Using Model Checking to Analyze the System Behavior of the LHC Production Grid

Daniela Remenska[a,c,*], Tim A.C.Willemse[b], Kees Verstoep[a], Jeff Templon[c], Henri Bal[a]

[a] *Department of Computer Science, VU University Amsterdam, The Netherlands*
[b] *Department of Computer Science, TU Eindhoven, The Netherlands*
[c] *NIKHEF National Institute for High-Energy Physics, Amsterdam, The Netherlands*

Abstract

DIRAC (Distributed Infrastructure with Remote Agent Control) is the grid solution designed to support production activities as well as user data analysis for the Large Hadron Collider "beauty" experiment. It consists of cooperating distributed services and a plethora of light-weight agents delivering the workload to the grid resources. Services accept requests from agents and running jobs, while agents actively fulfill specific goals. Services maintain database back-ends to store dynamic state information of entities such as jobs, queues, or requests for data transfer. Agents continuously check for changes in the service states, and react to these accordingly. The logic of each agent is rather simple; the main source of complexity lies in their cooperation. These agents run concurrently, and communicate using the services' databases as a shared memory for synchronizing the state transitions. Despite the effort invested in making DIRAC reliable, entities occasionally get into inconsistent states. Tracing and fixing such behaviors is difficult, given the inherent parallelism among the distributed components and the size of the implementation.

In this paper we present an analysis of DIRAC with mCRL2, process algebra with data. We have reverse engineered two critical and related DIRAC subsystems, and subsequently modeled their behavior with the mCRL2 toolset. This enabled us to easily locate race conditions and livelocks which were confirmed to occur in the real system. We further formalized and verified several behavioral properties of the two modeled subsystems.

Keywords:
model checking, process algebra, grid, LHC, distributed system, workflow

1. Introduction

The Large Hadron Collider beauty (LHCb) experiment [1] is one of the four large experiments conducted on the Large Hadron Collider (LHC) accelerator, built by the European Organization for Nuclear Research (CERN). Immense amounts of data are produced at the LHC accelerator, and subsequently processed by physics groups and individuals worldwide. The sheer size of the experiment is the motivation behind the adoption of the grid computing paradigm. The grid storage and computing resources for the LHCb experiment are distributed across several institutes in Europe. To cope with the complexity of processing the vast amount of data, a complete grid solution, called DIRAC (Distributed Infrastructure with Remote Agent Control) [2, 3], has been designed and developed for the LHCb community.

DIRAC forms a layer between the LHCb user community and the heterogeneous grid resources, to allow for optimal and reliable usage of these resources. It consists of many cooperating distributed services and light-weight agents which deliver the workload to the resources. Services maintain database back-ends to

*Corresponding author
Email addresses: danielar@nikhef.nl (Daniela Remenska), timw@win.tue.nl (Tim A.C.Willemse), c.verstoep@vu.nl (Kees Verstoep), templon@nikhef.nl (Jeff Templon), bal@cs.vu.nl (Henri Bal)

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 individual component is relatively simple; the overall system complexity emerges from the cooperation among them. Namely, these agents run concurrently, and communicate using the services' databases as a shared memory (blackboard paradigm [4]) for synchronizing state transitions of various entities.

Although much effort is invested in making DIRAC reliable, entities occasionally get into inconsistent states, leading to a loss of efficiency in both resource usage and manpower. Debugging and fixing the root of such encountered behaviors becomes a formidable mission due to multiple factors: the inherent parallelism present among the system components deployed on different physical machines, the size of the implementation ($\sim$150,000 lines of Python code), and the distributed knowledge of different subsystems within the collaboration. Depending on the frequency of certain heavier data (re)processing campaigns throughout the year, the occurrence of such inconsistencies can be up to several reported cases per month. However, since they have a blocking effect on these heavy grid activities with a strict deadline, locating and fixing the root cause of such a problem typically has a lower priority within the collaboration, compared to a temporary fix, such as manipulating database records or restarting processes manually.

In this paper we propose the use of rigorous (formal) methods for improving software quality. Model checking [5] is one such technique for analysis of an abstract model of a system, and verification of certain system properties of interest. Unlike conventional testing, it allows full control over the execution of parallel processes and also supports automated exhaustive state-space exploration. We used the mCRL2 language [6] and toolset [7] to model the behavior of two critical and related DIRAC components: the Workload Management (WMS) and the Storage Management System (SMS). Based on Algebra of Communicating Processes (ACP) [8], mCRL2 is able to deal with generic data types as well as user-defined functions for data transformation. This makes it particularly suitable for modeling the data manipulations made by DIRAC's agents. Visualizing the state space and replaying scenarios with the toolkit's simulator enabled us to gain insight into the system behavior and incrementally improve the model. Already using these techniques, critical race conditions and livelocks were detected and 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 application-specific properties. Once the model is in place, the toolset can automatically check for general system properties, such as deadlocks. We show how to formulate and check application-specific properties with the toolset. Furthermore, we outline a methodology to effectively tackle the classical state space explosion problem arising when the underlying model grows beyond the resources available to perform explicit model checking. For this, we borrow concepts from finite state machine formalisms for description of system properties. Extending the specification with a so-called monitoring automaton for tracking relevant transitions in the original model, can speed up the analysis and discovery of traces that lead to violation of a certain property. This idea is simple but flexible and generic enough to describe many temporal properties of concurrent programs.

The idea of modeling existing systems using formal techniques is as such not new. Earlier studies ([9–15]) mostly focused on modeling and verifying hardware or communication protocols, since the formal languages and tools at hand were not sufficiently mature to cope with more complex data-intensive distributed systems. More recently, success stories on modeling real-life concurrent systems with data have been reported ([16–21]). Tools for automatic translation of the system implementation language into formal language constructs can greatly simplify the analysis. However, this has so far been feasible only when the language of implementation is domain-specific, or alternatively, a reasonably small subset of a general-purpose language is considered for translation.

We believe that the challenges and results of this work are unique in a number of aspects. First, to the best of our knowledge, the code-base and the number of concurrent grid components engaged in providing DIRAC's functionality considerably outnumber most of the previous industrial cases. Second, the choice of Python as implementation platform has led to the prevailing usage of dynamic structures (whose types and sizes are determined at runtime) throughout DIRAC, challenging the transition to an abstract formal representation. We have established general guidelines on extracting a model outline from the implementation. These can be reused in many distributed systems to address concurrency issues arising from the use of shared storage for inter-process communication. Third, analysis of this kind is typically

performed after a problem has already surfaced in the real system, as a means to understand the events which led to it and test for possible solutions. We managed to come across an actual bug at the same time it was observed in practice, which increased our confidence in the soundness of the model.

The paper is organized as follows. Section 2 introduces the architecture of DIRAC, focusing on the 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. Work related to the domain of distributed systems verification is discussed in Section 5. Section 6 and 7 conclude and give directions for future work, respectively.

2. DIRAC: A Community Grid Solution

2.1. Architecture Overview

The development of DIRAC started in 2002 as a system for production of simulation data that would serve to verify physics theory, aspects of the LHCb detector design, as well as to optimize physics algorithms. It gradually evolved into a complete community grid solution for data and job management, based on a general-purpose framework that can be reused by other communities besides LHCb. Today, it covers all major LHCb tasks starting with the raw data transfer from the experiment's detector to the grid storage, several steps of data processing, up to the final user analysis. It provides a concurrent use of over 60K CPUs and 10M file replicas distributed over tens of grid sites, provided by the WLCG computing grid [22]. The community of users has grown to several hundreds, loading the LHC grid resources with up to 30K simultaneously running jobs (Fig. 1, top) and over 100K jobs executed per day, during peak processing periods. About 10PB of data is stored and visible in the LHCb file catalogs, with daily data transfers between storage units reaching 3Gbps (Fig. 1, bottom). Python was chosen as the implementation language, since it enables rapid prototyping and development of new features. DIRAC follows the Service Oriented Architecture (SOA) paradigm, accompanied by a network of lightweight distributed agents which animate the system. Its main components are depicted in Fig. 2.

The *services* are passive components that react to requests from their clients, possibly soliciting other services in order to fulfill the requests. They run as permanent processes deployed on a number of high-availability hosts (VO-boxes) at CERN, and store the dynamic system state information in database repositories. The user interfaces, agents or running jobs can act as clients placing the requests to DIRAC's services.

Agents are active components that fulfill a limited number of specific system functions. They run in different environments, depending on their mission. Some are deployed close to the corresponding services, while others run on the grid worker nodes. Examples of the latter are the so-called Pilot Agents, part of the Workload Management System explained in the following section. All DIRAC agents repeat the same logic in each iteration cycle: they monitor the service states, and respond by initiating actions (like job submission or data transfer) which update the states of various system entities.

Resources are software abstractions of the underlying heterogeneous grid computing and storage entities allocated to LHCb, providing a uniform interface for access. The physical resources are controlled by the site managers and made available through middleware services such as gLite [23].

The DIRAC functionality is exposed to users and developers through a rich set of command-line tools forming the DIRAC API, complemented by a Web portal for visually monitoring the system behavior and controlling the ongoing tasks. Both the Web and command-line *interfaces* ensure secure system access using X509 certificates.

The DIRAC hardware configuration of LHCb is still by far the most powerful one: it consists of about 10 servers (VO-boxes) located at CERN, hosting the central DIRAC services, including 6 MySQL database servers. Small fraction of the servers and databases are used for testing purposes, and are connected to the same pool of WLCG computing and storage resources. Each of the six LHCb Tier-1 computing centers accross Europe hosts an additional server as part of this cluster for redundancy, to ensure high service availability at any moment. Several other experiments are currently using DIRAC on an every day basis, with new user groups evaluating its functionality and performance before adopting it in production.

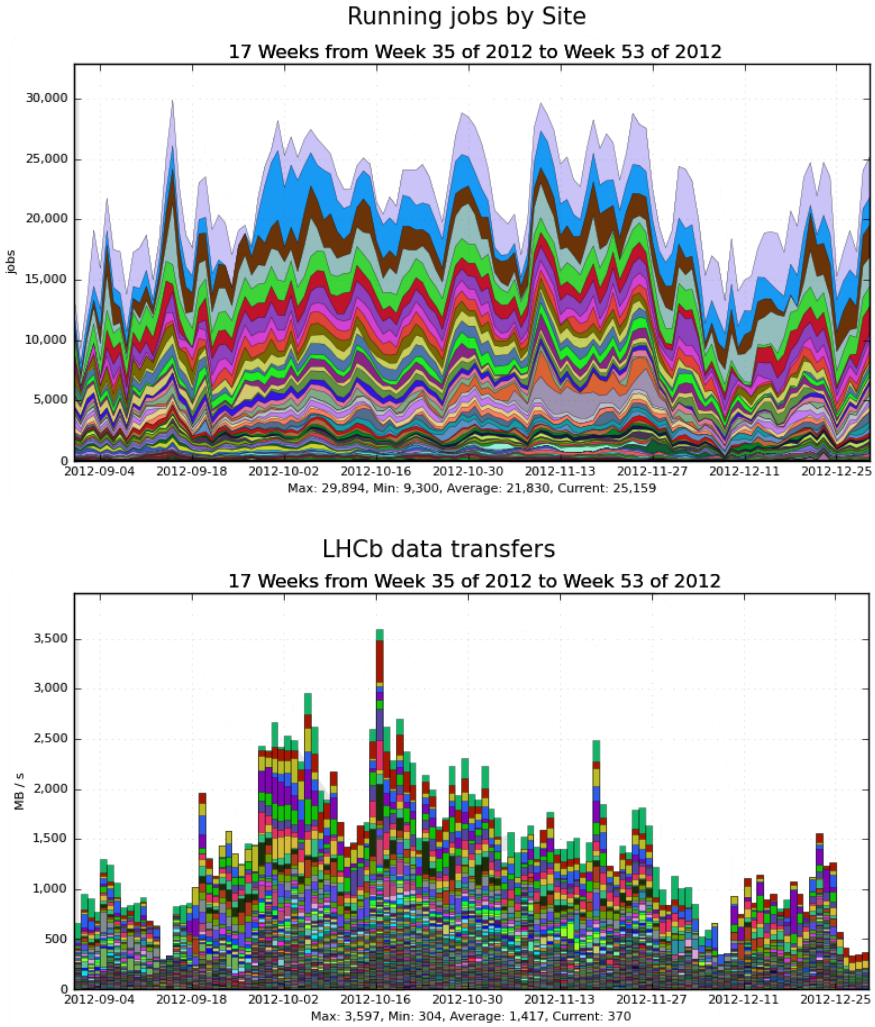


Figure 1: LHCb resource usage

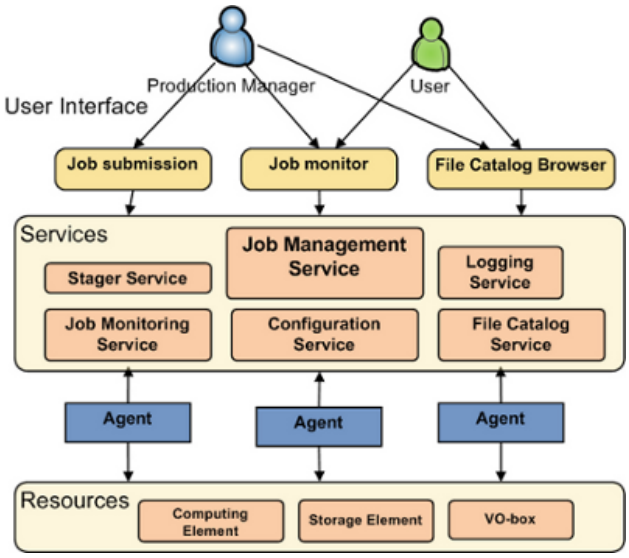


Figure 2: DIRAC Architecture overview

In the following, we focus on two related subsystems that are considered the backbone of DIRAC. These are the ones where problematic state changes are most often encountered, which are difficult to trace and correct. Understanding their behavior is essential for interpreting the issues that were discovered during the model analysis and verification.

2.2. 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*, illustrated in Fig. 3, was chosen as an efficient way to implement a pull scheduling mechanism. Pilot jobs are simply resource reservation processes without an actual payload defined a priori. They are submitted to the grid worker nodes with the aim of checking the sanity of the operational environment just before pulling and executing the real payload. This hides the fragility of the underlying resources and increases the job success rate, from the perspective of end users. Centralized *Task Queues* contain all the pending jobs, organized in groups according to their requirements. This enables efficient application of job priorities, enforcing fair-share policies and quotas, which are decided within the LHCb community, without the need of any effort from the grid sites. Originally, the DIRAC WMS submitted pilot jobs using the native gLite middleware tools, but this method was shown to be suboptimal. The gLite resource brokers

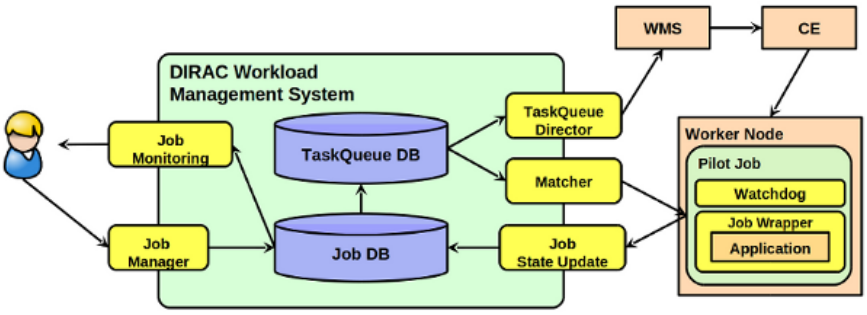


Figure 3: DIRAC Workload Management System [24]

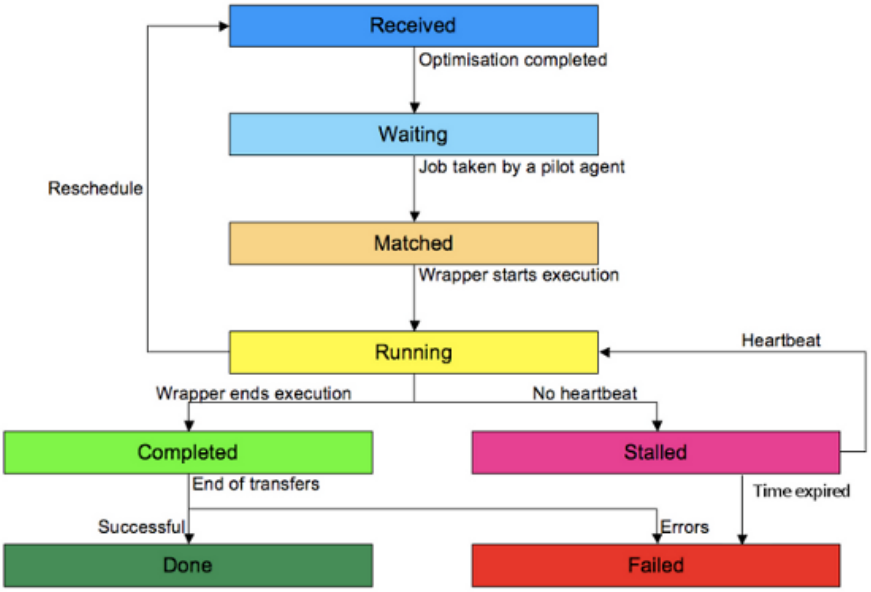


Figure 4: Job state machine [26]

are centralized components, and as such they do not have the capacity and flexibility to cope with the load of a community such as LHCb. Using multiple resource brokers, rather than a single one, was not the solution: all brokers use the same site state information and would always submit jobs to the most suitable one globally, causing overloads to certain sites while underloading others very often. In addition, the gLite information system was not reactive enough to propagate changes in site status information to the brokers. With the wide deployment of the CREAM CE service [25], DIRAC's WMS switched to submitting pilot jobs directly to sites. By interrogating the Computing Element status directly, the system reactiveness improved dramatically, making sites compete with each other for jobs and have a faster job turnaround.

The basic flowchart describing the evolution of a job's states is depicted in Fig. 4. After submission through the *Job Manager* ("Received"), 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"). The wrapper can catch any failure exit state reported by the running physics applications. 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 backends 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).

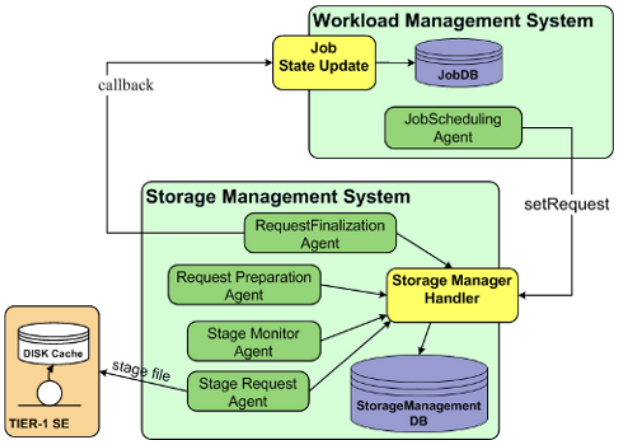


Figure 5: DIRAC Storage Management System

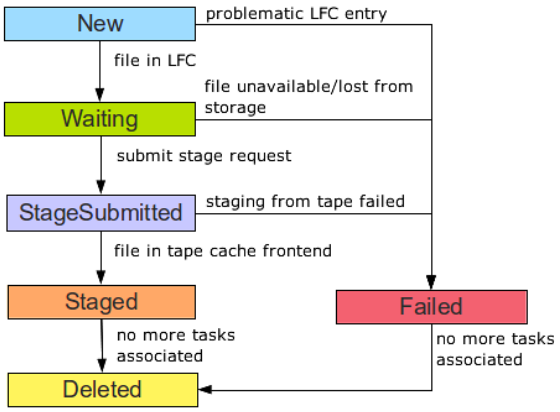


Figure 6: CacheReplicas state machine

2.3. Storage Management System

The DIRAC 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 Fig. 5. 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.

The processing begins with the *Request Preparation Agent*. It selects all the "New" replica entries, checks whether they are registered in a Logical File Catalog (LFC), 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. The state machine of the *CacheReplicas* entity is shown in Fig. 6.

This subsystem is still in an evolutionary phase. In multiple instances, tasks or replicas have become stuck, effectively blocking the progress of jobs. Tracing back the sequence of events which led to the inconsistent states is non-trivial. 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 are reported from unsuccessful attempts of the SMS service to update the state of non-existent table entries.

3. System modeling

3.1. The mCRL2 language and toolset

Distributed systems, such as the DIRAC system, 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, edges between nodes represent an atomic state change and the edge labelings 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 case edge labels are identical) in different ways.

In practice, even mundane distributed systems can give rise to rather large LTSs. Analyzing the behaviors described by such large LTSs, using e.g., *model checking*, can be quite expensive computationally. This can sometimes be overcome by minimizing the LTS. Of course, the behaviors, relevant to the analysis, should be preserved. In practice, one can safely use minimization with respect to *strong bisimulation* equivalence [27]; this equivalence allows one to remove duplications of behaviors and "fold" unfolded behaviors.

The language mCRL2 [6], which has its roots in process algebraic theories, can be understood as a language for specifying LTSs and reasoning about them modulo strong bisimulation. *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*, representing atomic events such as *read* and *send*. Special actions (constants) are defined, such as the *silent step* (τ) for denoting internal unobservable behavior, and the *deadlock* (δ) action upon which the system cannot perform any other action. If processes p and q represent some system, their alternative composition, denoted $p + q$, represents the system that behaves as p when the first action executed comes from p, and it behaves as process q otherwise; this models a *nondeterministic choice* between behaviors. In terms of the underlying LTS model, this may be understood as the operation that constructs a new LTS from the LTSs of p and q by creating a new top node that inherits all edges from the top nodes of the LTSs of p and q. Sequential composition simply allows to "glue" the behavior of two processes. That is, $p.q$ behaves just as process p, and, upon successful termination, it continues to behave as process q. In the underlying LTS models, this essentially creates a new graph that merges the top node of q's LTS with the nodes marked as terminal p's LTS. In addition, new processes can be specified through recursion, effectively defining LTSs that have loops, by filtering actions, by renaming (concurrently executed) actions, and by composing processes in parallel, etc.

Apart from describing processes, mCRL2 has a data language that facilitates describing realistic systems where data influences the system behavior. The data language has built-in support for standard data types such as Booleans ($Bool$), and integer (Int), natural (Nat) and positive (Pos) numbers, and infinite lists ($List(\_)$), sets ($Set(\_)$) and bags ($Bag(\_)$) over arbitrary data types. In addition, users have various ways in which they can define their own data types and operations on these. For instance, one can define one's own enumerated data types and reason about these.

Processes can be parametrized by data parameters and values. This way, the process behavior can be influenced by the values such parameters carry; mCRL2 offers two operators for this. First, given a process description p in which a variable d occurs unbound, process $\sum_{d:D}\ p$ describes the process in which non-deterministically a value is chosen for variable d. The $\sum$-operator can best be understood as a generalization of the binary + operator. Second, the *if-then-else* constructs of the form $b \rightarrow p \diamond q$ can be used to describe the process that behaves as process p if the Boolean expression b holds and as process q otherwise. Actions can carry data values, too, allowing one to model the exchange of information.

Concurrent processes can interact by enforcing synchronous communication. If actions *send* and *read* are intended to happen at the same time, rather than as single isolated actions in arbitrary order, this should be specified using the mCRL2 operator *comm* (Γ) in the following manner: $\Gamma_{\{send|read\ \rightarrow commAction\}}$

. In practise this means that that actions *send* and *read* must communicate to action *commAction*. All occurences of *send*|*read* will be replaced by *commAction*, provided that the data parameters that these two actions carry, match. Such features are indispensible for modeling distributed and concurrent systems.

3.2. From DIRAC to mCRL2

Any formal analysis uses a simplified description of the real system. Even in the best possible scenario, where the target implementation language and the modeling notation are very close ([20],[28]), it is practically impossible to avoid the use of abstraction to create a simplified model. Software implementations are large and often contain many details that are irrelevant for the intended analysis. In that sense, a model is an abstraction from reality, where certain implementation details are ignored.

Abstraction aims at reducing the program's state space in order to overcome the resource limitations [**?**] encountered during model-checking. Furthermore, specification languages describe *what* is being done, abstracting away from the details of *how* things are done. Then, the ultimate question is: *how do we establish correspondence between model and code?*

High-level design documents are a good starting point, but insufficient for building a sound model. In absence of more detailed up-to-date behavioral models, we based our models on the source code and discussions with developers. A popular abstraction technique for identifying code subsets that potentially affect particular variables of interest (slicing criteria) is program slicing [29]. It reduces the behavior of a program by removing control statements and data structures deemed irrelevant for the criteria. In the context of building a formal model, the slicing criteria depends on the focus of the final analysis, i.e., the intended properties to be checked. The result of such an approach is typically an over-approximation of the original program behavior. If the abstract model is checked and found to be in conformance with the property, one can be sure that the behavior of the original program satisfies this property too. In other words, the obtained abstraction is sound.

Unfortunatelly, so far the only practical applications of automated slicing have been on C/C++ and Java programs ([30–32]). Research has not yet matured on the topic of code slicing for dynamic languages, such as Python, so no automated tools exist to aid in the process of obtaining the abstract model. Therefore, we performed the program slicing manually, relying on the Eclipse IDE for reverse-engineering and dependency analysis of the subsystems. Given the recurrent invalid state transitions of entities within DIRAC, we considered the possible race conditions caused by multiple agents updating the 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). This determines our slicing criteria. In the following, we use the SMS as a case study for describing the established modeling guidelines. They can be directly applied to other DIRAC subsystems.

3.2.1. Control Abstractions

As already explained, all agents repeat the same logic in every subsequent iteration: first they read some entries of interest from the service database, then they process the cached data, and finally they may write back or update entries, based on decisions from the processing step. They can be naturally modeled as *recursive processes*. Each process represents a distinct agent, specifying its expected behavior when interacting with the service database. 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 illustrating the basic idea are kept. The first highlighted statement is the selection of all "New" CacheReplicas entries. What follows is retrieval of their metadata from the LFC, 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.

Listing 1: *RequestPreparationAgent.py* code excerpt

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

# Update the states of the files in the database
if terminalReplicaIDs:
  gLogger.info('%s replicas are terminally failed.' % len( terminalReplicaIDs ) )
  res = self.stagerClient.
  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.stagerClient.
  updateReplicaInformation( replicaMetadata )
return S_OK()
```

The logging statements, although critical for operational matters, will not affect the entries' states, and can be translated to *silent steps* in mCRL2. Furthermore, 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 entity contains more information besides the status of a file. Every database entry has a unique identifier, descriptive data such as the storage where it resides, its full path, checksum, timestamps, etc., as can be seen from the table description in Fig. 7. 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 in terms of changes to the *Status* field, we can collapse most of this descriptive data, and represent this entity as a *user-defined sort* (type) in mCRL2:

$$\begin{aligned} sort\ CacheReplicas = \textbf{struct}\ & Start\ |New\ | \\ & Waiting\ |StageSubmitted\ | \\ & Staged\ |Failed\ |Deleted\ ; \end{aligned}$$

This defines an enumerated data type with all possible states. The Tasks entity is modeled in the same manner. Lists of these sorts can be easily modeled in mCRL2 as $List(Tasks)$ and $List(CacheReplicas)$. To define the many-to-many relationship between Tasks and CacheReplicas, we join these data elements in a tuple:

$$sort\ Tuple = \textbf{struct}\ p(t : Nat, r : Nat, link : Bool)$$

The first two elements are the list positions of the Tasks and CacheReplicas entries, while the last one indicates whether a relation between them exists at a given moment of the system execution.

In reality agents operate on lists of IDs corresponding to the database entries, so functions for transforming items of type $List(CacheReplicas)$ and $List(Tasks)$ to a list of identifiers (i.e., positions) $List(Nat)$ are necessary. One such *functional data transformation* used in the model is the $t2id : List(Tasks) \times Tasks \mapsto List(Nat)$ function, which, given an existing list of Tasks and a specific Tasks value, returns the list positions matching the value. For example:

$$t2id([Staged, New, Staged, Failed], Staged) \rightarrow [0, 2]$$

Another example is the $id2cr : List(Nat) \times List(CacheReplicas) \times CacheReplicas \mapsto List(CacheReplicas)$ function, which can be used to update certain CacheReplicas list entries with a new value, in the following way:

$$id2cr([0, 1], [Waiting, Staged, New], Failed) \rightarrow$$
$$[Failed, Failed, New]$$

These data transformations provide a natural way of modeling the actual database operations.

The mCRL2 language does not support global variables (or a similar construct). Therefore, the shared database is modeled as a wrapper process that keeps the list of data entries as a parameter in its local memory. This recursive process continuously listens and responds to requests from other processes (agents), as illustrated below.

$proc\ CacheReplicaMem(d{:}List(CacheReplicas))=$
$\sum_{t:CacheReplicas} RPAgent\_selectCacheReplicas\_(cr2id(d,t),t).\ CacheReplicaMem(d)+$
$\sum_{l:List(Nat),t:CacheReplicas} RPAgent\_prepareNewReplicas\_(l,t).\ CacheReplicaMem(id2cr(l,d,t))+$
$\ldots +$
$\sum_{l:List(Nat),t:CacheReplicas} RFAgent\_removeReplicas\_(l,t).\ CacheReplicaMem(id2cr(l,d,t))+$
$CacheReplicaMem(d);$

The information exchange is arranged via actions of the respective processes that are enforced to communicate. The summation operator allows the process to accept any value of the $CacheReplicas$ and $List(Nat)$ sort passed by agents. To ensure that such synchronous communication between the memory process and the agents is possible, the mCRL2 *allow* (∇) and *comm* (Γ) constructs are used:

$\nabla_{\{RPAgent\_selectCacheReplicas, RPAgent\_prepareNewReplicas,\ \ldots, RFAgent\_removeReplicas\}}$
$\Gamma_{\{\_RPAgent\_selectCacheReplicas|RPAgent\_selectCacheReplicas\_ \rightarrow RPAgent\_selectCacheReplicas,\ \_RPAgent\_prepareNewReplicas|RPAgen}$
$\rightarrow RPAgent\_prepareNewReplicas, \ldots,\ \_RFAgent\_removeReplicas|RFAgent\_removeReplicas\_ \rightarrow RFAgent\_removeReplicas\}$

To complete the picture, we give an outline of the model for the *RequestPreparationAgent* process:

$RPAgent= \sum_{cc:List(Nat)} \_RPAgent\_selectCacheReplicas(cc,New).$

```
mysql> describe CacheReplicas;
+--------------+--------------+------+-----+---------+----------------+
| Field        | Type         | Null | Key | Default | Extra          |
+--------------+--------------+------+-----+---------+----------------+
| ReplicaID    | int(11)      | NO   | PRI | NULL    | auto_increment |
| Type         | varchar(32)  | NO   |     | NULL    |                |
| Status       | varchar(32)  | YES  |     | New     |                |
| SE           | varchar(32)  | NO   | PRI | NULL    |                |
| LFN          | varchar(255) | NO   | PRI | NULL    |                |
| PFN          | varchar(255) | YES  |     | NULL    |                |
| Size         | bigint(60)   | YES  |     | 0       |                |
| FileChecksum | varchar(255) | NO   |     | NULL    |                |
| GUID         | varchar(255) | NO   |     | NULL    |                |
| SubmitTime   | datetime     | NO   |     | NULL    |                |
| LastUpdate   | datetime     | YES  |     | NULL    |                |
| Reason       | varchar(255) | YES  |     | NULL    |                |
| Links        | int(11)      | YES  |     | 0       |                |
+--------------+--------------+------+-----+---------+----------------+
```

Figure 7: CacheReplicas table description

$((cc{\neq}[]){\rightarrow}\ (\_RPAgent\_prepareNewReplicas(cc,Failed).$
$\sum_{tt:List(Nat)} \_RPAgent\_selectTaskReplicas(tt,cc).$
$((tt{\neq}[]){\rightarrow}\_RPAgent\_prepareNewReplicasT(tt,tFailed) \diamond \tau) +$
$\_RPAgent\_prepareNewReplicas(cc,Waiting)) \diamond \tau)$
$.RPAgent;$

As can be seen, the translated process is a sequence of different actions for communicating with the CacheReplicas memory process. These include obtaining and modifying the data that this memory process maintains. Finally, the model is put together as a parallel composition of all communicating processes, in the *init* part of the specification.

$init\ RPAgent\ ||\ SRAgent\ ||\ ...\ ||\ CacheReplicaMem([]);$

Although the WMS model is larger, it is constructed using the same approach. The complete models are available at [33].

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 model is written, the state space can be 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.

System	States	mCRL2 LoC	Python LoC	Generation time
SMS	18,417	432	2,560	<10 sec.
WMS	160,148,696	663	15,042	∼12 hr.

Table 1: mCRL2 model statistics

4.1. Simulation and debugging

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. Modeling errors are not uncommon, especially in the first(draft) versions of the model, and XSim provides the necessary feedback, acting as a debugger. 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 (Fig. 8a), where a job had transited between two different terminating states, "Failed" and "Done". Replaying the behavior with XSim revealed the exact same trace in the WMS model (Fig. 8b). This was a result of several (difficult to reproduce in reality) 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

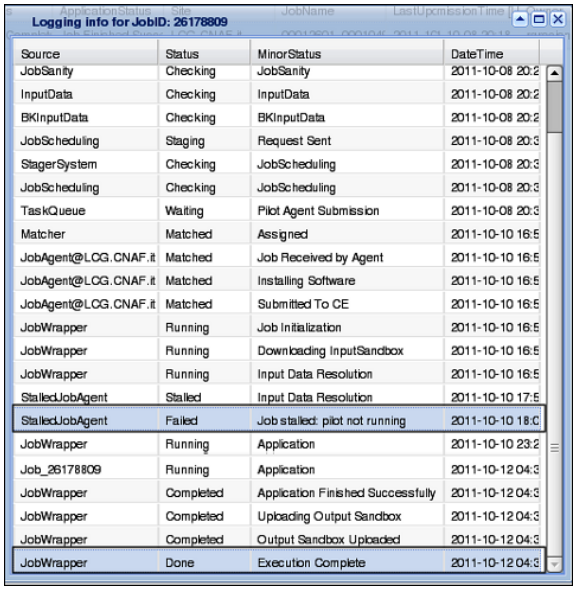
ultimately as non-responsive ("Failed"), since its child process was busy with data access attempt for a long time. However, once data access succeeded, the *JobWrapper* finally started executing the actual payload and reported a "Running" status. The *JobWrapper* assumes that nothing has happened to 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.

4.2. Visualization

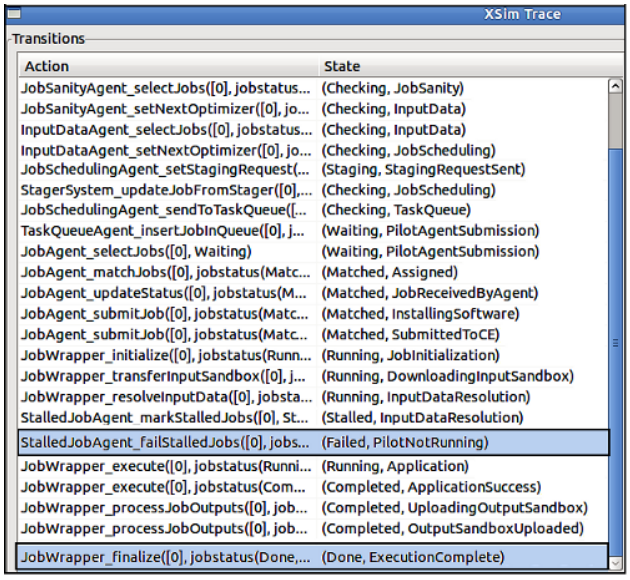
Reasonably small LTSs can be easily visualized with the interactive GUI tools, as the tools employ smart clustering techniques to reduce the complexity of the image. For instance, the SMS model, with around 18000 states, was visualized with DiaGraphica. Fig. 9 shows a projection (clustering) of the state space on the CacheReplicas memory process, which resulted in only 6 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 fact, in the modeling phase we discovered the first 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.

The ltsview tool can be used for a more sophisticated way of visualizing model state spaces in 3D. It also employs clustering of states based on structural properties, but serves to further explore symmetry in the behavior of the system and discover unexpected visual anomalies. The SMS state space is shown in Fig. 10. The behavior starts at the top of the cluster and proceeds downwards. Branching points indicate alternatives in the behavior of the system. It is possible to mark and color different regions based on values of the state vectors, as well as transitions based on their labels. The particular figure shows a suspicious transition between "Failed" and "Staged" tasks presented by distinct coloring of the regions. One can



(a) Logging info of a DIRAC job



(b) XSim simulator trace

Figure 8: Invalid job state transitions

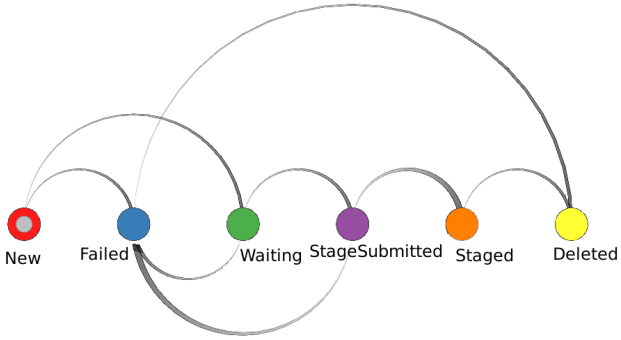


Figure 9: CacheReplicaMem process visualization with DiaGraphica

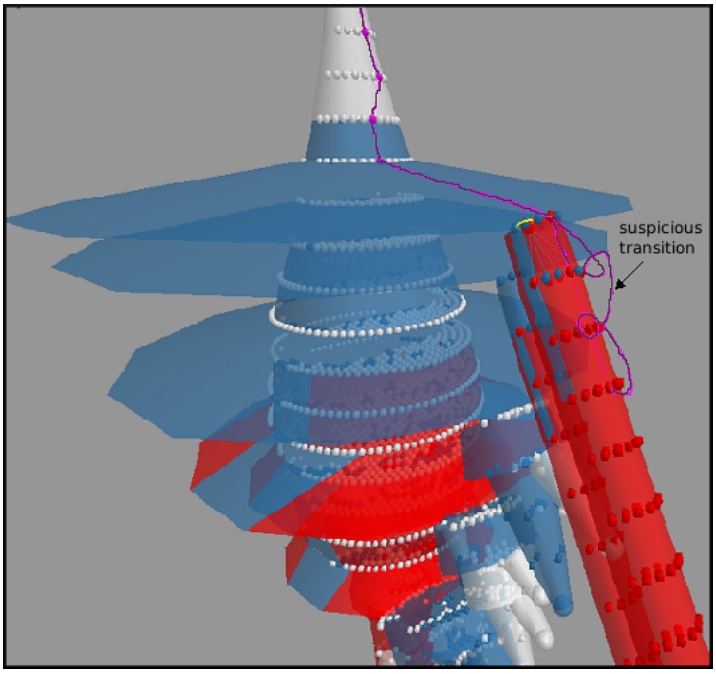


Figure 10: State-space visualisation of the SMS with ltsview

also interactively simulate the behavior while keeping track of the previous, current state, and the possible transitions.

4.3. Model checking

The default requirement specification language in the mCRL2 toolset is the modal μ-calculus [5], extended with regular expressions and data. For our purposes, the regular expressions suffice. These typically allow for specifying that certain behaviors must be absent or present, and *if-then* scenarios. Regular expressions are constructed using constants *true* and $false$ and the modalities "[]" and "<>". The constants *true* and $false$ have their usual meaning: *true* holds in every state, while $false$ does not hold in any state of the model. The modalities are used for expressing *necessity* ("[]") and *possibly* ("<>") statements in a formula. *Necessity* means that the formula should hold for every behavior within the brackets, while *possibly* means that there exists a behavior where the formula is satisfied. The behaviors inside the modalities can be specified concisely using *action formulae*. These can be constructed using actions and the operations "||" for union and "!" for the set complement; the action formula *true* represents the set of *all* actions. Any action or action formula can be turned into a set of behaviors using the "$*$" operator, which is used for expressing cardinality, meaning that actions can occur any number of times; the "." operator allows for concatenating behaviors. In addition to these constructs, one can use quantification over data variables, Boolean expressions, etc. In mCRL2, the model checking problem is often converted to an equation system solving problem, see [5]; such equation systems can be manipulated and solved, thereby answering the encoded model checking problem.

Using the gained understanding of the system behavior, as well as the reported and anticipated problems, we formulated several requirements for the model checker to systematically probe, see below. Typical expressions are of the form "$[A]false$", which holds when a system has no behaviors matching A, and "$[A]<B>true$", which holds if a system has a behavior matching B whenever a behavior matching A has occurred.

1. Each task in a terminating state ("Failed" or "Staged") is eventually removed from the system. (*progress*)

```
[true* .(state([tStaged]) || state([tFailed])).(!(state([tDeleted])))*]
<(!(state([tDeleted]))* .(state([tDeleted]))> true
```

2. A deleted task will never be referenced for transition to any other state. (*safety*)

```
[true*.state([tDeleted]). true*.
(state([tNew]) || state([tStageSubmitted]) || state([tStaged]) || state([tFailed]))]
    false
```

3. Once a job has been killed, it cannot ressurect and start running. (*safety*)

```
[true*.state([jobstatus(Killed,MarkedForTermination)]).true*.state([jobstatus(Running,
    JobInitialization)])]false
```

Model checking was carried out on a 64bit Intel Core™2 Duo (1.6GHz) machine with 2GB RAM. Both requirements 1 and 2 were found to be violated, as can be seen from the traces (Fig. 11). 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

#	Action	State
1	RPAgent_selectCacheReplicas([0, 1], New)	[tNew], [New, New]
2	RPAgent_prepareNewReplicas([0, 1], Waiting)	[tNew], [Waiting, Waiting]
3	SRAgent_selectCacheReplicas([0, 1], Waiting)	[tNew], [Waiting, Waiting]
4	SRAgent_issuePrestageRequests([0, 1], StageSubmitted)	[tNew], [StageSubmitted
5	SMAgent_selectCacheReplicas([0, 1], StageSubmitted)	[tNew], [StageSubmitted, StageSubmit
6	SRAgent_selectTaskReplicas([0, 0], [0, 1])	[tNew], [StageSubmitted, StageSubmit
7	SMAgent_monitorStageRequests([0, 1], Failed)	[tNew], [Failed, Failed]
8	SMAgent_selectTaskReplicas([0, 0], [0, 1])	[tNew], [Failed, Failed]
9	SMAgent_monitorStageRequestsT([0, 0], tFailed)	[tFailed], [Failed, Failed]
10	SRAgent_issuePrestageRequestsT([0, 0], tStageSubmitted)	[tStageSubmitted], [Failed, Failed]

#	Action	State
1	RPAgent_selectCacheReplicas([0, 1], New)	[tNew], [New, New]
2	RPAgent_prepareNewReplicas([0, 1], Waiting)	[tNew], [Waiting, Waiting]
3	SRAgent_selectCacheReplicas([0, 1], Waiting)	[tNew], [Waiting, Waiting]
4	SRAgent_issuePrestageRequests([0, 1], StageSubmitted)	[tNew], [StageSubmitted, StageSubmitted]
5	SMAgent_selectCacheReplicas([0, 1], StageSubmitted)	[tNew], [StageSubmitted, StageSubmitted]
6	SMAgent_monitorStageRequests([0, 1], Failed)	[tNew], [Failed, Failed]
7	SRAgent_selectTaskReplicas([0, 0], [0, 1])	[tNew], [Failed, Failed]
8	SMAgent_selectTaskReplicas([0, 0], [0, 1])	[tNew], [Failed, Failed]
9	SMAgent_monitorStageRequestsT([0, 0], tFailed)	[tFailed], [Failed, Failed]
10	RFAgent_selectTasks([0], tFailed)	[tFailed], [Failed, Failed]
11	RFAgent_clearFailedTasksT([0], tDeleted)	[tDeleted], [Failed, Failed]
12	SRAgent_issuePrestageRequestsT([0, 0], tStageSubmitted)	[tStageSubmitted], [Failed, Failed]

Figure 11: Violation of requirements 1 (top) and 2 (bottom)

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.

Model checking and debugging with explicit state space generation was not a viable option for the WMS, due to its size (13 concurrent processes giving rise to a state space of over 150 million states). We therefore resorted to using the LTSmin symbolic reachability tool [34] and the symbolic equation system solver built on top of LTSmin's equation system explorer [35]. These tools are almost instantaneous in traversing the state space and equation systems, respectively, taking less than 20 seconds. They rely on the use of clever data structures such as BDDs [36] for concisely representing the underlying LTS or equation system that is being explored. Even though all information for computing the explicit artifacts is present in such data structures, obtaining these from such data structures can again be costly, computationally.

The symbolic equation system explorer and solver effectively solves the model checking problems for our requirements. However, in case of violations of the requirements, the symbolic verification tools currently do not give counterexamples. In order to understand the violations, we employ a different trick. We use the symbolic state space exploration tool's option for tracking the occurrence of a certain (specified) action and reporting a trace if such an action is encountered during exploration. For this purpose, we extended the specification with a monitoring process which fires an *error* action if requirement 3 is violated. This monitoring process is set to run in parallel with the system, and observes the relevant actions and state changes that the system itself takes.

A generic monitoring process for checking temporal properties of the form:

```
[true*. state(l1). true*. state(l2)]false
```

is shown below.

$$
\begin{aligned}
&proc\ Monitor(l1,l2{:}List(Job),b{:}Bool)=\\
&\tau.Monitor(l1,l2,b)\\
&+\ (!b \rightarrow (\textstyle\sum_{l:List(Job)}.(l{==}l1) \rightarrow (monitor(l).Monitor(l1,l2,true))\\
&\qquad\qquad \diamond\ (monitor(l).Monitor(l1,l2,b)))\\
&\qquad \diamond\ (\textstyle\sum_{l:List(Job)}.(l{==}l2) \rightarrow (monitor(l).error(l1,l2))\\
&\qquad\qquad \diamond\ (monitor(l).Monitor(l1,l2,b))
\end{aligned}
$$

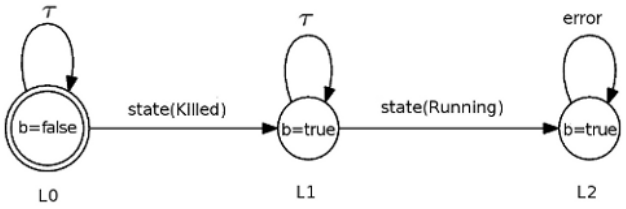


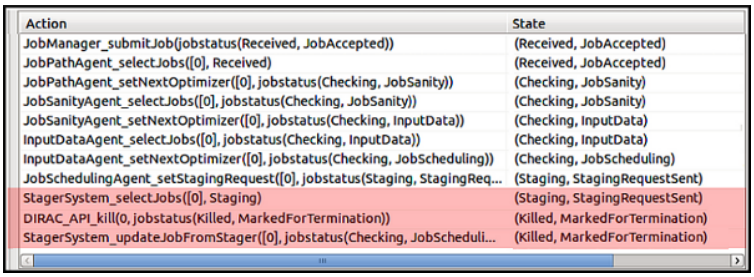
Figure 12: Monitor automaton for requirement 3

);

The automaton of the concrete monitor instance which accepts requirement 3 trace is shown in Fig. 12. Whenever the observed system changes the state of a job to "Killed", the automaton also changes its state to L1 by enforcing synchronous communication between the respective actions. The silent step (τ) can be executed as a result of any non-relevant system state change from the perspective of the automaton. As soon as the system changes the state of the job further to "Running", the automaton makes a move in a similar fashion, after which point the action *error* is executed, and can be detected by the LTSmin tool. If such a full cycle is accepted by the automaton, the temporal property is indeed violated. Using this technique, a counterexample (Fig. 13) 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 "Checking" (and eventually "Running") state even if it had been manually "Killed" (using DIRAC's command line API) 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 this particular case. The changes were also reflected back to the mCRL2 model [33]. Fig. 14 shows that the stager callback no longer affects the job state if it has been changed from its original "Staging" one in the meantime. The state space of the fixed model amounts to 157048664 states, the reduction being a result of the blocking of the unwanted behavior. However, we must note that model checking still discovered traces where the property is violated, even without the stager being involved. The current architecture does not permit an easy workaround for this problem. Protecting shared memory access using traditional locking techniques does not scale well. There is an ongoing effort in redesigning DIRAC's architecture, to accomodate a task dispatcher that takes care of persisting the state of the tasks to the storage backend and distributing them among so called "executors".This should replace the old agent components' polling architecture, and reduce the load they place on the storage, in addition

Action	State
JobManager_submitJob(jobstatus(Received, JobAccepted))	(Received, JobAccepted)
JobPathAgent_selectJobs([0], Received)	(Received, JobAccepted)
JobPathAgent_setNextOptimizer([0], jobstatus(Checking, JobSanity))	(Checking, JobSanity)
JobSanityAgent_selectJobs([0], jobstatus(Checking, JobSanity))	(Checking, JobSanity)
JobSanityAgent_setNextOptimizer([0], jobstatus(Checking, InputData))	(Checking, InputData)
InputDataAgent_selectJobs([0], jobstatus(Checking, InputData))	(Checking, InputData)
InputDataAgent_setNextOptimizer([0], jobstatus(Checking, JobScheduling))	(Checking, JobScheduling)
JobSchedulingAgent_setStagingRequest([0], jobstatus(Staging, StagingReq...	(Staging, StagingRequestSent)
StagerSystem_selectJobs([0], Staging)	(Staging, StagingRequestSent)
DIRAC_API_kill(0, jobstatus(Killed, MarkedForTermination))	(Killed, MarkedForTermination)
StagerSystem_updateJobFromStager([0], jobstatus(Checking, JobScheduli...	(Checking, JobScheduling)
JobSchedulingAgent_sendToTaskQueue([0], jobstatus(Checking, TaskQueue))	(Checking, TaskQueue)
TaskQueueAgent_insertJobInQueue([0], jobstatus(Waiting, PilotAgentSubm...	(Waiting, PilotAgentSubmission)
JobAgent_selectJobs([0], Waiting)	(Waiting, PilotAgentSubmission)
JobAgent_matchJobs([0], jobstatus(Matched, Assigned))	(Matched, Assigned)
JobAgent_updateStatus([0], jobstatus(Matched, JobReceivedByAgent))	(Matched, JobReceivedByAgent)
JobAgent_checkInstallSoftware([0], jobstatus(Matched, InstallingSoftware))	(Matched, InstallingSoftware)
JobAgent_submitJob([0], jobstatus(Matched, SubmittedToCE))	(Matched, SubmittedToCE)
JobWrapper_initialize([0], jobstatus(Running, JobInitialization))	(Running, JobInitialization)

Figure 13: "Zombie" job starts running after being killed



Action	State
JobManager_submitJob(jobstatus(Received, JobAccepted))	(Received, JobAccepted)
JobPathAgent_selectJobs([0], Received)	(Received, JobAccepted)
JobPathAgent_setNextOptimizer([0], jobstatus(Checking, JobSanity))	(Checking, JobSanity)
JobSanityAgent_selectJobs([0], jobstatus(Checking, JobSanity))	(Checking, JobSanity)
JobSanityAgent_setNextOptimizer([0], jobstatus(Checking, InputData))	(Checking, InputData)
InputDataAgent_selectJobs([0], jobstatus(Checking, InputData))	(Checking, InputData)
InputDataAgent_setNextOptimizer([0], jobstatus(Checking, JobScheduling))	(Checking, JobScheduling)
JobSchedulingAgent_setStagingRequest([0], jobstatus(Staging, StagingReq...	(Staging, StagingRequestSent)
StagerSystem_selectJobs([0], Staging)	(Staging, StagingRequestSent)
DIRAC_API_kill(0, jobstatus(Killed, MarkedForTermination))	(Killed, MarkedForTermination)
StagerSystem_updateJobFromStager([0], jobstatus(Checking, JobScheduli...	(Killed, MarkedForTermination)

Figure 14: Race condition involving the stager callback no longer occurs

to eliminating race conditions.

5. Related Work

Model checking belongs to a class of mathematically-based techniques for specification, analysis, and verification of software and hardware systems. Theorem proving, static checking, run-time verification, and model-based testing also belong to this category of formal methods. The main premise behind their use is the expectation that the initial investment in becoming proficient and developing models pays off in the long run, in terms of improved reliability and safety of the system. While traditional testing is almost exclusively used as a verification methodology in many practical industrial cases, formal methods have started receiving more attention in the last couple of decades. Testing is often based on the intuition and experience of the tester, and, as such, does not always reveal subtle system errors. Hence, particular efforts are directed at the practical use of formal methods in the area of critical, concurrent, and distributed systems. The application of such methods in communication protocols [9, 10, 14], safety-critical software [16, 20, 37], driver design [17, 38], among others, has been a success. Case studies in the domain of high-performance computing, and in particular concurrent/MPI-based program design, have surfaced as well [39–41], mostly relying on the the MPI extension of the popular model checker SPIN [42]. Ironically enough, the model checking tools themselves do not take advantage of distributed infrastructures, and as such do not scale well.

To overcome the problem of state space explosion, several variants to explicit model checking have gained popularity. Runtime verification [43, 44] aims to increase confidence in the correctness, but does not make claims for absence of defects in a system. The target program is instrumented manually, or automatically by scanning function calls and variable assignments, in order to capture and emit relevant events which are then monitored against a special component during execution. This component resembles our monitoring process automaton used to cope with state space explosion during model checking of DIRAC's WMS system. While runtime verification is useful for partially-capturing incorrect behavior of the system implementation, code instrumentation is not always a feasible option, considering the increase of memory consumption.

Similar to this lightweight form of verification is static program analysis [45]. Rather than code instrumentation and execution, the objective of static program analysis is to determine runtime properties of systems at compilation time. A special analyzer targeted for the implementation language maintains information on the control flow and function call graphs of the software. Approximation is used by such tools in practice, in order to improve scalability. The obvious advantage of this approach is that it works directly on the implementation. However, most static analysis tools aim at discovering more general and low-level implementation bugs, such as uninitialized variables, null pointer references, zero division, or out of bound array indices. It is almost impossible for a general static analysis tool to figure out the exact high-level intent of the implementation code taken as input. Because of the heavy approximation they use, static analysis tools are quite fast and applicable to large software projects, but result in many false positives reported. An additional drawback is their practical availability for languages other than C and Java. To the best of our knowledge, no such tool exists for Python. Both runtime verification and static analysis can be seen as additions to traditional model checking, rather than alternatives. Some progress has been made in the direction of combining the approaches [17, 20, 46]. For instance, Java PathFinder [20] relies on static

analysis to reduce the state space before systematically exploring it by executing the abstracted program. In [46] static analysis and model checking also operate in a feedback loop.

Static analysis is, in a sense, an extreme form of automated deductive verification. In its original form, deductive verification technology allowed for the use of logics such as Floyd-Hoare logic [47] for showing the correctness of a software or hardware system by constructing proofs thereof, based on mathematical descriptions of the behavior of the system. These days, the proofs involved are often discharged using a mechanical or an automated theorem prover. Moreover, more advanced logics such as Separation Logic [48] are used. Deductive verification is applied quite successfully on software code [49, 50] and hardware [51], but the effort required for conducting the verification can be prohibiting, and can compare poorly to model checking.

Instead of constructing a model of the system from scratch, when the only source of documentation is the code itself, a technique called process (or workflow) mining [52] can be used. It is a method of automatically distilling a process or software specification from a set of structured traces collected from real system executions. The mined specifications are typically expressed in a form of Message Sequence Graphs or Petri-Nets, graphic formalism variants of the classical finite-state machines. Given that the analysis of program execution traces is still the most prevalent debugging technique for distributed systems, and DIRAC's components (as well as the middleware) produce immense amounts of execution logs anyhow, it seems like an attractive approach to constructing models. There is a vast amount of work in this area [53–57]. A common assumption in process mining is that the events in each trace are totally ordered, which for concurrent systems does not always hold. To cope with this, most approaches require that the thread IDs or process IDs are recorded in the traces as well. Observing this as a realistic obstacle for obtaining an accurate model, Lou et al. in [54] propose an algorithm for constructing models from traces of concurrent systems by expressing events as partially ordered. Their algorithm identifies dependency relationships between pairs of events in interleaved traces. Another common assumption is a direct result of the underlying techniques used for constructing the models, namely data mining and machine learning. These techniques consider "erroneous activities" or activities that were not logged in the appropriate order for some reason, as noise, and disregard trace sequences with no statistical significance. As a result, a design flaw can be mistaken for a noise, or not captured at all if a particular execution path has a low probability of occurrence. A notable exception in this direction is [58], where the emphasis is not only on process discovery, but also on verification: desired or undesired properties can be expressed with Linear Temporal Logic, and the checker verifies whether the observed behavior in the traces matches the property.

While relying solely on process mining as a technique for extracting a formal model may not be the way to go, inferring a "fuzzy" model from the frequent interaction traces, rather than manual code analysis could be a helpful initial step. Nevertheless, the graphical models obtained must be translated into a process algebra formalism accepted by an actual toolset, if system correctness is to be established.

6. Conclusions

In this paper we have applied a formal methods approach for analyzing CERN's DIRAC grid system. 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 critical race conditions and livelocks, which were confirmed to occur in the real system. Despite the continuous DIRAC development for 10 years, these problems were notoriously difficult to discover and trace, considering the number of concurrent components involved. Testing of concurrent systems and reproducing error traces is often challenging because of the lack of control that the tester has over the execution of the concurrent processes. With the model at hand, replaying the traces and localizing the problems became 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. It is observed

that many large concurrent/distributed systems are essentially parametrized, meaning that they contain components or process descriptions which are behaviorally similar. Once becoming proficient with the formal language notation, modeling such components is reduced to reapplying similar abstraction rules. The methodology we have outlined is particularly suitable for such systems. In addition, the guidelines proposed can be applied when dealing with data-centered concurrent systems, where the only documentation is the implementation itself, and most of the problems arise from sharing data among processes. The biggest challenge for such systems is finding a reasonable abstraction level for the formal model, and we believe that the combination of code slicing and abstraction rules of thumb presented in Section 3 can be helpful in this direction.

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 descriptive guidelines we have proposed for obtaining an abstract model can be easily applied to other large-scale industrial systems of a similar nature. It goes without saying that the analyst must be careful to faithfully describe the system behavior in the abstract model. With this critical step in mind, formal methods are an important addition to the assembly of software analysis and verification techniques for concurrent distributed systems.

7. Future work

Besides discovering behavioral problems, the model gives opportunities to analyze the system performance in the future. Performance analysis aims at predicting the behavior of a system with respect to dynamic properties, such as the expected throughput, response time, resource usage, number of requests per unit time, etc. For this, the formal model must be enriched with timing and probability quantities attached to certain activities of interest. Adding this kind of information to traditional process algebra models means getting in the domain of stochastic process algebra. Random variables with exponential distribution are typically associated with the designated actions, to express system transitions with duration, which could be estimates of processing time or communication overhead. Tools based on stochastic process algebra formalisms can then generate the underlying Markov Chain model representation and solve it to obtain steady-state probability distributions over the state space. These measures can provide a good base for assessing the system's efficiency and reliability. Furthermore, by varying the quantitative inputs, alternative designs can be evaluated before actual implementation takes place. One such stochastic process algebra is PEPA (Performance Evaluation Process Algebra [59]), supported by an Eclipse Plug-in for modeling and analysis of systems. PEPA models are annotated with information about the duration of activities and their relative probabilities of occurence.

In the context of DIRAC, our future work will be focused on extending the mCRL2 models with information about polling intervals of the agents, the rate at which database queries can be executed, and the overhead due to communication with external components (such as the storage middleware and the LFC), in order to obtain estimates of the agents utilization, mean processing time before a file replica is staged, and the average cache occupancy. Furthermore, we want to assess an alternative DIRAC WMS architecture which is claimed to reduce the reaction time of the system with respect to the optimizers-chain part of the job workflow. We want to have an indication of the possible system bottlenecks under different load (job accumulation), by varying the number of instances of certain concurrent components. This will give us valuable information on how to proceed with deployment in production environments.

Fortunately, we do not have to use another formalism for enhancing the models with stochastic information. The CADP toolset [60] for analysis of stochastic models is well integrated with mCRL2 for this purpose. CADP can explore and manipulate mCRL2 specifications via an interface for a common LTS format representation [61] which both tools understand.

Acknowledgements

The authors would like to thank Wan Fokkink from VU University Amsterdam, Dragan Bosnacki from TU Eindhoven, as well as Ricardo Graciani, Philippe Charpentier and the rest of the LHCb collaboration

for their valuable contribution to this work.

References

[1] Large Hadron Collider beauty experiment.
URL http://lhcb-public.web.cern.ch/lhcb-public

[2] A. Tsaregorodtsev, et al., DIRAC: A Community Grid Solution, in: Proc. CHEP, IOP Publishing, 2007.

[3] A. Smith, A. Tsaregorodtsev, DIRAC: Reliable Data Management for LHCb, in: Proc. CHEP, IOP Publishing, 2007.

[4] J. McManus, W. Bynum, Design and analysis techniques for concurrent blackboard systems, SMC 26 (1996) 669–680.

[5] J. Groote, T. Willemse, Model-checking processes with data, SCP 56 (2005) 251–273.

[6] J. Groote, A. Mathijssen, M. Reniers, Y. Usenko, M. van Weerdenburg, The Formal Specification Language mCRL2, in: Proc. MMOSS, 2006.

[7] J. Groote, J. Keiren, F. Stappers, J. Wesselink, T. Willemse, Experiences in developing the mCRL2 toolset, SPE 41 (2011) 143–153.

[8] J. Baeten, T. Basten, M. Reniers, Process Algebra: Equational Theories of Communicating Processes, Cambridge University Press, 2009.

[9] Y.-T. He, R. Janicki, Verifying protocols by model checking: a case study of the wireless application protocol and the model checker SPIN, in: Proc. CASCR, IBM Press, 2004.

[10] K. Palmskog, Verification of the session management protocol, Master's thesis, Royal Institute of Technology (2006).

[11] G. Holzmann, Design and validation of computer protocols, Prentice-Hall Inc., 1990.

[12] W. Fokkink, Modelling Distributed Systems, Springer-Verlag, 2007.

[13] T. Ball, S. Rajamani, The SLAM Toolkit, in: Proc. CAV, Springer-Verlag, 2001.

[14] B. Badban, W. Fokkink, J. Groote, J. Pang, J. van de Pol, Verification of a sliding window protocol in μCRL and PVS, FAC 17 (2005) 342–388.

[15] S. Islam, M. Sqalli, S. Khan, Modeling and Formal Verification of DHCP Using SPIN, IJCA 3 (2006) 145–159.

[16] Y.-L. Hwong, V. Kusters, T. Willemse, Analysing the Control Software of the Compact Muon Solenoid Experiment at the Large Hadron Collider, in: Proc. FSEN, Springer-Verlag, 2011.

[17] D. Bošnački, A. Mathijssen, Y. Usenko, Behavioural Analysis of an I2C Linux Driver, in: Proc. FMICS, Springer-Verlag, 2009.

[18] K. Verstoep, H. Bal, J. Barnat, L. Brim, Efficient Large-Scale Model Checking, in: Proc. IPDPS, IEEE Computer Society, 2009.

[19] H. Hojjat, M. Mousavi, M. Sirjani, Formal Analysis of SystemC Designs in Process Algebra, FI 107 (2011) 19–42.

[20] W. Visser, P. Mehlitz, Model Checking Programs with Java PathFinder, in: Proc. SPIN, Springer, 2005.

[21] B. Ploeger, Analysis of ACS using mCRL2, Cs-report 09-11, Technische Universiteit Eindhoven (2009).

[22] I. Bird, Computing for the Large Hadron Collider, Ann.Rev.Nucl.Part.Sci. 61 (2011) 99–118.

[23] E. Laure, C. Gr, S. Fisher, Frohner, et al., Programming the Grid with gLite, in: Computational Methods in Science and Technology, 2006.

[24] A. Casajus, R. Diaz, S. Paterson, A. Tsaregorodtsev, DIRAC Pilot Framework and the DIRAC Workload Management System, in: Proc. CHEP, IOP Publishing, 2010.

[25] C. Aiftimiei, P. Andreetto, S. Bertocco, et al., Design and implementation of the gLite CREAM job management service, Future Generation Computer Systems 26 (4) (2010) 654–667.

[26] Grid Production Shifter Guide for LHCb.
URL https://twiki.cern.ch/twiki/bin/view/LHCb/ProductionShifterGuide

[27] D. Park, Concurrency and Automata on Infinite Sequences, in: Theoretical Computer Science, Vol. 104 of Lecture Notes in Computer Science, Springer Berlin / Heideberg, 1981, pp. 167–183.

[28] M. Musuvathi, D. Engler, Model Checking Large Network Protocol Implementations, in: Proc. NSDI, USENIX, 2004.

[29] J. Hatcliff, M. Dwyer, H. Zheng, Slicing Software for Model Construction 13 (2000) 315–353.

[30] Frama-C software suite.
URL http://frama-c.com

[31] CodeSurfer.
URL http://www.grammatech.com/products/codesurfer

[32] J. Corbett, M. Dwyer, J.Hatcliff, S. Laubach, C. Pasareanu, H. Zheng, Bandera: Extracting Finite-state Models from Java Source Code, in: Proc. ICSE, ACM Press, 2000.

[33] mCRL2 SVN Repository.
URL https://svn.win.tue.nl/viewvc/MCRL2

[34] S. Blom, J. van de Pol, Symbolic Reachability for Process Algebras with Recursive Data Types, in: Proc. ICTAC, Lecture Notes in Computer Science, Springer, 2008, pp. 81–95.

[35] G. Kant, J. van de Pol, Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games, in: Proc. EPTCS'12.

[36] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang, Symbolic Model Checking: 10^{20} States and Beyond, in: Proc. LICS, IEEE Computer Society, 1990, pp. 428–439.

[37] A. Mathijssen, A. J. Pretorius, Verified design of an automated parking garage, in: Proc. FMICS'06, Springer-Verlag.

[38] T. Ball, B. Cook, V. Levin, S. K. Rajamani, SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft, in: Proc. IFM'04, Springer-Verlag.

[39] S. F. Siegel, G. S. Avrunin, Verification of MPI-Based Software for Scientific Computation, in: Model Checking Software: 11th International SPIN Workshop, Springer-Verlag, 2004.
[40] O. S. Matlin, E. L. Lusk, W. McCune, SPINning Parallel Systems Software, in: Proceedings of the 9th International SPIN Workshop on Model Checking of Software, Springer-Verlag.
[41] S. F. Siegel, Model checking nonblocking MPI programs, in: Proc. VMCAI'07, Springer-Verlag.
[42] S. F. Siegel, Verifying parallel programs with MPI-spin, in: Proc. PVM/MPI'07, Springer-Verlag, Berlin, Heidelberg.
[43] S. Colin, L. Mariani, Run-Time Verification, in: Model-Based Testing of Reactive Systems, 2004.
[44] K. Havelund, Runtime Verification of C Programs, in: Proc. TestCom/FATES, Lecture Notes in Computer Science, Springer, 2008.
[45] D. A. Schmidt, B. Steffen, Program Analysis *as* Model Checking of Abstract Interpretations, in: Proc. SAS'98, Springer.
[46] G. Brat, W. Visser, Combining Static Analysis and Model Checking for Software Analysis, in: Proc. ASE'01, IEEE Computer Society.
[47] C. Hoare, An Axiomatic Basis for Computer Programming, Commun. ACM 12 (10) (1969) 576–580.
[48] J. C. Reynolds, Separation Logic: A Logic for Shared Mutable Data Structures, in: LICS'02, IEEE Computer Society, pp. 55–74.
[49] W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, F. Piessens, Sound Formal Verification of Linux's USB BP Keyboard Driver, in: A. Goodloe, S. Person (Eds.), NASA Formal Methods, Vol. 7226 of Lecture Notes in Computer Science, 2012, pp. 210–215.
[50] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood, seL4: formal verification of an operating-system kernel, Commun. ACM 53 (6) (2010) 107–115.
[51] B. Akbarpour, A. T. Abdel-Hamid, S. Tahar, J. Harrison, Verifying a Synthesized Implementation of IEEE-754 Floating-Point Exponential Function using HOL, Comput. J. 53 (4) (2010) 465–488.
[52] R. Agrawal, D. Gunopulos, F. Leymann, Mining Process Models from Workflow Logs, in: Proc. EDBT '98, Springer-Verlag.
[53] W. M. P. v. d. Aalst, Verification of Workflow Nets, in: Proc. ICATPN'97.
[54] J.-G. Lou, Q. Fu, S. Yang, J. Li, B. Wu, Mining program workflow from interleaved traces, in: Proc. SIGKDD'10, ACM.
[55] N. Walkinshaw, K. Bogdanov, Inferring Finite-State Models with Temporal Constraints, in: Proc. ASE '08, IEEE Computer Society.
[56] W. van der Aalst, A. Weijter, L. Maruster, Workflow Mining: Discovering process models from event logs, IEEE Transactions on Knowledge and Data Engineering 16 (9).
[57] G. Ammons, R. Bodík, J. R. Larus, Mining specifications, in: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '02, ACM.
[58] W. M. P. van der Aalst, H. T. de Beer, B. F. van Dongen, Process mining and verification of properties: an approach based on temporal logic, in: Proc. OTM'05, Springer-Verlag.
[59] G. Clark, S. Gilmore, J. Hillston, Specifying Performance Measures for PEPA, in: Proceedings of the Fifth International AMAST Workshop on Real-Time and Probabilistic Systems, LNCS, 1999, pp. 211–227.
[60] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes, in: Proc. TACAS'11.
[61] H. Garavel, H. Hermanns, On combining functional verification and performance evaluation using CADP, in: Proc. FME'02, Springer.