Actionable Management Policies

Reagan W. Moore, Arcot Rajasekar,
Hao XU
UNC-CH
216 Manning Hall
Chapel Hill, NC 27599-3360
01 919 962 9548
{rwmoore@email.unc.edu,
sekar@renci.org, xuh@cs.unc.edu}

Stanley Ahalt
Renaissance Computing Institute
100 Europa Drive, Suite 540
Chapel Hill, NC 27517
01 919 445 9641
ahalt@renci.org

ABSTRACT

In this paper, we describe the implementation of policies that automate the management and validation of collection properties. We use the integrated Rule Oriented Data System (iRODS) to enforce management policies on data collections. The policies are cast as periodic rules that verify whether desired collection properties have been maintained, identify problems, and automatically correct properties when errors are detected. An analysis is provided of the performance requirements for a production system, as well as the basic functions that are needed to implement production capable policies.

Categories and Subject Descriptors

H.4.2 [**Information Systems**]: Information Systems Applications – *decision support.*

General Terms

Management, Design, Performance

Keywords

Policy-based data management, Rule design

1. INTRODUCTION

Data grids based on the integrated Rule Oriented Data System (iRODS) now manage collections that are petabytes in size, that contain hundreds of millions of files, and that are distributed across multiple storage systems, administrative domains, and institutions [1]. Two examples of iRODS production data grids include the French National Institute for Nuclear Physics and Particle Physics [2], and the Wellcome Trust Sanger Institute genomics data grid [3]. Each data grid has a specific set of objectives for the organization of a shareable collection. For example, objectives may include the preservation of observational data, or the management of data products derived from experimental data, or the tracking of research results across multiple analyses. Given a set of objectives, each project defines a corresponding set of properties (assertions) to be applied to the federated collections. The properties may be related to authenticity, integrity, completeness, data formats used, or semantic descriptions. The data management system needs to provide mechanisms to enforce the desired properties, and verify that the properties have been conserved over time.

In distributed environments a shareable collection is subject to operational procedures that may be applied by local institutional administrators, may be housed on storage systems with varying preservation capabilities, and is susceptible to data loss mechanisms related to operational, software, and hardware errors. Thus a validation of the properties of a collection is only as good as the application of the most recent verification process. The development of a system that enforces the desired policies, automates administrative tasks, verifies the required properties, and automates repair of problems has strongly motivated the creation of the integrated Rule Oriented Data System.

In iRODS, collection properties are enforced by defining policies that are cast as computer actionable rules. Each client action is trapped at a set of policy enforcement points. A rule engine then queries a local rule base to decide which policies should be applied, and executes the appropriate procedures. The rules are written in a rule language that supports the definition of `Event: Condition: Procedure: Recovery-procedure`. Each procedure is composed by chaining together basic functions as a computer executable workflow. The basic functions are called micro-services because they can exchange information through memory structures, or through an exchange of parameters, or through a catalog of persistent state information, or through files. Further, the execution of each micro-service may generate state information that is persistently saved in a metadata catalog.

Validation of the management policies can be done by querying the persistent state information, or by evaluating the properties of each file to check that the persistent state information is (still) accurate. To verify compliance over time, audit trails may be parsed to check accesses, application of specific policies, and transformations performed upon the data.

The expectation is that through application of periodic rules, administrative tasks such as integrity checks can be automated. This includes identification of missing replicas and verification of the checksum of each file. Recovery operations such as creation of the required number of replicas can be automated, and logs of all operations can be maintained.

Section 2 of this paper reviews the properties of a computer actionable integrity policy that are needed for a production system. In Section 3 we examine the performance of a production rule, and in Section 4 we identify the basic functions that are needed to implement a production integrity rule. In Section 5 multiple strategies are proposed for alternate forms of the policy. The conclusion describes additional types of policies that can be implemented in a policy-based data management system. The expectation is that through choice of the appropriate policies and procedures, required policies for data sharing, collection formation, data publication, data analysis, and data preservation systems can be implemented and automated.

2. PRODUCTION INTEGRITY POLICY

The creation of a production policy needs to address production management challenges, as well as the enforcement of a specific policy. The production challenges typically involve management of the execution of the policy itself, with a goal of minimizing the amount of labor required to execute the policy.

The simple production policy we will examine is designed to meet the following production requirements. In the list, the generic challenges (needed for production systems) are listed in italics.

1. *Verify all input parameters for consistency.*
2. *Query the iRODS metadata catalog to retrieve information* about the number of files in a collection, their sizes, and the location of replicas.
3. Verify the integrity of each file in a collection by comparing the saved checksum with a new evaluation of each checksum. This requires reading each file.
4. Update all replicas to the most recent version.
5. *Minimize the load on production services.* The average storage system I/O rate needs to be as small as possible. We implement a deadline scheduler to ensure that the checksums are calculated at the slowest possible rate to meet the specified deadline.
6. *Differentiate between the logical name for a file and the physical replica locations.*
7. Identify all missing replicas and *document their absence*.
8. Create new replicas to replace missing replicas.
9. *Implement load leveling* to distribute the new replicas across the storage systems that are being used to support the collection.
10. *Create a log file that records all repair operations performed upon the collection.*
11. T*rack progress of the policy execution.*
12. *Initialize the rule for the first execution.* This includes initializing variables, setting up a directory in which log files are stored, calculating the required I/O rate to meet the deadline, creating collection status flags for tracking progress, identifying the storage systems that are being used for the collection replicas, and verifying that the number of storage systems is greater than or equal to the number of required replicas.
13. *Enable restart of the process from the last set of checked files in case of a system halt.*
14. *Manipulate files in batches of 256 files at a time to handle arbitrarily large collections.*
15. *Minimize the number of sleep periods used by the deadline scheduler.* This is set by specifying a minimum amount of time to sleep when the execution rate is too fast.
16. *Include the checking of new files that have been added during the execution of the policy if a restart is needed.*
17. *Write out statistics* about the effective execution rate, and the number of files checked.

We note that of the 17 objectives, only three objectives are specific to the integrity policy. The expectation is that the remaining 14 objectives can be cast as a template for the execution of other production policies.

2.1 Implementation

Each of these objectives is expressed as an executable workflow that is applied by the iRODS rule engine. The iRODS workflow language provides basic constructs (implemented as micro-services) that can be used to control the operations. The basic operations include:

- Support for variables – integers, strings, binary, double, Boolean, lists
- Arithmetic – add, subtract, multiple, divide
- String manipulation – subset, concatenate
- Loops – for, foreach, while
- Conditional tests – if then else, and, or
- Breaks – conditional exit from a loop

The basic operations are augmented with micro-services that encapsulate specific manipulation functions, such as querying the metadata catalog, metadata manipulation, file and directory manipulation, evaluating a checksum, and updating replicas to the most recent version. In iRODS, there are currently about 250 micro-services that are provided to support workflows, data and metadata manipulation, message passing, interaction with web services and remote systems, and debugging [4].

We illustrate selected production policy objectives with the actual iRODS rule language code. In the iRODS rule language, each variable name is denoted by a leading asterisk. String variable values are specified using double quotes. The input to the integrity rule is four variables:

- The collection that will be examined, *Coll
- The desired total run time, *Delt
- The required number of replicas, *NumReplicas
- The resource where the log file will be stored, *Res

A simple initiation test is to verify that the *Coll variable actually specifies a collection, and not some other entity such as a file. The iRODS micro-service msiIsColl can be used to do this:

```
--------------------------------------------------------------------------
# check whether a collection was defined
  msiIsColl(*Coll, *Result, *Status);
  if(*Result == 0 || *Status < 0) {
    writeLine("stdout","Input path *Coll is not a collection");
     fail;
  }
--------------------------------------------------------------------------
```

The msiIsColl micro-service takes the collection name as an input parameter, and returns a result flag *Result that has the value 1 if the pathname is a collection, and a status flag *Status that is negative if it has an execution error. If either test fails, an output line is written to standard out by the writeLine micro-service, and the rule is then terminated.

To test whether the rule execution is a restart or an initial execution, a metadata attribute, called TEST_DATA_ID is associated with the collection name. If this attribute is missing, then the rule is being run for the first time and a default value is added as an attribute on the collection. Otherwise the value of this metadata attribute is used to track which DATA_ID was last successfully checked.

```
----------------------------------------------------------------------------
# check whether the attribute TEST_DATA_ID has been set from
a prior execution
*Val = "0";
msiExecStrCondQuery("SELECT
COUNT(META_COLL_ATTR_NAME) where COLL_NAME =
'*Coll' and META_COLL_ATTR_NAME = 'TEST_DATA_ID'",
*GenQOut2);
foreach (*GenQOut2) { msiGetValByKey(*GenQOut2,
"META_COLL_ATTR_NAME", *Val);   }
if(int(*Val) == 0) {
  *Str1 = "TEST_DATA_ID=0";
  msiString2KeyValPair(*Str1,*kvp);
  msiAssociateKeyValuePairsToObj(*kvp,*Coll,"-C");
  writeLine("*Lfile","added TEST_DATA_ID attribute to
collection *Coll");
}
# on a restart TEST_DATA_ID will be greater than 0
msiMakeGenQuery("META_COLL_ATTR_VALUE",
"COLL_NAME = '*Coll' and META_COLL_ATTR_NAME =
'TEST_DATA_ID'", *GenQInp2);
msiExecGenQuery(*GenQInp2,*GenQOut2);
foreach(*GenQOut2) {
  msiGetValByKey(*GenQOut2,
"META_COLL_ATTR_VALUE", *colldataID);
}
# *colldataID is the string identifier of the last file that has been
checked
```

The query that is issued to the metadata catalog by the msiExecStrCondQuery micro-service counts the number of times the metadata attribute with the name TEST_DATA_ID is present on the collection *Coll. Note that schema indirection is used for status metadata, with the name of the collection attribute stored in META_COLL_ATTR_NAME. The value of the collection attribute is stored in META_COLL_ATTR_VALUE. The values returned from the metadata catalog are returned as strings. Thus the number of appearances of the metadata attribute (*Val) has to be converted to an integer with the "int" micro-service. To add a metadata attribute to the collection, a key-value pair defined in the string "*Str1" is converted to a key-value pair structure by the micro-service msiString2KeyValPair. The structure is then used as input to the micro-service, msiAssociateKeyValuePairsToObj, that loads the well-formed metadata into a collection attribute.

iRODS can separate the formation of a query from the execution of a query. The arguments for the micro-service msiMakeGenQuery specify the selection variables, the condition, and the name of the string variable that holds the query, *GenQInp2. The arguments for the micro-service msiExecGenQuery specify the string holding the query and a structure that holds up to 256 rows of the query result, *GenQOut2. The foreach loop over the query result iterates through each row. The micro-service msiGetValByKey extracts the value of the specified state information from the structure and stores the value in a workflow variable, *colldataID.

This illustrates the type of operations that are performed within a data management workflow. Queries are made on the metadata catalog and returned through in-memory structures. Data can be read from the in-memory structures and used to initialize variables, which can then be manipulated or tested.

The deadline scheduler is implemented by comparing the rate at which the checksums are being performed against an expected execution rate. The desired I/O rate (*Fac) is calculated by summing the size of all of the files in the collection, *Coll, and dividing by the input total duration, *Delt. The actual average I/O rate is calculated by tracking the run time and the size of the data that have been checksummed. The "msiGetSystemTime" micro-service returns the time in seconds from an initial epoch. This is called at the start of the policy execution to establish the start time, *TimeS. A variable, *Runsize, is incremented by the size of each checksummed file. After completion of each batch of checksums (256 files at a time), the current system time is retrieved, *timei. The time when the checksums should have completed (*timerun = int(*TimeS) + *Runsize / *Fac) is compared with the current time. If the difference is greater than the minimum sleep time, typically four seconds, the average execution rate is slowed down through an explicit sleep call, msiSleep.

The management of restarts depends on the use of a unique internal iRODS identifier for each file, DATA_ID. This identifier is incremented each time a file is added, and counts the total number of files that have been added to the data grid. Thus the DATA_ID identifier is monotonically increasing and can be used to order result sets from queries made on the iCAT catalog. The query:

```
msiMakeGenQuery("order(DATA_ID), DATA_SIZE,
DATA_NAME, COLL_NAME, DATA_CHECKSUM",
"COLL_NAME = '*Coll' and DATA_ID > '*colldataID'",
*GenQInp);
```

generates a query for a monotonically increasing list of all DATA_ID values of files in the collection *Coll, that have a DATA_ID value greater than the restart value *colldataID, and returns the DATA_ID, the size of the file (DATA_SIZE), the name of the data file (DATA_NAME), the name of the collection (COLL_NAME), and the value of the checksum (DATA_CHECKSUM). Since the collection *Coll may contain subcollections, the current collection name is retrieved to define the correct logical name for the file. The logical name is constructed as a string, "*Colln/*Name", where *Colln and *Name are retrieved from the query result structure using the msiGetValByKey micro-service.

The integrity rule loops over the result set from this query, which contains a list of all of the logical file names in the collection. For each logical name, a separate query is made to the metadata catalog to retrieve information about all of the replicas associated with the logical name. Thus the policy has to support nested loops, a "while" loop that iterates over batches of 256 logical names, a "foreach" loop that iterates over the logical names, and a second "foreach" loop that iterates over the physical replica locations for each logical file name.

The implementation of load leveling across the storage systems that are used for replicas for the collection is implemented by constructing a list of all resources used by the collection. A simple query on the metadata catalog

```
 msiMakeGenQuery("DATA_RESC_NAME","COLL_NAME=
'*Coll'", *GenQInpr);
```

retrieves all of the resources that are used to store file for the collection, *Coll. The names of the resources are extracted from the query result structure and stored in a list. When the location of the replicas for a specific logical file name is found, a second list is created that marks whether or not the resources from the resource list have been used. To implement load leveling, a pointer is maintained to the last resource that was used from the resource list. After the creation of a new replica, the point is incremented modulo the length of the replica list. This ensures that the missing replicas will not be re-created on a single storage system.

After each batch of 256 files, the collection attribute value for TEST_DATA_ID is reset to the DATA_ID of the last file that was checked. This ensures that on a restart, the system will be able to skip files that have already been verified. This approach works because the DATA_ID for each file is a unique, persistent, monotonically increasing identifier.

The complete rule is shown in Section 7.

3. PERFORMANCE

The tests of production rules for verifying properties of a collection include the validation of the policy, analysis of the execution time, and determination of some idea of the load on the system. We track the time needed to loop over the data sets, the time needed to execute the micro-service, and the time needed to interact with files on a local disk.

The tests were run on an Ubuntu operating system within a Virtual Box emulator on a MacPro 2.53 GHz Intel Core i5 computer. The disk had a rotational rate of 5400 rpm (disk rotational latency of 11 milliseconds). Data transfer time to put a 200 MByte file into an iRODS local disk vault was 4 seconds, implying an effective transfer rate of 50 Mbytes/second.

The version of the iRODS software was 3.1, revision 4882 from the SVN repository. This revision included modifications to the writeLine micro-service to support writing to a log file, revisions to the rule engine for parsing rules, and revisions to the msiCloseGenQuery micro-service for closing buffers. These revisions are needed to implement all of the features of the integrity policy.

To test performance, collections were created that contained 21,000 files, 40,0000 files, and 100,000 files. Each file was 877 bytes in size. The performance results were strongly dominated by the latency of the system, since the size of the files was very small. The time needed to read a file from a directory on the Mac operating system and write the file into an Ubuntu directory averaged 18.2 milliseconds. This time is a combination of the rotational latency (11 milliseconds) and the seek latency (about 5 milliseconds). Manipulations of files (such as a checksum calculation) are expected to take at least one spin rotational latency time.

To test the execution rate of the rule engine, a simple loop test was constructed that looped over a counter one million times. The time per loop iteration (while statement exit test, counter increment) was 35 microseconds per iteration. Thus a single invocation of the rule engine will take about 18 microseconds on average, which is 1000 times faster than the observed time to read and write a small file.

A second performance test was constructed that looped over the files in the 100,000-file collection, retrieving the file name from the query result structure. The time per loop iteration was 160 microseconds. The loop made queries to the iCAT metadata catalog to retrieve information in batches of 256 files. Hence the cost of interacting with the database was effectively amortized.

To better estimate the time for simple queries to the database, a test was constructed that first looped over all the files in a collection, and then for each file made an additional query for the DATA_NAME, COLL_NAME, and DATA_RESC_NAME. This was applied to a collection with 21,000 files, and took on average 714 microseconds per query. These analyses show that the latency of interaction with the disk at 11 milliseconds dominates the time needed to either query the metadata catalog at 0.71 milliseconds per query or perform operations at 0.16 milliseconds per operation. The latency of the disk is about 100 times slower than the combination of the rule engine and metadata catalog latencies.

The full replication rule was run on a collection of 21,000 logical file names with two physical replicas for each logical file name (42,000 physical files total). The time to run the policy was 132 seconds when file access was turned off. This is the time needed to perform the nested queries against the metadata catalog, and apply the logic to control the checksums and replica counts. The average loop and query time per file was 6.3 milliseconds. With file access (and checksums) turned on, the run time increased to 920 seconds, or 21.9 milliseconds per file. This is close to the sum of a rotational latency plus the loop and query time plus a seek latency of 4.6 milliseconds. Since the replicas are stored on a separate directory on the same disk, the system did have to move the disk head back and forth for every two files.

The observed performance is quite reasonable, indicating that rules that use several hundred lines of workflow and micro-service operations can keep up with disk rotational latencies.

4. BASIC FUNCTIONS

A second important characterization is the type and number of basic functions that are needed to implement a production rule. For the integrity test example, the following workflow operations were required:

- Arithmetic (+, -, *, /)
- Boolean tests (==, !=, &&, ||, >, <, >=)
- Conditional statements
  - if
  - then
  - else
- Control
  - break
  - fail
- Loops
  - for
  - foreach
  - while
- List manipulation
  - initialization
  - list addition (cons)
  - extracting an element from a list (elem)
  - updating an element in a list (setelem)
- Variable manipulation
  - initialization
  - type conversion (int, double, str)
  - string concatenation

These operations comprise a minimal set of workflow operations needed to implement validation policies.

Of greater interest is the set of basic operations that were invoked during the execution of the integrity policy. They have been roughly organized into categories for metadata catalog interactions, data and directory manipulations, and system functions.

Metadata catalog manipulation

msiGetValByKey	get metadata from structure
msiExecStrCondQuery	execute string conditional query
msiString2KeyValPair	convert string to key-value pair
msiAssociateKeyValuePairsToObj	add metadata
msiMakeGenQuery	create a query
msiExecGenQuery	execute a query
msiCloseGenQuery	release query buffers
msiGetContInxFromGenQueryOut	check for more rows
msiRemoveKeyValuePairsFromObj	remove metadata
msiGetMoreRows	get more rows from query

Data and directory manipulation

msiIsColl	check whether name is a collection
msiCollCreate	create a collection
msiDataObjCreate	create a file
msiDataObjRepl	replicate a file
msiDataObjChksum	checksum a file
msiDataObjUnlink	delete a file

System functions

msiGetSystemTime	get the system time
writeLine	write a line to a file or standard out
msiSleep	sleep

The complete set of workflow operators and basic micro-services are listed in [4]. Each community that implements management policies or validation criteria adds micro-services that implement required functions. Examples of additional micro-services include support for message passing (track status and debug support), partial I/O on files, invocation of remote web services, remote procedure execution, delayed and periodic execution of policies, template-based pattern analysis and extraction of metadata, XML parsing, bulk metadata loading, etc.

5. STRATEGIES

The performance that was demonstrated with the integrity rule can be improved. For the test environment, disk rotational latency dominated. There are multiple ways to minimize the latency and decrease the time spent by the rule engine:

- Aggregate files into a container such as a tar file. The system performance will then be limited primarily by the time to read and checksum a large file. The number of queries of the metadata catalog and the number of iterations will be minimized. The downside is that if a problem is found, a large file will need to be written.
- Implement a workflow operator that integrates queries on the metadata catalog with loops over the result set. This simplifies the logic, and minimizes the time required for executing the loop logic. However, at best this saves 6 milliseconds out of a per file execution time of 21 milliseconds.
- Implement the entire logic in a micro-service. This moves manipulations out of the rule language into C code. This approach has been implemented in the micro-service msiAutoReplicateService. The micro-service input parameters are the collection name, whether recursion is enabled across sub-directories, the number of required replicas, the name of the resource group that contains all of the storage locations, and an optional e-mail address for sending completion notifications. The msiAutoReplicateService micro--service does not implement a log file for operations performed, does not control the execution rate, and does not do load leveling. However it does verify the checksum of each file and replace missing replicas.

6. CONCLUSION

The implementation of policies that automate administrative tasks, and validate assessment criteria is straightforward within the iRODS policy-based data management system. The extensibility of iRODS makes it possible to add new policies and add new procedures for data management related tasks. Part of the power of the system comes from characterization of data management as operations applied on virtual name spaces. The iRODS data grid manages virtual name spaces for users, objects collections, storage systems, state information, policies, and procedures. For each name space, a set of operations are defined that can manipulate the associated entities. For each set of operations, a virtualization mechanism is defined that enables application of the operations across multiple types of storage and data management infrastructure.

Examples of the types of operations are shown in the following chart.

Name Space	**Operations**	**Virtualization interface**
Users	Authentication, authorization, groups	GSSAPI / PAM
Objects	Partial I/O, move, copy, replicate, share	Posix I/O & staging
Collections	Organization, browsing	System metadata
State information	Add, update, delete, query	Catalog interface to DBMS
Resources	Load leveling, fault tolerance, grouping	Storage drivers
Policies	Management, versions, administrative, verification	Policy language
Procedures	Basic functions on each name space	Workflows

The iRODS data grid provides a single sign-on environment for users to access a shared collection that may be stored across multiple administrative domains. The user name space enables iRODS to manage access controls across the systems without having to establish accounts for each user at the remote storage location. The iRODS system is being upgrade to use Pluggable Authentication Modules to enable interaction with modern authentication systems. The Grid Security Service API (GSSAPI) supports authentication via Grid Security Infrastructure, Kerberos, and challenge response mechanisms.

The object name space can be used to register files, database queries, and soft links to data in remote resources. The iRODS data grid is being extended to support registration of workflows. This tightly couples input parameter files for a workflow to the output files created by running the workflow. Since the workflow is registered as an iRODS object, the same access controls apply to workflows as to files. It is possible to share workflows, put access controls on workflows, and re-execute workflows. This is an essential capability for reproducible science. The workflow that is executed uses the same rule language that is used to create procedures for policy enforcement. This means that one can register a policy into the data grid, manage the parameters associated with the policy, and archive the log files from each execution of the policy.

The formation of shared collections enables the organization of distributed files into a logical collection. It is then possible to browse the logical collection, associate metadata with the collection, and manage status flags on processing the collection. These capabilities were essential for implementing the integrity policy.

The management of persistent state information was also essential for the execution of the integrity policy. Metadata can be associated with an entity in any of the name spaces. Thus metadata can be applied to users, objects, collections, and storage systems. The mechanisms to update metadata, add new metadata attributes, and query metadata make it possible to manage long-running processes. All of the metadata are stored in a metadata catalog.

The resources name space is used to implement compound resources (disk caches in front of tape archives) and groups of resources. If the data are stored on a tape archive, the data are staged to a disk where the policies are then applied. Thus a request to checksum a file that is stored on tape automatically causes the file to be staged to a disk, and the checksum is then performed. Groups of resources can be used to define storage systems that support collective operations. A group of resources could be used with a policy that causes files to be automatically replicated across all storage systems in the group.

The policy name space supports versions of policies. The policies are stored in the metadata catalog. It is possible to add new versions of policies, define which policies will be applied at a specific storage location, distribute policies to storage systems, and list the set of policies that are being applied.

The procedure name space supports versions of micro-services. A policy can be defined that applies to a specific collection or type of file, or group of persons, with a specific version of a micro-service used to implement the associated procedure. This makes it possible for the data management system to support multiple collections that have different policies and procedures within the same generic infrastructure.

The expectation is that the extensibility enabled by the management of the multiple virtual name spaces, makes it possible for the same generic infrastructure to support data sharing (data grids), data publication (digital libraries), data processing (data pipelines), and data preservation (archives).

7. Listing of the Integrity Rule

```
schedulerReplicas {
# This rule requires iRODS version 3.1 (msiCloseGenQuery mods)
# The replicas for each file are updated to the most recent version
# Each file is checked to verify whether all required replicas exist and have valid checksums
# As replicas are created, the algorithm round robins through available storage vaults
# Checks that the number of storage resources used within a collection is greater than or
#   equal to the number of desired replicas.
# This uses a just in time scheduler that slows down the processing rate
#   to complete the task within the specified number of seconds (*Delt)
# Checks a TEST_DATA_ID parameter associated with the collection
#   to determine enable restarts after system interrupts
# Writes a log file stored as Check-Timestamp in directory *Coll/log
# get current time, Timestamp is YYY-MM-DD.hh:mm:ss
  msiGetSystemTime(*TimeS,"unix");
  msiGetSystemTime(*TimeH,"human");
  *NumBadFiles = 0;
  *NumRepCreated = 0;
  *NumFiles = 0;
  *Runsize = double(0);
  *Sleeptime = 0;
  *colldataID = "0";
#this is used to round robin through available storage resources
  *Jround = 0;
# check whether a collection was defined
  msiIsColl(*Coll,*Result, *Status);
  if(*Result == 0 || *Status < 0) {
    writeLine("stdout","Input path *Coll is not a collection");
    fail;
  }
#============ create a collection for log files if it does not exist ===============
  *LPath = "*Coll/log";
  msiIsColl(*LPath,*Result, *Status);
  if(*Result == 0 || *Status < 0) {
    msiCollCreate(*LPath, "0", *Status);
    if(*Status < 0) {
      writeLine("stdout","Could not create log collection");
      fail;
    }
  }
# create file into which results will be written
  *Lfile = "*LPath/Check-*TimeH";
  *Dfile = "destRescName=*Res++++forceFlag=";
  msiDataObjCreate(*Lfile, *Dfile, *L_FD);
# check whether the attribute TEST_DATA_ID has been set from a prior execution
  *Val = "0";
  msiExecStrCondQuery("SELECT COUNT(META_COLL_ATTR_NAME) where COLL_NAME = '*Coll' and
META_COLL_ATTR_NAME = 'TEST_DATA_ID'",*GenQOut2);
  foreach (*GenQOut2) {
    msiGetValByKey(*GenQOut2, "META_COLL_ATTR_NAME", *Val);
  }
  if(int(*Val) == 0) {
    *Str1 = "TEST_DATA_ID=0";
    msiString2KeyValPair(*Str1,*kvp);
    msiAssociateKeyValuePairsToObj(*kvp,*Coll,"-C");
    writeLine("*Lfile","added TEST_DATA_ID attribute to collection *Coll");
  }
# on a restart TEST_DATA_ID will be greater than 0
  msiMakeGenQuery("META_COLL_ATTR_VALUE", "COLL_NAME = '*Coll' and META_COLL_ATTR_NAME =
'TEST_DATA_ID'",*GenQInp2);
  msiExecGenQuery(*GenQInp2,*GenQOut2);
  foreach(*GenQOut2) {
    msiGetValByKey(*GenQOut2, "META_COLL_ATTR_VALUE",*colldataID);
  }
# *colldataID is the string identifier of the last file that has been checked
  msiCloseGenQuery(*GenQInp2, *GenQOut2);
  msiMakeGenQuery("count(DATA_NAME), sum(DATA_SIZE)","COLL_NAME = '*Coll' and DATA_ID > '*colldataID'", *GenQInp2);
# this counts all files that have not yet been checked including replicas
  msiExecGenQuery(*GenQInp2, *GenQOut2);
  foreach(*GenQOut2) {
    msiGetValByKey(*GenQOut2, "DATA_NAME", *num);
    msiGetValByKey(*GenQOut2, "DATA_SIZE", *sizetotal);
  }
  msiCloseGenQuery(*GenQInp2, *GenQOut2);
  *Size = double(*sizetotal);
  *Num = int(*num);
# expected execution time = 0.0161 (sec) * (number of files) + (total size) / (50 MBytes/sec)
  *Timeest = int(*Num / 62) + int(*Size / 50000000);
  writeLine("*Lfile","Estimated time is *Timeest seconds, total time is *Delt seconds, number of files is *Num,  and total size is *Size
bytes");
  writeLine("*Lfile","Number of required copies of a file is *NumReplicas");
  if(*Delt > 0 && *Size > 0) {
    *Fac = *Size / *Delt;
    writeLine("*Lfile", "Required analysis rate is *Fac bytes/second");
# identify the resources that were used for the collection
    msiMakeGenQuery("DATA_RESC_NAME","COLL_NAME = '*Coll' and DATA_ID > '*colldataID'",*GenQInpr);
    msiExecGenQuery(*GenQInpr,*GenQOutr);
    *Ir = 0;
    *Rlist = list();
    *Ulist = list();
    foreach(*GenQOutr) {
      msiGetValByKey(*GenQOutr,"DATA_RESC_NAME",*Str1);
      *Rlist = cons(*Str1,*Rlist);
      *Ulist = cons("0",*Ulist);
      writeLine("*Lfile","Collection *Coll uses storage resource *Str1");
      *Ir = *Ir + 1;
    }
    *Ulist0 = *Ulist;
    *Irm1 = *Ir - 1;
    if(*Ir < *NumReplicas) {
      writeLine("stdout","Required number of replicas, *NumReplicas, exceeds the number of storage vaults, *Ir");
      writeLine("*Lfile","Required number of replicas, *NumReplicas, exceeds the number of storage vaults, *Ir");
      fail;
    }
    msiCloseGenQuery(*GenQInpr, *GenQOutr);
    msiMakeGenQuery("order(DATA_ID), DATA_SIZE, DATA_NAME, COLL_NAME, DATA_CHECKSUM","COLL_NAME =
'*Coll' and DATA_ID > '*colldataID'",*GenQInp);
    msiExecGenQuery(*GenQInp, *GenQOut);
    msiGetContInxFromGenQueryOut(*GenQOut,*ContInxNew);
    *ContInxOld = 1;
    while (*ContInxOld > 0) {
      foreach(*GenQOut) {
        msiGetValByKey(*GenQOut, "DATA_SIZE", *Sizedata);
        msiGetValByKey(*GenQOut, "DATA_ID", *newdataID);
        msiGetValByKey(*GenQOut, "DATA_NAME", *Name);
        msiGetValByKey(*GenQOut, "COLL_NAME", *Colln);
# first update all replicas to the most recent version
        msiDataObjRepl("*Colln/*Name","updateRepl=++++irodsAdmin=",*Status2);
        if(*Status2 != 0) {
          writeLine("*Lfile","Unable to update replicas to most recent version for *Colln/*Name");
        }
# get all replica numbers for this file
        msiMakeGenQuery("DATA_REPL_NUM,DATA_CHECKSUM,DATA_RESC_NAME", "COLL_NAME = '*Colln' and
DATA_NAME = '*Name'", *GenQInp4);
        msiExecGenQuery(*GenQInp4, *GenQOut4);
        *Numr = 0;
        *Ulist = *Ulist0;
        foreach(*GenQOut4) {
          *Numr = *Numr + 1;
          msiGetValByKey(*GenQOut4, "DATA_REPL_NUM", *Repln);
          msiGetValByKey(*GenQOut4, "DATA_CHECKSUM", *Chk);
          msiGetValByKey(*GenQOut4, "DATA_RESC_NAME", *Rescn);
          msiDataObjChksum("*Colln/*Name", "replNum=*Repln++++forceChksum=", *Chkf);
          if(int(*Chk) == 0) {
            *Chk = *Chkf;
          }
# save list of resources and pick resource to use as source
          if(int(*Chk) == int(*Chkf)) {
            for(*J=0;*J<*Ir;*J=*J+1) {
              if(elem(*Rlist,*J) == *Rescn) {
                *Ulist = setelem(*Ulist,*J,"1");
                *Resource = *Rescn;
                break;
              }
            }
          }
          if (int(*Chk) != int(*Chkf)) {
            writeLine("*Lfile","Bad checksum for replica *Repln of file *Colln/*Name with DATA_ID *newdataID.");
            *NumBadFiles = *NumBadFiles + 1;
#           msiDataObjUnlink("objPath=*Colln/*Name++++replNum=*Repln", *Status);
            writeLine("*Lfile","Deleted replica *Repln of file *Colln/*Name");
            *Numr = *Numr - 1;
          }
        }
# test whether the required number of replicas exists
        if (*Numr != *NumReplicas) {
          *N = *NumReplicas - *Numr;
          if(*N > 0) {
            writeLine("*Lfile","File *Colln/*Name is missing *N replicas");
            for(*I = 0;*I<*N;*I=*I+1) {
# pick resource to use for storing replica, round robin through storage systems without a replica
              *Check = false;
              for(*L = 0;*L<*Ir;*L=*L+1) {
                *J = *L + *Jround;
              if(*J >= *Ir) {
                *J = *J - *Ir;
              }
              *Stu = elem(*Ulist,*J);
              if(*Stu == "0") {
                *Resu = elem(*Rlist,*J);
                msiDataObjRepl("*Colln/*Name","destRescName=*Resu++++rescName=*Resource++++irodsAdmin=",*Status1);
                *NumRepCreated = *NumRepCreated + 1;
                *Ulist = setelem(*Ulist,*J,"1");
                *Check = true;
                *Jround = *J + 1;
                if(*Jround >= *Ir) {
                  *Jround = 0;
                }
                if(*Status1 < 0) {
                  *NumRepCreated = *NumRepCreated - 1;
                  writeLine("*Lfile","Unable to create a replica for *Colln/*Name on resource *Resu");
                  *Check = false;
                }
              }
              if(*Check == true) {
                break;
              }
            }
          }
        }
      }
      msiCloseGenQuery(*GenQInp4,*GenQOut4);
# slow rate at which are processing collection
      *Runsize = *Runsize + double(*Sizedata);
      msiGetSystemTime(*timei, "unix");
      *timerun = int(*TimeS) + *Runsize / *Fac;
      *delt = *timerun - int(*timei);
      if (*delt > 4) {
        msiSleep(str(*delt), "0");
        *Sleeptime = *Sleeptime + *delt;
      }
      *NumFiles = *NumFiles + 1;
    }
    *Str1 = "TEST_DATA_ID=*colldataID";
    msiString2KeyValPair(*Str1, *kvp1);
    msiRemoveKeyValuePairsFromObj(*kvp1, *Coll, "-C");
    *colldataID = *newdataID;
    *Str2 = "TEST_DATA_ID=*colldataID";
    msiString2KeyValPair(*Str2, *kvp);
    msiAssociateKeyValuePairsToObj(*kvp, *Coll, "-C");
    writeLine("*Lfile", "Reset TEST_DATA_ID to *colldataID for collection *Coll");
    *ContInxOld = *ContInxNew;
    if (*ContInxOld > 0) {
      msiGetMoreRows(*GenQInp,*GenQOut,*ContInxNew);
    }
  }
  writeLine("*Lfile", "Number of logical file names tested is *NumFiles, total size checked is *Runsize bytes, and total time slept is
*Sleeptime seconds");
  writeLine("*Lfile", "Number of bad files is *NumBadFiles, and number of replicated files created is *NumRepCreated");
 #reset TEST_DATA_ID status flag to zero
  msiExecStrCondQuery("select META_COLL_ATTR_VALUE where COLL_NAME = '*Coll' and META_COLL_ATTR_NAME =
'TEST_DATA_ID'", *GenQOut2);
  foreach(*GenQOut2) {
    msiGetValByKey(*GenQOut2, "META_COLL_ATTR_VALUE",*colldataID);
  }
  *Str1 = "TEST_DATA_ID=*colldataID";
  msiString2KeyValPair(*Str1, *kvp1);
  msiRemoveKeyValuePairsFromObj(*kvp1, *Coll, "-C");
```

```
    *Str2 = "TEST_DATA_ID=0";
    msiString2KeyValPair(*Str2, *kvp);
    msiAssociateKeyValuePairsToObj(*kvp,*Coll,"-C");
    writeLine("*Lfile", "Reset TEST_DATA_ID to 0 indicating a successful completion of the integrity check");
  }
# Calculate actual elapsed time
  msiGetSystemTime(*TimeE, "unix");
  *Del = int(*TimeE) - int(*TimeS);
  writeLine("*Lfile","Total elapsed time is *Del seconds");
}
INPUT *Coll=$"/tempZone/home/rods/test21000", *Delt=10, *NumReplicas = 2, *Res="demoResc"
OUTPUT ruleExecOut
```

8. ACKNOWLEDGMENTS

The development of the iRODS data grid and the research results in this paper were funded by the NSF Office of Cyberinfrastructure OCI-0848296 grant, "NARA Transcontinental Persistent Archive Prototype", (2008-2012), the NSF ITR 0427196 grant, "Constraint-based Knowledge Systems for Grids, Digital Libraries, and Persistent Archives" (2004-2007), the NSF SDCI 0721400 grant, "SDCI Data Improvement: Data Grids for Community Driven Applications" (2007-2010), the NSF OCI-1032732 grant, "SDCI Data Improvement: Improvement and Sustainability of iRODS Data Grid Software for Multi-Disciplinary Community Driven Application," (2010-2013), and the NSF Cooperative Agreement OCI-094084, "DataNet Federation Consortium", (2011-2013).

9. REFERENCES

[1] Rajasekar, R., Wan, M., Moore, R., Schroeder, W., Chen, S.-Y., Gilbert, L., Hou, C.-Y., Lee, C., Marciano, R., Tooby, P., de Torcy, A., and Zhu, B.. 2010. *iRODS Primer: Integrated Rule-Oriented Data System*, Morgan & Claypool. DOI= 10.2200/S00233ED1V01Y200912ICR012.

[2] Nief, J.-Y. 2010. iRODS usage at CC-IN2P3. In *Proceedings iRODS User Group Meeting 2010: Policy-Based Data Management, Sharing, and Preservation.*CreateSpace. ISBN-13: 978-1452813424.

[3] Chiang, G.-T., Clapham, P., Qi, G., Sale, K., and Coates, G. 2011. Implementing a genomic data management system using iRODS in the Wellcome Trust Sanger Institute. In *BMC Bioinformatics* 12:361. DOI=10.1186/1471-2105-12-361.

[4] Ward, J., Wan, M., Schroeder, W., Rajasekar, A., de Torcy, A., Russell, T., Xu, H., Moore, R. 2011. *The integrated Rule-Oriented Data System (iRODS 3.0) Micro-service Workbook*, DICE Foundation, November 2011, ISBN: 9781466469129, Amazon.com