# The Sun Stone Scheduling Framework

In this section, we present an architecture and design features of a workflow and parallel job grid scheduling simulation framework developed as an extension to Teikoku Grid Scheduling Framework (tGSF) (Grimme et al. 2007). tGSF is a generic standard-based parallel job scheduling framework. It provides mechanisms for parallel job interchange between sites in a computational grid. Site acceptance and distribution of jobs are subject to policies; queuing and scheduling strategies are configurable; provenance information is associated to jobs during their execution life cycle and optionally stored to permanent storage. The job model is based on the Standard Workload Format (SWF) (Chapin et al. 1999). tGSF is a flexible configurable and extensible parallel grid scheduling framework that complies with standards and offers a controllable research environment.

The tGSF has been extended with the capabilities of: (1) scheduling workflows and parallel jobs at the grid layer; (2) query real-time or stale site state information; (3) query job execution time estimates; (4) monitoring/storing performance accounting data; and (5) model data center power consumption. With such a functionality, tGSF users can perform performance evaluation studies that include scheduling of parallel and workflow jobs. As of 2016, we have relabeled our development branch Sun Stone Scheduling Framework (SStone Sch.) The architectural design for supporting the previously mentioned functionality is given in section IV.2.

## IV. 1 The Teikoku Grid Scheduling Framework (tGSF)

The tGSF was developed to provide a simulation framework for grid scheduling. It is a Java based application developed by the Grid Scheduling Architecture Research Group (GSA-RG) of the open Grid Forum and a research group of equal name within the CoreGrid (Priol, 2009). tGSF uses, wherever possible, standards for workload representation, performance metrics, and the scheduling architecture. It was initially designed for parallel job interchange between computational sites. It uses job acceptance, distribution, and location policies to achieve load balancing. tGSF is structured in the following four layers (see Fig. 8):

- **Foundation layer** is an event-driven kernel that manages global time and event dispatching. Timed events are registered and dispatched by the kernel. The run time environment allows carrying out real-time, simulation, or debugging scheduling setups.
- **Commons layer** provides an abstraction for modeling of jobs, metrics, and persistence. A job is an aggregation of a description, life cycle, and provenance information. The description holds static attributes that define job ownership, group membership, and resource requirements. The job-cycle holds information that describes job current and historic states. Provenance stores the job execution path from the job release site to the location of the host that satisfied the resource request. Metrics provide mechanisms for evaluating job performance. Lastly, persistence provides mechanisms to access permanent storage, such as, relational databases and files.
- **Site layer**, resource administration is performed at site layer by means of an abstract scheduler. Strategies are used to advice the scheduler of possible job assignments. The scheduler can evaluate multiple strategies and select the most appropriate one. Parallel job scheduling strategies such as easy backfilling and FCFS are used. This layer also provides data warehousing to facilitate retrieving resource state information, used by strategies to make job allocation decisions.
- **Grid layer**, previous version of tGSF 1.0 did not provide grid scheduling support. The grid layer enables job interchange between sites by means of acceptance, location, and distribution policies. The decision maker may be operated under different settings. In a centralized set-up, its responsibility is to accept jobs and delegate them to the local site scheduler. In a decentralized set-up, job delegation is also performed based on a distribution policy. Centralized grid scheduling support is added in this work.

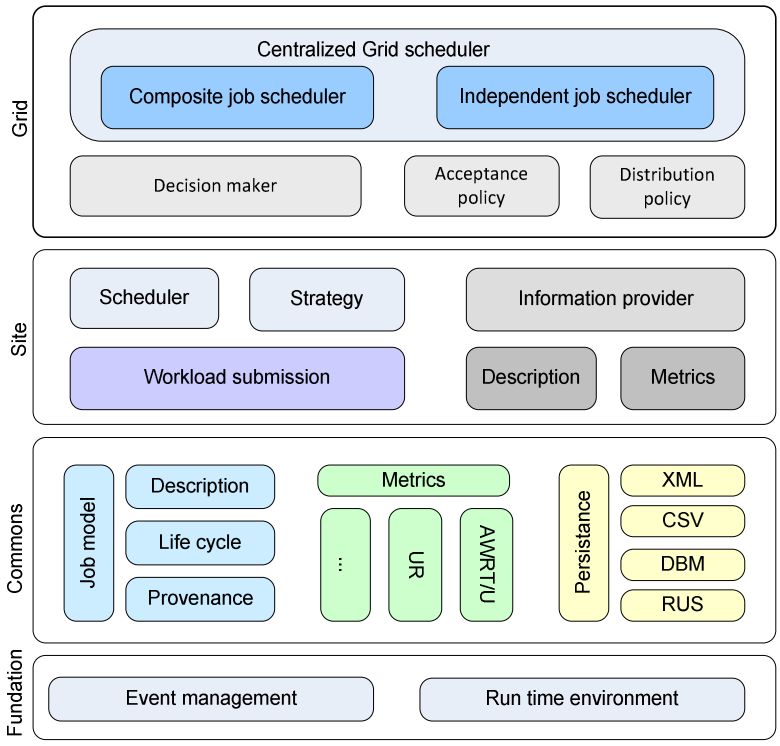



Figure 1. Teikoku grid scheduler layered architecture, workflow and parallel job scheduling support is provided by the composite and independent job schedulers correspondingly

A site may model a computational, data, or memory intensive set of resources. Only computational sites are available in tGSF version 1.0. Computational sites provide processing capabilities to execute parallel jobs. A computational site is an ensemble of components from the commons, site, and grid layer, for instance: an activity broker, parallel scheduler, job submission component, metrics, an information provider, and acceptance and distribution policies.

Parallel jobs are submitted to computational sites via a local submission component or forwarded from an external site. Locally submitted jobs may be processed or delegated to other sites. The delegated job acceptance or denial is determined by local acceptance policies.

We extend tGSF scheduling capabilities by adding workflow and parallel job scheduling support at the grid layer. A centralized grid scheduler is responsible for orchestrating job placement to computational sites in the grid. Two levels of scheduling are distinguished namely, site scheduling and grid scheduling (see Fig. 9). The grid scheduler queues parallel jobs and jobs with broken precedence constraints. It also buffers workflow control and state information used during workflow execution.

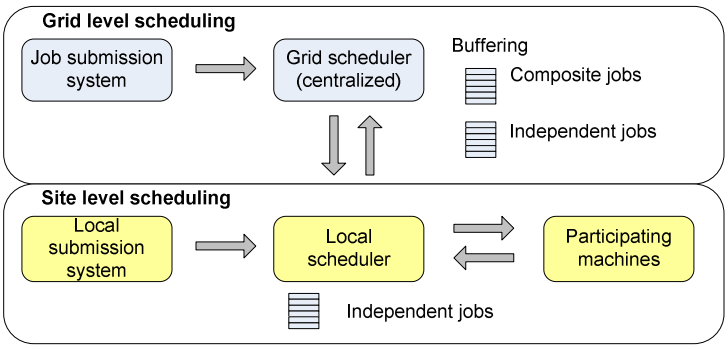



Figure 2. tGSF new two layer scheduling architecture

## IV. 2 Integrating Grid scheduling support

### IV. 2.1 The job model in tGSF

The original version of tGSF models parallel jobs using the Standard Workload Format (SWF) with chain support (Chapin et al. 1999). We extend tGSF abstract job model so that workflows can be modeled and scheduled at the grid layer. We added four new attributes to the existing format, namely: composite job ID, composite job type, vector of successor IDs, and a vector of predecessor IDs. Hereon, we use the term workflow or composite job interchangeably to refer to precedence constraint jobs.

Composite jobs are modeled by Directed Acyclic Graphs (DAG). A DAG is composed by a set of SWFjobs and a set of precedence constrains stored in a hypergraph data type (O'Madadhain, 2010), which models jobs with directed, undirected, cyclic, forest, sparse, or other graph properties.

Parallel jobs are modeled as SWFJob objects, such objects include most attributes found in the SWF file format (see Appendix A.IV), i.e. job id, release time, requested memory, allocated memory, etc. A total, 18 SWF attributes are included in the parallel job object. Not all are useful. Nevertheless, they are kept for compatibility purposes. Since the focus of this section is on workflow scheduling support, we do elaborate design details of the parallel job model. A thorough and extensive API documentation is found in (Grimme, C. 2009).

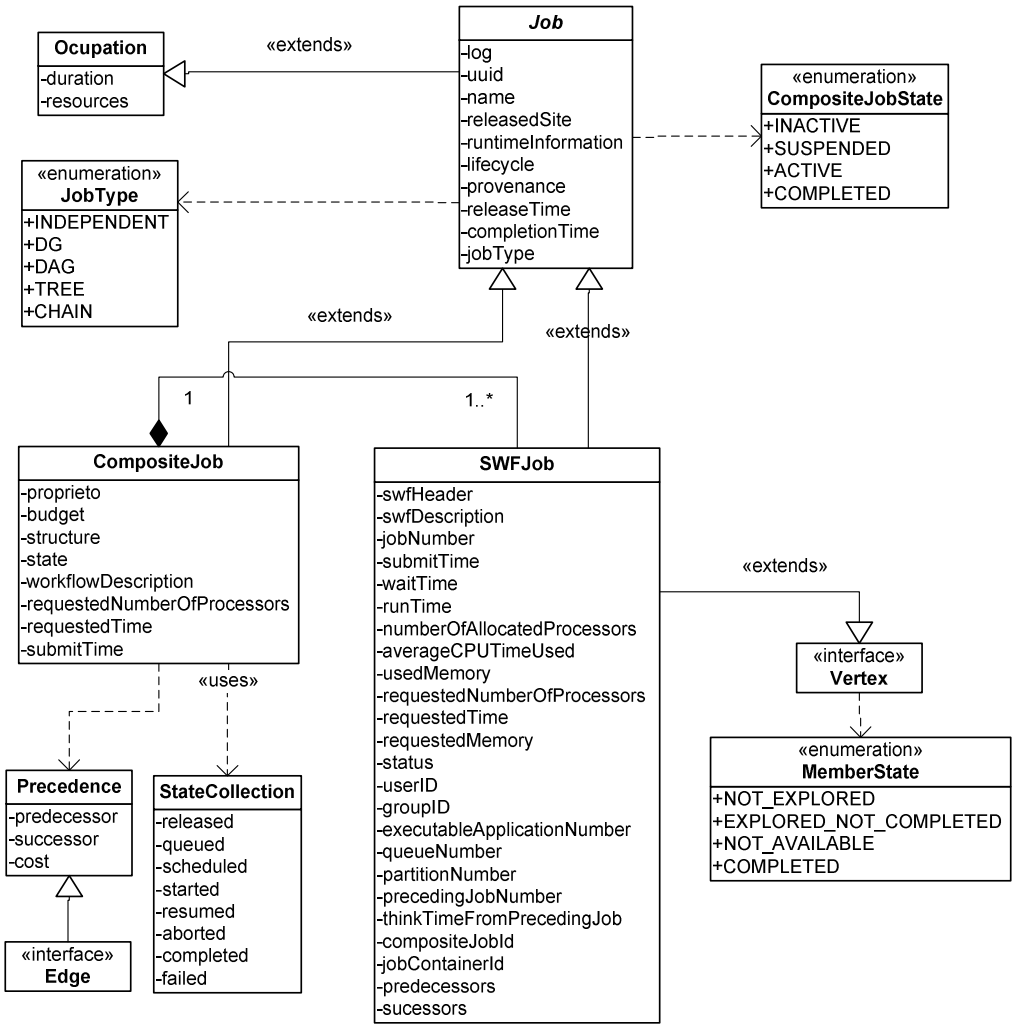



Figure 3. Teikoku static job model

As stated earlier, SWF jobs can only model precedence constraints in form of chains, via the preceding job and think time attributes. Due to the limited capability to model other types of precedence constraints jobs, we have created the CompositeJob object that can be used to model jobs as directed graphs. However, in our implementation, only scheduling of DAGs is supported. Composite job object attributes are the following:

- Proprietor, a string holding the identifiers of the proprietors
- Budget, a monetary amount of resources available to the job
- State $\in \{\text{inactive}, \text{active}, \text{suspended}, \text{completed}\}$
- Requested time, a user runtime estimate for the entire workflow

- Submit time, the release time
- Requested number of processors, the number of requested processors
- Structure, a graph data structure that holds jobs and precedence constraints, modeled as a JUNG DAG (O'Madadhain, 2010)

The static model of the job objects is shown in Fig. 10 (above). The number of attributes and methods in most classes are numerous. We restrain to explain only features necessary to understand relations and interactions among objects.

The parallel and composite job parent class is Job. Job is an abstract class used as an attribute in many interfaces. An instantiated job can be made concrete by first reading its type attribute. Four job types are distinguished: independent, DAG, tree, and chain, defined in the JobType enumeration. An independent job is equivalent to a parallel job. When a job is of type chain, tree, or DAG the successor set, predecessor set, and composite job id can be used to determine precedence constraints.

A correct parallel job execution path initiates in the release state. Once the grid scheduler finds a suitable resource, it allocates the job to it. Upon the job arrival, the target hosts queues the job in a local job queue causing the job state to transition to queued state. The job transitions from queued state to schedule state, when the local schedule de-queues the job and chooses a local resource. As soon the jobs is allocated to the resource it initiates execution, this action causes the job to transition to start state. Lastly, the job transitions to complete state when it finishes execution. Other transitions are possible, as illustrated in the Fig. 11, but correct jobs are only supported in tGSF Ver. 1.0.

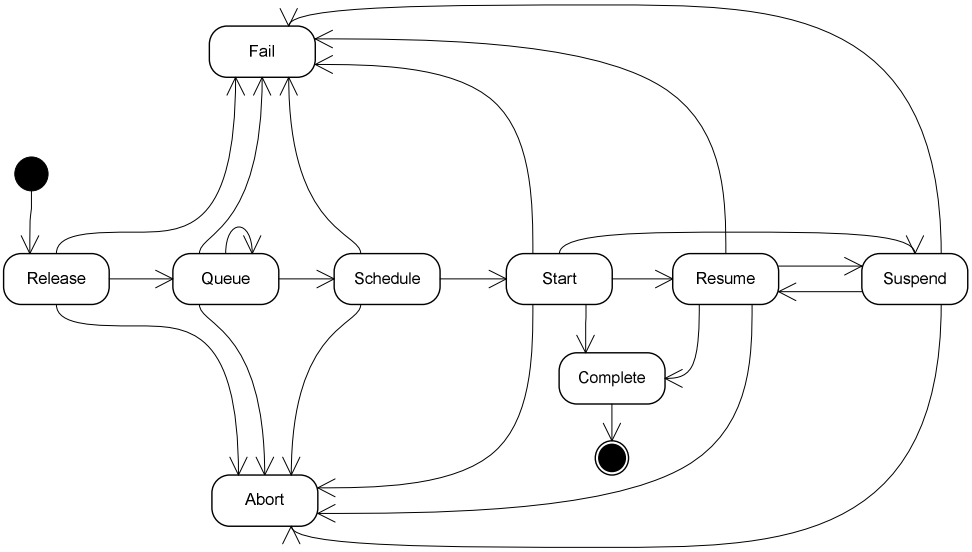



Figure 4. Parallel jobs state transition diagram

Composite job state transitions are more intricate, as they depend on the state of member tasks. Member jobs are parallel jobs subject to precedence constraints, hereon they are referred to as member tasks. Each member task holds its own state and is subject to state transitions as described above. The states of all member tasks are stored in an instantiated StateCollection object and managed according to the rules described in Table I.

TABLE I
Composite job state transitions

| Event | Action | From | To |
|---|---|---|---|
| The composite job is released | Initiate composite job PCB | none | Initiated |
| At least one member task is released | Broker creates a plan and puts it in PCB | initialized | Active |
| One member job completed execution, no other member jobs are executing | Break precedence constraints if any | active | Initiated |
| At least one member job is executing, there are pending suspended jobs | | active | Active |
| At least one member job is suspended, no member job is in execution | | active | Suspended |
| At least one member job is released or reactivated after being suspended | | suspended | Active |
| All member jobs complete<br>At least one member job is terminated | Apply metrics and free PCB | active<br>suspended, initiated, active | Completed<br>Terminated |

A Process Control Block (PCB) is a data structure used to hold workflow runtime information, namely: an allocation set, tasks absolute finishing times, jobs labels (rank), a sorted set of critical path jobs, and the composite job id. The allocation set is used to buffer tasks whose target machine has been determined, buffering task- target machine 2-tuples. Labels and Labeling strategies are described in Section III.2. Critical path tasks are stored in topological order to preserve task precedence constraints.

Composite jobs are marked with a label that describes its state during the workflow execution, as Not Explored (NE), Not Available (NA), Explored but not Completed (EC), and Completed (C).

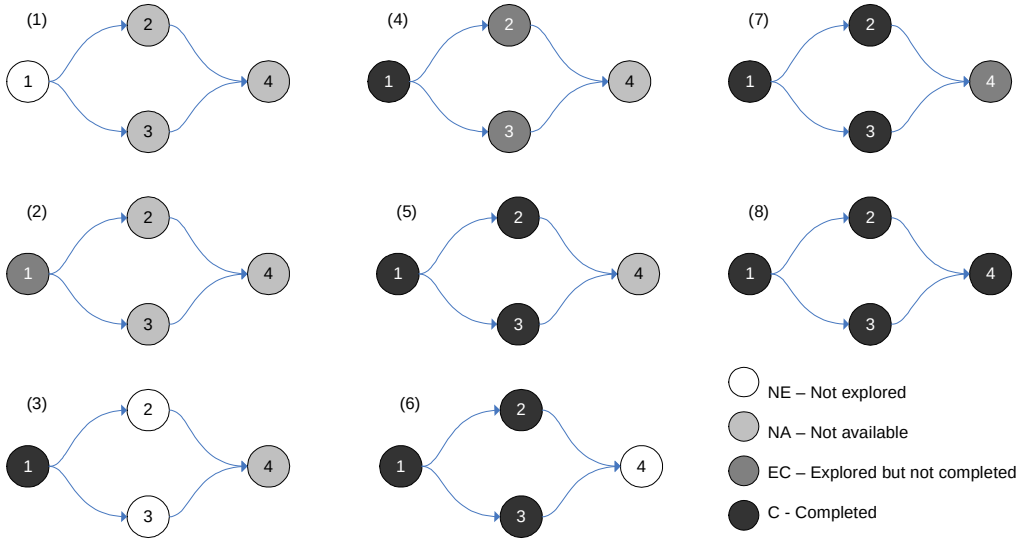



Figure 5. Workflow task state transitions

State transitions are handled by re-labeling tasks as precedence constraints are met. Initially entry tasks are set to NE state and all other tasks are labeled NA as in step 1 in Fig. 12. The grid scheduler can only allocate tasks in NE state. Once a task is allocated to a machine, scheduling and computation delays occur, during this time interval the task label is set to NA, i.e. steps 2,4, and 7. Upon task completion the task label is set to completed state (C) and immediate successor tasks are set to NE state, as in steps 3 and 6. This procedure continues until all tasks are labeled (C).

**IV. 2.2 Workflow scheduling phases**

Site level schedulers in tGSF receive jobs that arrive over time and allocate them to resources, so that there are no guarantees of producing optimal schedules. Therefore, tGSF falls under the online scheduling model. In spite of the current scheduling model, the simulator can be used to study other scheduling problems by setting problem parameters accordingly. For instance, to study deterministic scheduling problems, all job release times are set to a constant value.

Based on the fact that numerous deterministic workflow scheduling strategies bare common functionality, we have procured the design of a loosely coupled workflow scheduling architecture. Such a design is attain by defining interfaces for brokering, allocation, and sorting services of jobs. Services are implemented as policies.

Static workflow –DAG- scheduling strategies generally employ three phases (Topcuouglu et al. 2002):

- **Labeling phase** prioritizes each job in the workflow. Priorities are later used to establish the order in which jobs are to be scheduled. Classical Labeling strategies include: upper rank, downward rank, and as late as possible.
- **Job selection phase**, jobs are selected based on the objective of the selection criteria. Higher or lower priority jobs are selected first and passed to the assignment phase.
- **Job assignment phase**, an optimization function is used to determine the best location for job placement. Typical optimization functions include: earliest start time and absolute latest start time.

These three phases have been rigorously used in many designs of static and dynamic DAG scheduling strategies (Kwok and Ahmad, 1999; Topcuouglu et al. 2002; Bittencourt and Madeira, 2010). In this work, prioritization, selection, and assignment phases are defined as interfaces. Furthermore, phases are implemented as policies. For instance, prioritization phase policies can be downward rank, upper rank, as late as possible, or other. Such a design, enables exploring different scheduling settings. To this writing, we have extended tGSF to support two workflow scheduling strategies, namely HEFT (Heterogeneous Earliest Finishing Time) and CPOP (Critical Path on Processor) (Topcuouglu et al. 2002). Components that enable workflow support and their interactions are subject of the following section.

### IV. 2.3 Workflow scheduling

The primary goals of the design of the proposed workflow scheduling environment are to provide an extensible, highly cohesive, and loosely coupled infrastructure. Extensibility is supported by using wherever possible interfaces or abstract classes that provides specializing functionality, such as scheduling policies, optimization functions and metrics, schedulers, and job administration brokers. High cohesion and loosely coupling are partially attained by making components functionality self contained, and providing mechanisms for processing shared data. This objective is achieved by abstracting each job's control and state data into independent Job Control Blocks (JCB). Such a design principle is commonly used to model process and threads in modern operating systems (Silberschatz et al. 1991; Tanenbaum, 2007).

Workflow scheduling support is based on the scheduling phases described in the previous section. It is an aggregation of specialized schedulers, strategies, information providers, and brokers that orchestrate scheduling of workflows. The following components enable workflow scheduling support:

- **Abstract Grid Broker**: The Abstract Grid Broker (AGB) receives jobs from a grid submission component, and forwards them to a parallel job scheduler or to a workflow scheduler. Several grid broker implementations are possible, for instance, a distributed, hierarchical, or centralized. Differences between grid broker implementations may lie in the visibility of resources, interconnection topology, among other factors. A Centralized Grid Broker (CGB) with complete view of all computational sites in the grid is designed.
- **Abstract Scheduler**: The Abstract Scheduler (AS) manages jobs by applying a parallel job or workflow scheduling scheme. Two concrete schedulers are distinguished: Parallel Job Scheduler (PJS) and Workflow Scheduler (WS). WS processes workflow jobs by ranking independent jobs, ordering them based on a selection criterion, and determining job destinations. The preceding process is repeated as precedence constraints are broken.
- **Strategy**: A strategy is an extensible interface used to define job assignment policies, for example, random, minimum parallel load, minimum lower bound, etc. Polices are discussed later in Section V.1.2.

- **Information system**: The Information system (IS) provides a query interface for status information of compute sites and jobs execution time estimates. Assignment policies may be used by some strategies to gather information for decision making purposes.
- **Dispatcher**. The dispatcher forwards jobs to the destination site determined by a strategy.

The sequence of events performed during the collaboration of workflow scheduling components may not be unique, as workflow scheduling strategies may omit some steps of the scheduling scheme. Others may reevaluate values as the scheduling process progresses. This is the case of dynamic critical path workflow scheduling strategies. In spite of the differences in sequencing of events, we illustrate a general set of events for scheduling workflows in Fig. 13 and describe them next.

1. The scheduling process is initiated, when jobs are submitted –offered- to the CGB via a grid submission component. Sessions are used to send one or more jobs as a single request. Session duration is bounded, if session torn down occurs before consuming all session jobs, the CGB does not schedule them. Note that the CGB is a concrete implementation of AGB.
2. Upon reception of a workflow, jobs are pushed into the Composite Job Queue (CJQ). The CGB forwards the job to the PJS or the WS based on the job type.
3. The WS manages the job by creating or updating a workflow JCB. Job management includes setting the workflow state, internal workflow jobs states, and storing accounting data. Afterward, a Labeling policy is used to rank all workflow jobs. A selection policy criterion is used to select a set of independent jobs ordered according to an ordering policy.
4. Jobs in the independent job set are processed based on their relative ordering. The WS schedules each job by calling an Assignment Strategy (AS) to determine which site will host a given job. Note that one or more assignment strategy can be used.
5. During the execution of the strategy the Grid Information Broker (GIB) polls job execution time estimates (i.e. earliest start time, earliest finishing time, etc.) or site status information (i.e. number of waiting jobs, total running jobs, number of resources, etc.). Ultimately, the strategy selects a destination site and adds localization information to the JCB.

6. As job assignment is determined, the WS pushes jobs into the Independent Job Queue (IJQ), and creates a Job Queued Event (JQE) for the job.
7. On the occurrence of a JQE event, the PJS retrieves –pop- localization information from the IJQ and calls the dispatcher.
8. The dispatcher adds provenance information to the job and dispatches it to its destination site.

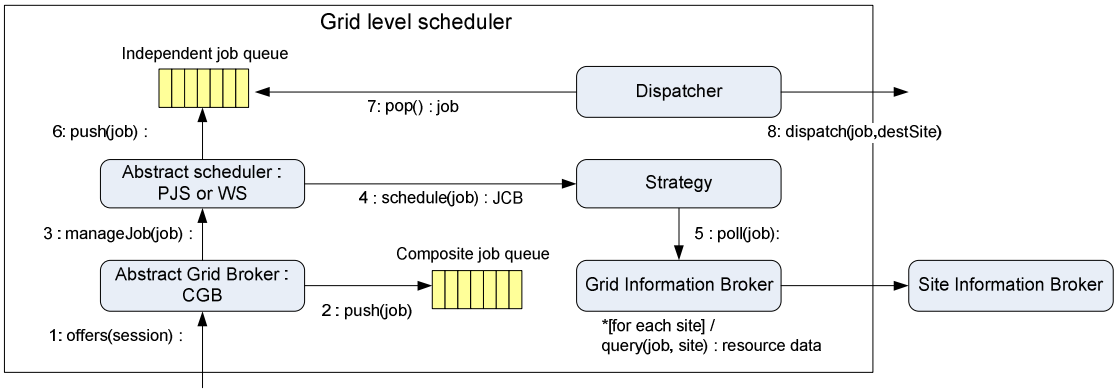



Figure 6. Parallel and composite job tGSF extended grid scheduling support

In tGSF, the policies and strategies referenced in steps 4 to 6 are defined in an ASCII text based configuration file. They are dynamically loaded during simulation runtime.

### IV. 2.4 Parallel Job Scheduling

In order to define a suitable scheduling methodology, the following model restrictions are considered. A computational site schedule is only accessible to its local scheduler. The grid scheduler does not have access to the computational site schedules, it can only request state data and job execution time estimates. Estimates do not guarantee resource availability, nor that reservations performed.

Parallel job scheduling is thus straightforward. Given a parallel or independent job, an assignment policy is used to evaluate the candidacy of computational sites. First, a subset of admissible sites is built by selecting computational sites that can fulfill a given job resource requirements. Then, the candidate site is selected according to the optimization criterion of the allocation strategy. Once the target site is determined, the job is sent to the computational site.

The parallel job grid scheduling strategies proposed in (Ramirez-Alcaraz, et al. 2011) have been implemented. Ramirez-Alcaraz et al. (2011) classified job allocation strategies based on the amount of information required for decision making. Four levels are distinguished, strategies of level 1, 2, and 3 use site status information, while strategies of level 4 require job execution time estimates. See Chapter V for parallel job allocation strategies implemented in the new grid scheduling framework.

Level 1 throughout 3 strategies use the GIB to query site status information. Status information is used by assignment strategies to select a target computational site that will host a given job. Level 4 strategies use the GIB to pull job completion times $C_j$. Each computational site $i$ that can host a given job $j$ forwards $C_j^i$ to the GIB.

Providing job completion time estimates is expensive, for it may require creating a schedule at each queried site. Furthermore, estimates do not guarantee that computational resources are reserved, nor that the validity of estimates will hold once jobs has been allocated.

**IV. 2.5 Parallel and composite job static model**

Scheduling strategies are used to select suitable resources given job resource requirements. In order to have support for both parallel and composite job scheduling at grid layer, two interfaces were included in Teikoku, namely: *RStrategy* and *CStrategy*. The first one is used to include parallel job allocation strategies, and the second one to provide both Labeling and allocation support for workflows (See Fig. 14).

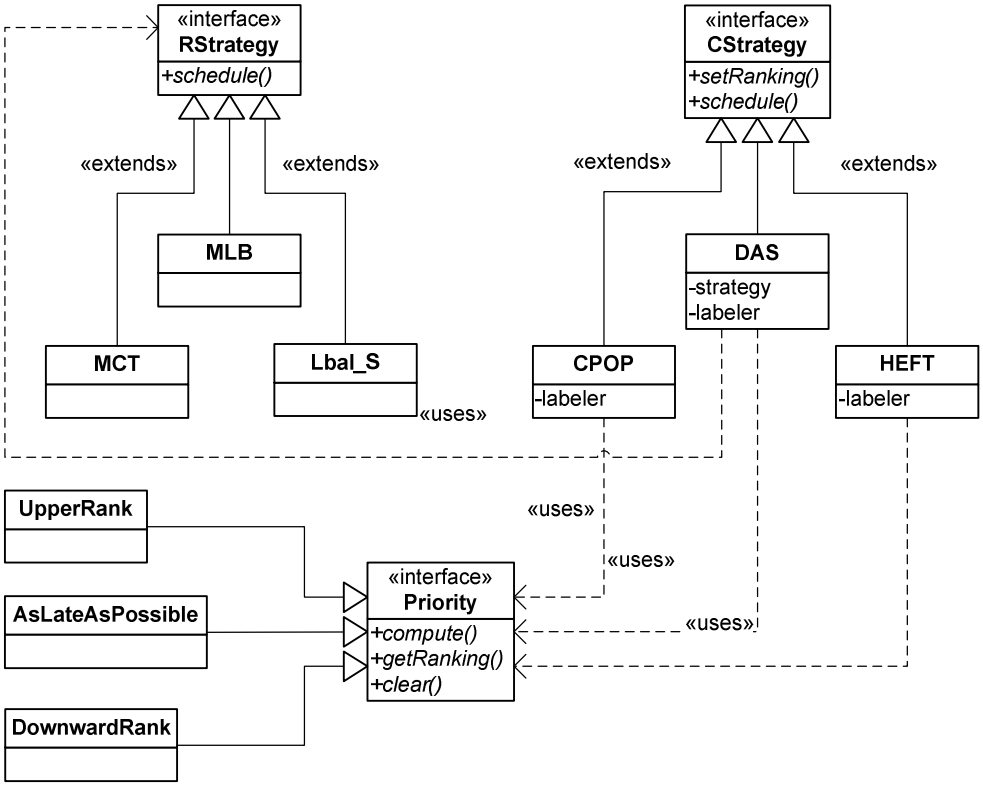



Figure 7. Parallel and composite job static model

Workflow scheduling heuristics cannot be modeled by the strategy design pattern. Since there exist various approaches –paradigms- to schedule DAGs and they often differ in the goals they intend to achieve. For instance, clustering strategies aim to eliminate communication costs by grouping tasks and allocating them to the same resource, while most list scheduling strategies maintain an order list of independent jobs, allocating them in such an order. Clearly, it is complex to find a proper representation that can model such differences in a generic way.

Most workflow scheduling strategies designs are highly cohesive. Many algorithms cannot be decomposed into finer grain components. Algorithms that are subject to such limitations are modeled as concrete classes. CPOP and HEFT are example of these.

The Dynamic Allocation Strategy (DAS) heuristic has been design to model algorithms based on the list-scheduling paradigm. It uses a Labeling strategy to assign a numerical value to a tasks and an allocation strategy to select resources. DAS allows binding a Labeling and allocation strategy at runtime. Therefore, multiple configurations are possible. For a complete list of allocation strategies and Labeling strategies, see Chapter V.

**IV. 2.6 Providing information to strategies**

Scheduling strategies may require accounting data or job execution estimates for decision-making purposes. In Teikoku, a grid information component provides services for consulting resource availability, state, and architectural characteristics. Its design is based on the Grid laboratory Uniform Environment (GLUE) schema (Balazs Konya, 2009).

Teikoku's grid information component is composed by a Grid Information Broker (GIB) and the Site Information Broker (SIB). The GIB provides services that enable sampling data from one or more SIB's, whereas SIB retrieves information from local resources.

State and accounting data are stored in a Compute Site Information (CSI) data structure. tGSF version 0.1 only supports CSI that describes the load of the machine, all do the architecture is extensible so that other resources can be included.

State data are obtained using a stateful pull querying mechanism. At each pull request, a copy of the requested information is buffered at the SIB side. A time out is associated to the buffered data. When it expires the data are flushed. The time out period can be set by the researcher. A time out period of -1 indicates that buffering is not allowed, therefore each time a pull is done the SIB retrieves state data from its local resources. Sampling of the resource state is allowed only once between two consecutive time out events, when the time out value (in milliseconds) is greater than zero buffered data may become stale. Such a scenario occurs when the time out intervals is large and the machine state changes rapidly. The sensibility of stale data on allocation strategies is address in Section VII.3.

The CSI state data structure contains variables that account for compute site capacity bounds and accounting information. The first gives quotas on the maximum number of jobs, running jobs, and waiting jobs. Accounting data inform the load condition of the machine, for instance: total number of executed jobs, running jobs, waiting jobs, etc.

A job runtime execution estimate gives a forecast on the task processing time. Forecasting is complex, since it must consider: the state of the machine schedule; waiting jobs; and past forecasts. The SIB stores past forecasts and uses a mechanism that enables it to know when these are no longer valid. Two types of forecasts are supported: earliest start time and earliest finishing time.

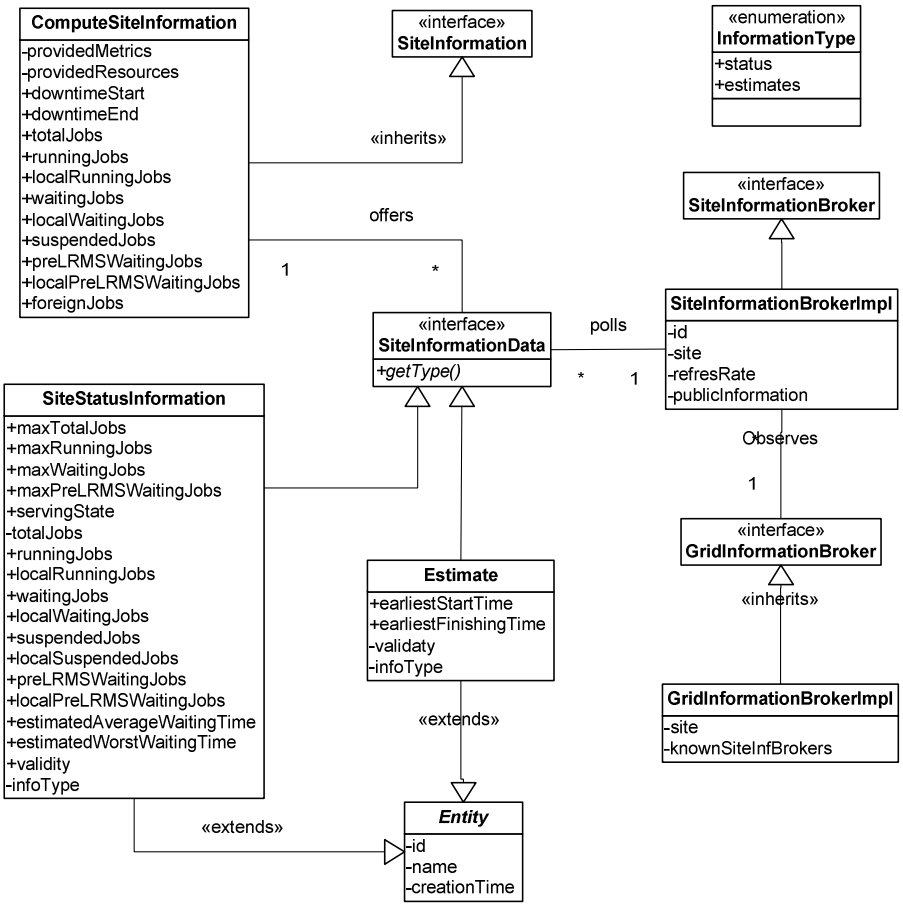



Figure 8. Information broker static model

The static diagram of the grid information component is illustrated in Fig. 15 (bellow). Concrete implementations for the site information broker and grid information broker are given by Impl suffix named classes.

In real production systems, a compute site can be viewed as an aggregation of computational, storage, or other forms of devices, each containing attributes that describe their state. Only computational resources are modeled in Teikoku via the ComputeSiteInformation object that describes the machine:

- **Resource quotas**, such as the maximum number of jobs, running jobs, and waiting jobs
- **Runtime information**, such as: total jobs in the system; number of local and foreign: waiting, running, and suspended jobs

- **Servicing state**, which indicates whether the machine is operational or not
- **A time out attributed** referred to as validity.

The information broker can obtain either status information or job runtime estimates from computational resources. They are modeled by the SiteStatusInformation and Estimate objects correspondingly. Status information is easily acquired by consulting the state of queues, local schedules, etc. Obtaining job runtime estimates is more complex, since it requires knowledge of the real schedule and of past job runtime requests.

The SIB encloses estimates or status information in messages -SiteInformationdata- and forwards them to the GIB.SIB is stateful, it makes use of a createTime attribute to discard obsolete messages. Therefore, each pulled message is tagged with a creation time.

At grid level workflow scheduling algorithms and parallel job allocation strategies use knowledge of the execution context. In order to retrieve such a knowledge, heuristics call the GIB and pull data of interest from the SIBs. Given a job, the GIB is also capable of returning a list of admissible resources, is sorted in non-decreasing order of machine size.

**IV. 2.7 The event handler**

An event is an internal or external stimulus used for synchronization purposes. These can be produced by an external interrupt, timer expiration, signal, or the arrival of a message. In tGSF the most common type of event is the timer expiration event. Events occur at discrete points of time.

The event manager (EM) binds events to actions. When an event occurs, the EM initiates some data treatment process. In tGSF the data treatment process consists of three phases: pre-processing, processing, and post-processing. Pre and post processing phases are used for logging, evaluation, or accounting purposes. While in the processing phase, the EM calls one or more components to carry out some computation. Table II summarizes component-event associations present in the simulation environment.

The term job is used to refer to a workflow and task to an independent rigid or sequential application. State transitions and actions vary depending were the job originates. In tGSF jobs can arrive at site and grid layers. Table X shows job states when these arrive at grid layer.

TABLE II
Parallel and workflow job events

| Event | Event handler calls |
|---|---|
| Released | Job submission component,<br>- Forwards the job to the parallel or workflow job broker at grid layer<br>- Uses a scheduling heuristic and selects the target machine<br>- Inserts precedence constraint free tasks in the independent task queue<br>- Creates an allocation event |
| Queued | Site scheduler, uses the parallel machine scheduler to<br>- Determine the task start time<br>- Creates a task started event |
| Started | Site executer, emulates execution of the task, it<br>- Uses a local scheduling strategy to determine the absolute task completion time, it includes task waiting times if defined<br>- Creates a task completed event |
| Completed | Site scheduler, uses the parallel machine scheduler to<br>- Determine if the task source is foreign, if so, it creates a task completed in foreign site event<br>- The performance of the task is evaluated at site level |
| Completed on foreign site | Grid broker, uses a parallel or workflow job broker to evaluate the performance of the task or to signal the grid broker<br>- If the task is parallel, it evaluates its performance via some metric<br>- If the task belongs to a composite job, it frees precedence constrains if any.<br>- If all workflow tasks completed, it creates a workflow completed event |
| Allocation | Dispatcher,<br>- Allocates the task to the preselected target machine |
| Workflow completed | Metric,<br>- Evaluates the performance of the workflow job |

In order to provide grid layer parallel and workflow job scheduling support, the job allocation and workflow completed events were included in Teikoku. In addition, the job completed on foreign site is used to advice the grid broker when tasks complete execution at foreign hosts.

**IV. 2.8 Performance metrics**

User and system centric metrics evaluate the performance of the grid infrastructure. Evaluation is conducted after the expiration of an event, after a job completion event or workflow completion event. Performance may also be evaluated during a pre-processing, processing, and post-processing phase. Teikoku ver. 0.1 provides support for evaluating the average response time, average slowdown, average wait time, average weighted response time, average weighted slowdown, and average weighted wait time of parallel jobs.

We have implemented three workflow scheduling metrics: approximation factor, mean waiting time, and mean critical pat slowdown. See Chapter V for details. No significant changes in

Teikoku are necessary to include workflow metrics. As discussed in the previous section, the event manager need only capture the job completion on foreign site and workflow completion events and call upon the appropriate metric.

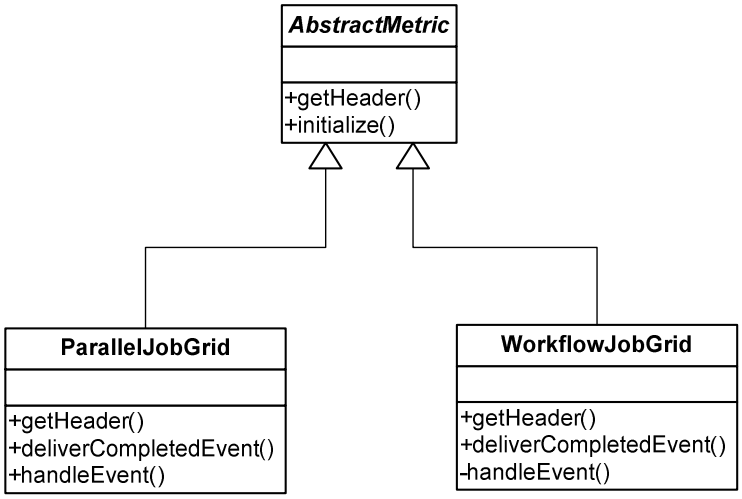



Figure 9. Parallel and workflow job metrics static model

Metrics are added by extending the AbstracMetric abstract class. Concrete classes must overload the getHeader method, which label the columns to be written to a trace file. Each column represents a metric of interest and lines, task performance data.

During a simulation, the deliverCompleEvent method captures the event notification forwarded by the local event handler and verifies if the current machine is the recipient. If so, the event is passed to the handleEvent method, which implements the metric. Fig. 16 illustrates the static class model of the metric class. In it, parallel and workflow metrics are contained in the ParalleJobGrid and WorkflowJobGrid classes correspondingly.

Stakeholders bind metrics to job types by defining an association in a Java properties file, named teikoku.properties, by setting each machine metric attribute to the name of the metric of interest. As illustrated in the following example

sites.site0.registeredMetric.ref = workflow_grid

Here, machine label site0 contains the grid broker. The registeredMetric.ref label is set to the workflow_grid, whose behavior is defined by the following properties

metrics.id = workflow_grid

***metrics.workflow_grid.class = mx.cicese.dcc.teikoku.metrics.Grid.WorkflowJob_Grid***

metrics.workflow_grid.writer.class = de.irf.it.rmg.core.util.persistence.CSVFilePersistentStore

metrics.workflow_grid.writer.mayOverwrite = true

metrics.workflow_grid.writer.outputfile = 1

In the previous code except, metrics are implemented in the WorkflowJob_Grid class. Performance data produced during the simulation can be written to persistent storage by sending it to a relational database or to a CVS (Comma Separated Values) file. The CSVFilePersistentStore class is used to write performance data to a CVS filed named after the performance metric, in this case a workflow_grid_1.cvs file stored in the project working directory. The file may be overwritten if the simulator is re-executed; this is done by setting the mayOverwrite attribute to true. The outputfile attribute is used as a prefix of the output file name, for instance by making outputfile equal 2, the name of the output file becomes workflow_grid_2.cvs.

**IV. 2.9 The simulation environment**

In tGSF version 0.1, the simulation environment is setup by defining a set of sites and their corresponding properties in a Java properties file. Properties are used to define: the number of available resources, session life time, queuing policy, site information broker caching life time, and a set of performance objectives. Properties are set by initiating tGSF system constants, such as: numberOfProvidedResources, informationBroker.class, etc. System constants can be found in tGSF package API documentation (Grimme, 2009).

A site can be configured to function as computational site or as a centralized grid scheduler; this is performed by initiating the activityBroker.class or the gridActivityBroker.class properties. Only one grid activity broker can be set, since the implementation of the grid activity broker is centralized.

A single computational site and centralized grid scheduler site Java properties file configuration excerpt is illustrated in the following pseudo code. Only key configuration features are presented.

**# Site 1 runtime information and properties**

**# Runtime information**

runtime.workloadSource.id = <id1>

runtime.< id1>.associatedSite.ref = site1

runtime.< id1>.url = file:/C:/<workload file>.swf

**# Site 1 properties**

sites.<id1>.numberOfProvidedResources = 1

sites.< id1>.submissioncomponent.sessionlifetime = 100

**# Activity Broker**

sites.<id1>.gridActivityBroker.class  = ... CentralizedGridActivityBroker

**# Grid information server**

sites.<id1>.gridInformationBroker.class =... GridInformationBrokerImpl

**# Composite job strategy**

sites.< id1>.grid.composite.strategy.class = ... HEFT

**# Rigid job strategy**

sites.<id1>.grid.rigid.strategy.class =  …Rand

**# Optimization function**

sites.< id1>.registeredMetric.ref = bounded_slowdown

**#Site 2 runtime information and properties**

**# Runtime information**

runtime.workloadSource.id = <id2>

runtime.< id2>.associatedSite.ref = site2

runtime. <id2>.url = file:/C:/<workload file>.swf

**# Site2 properties**

sites.<id2>.numberOfProvidedResources = 100

sites.<id2>.localsubmissioncomponent.sessionlifetime = 100

**# Activity Broker**

sites.<id-2>.activitybroker.class = …StandardActivityBroker

**# Information broker**

sites.<id2>.informationBroker.class = … SiteInformationBrokerImpl

sites.<id2>.informationBroker.refreshRate = -1

…

**# Acceptance policy**

sites.<id2>.activitybroker.acceptancepolicy.class = … StandardAcceptancePolicy

**# Scheduler and scheduling policy**

sites.<id2>.scheduler.class =…ParallelMachineScheduler

sites.<id2>.scheduler.localstrategy.class = …FCFSStrategy

**# Queue policy**

sites.<id2>.scheduler.localqueuecomparator.class = …

NaturalOrderComparator

sites.<id2>.scheduler.localqueuecomparator.ordering = ascending

**# Metrics**

sites.<id2>.registeredMetric.ref = art

Two brokers are defined, site 1 role is set as a grid activity broker (CentralizedGridActivityBroker) and site 2 role is set as a computational site (StandardActivityBroker). Workloads are associated to each site by setting the url property, for example, site 1 workload is runtime.site1.url = file:/C:/gridWorkload.swf. The number of processors of a site is defined by the property numberOfProvidedResources. Site 2 hosts 100 processors.

Recall that the information system is used to provide strategies state or job execution estimates. Site information broker refresh rate informationBroker.refreshRate. A value of -1 disables caching state data, it ensures that sampling is always performed. State data become stale if they are buffered more than a predefine amount of time set by the informationBroker.refreshRate to a positive integer value greater than one (milliseconds). A value of 600,000 indicates that information queries within a 10 minute time frame.

The centralized grid scheduler scheduling policies are set by initiating the grid.composite.strategy property to HEFT, for workflow scheduling support, and the grid.rigid.strategy property to Rand, for parallel job scheduling support.

Computational sites may perform load balancing of locally submitted jobs, by setting the transfer, location, and distribution policies properties. This feature may be used to create

dynamicity at site level scheduling layer, or may be disable so that once a job is assigned to a computational site, it cannot be delegated to others sites.

### IV.3 Grid workflow scheduling systems

Grid workflow scheduling systems are used to manage resources that are geographically distributed. Their development has been driven by the complexity of modeling and processing large-scale problems or complex services in academic and industrial settings. Over the years many grid workflow scheduling systems have been proposed, amongst the most well-known include Pegasus (Deelman et al. 2005), DAGMan/Condor (Couvares and Wenger, 2007), and Nimrod/G (Buyya et al. 2000). Many of these use the Globus (Chicago, 2011) middleware to aggregate and access grid resources transparently. Although there has been significant progress in providing tools and languages for modeling and executing workflows, a significant number of developers, large learning curve, and access to distributed resources are needed to deploy them.

Validating the performance of workflow scheduling strategies in production systems is complex and time consuming. To perform such a task, it is often necessary access to resources, source code, and instrumentation tools. Furthermore, workload and resource utilization conditions are unlikely to be reproducible in shared execution contexts. Because research reproducibility is a basic principal of quantitative experimental analysis, simulation thus seems a feasible method for studying resource administration problems.

A large body of work has been devoted in developing grid scheduling simulators. Out of the reviewed works, SimGrid (Casanova, 2001) and GridSim (Buyya and Murshed, 2002) have experienced widespread acceptance from the research community, as they have been used to evaluate the performance of multiple scheduling heuristics in: computational, economic, and data grids. Often, simulators simplify the scheduling problem. Other works aim at building simulation tools that are scalable, fast, and that can conduct large simulation experiments (Phatanapherom et al. 2003).

The base line feature that is found in all grid simulators is the capacity to model computational grids. More advance tools can model other features, such as hierarchical heterogeneous grids (Bittencourt and Madeira, 2010); resource interconnectivity, capacity constraints, among other characteristics. In the following paragraphs, we briefly summarize their features and know application.

SimGrid is an event-driven simulation toolkit used to study scheduling algorithms in distributed environments (Casanova, 2001). Originally, design to model computational grids. It optionally uses workload traces to simulate background load on time-shared resources, emulating resource contention phenomenons. In SimGrid a problem is modeled as a DAG, with tasks representing computations and edges data dependencies between tasks. Data dependencies are also modeled as tasks, therefore two types of tasks are distinguished: computation and communication tasks. The grid topology is configurable, being responsibility of the user to set resource capabilities and communication dependencies among them. Authors discuss that SimGrid performance is affected when high quantities of I/O operations on large trace files are performed.

GridSim is a discrete event simulator developed on top of the SimJava API. It supports a wide variety of features: modeling of heterogeneous resources, space-shared and time-shared simulation capabilities, resource reservations, support for modeling parallel applications, among other features, see (Buyya and Murshed, 2002) for features. GridSim can model computational and data grids. Aldo the simulator models many issues present in real grid production schedulers, performance and scalability issues render it unsuitable for large-scale simulations.

GangSim is design with the objective of studying scheduling strategies in grids (Dumitrescu and Foster, 2005). GangSim differs from other grid simulators in that it allows modeling Virtual Organizations (VO). The simulator has been used to study interactions between local and community resource allocation policies. Authors show that simulation results are comparable, in some aspects, to those generated by a real grid production system (Grid3), but fail to reproduce fine grain results of local schedulers.

Xavantes is a distributed process and workflow execution context based on Java-RMI (Bittencourt and Madeira, 2010). It provides a programming model based on BEPL (Business Process Execution Language) to model precedence constraints jobs. It provides support for scheduling workflows independently or in a time-shared manner. In Xavantes, machines and cluster of machines can be arbitrarily interconnected. A cluster can have distinct processing speeds. Communication capabilities between clusters can be heterogeneous. Xavantes provides a two-layer grid scheduling model, cluster and grid layer

A distributed scheduler with load balancing capabilities is proposed in (Shan et al. 2003). The scheduling architecture is composed by a set of autonomous local schedulers that delegate

workload via a grid middleware referred to as SuperScheduler (SS). Two main components in the SS architecture are distinguished, a job transferring and a resource selection policy. Three resource selection policies are proposed: sender-initiated, a proactive policy where the source machine sends the job resource requirements to peer-hosts and chooses the one that minimizes an objective of interest; receiver-initiated, a reactive policy where target machines check their local load periodically. If it drops below a threshold, it volunteers itself by informing other hosts of its low utilization; and symmetrically-initiated, works in both proactive and reactive modes.

Economic grid simulators aim is on provisioning an environment in which scheduling decisions are made dynamically and are directed by the end-user requirements. The major challenges in managing resources in economic grids are addressed in Buyya et al. (2005) and Broberg et al. (2008) works.

EcoGrid is used to simulate economic and non-economic based grids (Mehta et al. 2011). It was design to study the impact of load balancing strategies in time-shared and space-shared grids. It is an object oriented simulator that models different types of resources, supports advance reservation, and can integrate new scheduling strategies by extending a scheduling policy interface.

Data grid simulators provide mechanisms to emulate resource capacity constraints, machine topology, and communication costs. Data intensive scheduling problems have received less attention, many interesting problems and challenges in scheduling data-intensive workflows in grids are presented in (Deelman and Chervenak, 2008).

OptorSim is a time-based and event-based java simulator developed as part of the European DataGrid project (Cameron et al. 2003). It was design to study scheduling problems on computation and data grids. OptorSim scheduling strategies include: random; access control, which optimizes job allocation based on the time needed to access all files required by the job; queue size; and queue access cost, which considers the combine access cost of all jobs in the queue. The simulator includes task replication criteria for parallel jobs, namely: no replication; LRU (Least Recently Used), and LFU (Least Frequently Used) policies. An auction-based strategy uses the binomial economic model and the Zipf economic model that negotiates file replication. The simulator is implemented in Java.

## IV.4 Summary

This chapter presents tGSF extension for workflow and parallel job grid scheduling support, provided as a second layer of the scheduling stack. The job labeling, selection, and assignment strategies are self-contained and computed independently at runtime. The proposed components are loosely coupled, allowing study of different combinations of scheduling policies. To promote usage of the simulator, a setup example is given.

This work is an on process endeavor. The Teikoku research community entails many research branches, members, and institutions. Our branch deals with the study of workflow and parallel job scheduling heuristics in a computation grid. Other works deal with energy conservation, service level agreements, and peer-to-peer computing problems.

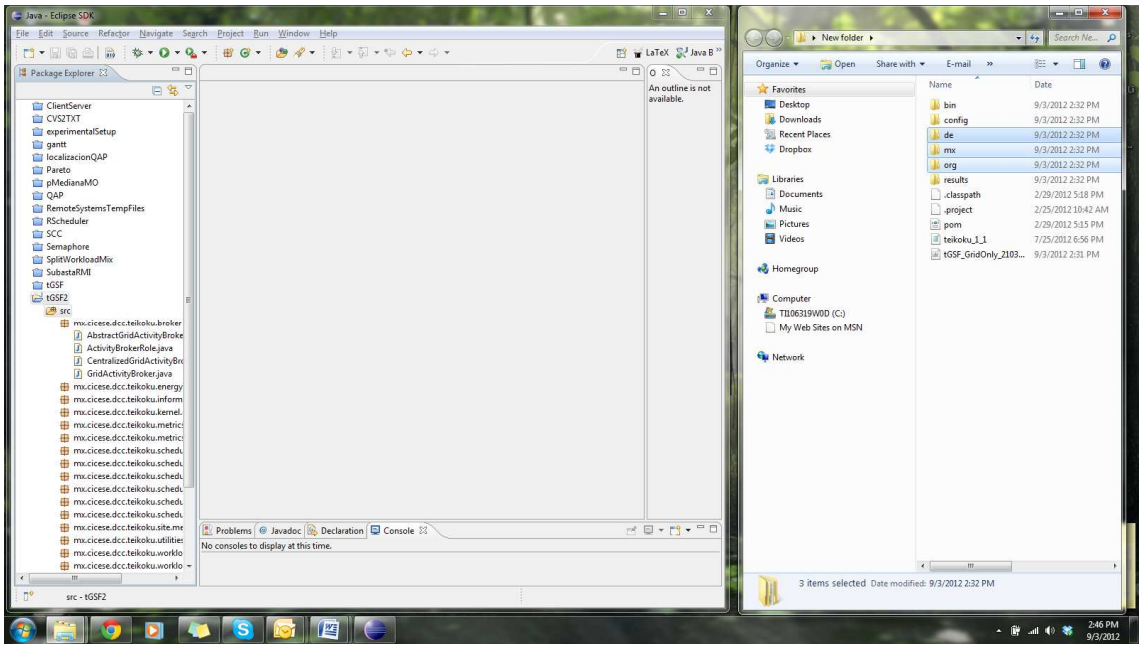
# Appendix A: Configuring the simulator in IBM Rational Architect

This section describes how to configure tGSF using the Eclipse development environment. The project can be configured with Maven. If for some reason it is not possible to get Maven stated, the programmer limit to create a java application and install jar files manually.

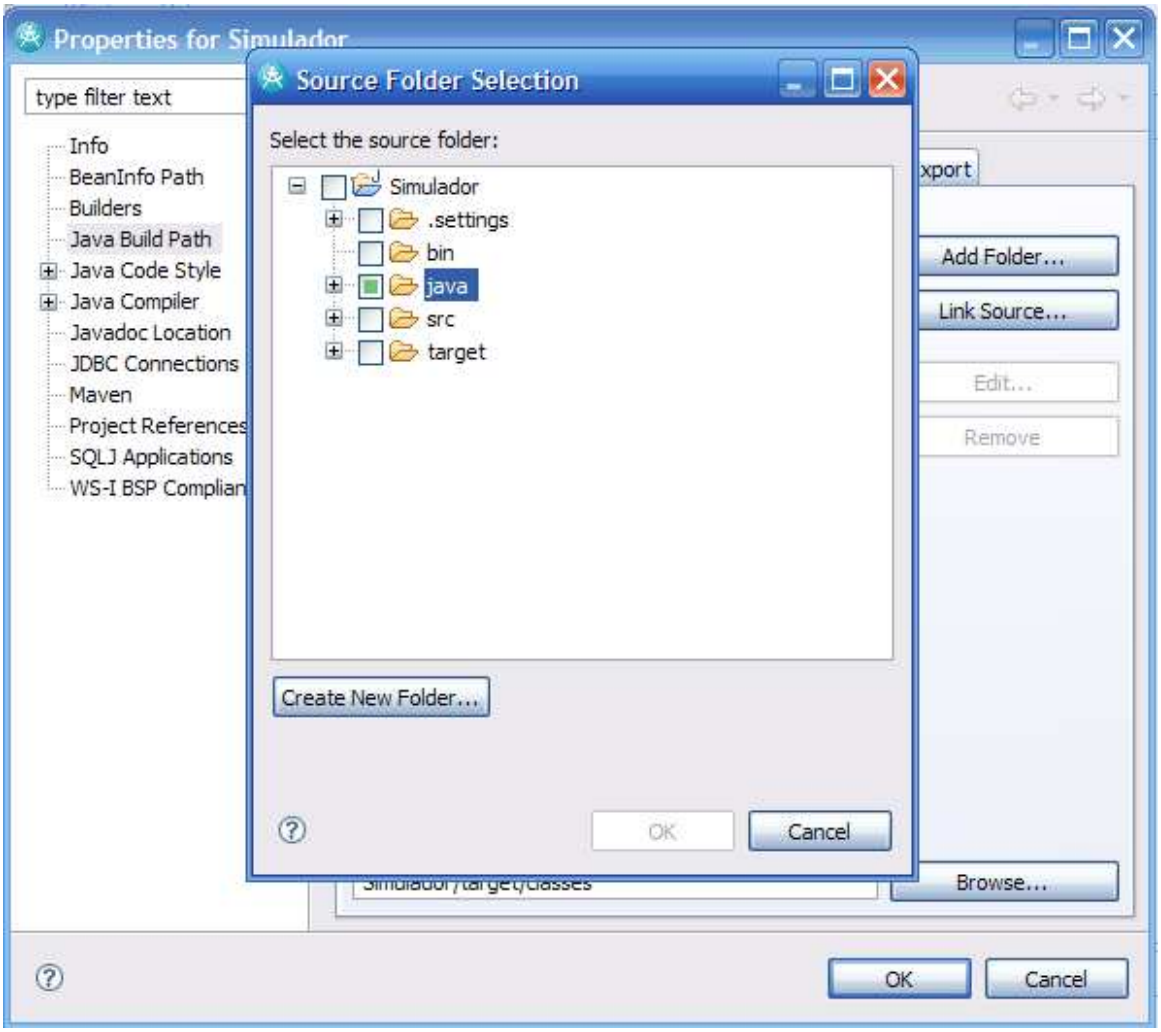
## Configuration of a new project

1. Create a Java application project in Eclipse: File →New → Java Project. The Create a Java Project window will be displayed. Type the project name and select the javaS-1.5 environment JRE. Select the finish button.

2. Decompress *teikoku.src.jar* file. It will create several directories, there are three main source subdirectories (de, mx, and org). The subdirectories and all their contents must be inserted in the eclipse src subdirectory. You can do this just by dragging and dropping the fore-mention subdirectories to the target src subdirectory.

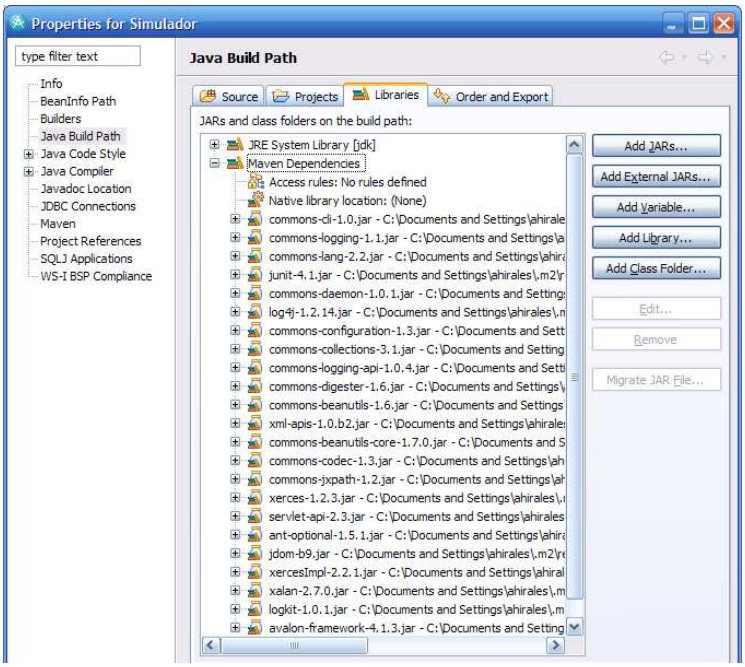



3. Edit the project properties. Select *the java build path* option. Select the source tab and press the *Add Folder* button. Such an action will display the *Source Folder Selection*, select the source directory (src). Confirm all actions by pressing Ok until the Properties for your project closes.

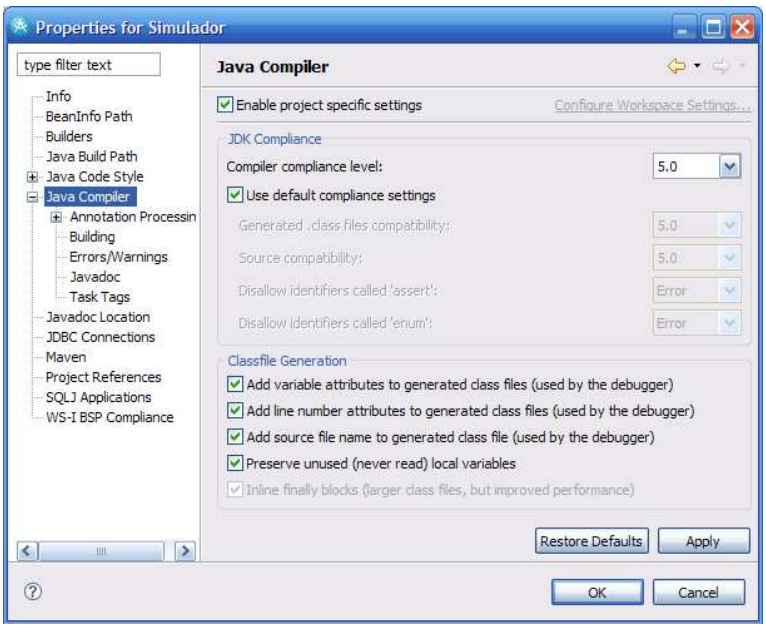



4. Select the project. Press the right mouse button. Activate Maven 2 dependency management.

5. Create a POM (pom.xml) file in the project root directory. The teikoku.src.jar contains an example POM maven configuration file, such can be drag and drop in the project root directory. After configuring the POM file, *update Maven dependencies*. Maven will download and include libraries (jar files) shown in the following image.



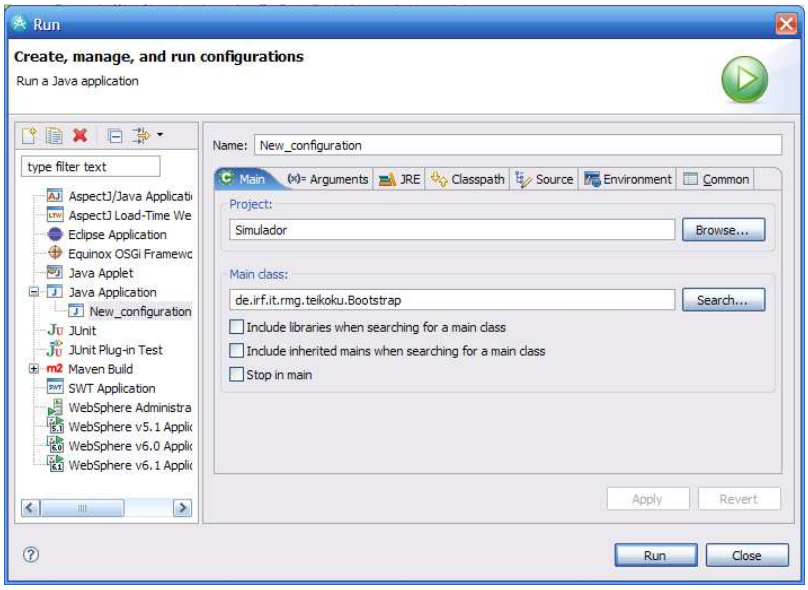

6. In the project properties window, chose the *java compiler option*. The *Compiler compliance level* must be set to 5.0 or greater (Note: this step is done if not conducted during step (1)).



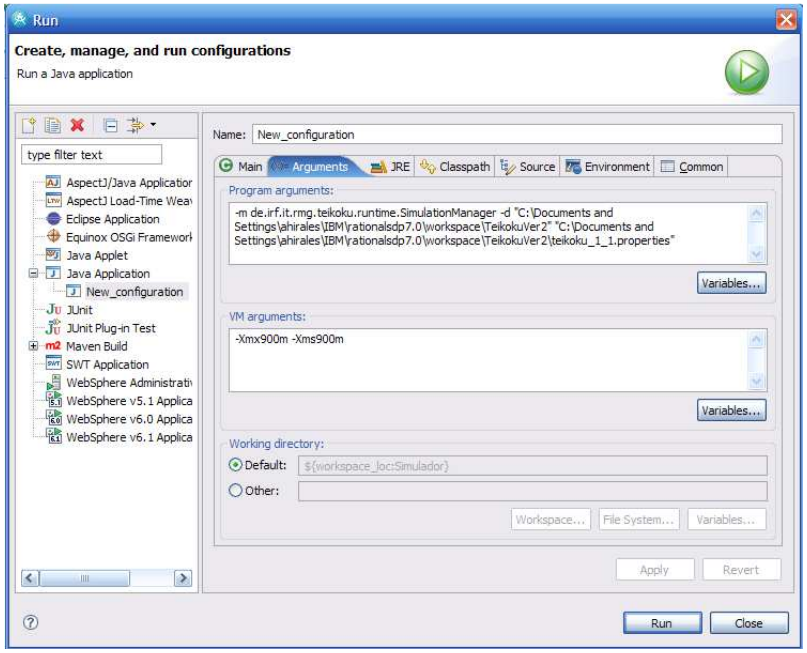
## Executing tGSF

To execute the simulator the following steps must be performed.

1. In the Main tab, specify the configuration name (New_configuration) and the main class (de.irf.it.rmg.teikoku.Bootstrap).




2. Select the Arguments tab, insert the program and VM arguments shown in the next figure.



**Program arguments are the following:**

-m de.irf.it.rmg.teikoku.runtime.SimulationManager -d "C**:\Documents and Settings\ahirales\IBM\rationalsdp7.0\workspace\TeikokuVer2**" "C**:\Documents and Settings\ahirales\IBM\rationalsdp7.0\workspace\TeikokuVer2**\teikoku_1_1.properties".

**VM arguments are the following:**

-Xmx900m -Xms900m

Note the path (bold characters in the program path arguments) may be different for each user configuration. Make sure that the path the eclipse project and to the teikoku_1_1.properties file is correct.