Analysis of DIRAC’s behavior using model checking with process algebra

Daniela Remenska¹,³, Jeff Templon¹, Tim Willemse⁴, Henri Bal³, Kees Verstoep³, Wan Fokkink³, Philippe Charpentier², Ricardo Graciani Diaz⁵, Elisa Lanciotti², Stefan Roiser², Krzysztof Ciba²

¹NIKHEF (NL)  ²CERN  ³Free University Amsterdam (NL)  ⁴Technical University Eindhoven (NL)  ⁵University of Barcelona (ES)

E-mail: danielar@nikhef.nl

Abstract. DIRAC is the grid solution developed to support LHCb production activities as well as user data analysis. It consists of distributed services and agents delivering the workload to the grid resources. Services maintain database back-ends to store dynamic state information of entities such as jobs, queues, staging requests, etc. Agents use polling to check and possibly react to changes in the system state. Each agent’s logic is relatively simple; the main complexity lies in their cooperation. Agents run concurrently, and collaborate using the databases as shared memory. The databases can be accessed directly by the agents if running locally or through a DIRAC service interface if necessary. This shared-memory model causes entities to occasionally get into inconsistent states. Tracing and fixing such problems becomes formidable due to the inherent parallelism present. We propose more rigorous methods to cope with this. Model checking is one such technique for analysis of an abstract model of a system. Unlike conventional testing, it allows full control over the parallel processes execution, and supports exhaustive state-space exploration. We used the mCRL2 language and toolset to model the behavior of two related DIRAC subsystems: the workload and storage management system. Based on process algebra, mCRL2 allows defining custom data types as well as functions over these. This makes it suitable for modeling the data manipulations made by DIRAC’s agents. By visualizing the state space and replaying scenarios with the toolkit’s simulator, we have detected race-conditions and deadlocks in these systems, which, in several cases, were confirmed to occur in the reality. Several properties of interest were formulated and verified with the tool. Our future direction is automating the translation from DIRAC to a formal model.

1. Introduction
The grid storage and computing resources for the LHCb experiment are distributed worldwide. To cope with processing the vast amount of data generated by the LHC, a solution called DIRAC [1, 2], has been developed by the LHCb community. DIRAC consists of many distributed services and light-weight agents which deliver workload to the resources. Services maintain database back-ends to store dynamic state information of entities such as jobs, queues, staging requests, etc. Agents use polling to check and possibly react to changes in the service states. The logic of each agent is relatively simple; the overall system complexity emerges from the cooperation
among them. These agents run concurrently, and communicate using the services’ databases as a shared memory.

This shared-memory model causes entities to occasionally get into inconsistent, or incorrect states. Fixing such behaviors becomes formidable due to multiple factors: the inherent parallelism present among the system components deployed on different physical machines, the size of the implementation (around 150000 lines of Python code), and the distributed knowledge of different subsystems within the collaboration.

In this paper we propose the use of rigorous (formal) methods for improving software quality. Model checking is one such technique for analysis of an abstract model of a system. Unlike conventional testing, it allows full control over the execution of parallel processes, and supports automated exhaustive state-space exploration. We used the mCRL2 language and toolset [3] to model the behavior of two related DIRAC systems: the Workload Management (WMS) and the Storage Management System (SMS). Based on process algebra laws, mCRL2 allows defining custom data types as well as functions over these. This makes it suitable for modeling the data manipulations made by DIRAC’s agents. Visualizing the state space and replaying scenarios with the toolkit’s simulator helped us gain insight into the system behavior, incrementally improve the model, and to already detect race-conditions and deadlocks, which were confirmed to occur in the real system. Some of them were a result of simple coding bugs; others unveiled more elementary design problems. We further formulated, formalized and verified several properties of interest, that the model checker could automatically probe.

The paper is organized as follows. Section 2 introduces DIRAC, focusing on two subsystems chosen as case studies. Section 3 gives a brief overview of the mCRL2 language, and describes our approach to abstracting and modeling the behavior of these subsystems. Section 4 presents the analysis with the mCRL2 toolset and the issues detected. Section 5 concludes and discusses future work.

2. DIRAC background

Since the beginnings of its development in 2002, until today, DIRAC gradually evolved [4] into a complete community grid solution for data and job management. Today it covers all major LHCb tasks starting with the raw data transfer from the detector to the grid storage, several steps of data processing, up to the final user analysis. Python was chosen as the implementation language, as it enables rapid prototyping of new features. DIRAC follows the Service Oriented Architecture (SOA) paradigm. More details about its main components can be found in [1]. In the following, we focus on two related subsystems of DIRAC. These are the ones where we felt that problematic state changes are often encountered. Understanding their behavior is necessary for interpreting the issues that were discovered during the model analysis and verification.

2.1. Workload Management System

The driving force of DIRAC is the Workload Management System (WMS). Taking into account the heterogeneous, dynamic, and high-latency nature of the distributed computing environment, a Pilot Job paradigm was chosen as an efficient way to implement a pull scheduling mechanism.

The flowchart describing the evolution of a job’s states is depicted in Figure 1. The state-changes in the lifetime of a job are orchestrated by plethora of WMS agents and services. After submission, the complete job description is placed in the DIRAC job repository (the Job DB). Before jobs become eligible for execution, a chain of optimizer agents checks and prioritizes them in queues, utilizing the parameters information from the Job DB. If the requested data resides on tape storage, the Job Scheduling Agent will pass the control to a specialized Stager service (part of the SMS explained in the next section), before placing the job in a Task Queue (“Waiting”). Based on the complete list of pending payloads, a specialized Task Queue Director agent submits pilots to the computing resources via the available job submission
middleware (e.g., gLite WMS). After a Matcher service pulls the most suitable payload for a pilot (“Matched”), a Job Wrapper object is created on the worker node, responsible for retrieving the input sandbox, performing software availability checks, executing the actual payload on the worker node (“Running”), and finally uploading any output data necessary (“Done” or “Failed”).

At the same time, a Watchdog process is instantiated to monitor the behavior of the Job Wrapper and send heartbeat signals to the monitoring service. It can also take actions in case resources are soon to be exhausted, the payload stalls, or a management command for killing the payload is received.

Although the grid storage resources are limited, it is essential to keep all data collected throughout the experiment’s run. Tape back-ends provide a reliable and cheap solution for data storage. The additional workflow step necessary for input data files residing on tape is carried out inside the Storage Management System (SMS).

2.2. Storage Management System
The SMS provides the logic for pre-staging files from tape to a disk cache frontend, before a job is able to process them. Smooth functioning of this system is essential for production activities which involve reprocessing older data with improved physics software, and happens typically several times per year.

A simplified view of the system is shown in Figure 2. The workflow is initiated with the Job Scheduling Agent detecting that a job is assigned to process files only available on tape storage. It sends a request for staging (i.e., creating a cached replica) to the Storage Manager Handler service with the list of files and a callback method to be invoked when the request has been processed. The Storage Management DB is immediately populated with records which are processed by a sequence of agents in an organized fashion. The relevant tables in the SMS DB are the Tasks and CacheReplicas, whose entities maintain a state observed and updated by these agents. Tasks maintain general information about every job requesting a service from the SMS. The details about every file (i.e., the Storage Element where it resides, the size, checksum, number of tasks that requested it), are kept in the CacheReplicas table. Other auxiliary tables maintain the relationship between these entities.

Figure 1. Job state machine [5]

Figure 2. Storage Management System
The processing begins with the Request Preparation Agent. It selects all the “New” replica entries, checks the integrity of the request by confirming that the requested replicas are properly registered in the File Catalog (FC), and retrieves their metadata. In case of problematic catalog entries, it can update the state of the CacheReplicas and the related Tasks entries to “Failed”. Non-problematic files are updated to a “Waiting” state. The Stage Request Agent is responsible for placing the actual staging requests for all “Waiting” entries, via dedicated storage middleware that communicates with the tape backends. These requests are grouped by Storage Element (SE) prior to submission, and carry information about the requested (pin) lifetime of the replicas to be cached. If certain pathologies are discovered (i.e., lost, unavailable, or zero-sized files on tape), it can update the corresponding entries to “Failed” in a similar manner. Otherwise, they are promoted to “StageSubmitted”. The agent responsible for monitoring the status of submitted requests is the Stage Monitor Agent. It achieves this by interrogating the storage middleware to see if the “StageSubmitted” files are successfully replicated on disk cache. In case of success, the CacheReplicas and their corresponding Tasks entries are updated to “Staged”. Various circumstances of tape or middleware misbehavior can also fail the staging requests. The last one in the chain is the Request Finalization Agent. The Tasks which are in their final states (“Staged” or “Failed”) are cleared from the database, and callbacks are performed to the WMS, which effectively wakes up the corresponding jobs. If there are no more associated Tasks for particular replicas, the respective CacheReplicas entries are also removed.

Observing the behavior of this system in 2011 reprocessing campaign, there were multiple instances where tasks or replicas have become stuck, effectively blocking the progress of jobs. To temporarily alleviate such problems, the status of these entries is typically manually reset to the initial “New” state, so that agents can re-process them from scratch. Occasionally, error messages were reported from unsuccessful attempts of the SMS service to update non-existent table records.

3. Modeling DIRAC with mCRL2
Distributed systems, such as DIRAC, are commonly modeled by a directed, edge-labeled graph referred to as a Labeled Transition System (LTS). The nodes in the graph represent the states of the system and the edge labels represent atomic events such as reading, sending and successful communications within the system. Behaviors of a distributed system are modeled by the sequence of edge labels obtained by traversing along the edges of the graph. Multiple edges emanating from a single node indicate that the state represented by the node may possibly evolve (non-deterministically) in different ways.

3.1. The mCRL2 language and toolset
The language mCRL2 [6], which has its roots in process algebraic theories, can be understood as a language for specifying LTSs. Processes, representing LTSs, can be composed using operators such as sequential composition and alternative composition to obtain new processes. The basic building blocks of the language are actions such as read and send, which can carry data parameters. If processes $p$ and $q$ represent some system, their alternative composition, denoted $p + q$, models a nondeterministic choice between behaviors. The system can either behave as $p$, or as $q$, and in the systematic exploration of the model, both cases are taken into account. Sequential composition simply allows to “glue” the behavior of two processes. That is, $pq$ behaves just as process $p$, and, upon successful termination, it continues to behave as process $q$. In addition, new processes can be specified through recursion, by composing processes in parallel, etc.

Apart from describing processes, mCRL2 has a data language, to describe realistic systems where data influences behavior. The data language has built-in standard data types such as booleans, integers, infinite lists and sets over arbitrary data types. In addition, users can define
their own custom data types. Process behavior can be influenced by the values of the data parameters they carry. For this, mCRL2 offers if-then-else constructs of the form \( b \rightarrow p \bowtie q \) that behave as process \( p \) if the boolean expression \( b \) holds and as process \( q \) otherwise. Such features make the language quite suited for modeling distributed and concurrent systems.

3.2. From DIRAC to mCRL2

Any formal analysis uses a simplified description (abstraction) of the real system. Abstraction aims at reducing the program’s state space in order to overcome the resource limitations encountered during model-checking. Given the recurrent invalid state transitions of entities within DIRAC, we considered the possible race conditions caused by multiple agents updating the shared service states to be the target of our analysis. We limited the scope to the analysis of the following entities: Tasks (SMS), CacheReplicas (SMS) and Jobs (WMS). In the following, we use the SMS as a case study for describing the general modeling approach. The same principles can be applied to other DIRAC subsystems.

3.2.1. Control Abstractions

As already mentioned, all agents repeat the same logic in every subsequent iteration: first they read some entries of interest from the database, then they process the cached data, execute some actions, and finally they may write back or update entries, based on decisions from the processing step. They can be naturally modeled as recursive processes. Take as an example the code snippet in Listing 1, from the Request Preparation Agent. Although much of the code is omitted for clarity, the necessary parts for illustration of the basic idea are

---

Listing 1. RequestPreparationAgent.py code excerpt

```python
def prepareNewReplicas( self ):
    res = self.getNewReplicas()
    if not res['Value']:
        gLogger.info('There were no New replicas found')
        return res

    # Obtain the replicas from the FileCatalog
    res = self.__getFileReplicas( fileSizes.keys() )
    if not res['OK']:
        return res

    failed = update( res['Value'][ 'Failed' ]
    terminal = res['Value'][ 'ZeroReplicas' ]
    fileReplicas = res['Value'][ 'Replicas' ]

    replicaMetadata = []
    for lfn, requestedSEs in replicas.items():
        for requestedSE, replicaID in requestedSEs.items():
            if not requestedSE in fileReplicas.keys():
                terminalReplicaIDs[ replicaID ] = "LFN not registered at requested SE"
                replicas[lfn].pop(requestedSE)
                else:
                    replicaMetadata.append( ( replicaID, lfnReplicas[ requestedSE ]

    # Update the states of the files in the database
    if terminalReplicaIDs:
        gLogger.info("%s replicas are terminally failed.", len(terminalReplicaIDs))
        self.storeClient.updateReplicaFailure(terminalReplicaIDs)

    if replicaMetadata:
        gLogger.info("%s replica metadata to be updated.", len(replicaMetadata))
        # Sets the Status=’Waiting’ of CacheReplicas records
        # that are OK with catalogue checks
        res = self.storeClient.updateReplicaInformation( replicaMetadata )

    return S_OK()```

---

5
kept. The first highlighted statement is the selection of all "New" CacheReplicas entries. What follows is retrieval of their metadata from the FC, external to this subsystem. Subsequently, list and dictionary manipulations are done to group the retrieved data depending on the outcome. Two lists of replica IDs are built before the last step: one for the problematic catalog entries, and one for the successful sanity checks.

Finally, the last two highlighted code segments update the states of the corresponding CacheReplicas to "Failed" and "Waiting" respectively.

Instead of tracing back and modeling all variables on which the two final lists depend, we can use nondeterminism. It is not known upfront which branch execution will follow for a particular replica, as it depends on external behavior (i.e. the interactions of the system with its environment). By stubbing out the communication with the LFC and most of the local variable manipulations that follow, and replacing them with a nondeterministic choice between the two ultimate state updates, we can include both possibilities in the model behavior, and still preserve correctness. Of course, depending on the context, some variable values cannot be ignored, in which case determinism can be added using an if-then-else mCRL2 statement. All relevant selection and update statements are translated into actions parametrized with data.

### 3.2.2. Data Abstractions

The CacheReplicas table contains more information besides the Status field. Each record has a unique identifier, descriptive data such as the storage where it resides, its full path, checksum, timestamps, etc. Model checking can only be performed on closed models, where the domains of all variables are finite. Since we are only interested in state transitions, we can collapse most of this descriptive data, and represent this entity as a user-defined type in mCRL2:

\[
\text{sort CacheReplicas = struct Start | New | Waiting | StageSubmitted | Staged | Failed | Deleted;}
\]

This defines an enumerated data type with all possible states. The Tasks entity is modeled in a similar manner.

The shared database is modeled as a process that keeps the data described above in its local memory. The processes modeling the agents exchange information with this database process via actions that are enforced to communicate. Finally, the model is put together as a parallel composition of all communicating processes. Although the WMS model is larger, it is obtained using the same approach. The complete models are available at [7].

### 4. Analysis and Issues

The purpose of model checking is to prove that the modeled system exhibits certain behavior (requirement), or alternatively, to discover problems. The operation of a model checker closely resembles graph traversal. Nodes of the graph represent the states of the system, while the edges connecting them represent transitions, or state changes. The collection of all possible states and transitions forms the state space, or an LTS. Typically, model checkers examine all reachable states and execution paths in a systematic and fully automated manner, to check if a certain property holds. In case of violation of the examined property, a counterexample is provided as a precise trace in the model, showing which interleavings of actions of the parallel components led to the violation.

After the models were written, their state space was explicitly generated and stored. The execution times and the resulting numbers of states are presented in Table 1. Generating the LTS with the mCRL2 toolset can be a time consuming process, since the state space typically grows exponentially with the number of parallel processes in the model.
Table 1. mCRL2 model statistics

| System | States       | mCRL2 LoC | Python LoC | Generation time |
|--------|--------------|-----------|------------|-----------------|
| SMS    | 18,417       | 432       | 2,560      | <10 sec.        |
| WMS    | 160,148,696  | 663       | 15,042     | ~50 hr.         |

4.1. Simulation

Apart from verification, the mCRL2 toolset has a rich set of tools for analysis of the modeled system. The XSim simulator allows to replay scenarios and inspect in detail the current state and all possible transitions in the model, for every execution step. This process was already valuable for understanding the system and identifying interesting behaviors that were later included in the requirements. We want to stress that this is especially useful when building models of existing implementations, where at first glance it is not very clear which (un)desired properties need to be formulated and automatically probed, before they actually surface in the real system. One such problematic behavior was recently reported in DIRAC’s production (Figure 3a), where a job had transited between two different terminating states, “Failed” and “Done”. Replaying the behavior with XSim revealed the same trace in the WMS model (Figure 3b).

This was a result of several (difficult to reproduce in production) circumstances: The JobWrapper process was continuously trying to access a file already migrated to tape. Meanwhile, the Stalled Job Agent responsible for monitoring the pilot declared the job as “Stalled”, and ultimately as non-responsive (“Failed”), since its child process was busy with data access attempt for a long time. However, once data access succeeded, the JobWrapper finally started executing the actual payload and reported a “Running” status. The JobWrapper assumes that nothing has happened to the status of a job once it brings it to a “Running” state, and only reports a different MinorStatus value afterwards. The logic is such because it is naturally expected that this process has exclusive control over the rest of the job workflow, once it starts running. Fixing such a problem without compromising efficiency is not trivial.

![Logging info of a DIRAC job](image1)

![XSim simulator trace](image2)

Figure 3. Invalid job state transitions
4.2. Visualization
Reasonably small LTSs can be easily visualized with the interactive GUI tools, as the tools employ smart clustering techniques to reduce the complexity of the image. For instance, the SMS model, with around 18000 states, was visualized with DiaGraphica. Figure 4 shows a projection (clustering) of the state space on the CacheReplicas memory process, which resulted in only 7 interesting states for this process alone. The clustering allowed us to see precisely which chain of actions by the concurrent agents advanced the state of the CacheReplicas entity.

In the modeling phase we discovered a problematic SMS behavior, blocking the progress of staging tasks (and as a result the progress of jobs). The origin of the deadlock had been a trivial human logic flaw introduced in coding. At a particular circumstance, when a “New” task enters the system and all associated replicas are already “Staged” (possibly by other jobs requiring the same input files), its status would immediately be promoted to “Done”. Such tasks would never be picked up by SMS and called-back to the appropriate job, since the agent responsible for this final step would only look for “Staged” tasks. A proper state machine implementation instead of hardcoding states in SQL queries can prevent such bugs.

4.3. Model checking
In addition to simulation and visualization of the models, we formulated several requirements on both systems, that the model checker would automatically probe:

1. Each task in a terminating state (“Failed” or “Staged”) is eventually removed from the system. (progress)
2. A deleted task will never be referenced for transition to any other state. (safety)
3. Once a job has been killed, it cannot resurrect and start running. (safety)

Both requirements 1 and 2 were found to be violated, as can be seen from the traces (Figure 5). The race conditions manifest themselves in a few subtle interactions between the SMS storage and the agents. In both cases, at step 4 the Stage Request Agent issues prestage requests and as a result the CacheReplicas entries (second element of the State column) are updated to “StageSubmitted”. Between the moment that this agent has selected the corresponding Task to update to the same state, to the point where the update is done (last step in both traces), other agents may have monitored these replica records and updated them to a new state, along with the associated Task. In practice, the manifestation of such race conditions depends on the speed of the agents propagating the state changes between the selection and update done by the Stage Request Agent. Such cases are nevertheless encountered in reality, when this agent has large lists of records to process. This results in a deadlock situation, as the Task will have no further state updates made by other agents, since from their perspective the state propagation is finished. To avoid these (and similar) deadlock situations, the logic was adapted: (1) to give precedence to CacheReplicas entries for files which are already in the disk cache, (2) to consider
all files that belong to the same Task within a single tape request, and (3) to explicitly re-check the status of all potential Tasks in each agent cycle, before updating them.

With respect to requirement 3, a counterexample (Figure 6) showed that, when staging was involved in the workflow of a job, the callback from the SMS was not properly handled. It awakened the job immediately to a “Waiting” (and eventually “Running”) state even if it had been manually “Killed” by production managers meanwhile. Such zombie jobs were discovered in the system on several occasions, and with the model at hand it was easy to replay the behavior and localize the problem. The implementation was fixed to properly guard the callback against such transitions.

![Figure 5. Violation of requirements 1 (top) and 2 (bottom)](image)

![Figure 6. “Zombie” job starts running after being killed](image)
5. Conclusions
In this paper we have applied a formal methods approach for analyzing DIRAC. We used two components as case studies: the Workload Management and the Storage Management System, which are the driving force of DIRAC. By creating an abstract model, simulating, visualizing, and model checking it with the mCRL2 toolset, we were able to gain insight into the system behavior and detect race-conditions and deadlocks, which were confirmed to occur in the real system. These problems are difficult to discover and trace in production, considering the number of concurrent components involved. With the model at hand, replaying the traces and localizing the problems becomes much more straightforward, overcoming many limitations of traditional testing techniques for large-scale distributed systems.

The investment in writing an abstract model in a formal language pays off proportionally to the size (number of components) of the distributed system. This holds especially for grid systems where a large number of concurrent components of the same kind deliver the overall system functionality. Once becoming proficient with the formal language notation, modeling such components is reduced to reapplying similar abstraction rules. Practitioners of model checking have already built sufficiently mature tools that can be utilized (almost) as blackboxes by more traditional software developers, hiding the mathematical theories under the hood. Although some domain-specific knowledge is always necessary, the approach we have outlined for obtaining an abstract model can be easily applied to other large-scale industrial systems of a similar nature. Our future efforts will be focused on automating (to a certain degree) the formal model generation, using the DIRAC implementation as input. Supplied with a proper Python grammar, tools like ANTLR[8] are of a great aid for translating between different implementation languages.

References
[1] Tsaregorodtsev A et al. DIRAC: A Community Grid Solution (Proc. CHEP 2007)
[2] Smith A and Tsaregorodtsev A DIRAC: Reliable Data Management for LHCb (Proc. CHEP 2007)
[3] Groote J, Keiren J, Stappers F, Wesselink J and Willemse T 2011 SPE 41 143–153
[4] Tsaregorodtsev A et al. DIRAC3 the new generation of the LHCb grid software (Proc. CHEP 2009)
[5] Grid production shifter guide for LHCb URL https://twiki.cern.ch/twiki/bin/view/LHCb/ProductionShifterGuide
[6] Groote J, Mathijssen A, Reniers M, Usenko Y and van Weerdenburg M 2006 Proc. MMOSS
[7] mCRL2 SVN repository URL https://svn.win.tue.nl/viewvc/MCRL2/trunk/examples/
[8] Parr T J, Parr T J and Quong R W 1995 Antlr: A predicated-ll(k) parser generator