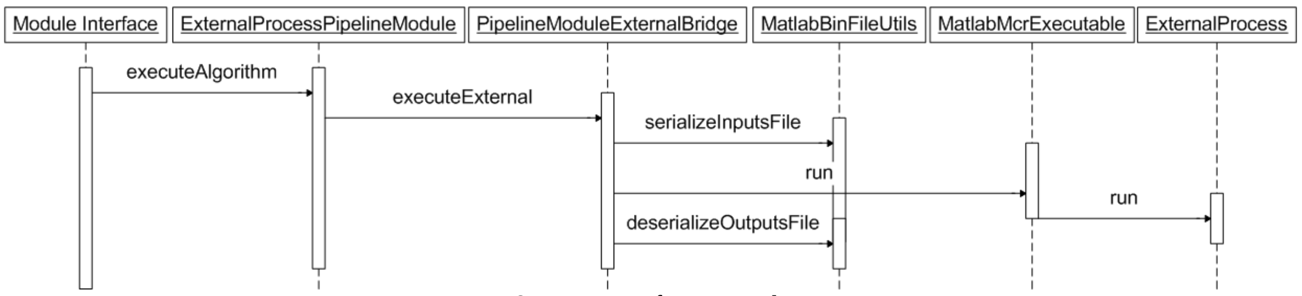

Figure 8: Execution of an External Process

First, the module interface code invokes the executeAlgorithm method of ExternalProcessPipelineModule, passing it a fully-populated inputs object and an empty outputs object. This call will block until the external process completes (successful or otherwise) and the outputs are deserialized.

The executeAlgorithm method creates a new instance of PipelineModuleExternalBridge and invokes the executeExternal method on that object.

PipelineModuleExternalBridge uses WorkingDirManager to initialize the working directory for the MATLAB process. This directory will contain the input and output SDF files for the algorithm as well as any temporary files created by the algorithm. The contents of the working directory are useful for debugging problems or for off-line analysis of the data, so they are not deleted immediately after the job processing completes. Instead, WorkingDirManager keeps track of all of the working directories created by a worker process and deletes the oldest directories when the total number of directories exceeds a configurable threshold.

PipelineModuleExternalBridge then uses MatlabBinFileUtils to serialize the inputs object tree to an SDF file. MatlabBinFileUtils is a thin wrapper around the Persistable*Stream classes and manages things like the filenames for the SDF files and error files generated by the MATLAB process (described below). Some pipeline modules invoke the MATLAB algorithm multiple times for a single unit-of-work, for example when the algorithm can't handle all of the data for the unit-of-work because of memory constraints. PipelineModuleExternalBridge supports this by maintaining a sequence number that is incremented for each invocation. This sequence number is used in the filenames of the SDF files and in MATLAB log files so that files from one invocation do not overwrite files from a previous invocation.

At this point the inputs SDF file exists in the MATLAB working directory and we are ready to launch the MATLAB process. This is done by calling the run method on `MatlabMcrExecutable` class. This class manages the setup needed to run an executable generated by the MATLAB compiler, including telling the process where to find the MATLAB compiler runtime. The first time a MATLAB executable runs, it extracts certain archive files needed by the process at run time, and it is important that the executable not be invoked by one worker thread while another worker thread is busy extracting the archive. The `MatlabMcrExecutable` class makes sure this doesn't happen.

The actual launching and monitoring of the MATLAB process is done by the `ExternalProcess` class. This class supports synchronous and asynchronous execution of external processes as well as management of their output to the `stdout` and `stderr` streams. `ExternalProcess` starts a new thread for each stream that consumes output from the process so that it does not block on I/O. The stream output can be configured to go to a file, a memory buffer, or to be thrown away. In the case of MATLAB executables, synchronous execution is used and stream output is redirected to log files in the working directory. `ExternalProcess` also supports a retry mode, where the external process will be re-launched a configurable number of times in the event of an abnormal exit. We use a retry count of one to minimize the chances of stalling the pipeline due to a transient error.

On the MATLAB side, the main function deserializes the inputs file using the framework function `sdf_to_struct.m`. Control is then handed over to the MATLAB controller for the module. The algorithm developer is responsible for writing the controller function. By convention, this call takes the following form.

```
outputsStruct = tps_matlab_controller(inputsStruct);
```

If the controller call completes successfully, the resulting outputs struct is serialized to an SDF file using `struct_to_sdf.m`.

If an error is thrown by the controller or some other code called by the controller (either other algorithm code or MATLAB built-in functions), that error is caught by the main function and the error message as well as the call stack are serialized to an error SDF file for subsequent processing by the Java side as described above.

Once the process completes (or exhausts the retry attempts), `MatlabBinFileUtils` checks for the existence of an error SDF file. The auto-generated MATLAB main function creates this file if the algorithm throws an error. The error SDF file contains the text of the error message as well as the MATLAB stack at the time of the error. If the error file exists, it is deserialized and an exception containing the error is thrown. The exception is caught by the framework code that initiated the processing and the job transaction is rolled back. The module code does not need to handle this exception and can assume that if control is returned from the `executeAlgorithm` call that the MATLAB processing completed successfully. The error information is propagated to the operator via the console, who can then forward it to the developer to aid in debugging problems. If the error file is not present, then the MATLAB process completed successfully and the outputs SDF file is deserialized. The resulting outputs object tree is then returned to the module code for storage in the data store.

ACKNOWLEDGMENTS

Funding for this work has been provided by the NASA Science Mission Directorate (SMD).

REFERENCES

[1] Middour, C., et al., "Kepler Science Operations Center architecture," Proc. SPIE, this volume (2010).

[2] Koch, D.G., et al., "Kepler Mission design, realized photometric performance, and early science", Astrophysical Journal Letters, 713, L79-L86 (2010).

[3] Klaus, T.C., et al., "Kepler Science Operations Center pipeline framework," Proc. SPIE, this volume (2010).

[4] McCauliff, S., et al., "The Kepler DB, a database management system for arrays, sparse arrays and binary data," Proc. SPIE, this volume (2010).

[5] Twicken, J.D., et al., "Presearch data conditioning in the Kepler Science Operations Center pipeline," Proc. SPIE, this volume (2010).

[6] Jenkins, J.M., et al., “Transiting planet search in the Kepler pipeline,” Proc. SPIE, this volume (2010).
[7] Wu, H., et al., “Data validation in the Kepler Science Operations Center pipeline,” Proc. SPIE, this volume (2010).
[8] Quintana, E.V., et al., “Pixel-level calibration in the Kepler Science Operations Center pipeline,” Proc. SPIE, this volume (2010).
[9] Twicken, J.D., et al., “Photometric analysis in the Kepler Science Operations Center pipeline,” Proc. SPIE, this volume (2010).
[10] Li, J., et al., “Photometer performance assessment in Kepler science data processing,” Proc. SPIE, this volume (2010).
[11] Jenkins, J.M., et al., “Semi-Weekly Monitoring of the Performance and Attitude of Kepler Using a Sparse Set of Targets,” Proc. SPIE, this volume (2010).